

## **JSpider**

## **JAVA**

### **JAVAC (JAVA COMPILER)**

- It checks the given program to find syntax mistakes.
- If there is any syntax mistakes then java compiler throws an error.
- The errors which is throw by javac are called as compile time errors.
- The java compiler generates .class file if there are no syntax errors in the program.

### **JVM (JAVA VIRTUAL MACHINE)**

- JVM is an interpreter which is going to
  - (i) Read one line of code.
  - (ii) Understand it.
  - (iii) Executes it.
- If JVM is not able to understand a line of code then JVM throws an error at runtime.
- The errors thrown by JVM are called as runtime errors or exceptions.

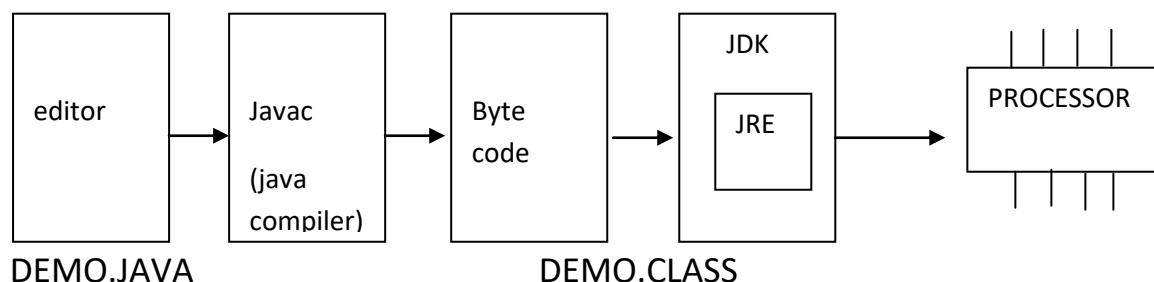
### **JRE (JAVA RUNTIME ENVIRONMENT)**

- It is used to setup the environment in with the help of operating system to execute the program of JVM.

### **JDK (JAVA DEVELOPMENT KIT)**

- It is a software package which contains java compiler, jvm and other necessary files which are required to compile and executes java programs.

### **JAVA WORK FLOW**



# CHAPTER 1 : KEYWORDS, IDENTIFIERS & VARIABLES

## KEYWORDS

- They are reserved words which have a predefined meaning.
- It is not possible to change the meaning of keywords in any programming language.
- Keywords are used to define a class, declare a variables or define a method etc. in java.

## Keywords in Java

Abstract	Continue	For	New	Switch
Assert ***	Default	Goto*	Package	Synchorinised
Boolean	Do	If	Private	This
Break	Double	Implements	Protected	Throws
Byte	Else	Imports	Public	Throws
Case	Enum****	Instanceof	Return	Transient
Catch	Extends	Int	Short	Try
Char	Final	Interface	Static	Void
Class	Finally	Long	Strictfp**	Volatile
Const*	Float	Native	Super	while

\* not used

\*\* added in 1.2

\*\*\* added in 1.4

\*\*\*\* added in 5.0

## IDENTIFIERS :

- It is used to identify class or interface, methods, variables etc.

## Rules of Identifiers :

- Identifiers can be alphanumeric.
- Identifiers should not start with numbers.
- \$ and \_ (underscore) are the only two special character allow.
- An identifier can start with \$ and \_.
- Keywords can not be used as identifiers.

## **VARIABLES :**

- A variables is a named memory location which holds the value for the program.
- To use a variables we have to follow three steps :
  1. Declaration
  2. Initialization
  3. Utilization (Usage)

### **Declaration of a variables :**

#### **Syntax :**

**Datatype variablename;**

**Eg : int age;**

- It is a statement which defines what type of data will be stored in the given variables.

### **Initialization of a variable :**

#### **Syntax :**

**Variable=value;**

**Eg: age=25;**

- It is a statement which is written to store the value into the variable using assignment operator.

### **Utilization of a variables:**

- Statements which are written to use the values of the variables are called as utilization.

## **Primitive Data types in java**

				<b>Default value</b>
<b>Integer</b>	Byte	8 bits	1 byte	0
	Short	16 bits	2 byte	0
	Int	32 bits	4 bytes	0
	Long	64 bits	8 bytes	0l
<b>Decimal</b>	Float	32 bits	4 bytes	0.0f
	Double	64 bits	8 bytes	0.0

	Char	16 bits	2 bytes	Blank space
	Boolean	8 bits	1 bytes	false

## CHAPTER 2 : OPERATORS

- It performs operations on the given operands and produce results.

### Arithmetic Operator

- To perform any arithmetic operator
- +, -, \*, /, %

Note- Variables of the program should not be printed with “double quotes”.

Note – System.out.print – prints & keep cursor in same line.

System.out.println – Prints & makes the cursor to next line.

### Formula to get how much a datatype maximum and min value range

$$-2^{n-1} \text{ to } 2^{n-1}-1$$

**Eg : for bytes**

$$-2^7 \text{ to } 2^7-1$$

$$-128 \text{ to } 127 \text{ (bytes)}$$

If we perform any arithmetic operation with the help of arithmetic operators with the help of arithmetic operators the result variables has to be decided by the following method.

**Max(int, type of 1<sup>st</sup> operand, type of 2<sup>nd</sup> operand)**

### Program :

(a)

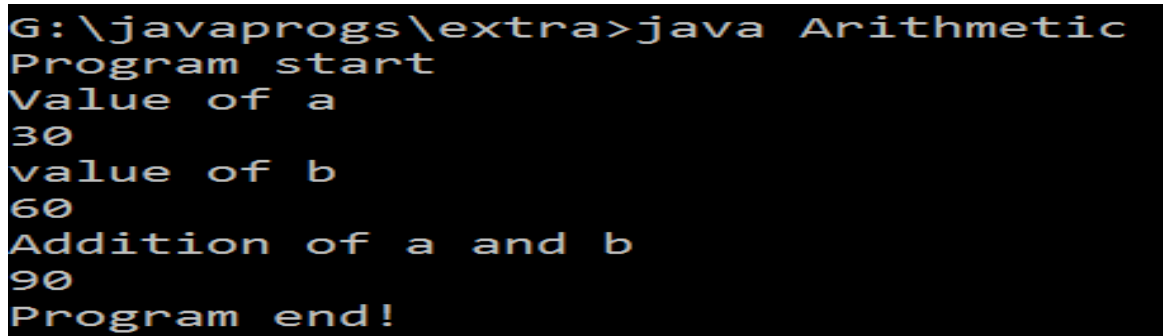
```
class Arithmetic
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        //declaration
        int a;
        int b;
```

```

int res;
//initialization
a=30;
b=60;
res=a+b;

        System.out.println("Value of a");
                System.out.println(a);
        System.out.println("value of b");
                System.out.println(b);
        System.out.println("Addition of a and b");
                System.out.println(res);
        System.out.println("Program end!");
    }
}

```



```

G:\javaprogs\extra>java Arithmetic
Program start
Value of a
30
value of b
60
Addition of a and b
90
Program end!

```

(b)

```

class Arithmetic2
{
    public static void main(String[] args)
    {

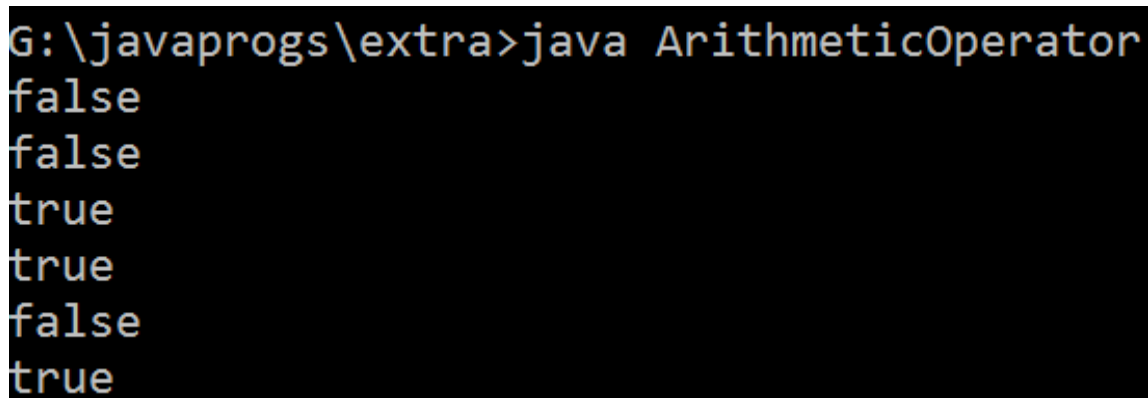
        int a=10; //declaration and initialization at same place
        int b=20;
        int c=a+b;

        System.out.println("value of a"+a);
        System.out.println("Value of b"+b);
        System.out.println("Addition of a and b"+c);
    }
}

```

(c)

```
class ArithmeticOperator
{
    public static void main(String[] args)
    {
        int a=5;
        int b=10;
        System.out.println(a>b);
        System.out.println(a>=b);
        System.out.println(a<b);
        System.out.println(a<=b);
        System.out.println(a==b);
        System.out.println(a!=b);
    }
}
```



```
G:\javaprogs\extra>java ArithmeticOperator
false
false
true
true
false
true
```

### Concatination Operator

- It helps in concatenating (joining) a string value with any other value.

### Combination of Concatination

- 2+ "hello" – 2hello
- "hello"+123 – hello123
- "hello" + "World" – helloworld
- 2+3+ "hello" – 5hello
- "hello"+2+3 – hello2+3 – hello23
- 2+ "hello"+3 – 2hello3

### Increment & Decrement Operator :

- Those operators are used to increase or decrease the value of a variable by 1 units.

## Increment Operator (++) :

- There are two types of increment operator.

### 1. Pre-increment.

### 2. Post-increment.

- If you write post or pre increment operators with a variables independently without any mathematical of expressions then, both operators will have same results.

Pre Increment	Post increment
First increment	Substitute
Substitute	Perform operation
Perform operation	Increment value

- Increment operator can not be used with Boolean datatypes.
- Any value can not be used directly with increment operators.
- Note :

```
Char a1= 'A';  
a1 = a1+1; //can not write like this  
a1=66;  
a1= 'B'
```

- **V.V.I notes**

<b>Byte b1=10; B1++; S.o.p(b1);</b>	<b>Byte b1=10; B1=b1+1; s.o.p(b1);</b>
<b>Does not show error</b>	<b>Show error just because of max function</b>

## Program

(a)

```
class Increment  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Program start");  
        int a1=10;  
        int b1=10;  
        int res;  
        System.out.println("a1 = "+a1);  
        System.out.println("b1 = "+b1);  
    }  
}
```

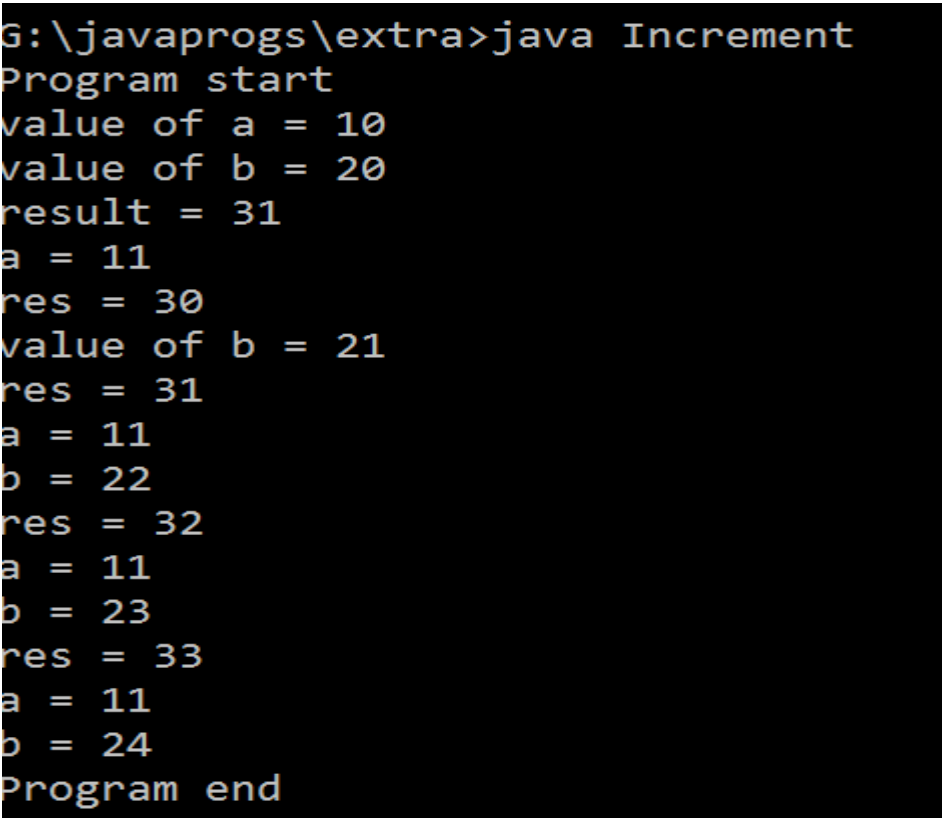
```

        res=++a1 + 10;
        System.out.println("res = "+res);

        res=b1++ + 10;
        System.out.println("res = "+res);
        System.out.println("a1 = "+a1);
        System.out.println("b1 = "+b1);

        res=b1++ + 10;
        System.out.println("res = "+res);
        System.out.println("a1 = "+a1);
        System.out.println("b1 = "+b1);
        res=b1++ + 10;
        System.out.println("res = "+res);
        System.out.println("a1 = "+a1);
        System.out.println("b1 = "+b1);
        System.out.println("Program end");
    }}

```



```

G:\javaprogs\extra>java Increment
Program start
value of a = 10
value of b = 20
result = 31
a = 11
res = 30
value of b = 21
res = 31
a = 11
b = 22
res = 32
a = 11
b = 23
res = 33
a = 11
b = 24
Program end

```



(b)

```
class Increment1
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
        int a1=10;
        System.out.println("a1 = "+a1);
        a1++;
        System.out.println( "a1 = "+a1);
    }
}
```

```
G:\javaprogs\extra>java Increment1
Hello World!
a1 = 10
a1 = 11
```

(C)

```
class Increment2
{
    public static void main(String[] args)
    {
        int a1=10;
        System.out.println(a1++);
        System.out.println(++a1);
    }
}
```

```
G:\javaprogs\extra>java Increment2
10
12
```

(d)

```
class Increment3
{
    public static void main(String[] args)
    {
        byte b1=10;
        b1++;
        System.out.println(b1);} }
```

```
G:\javaprogs\extra>java Increment3
11
```

## Decrement Operator: (--)

- Those operators are used to decrease the value of a variable by 1 units.
- There are two types of decrement operator.
  3. Pre-decrement
  4. Post-decrement
- If you write post or pre decrement operators with a variables independently without any mathematical of expressions then, both operators will have same results.

Pre decrement	Post decrement
First increment	Substitute
Substitute	Perform operation
Perform operation	Increment value

- decrement operator can not be used with Boolean datatypes.
- Any value can not be used directly with decrement operators.
- Note :

```
Char a1= 'B';  
a1 = a1-1;  
a1=65;  
a1= 'A'
```

## CHAPTER 3 : FLOW CONTROL STATEMENT

- It is a statement which is used to control the execution flow of a program.
- There are two types of flow control statements.
  1. Branching Statement (Decision making Statement)
  2. Looping Statement

### Branching Statement (Decision Making)

- These statements are used to execute a group of statements based on a Boolean condition.
- The type of decision making are :-
  - (a) If statement
  - (b) If else statement
  - (c) If else if statement
  - (d) Switch case statement

## IF STATEMENT

- If statement executes gives groups of statements within its body only if the Boolean condition result is true.
- Syntax :

```
If(condition) – true
{
    Statement ;
}
```

## PROGRAM

```
class IfStatement
{
    public static void main(String[] args)
    {
        System.out.println("Program Start");
        int v1=50;
        if(v1>10)
        {
            System.out.println("v1 is greater than 10");
        }
        System.out.println("Program end");
    }
}
```

## IF-ELSE STATEMENT

- Once the statements written in if block will be executed only if the Boolean condition is true and if Boolean is false then statements of else block will be executed.
- **Syntax:**

```
If(condition) true
{
    Statement;
}
Else
{
    Statement;
}
```

### **Program**

```
class IfElseStatement
{
    public static void main(String... args)
    {
        int a=97;

        if (a>b)
        {
            System.out.println("print a");
        }
        else
        {
            System.out.println("print b");
        }
    }
}
```

### **IF-ELSE-IF STATEMENT**

#### **SYNTAX :**

```
    If(Boolean condition 1)
    {
        Statement;
    }
    Else if(condition 1 )
    {
        Statement;
    }
    Else {}
```

### **Program**

```
class Ifelseif
{
    public static void main(String[] args)
    {
        System.out.println("Program Start");
        int marks=25;
        if(marks>79 && marks<=100)
        {
            System.out.println("first class distinction");
        }
        else if (marks>=60 && marks<=79)
        {
            System.out.println("first class ");
        }
    }
}
```

```

    }
    else if (marks>=50 && marks<=59)
    {
        System.out.println("second class ");
    }
    else if (marks>=35 && marks <=49)
    {
        System.out.println("third class ");
    }
    else if (marks>100)
    {
        System.out.println("Invalid number");
    }
    else
    {
        System.out.println("You are fail");
    }
}
}

```

**\*\* note :** logical operator are used to combine multiple conditions.

And operator (&&)				Or operator (  )		
C1	C2	Result		C1	C2	Result
T	T	T		T	T	T
T	F	F		T	F	T
F	T	F		F	T	T
F	F	F		F	F	F

## SWITCH CASE

- It is used whenever we have to compare the given value only equals (==) conditions.
- Switch case statements provides more readability for the program.
- It is not possible to compare the conditions other than equals conditions.
- Syntax :
 

```

Switch(choice)
{
    Case 1: Statement;
        Break;
    Case 2: Statement;
        Break;
    Default : Statement;
}

```
- Note : Break statement stop the execution of the given block at a given line of code. Writing break statement after default case is not mandatory in switch case statement.

## Program

```
class Switch
{
    public static void main(String[] args)
    {
        System.out.println("Program Start");
        char a='C';
        switch(a)
        {
            case 'A': System.out.println("alphabet A ");
            break;
            case 'C' : System.out.println("aphabet C ");
            break;
            default : System.out.println("Invalid ");
        }
    }
}
```

## LOOPING STATEMENTS

- They are used to perform repetitive task in the given program with lesser lines of code.
- Different types of loops are :-
  - (a) For loop
  - (b) While loop
  - (c) Do-while loop
  - (d) For each loop /advanced/enhanced loop

### FOR LOOP (ITERATION) :

- One complete execution cycle of a loop is called as iteration.
- The number of iteration of a loop depends on the condition of the loop.
- Syntax :

```
For(initialization; condition; counter)
{
    Statement;
}
```
- For loop is a type of loop which is used whenever the logical start and logical end is well defined.

**Basic Program :**

```

Class Basic
{
    Public static void main (String ar[])
    {
        (1)      (2)      (4)
        For(int count=1; count<=3; count++)
        {
            (3)
            System.out.println("Hello World");
        }
    }
}

```

**Op – Hello World**  
**Hello World**  
**Hello World**

Iteration 1<sup>st</sup> – 1, 2, 3, 4

Iteration 2<sup>nd</sup> – 2,3,4,

Iteration 3<sup>rd</sup> – 2,3,4

**Tracing :**

Count=1	
Count<=3, 1<=3, true print hello world	
Count=2	
Count<=3, 2<=3, true print hello world	
Count=3	
Count<=3, 3<=3, true print hello world	
Count=4	
Count<=3, 4<=3, false terminate	

**Prgm ; wap to print 1 to 5**

```

Class Pgm2
{
    Public static void main (String ar[])
    {
        For(int i=1; i<=5;i++)
        {
            System.out.println(i);
        }
    }
}

```

**Tracing**

I=1
1<=5 true print i=1

I=2 2<=5 return true print i=2
I=3 3<=5 return true print i=3
I=4 4<=5 return true print i=4
I=5 5<=5 return true print i=5
I=6 6<=5 return false for loop terminate

Pgm : wap to print the numbers between 33 to 74 in descending order.

Class A

```
{
    Public static void main (String ar[])
    {
        For (int i=74; i>=33;i--)
        {
            System.out.println(i);
        }}}
```

### Tracing

I=74 74>=33 true print i=74
I=73 73>=33 true print i=73
I=72 72>=33 return true print i=72
I=32 32>=33 return false and terminate

**Prgm : wap to display only even no between 1 to 100?**

```
For(int i=1; i<=10;i++)
{
    If(i%2==0)
    {
        S.O.P. (i);
    }
}
```

### Tracing

I=1 I<=10, 1<=10 return true then if block execute and check the condition If(1%2==0) return false so it terminate and return back to for block
I=2



I<=10, 2<=10 return true then if block execute and check the condition If(2%2==0) return true and print 2 and condition goes to again for loop block
I=3 I<=10, 3<=10 return true then if block execute and check the condition If(3%2==0) return false so it terminate and return back to for block
I=4 I<=10, 4<=10 return true then if block execute and check the condition If(4%2==0) return true and print 2 and condition goes to again for loop block -----
I=11 I<=10, 11<=10 return false so the for block is terminate and program is end.

Pgm : wap to display number between 1 to 100 which are divisible by both 2 and 5.

```

For( int i=1; i<=100;i++)
{
    If(i%2==0 && i%5==0)
    {
        System.out.println(i);
    }
}

```

### **NESTED FOR LOOPS**

- A for loop written within the body of another for loop is called as nested for loop.
- Syntax :

```

For(initialization; condition; counter)
{
    For(initialization; condition; counter)
    {
        Statement ;
    }
    Statement
}
}

```

**Pgrm :**

```

Class A
{
    Public static void main (String ar[])
    {
        For(int i=1;i<=3;i++)
        {
            System.out.println("outer loop");
            For(int j=1; j<=3;j++)
            {
                System.out.println("inner loop");} } }

```

## Tracing

I=1  
I<=3, 1<=3 return true print outer loop and goes to inner loop  
J=1  
J<=3, 1<=3 return true print inner loop  
J=2  
J<=3, 2<=3 return true print inner loop  
J=3  
J<=3, 3<=3 return true print inner loop  
J=4  
J<=3; 4<=3 return false inner loop terminate and outer loop execute again

I=2  
I<=3, 2<=3 return true print outer loop and goes to inner loop  
J=1  
J<=3, 1<=3 return true print inner loop  
J=2  
J<=3, 2<=3 return true print inner loop  
J=3  
J<=3, 3<=3 return true print inner loop  
J=4  
J<=3; 4<=3 return false inner loop terminate and outer loop execute again

I=3  
I<=3, 3<=3 return true print outer loop and goes to inner loop  
J=1  
J<=3, 1<=3 return true print inner loop  
J=2  
J<=3, 2<=3 return true print inner loop  
J=3  
J<=3, 3<=3 return true print inner loop  
J=4  
J<=3; 4<=3 return false inner loop terminate and outer loop execute again

I=4  
I<=3, 4<=3 return false and terminate outer loop

### Prgm : wap to print pattern

\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

```
For(int i=1;i<=3;i++)  
{  
    For(int j=1;j<=3;j++)  
    {  
        System.out.print(" * ");  
    }  
    System.out.println(); // moves to next line  
}
```

#### Tracing

I=1  
I<=3; 1<=3, return true , inner loop is executed  
J=1  
J<=3;1<=3 return true print \*  
J=2  
J<=3;2<=3 return true print \*  
J=3  
J<=3;3<=3 return true print \*  
J=4  
J<=3;4<=3 return false and inner loop is terminated, condition again goes to outer for loop.

I=2  
I<=3; 2<=3, return true , inner loop is executed  
J=1  
J<=3;1<=3 return true print \*  
J=2  
J<=3;2<=3 return true print \*  
J=3  
J<=3;3<=3 return true print \*  
J=4  
J<=3;4<=3 return false and inner loop is terminated, condition again goes to outer for loop.

I=3  
I<=3; 3<=3, return true , inner loop is executed  
J=1  
J<=3;1<=3 return true print \*  
J=2

J<=3;2<=3 return true print * J=3 J<=3;3<=3 return true print * J=4 J<=3;4<=3 return false and inner loop is terminated, condition again goes to outer for loop.
I=4 I<=3; 4<=3, return false, outer loop is terminated.

## WHILE LOOP

Syntax :

```
While(Boolean_condition)
{
    Statement1;
}
```

- It is a type of looping statement. It is used whenever the logical start and logical end is not well defined.
- Pgm

```
Int count=1;
While(count<=5)
{
    System.out.println(count);
    Count;
}
```

## DO-WHILE LOOP

Syntax :

```
Do
{
    Statement;
}
While(Boolean condition);
```

Prgm :

```
Int count=1;
Do
{
    System.out.println(count);
    Count++;
}
While(count<=5);
```

## CHAPTER 4 : METHODS IN JAVA

- A method is named block of codes which perform a specific task and it may or may not return a value.
- **Syntax**  
Access\_specifier access\_modifier return\_type name(arg list)  
{  
    Statement;  
    Return;  
}
- The return type of the method can be any primitive datatype or class type or void.
- The return statement of the method is used to return the control from method back to calling method
- Calling method : A method which is calling another method is known as calling method.
- A method which is called by another method is known as called method.
- Every method should be written within only the scope or body of a class.
- No methods will be executed without calling the method.
- If a method is expecting the arguments then you should call the same method by passing the required values matching the datatype.

**Program : Wap to print circumference of circle, perimeter of square, volume of a cube, total surface area of cylinder, and simple interest using method.**

class Assignment

```
{  
    public static void Circum(double rad)  
    {  
        Double cir=2*3.142*rad;  
        System.out.println("circumeferance of a circle is "+cir);  
        return;  
    }  
    public static void Square(double side)  
    {  
        double perimeter=4*side;  
        System.out.println("perimeter of a square " + perimeter);  
        return;  
    }  
    public static void Cylinder(double rad, double h)  
    {
```

```

        double area=2*3.142*rad *(rad+h);
        System.out.println("Total surface area of a cylinder is "+area);
        return;
    }
    public static void Cube(double side)
    {
        double volume=side*side*side;
        System.out.println("Volume of a cube is = " +volume);
        return;
    }
    public static void SimpleInterest(double p, double r, double t)
    {
        double si=(p*r*t)/100;
        System.out.println("Simple Interest is " + si);
        return;
    }
    public static void main(String[] args)
    {
        Circum(5);
        Square(5);
        Cylinder(5,7);
        Cube(5);
        SimpleInterest(1000, 2, 2);
    }
}

```

- If you declare the return type of the methods as **void** then the **method returns only the control** from called method to back o calling method.
- If the return type of the method is void then the written statement will be written by the compiler even if the programmer miss it.

## **METHOD WITH RETURN VALUE**

- If a method is returning a value then the return type of the method should be a primitive datatype or a class type matching the returned value.
- If a method is returning a value then you should store the returned value within a variables matching the return type of the method.
- From a method we can return only one single value.

### **Pgm : WAP TO PRINT AVG OF A NUMBER USING METHOD**

```
class Methods
{
    public static double calcArea(int a, int b, int c)
    {
        double avg;
        avg=(a+b+c)/3;
        return avg;
    }
    public static void main(String[] args)
    {
        double res=calcArea(10,20,30);
        System.out.println("res = " + res*2);
        double res1=calcArea(20,10,30);
        System.out.println("res1 = " +res1*0.3);
    }
}
```

**Op – res = 60.0**

**Res1= 9.0**

### **METHODS WITH NO ARGUMENTS:**

```
Class pgm3
{
    Public static void displayMsg()
    {
        S.O.P. ("Hello");
        S.O.P. ("good morning");
    }
    Public static void main (String ar[])
    {
        displayMsg();
        displayMsg();
    }
}
```

## CHAPTER 5 : ARRAYS (BASICS)

- An array is homogeneous group of elements which has fixed size and index.
- Using array we can manage the data easily to perform different operations.

### (a) Array Declaration

#### Syntax :

Datatype [] array\_name;

Eg : int [] marks;

Or Datatype array\_name[];

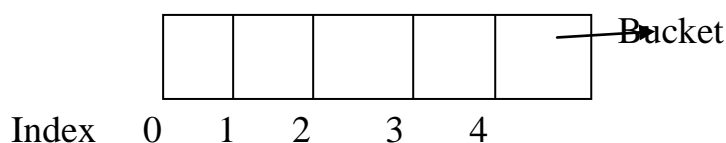
Int marks[];

### (b) Array Creation

#### Syntax :

Array\_name = new datatype [size];

Eg marks = new int[5];



### To Declare and Create Array in Single line

#### Syntax :

Datatype arrayname [] = new datatype [size];

Eg int marks[]=new int[5];

- After array creation every bucket of the array will be initialize with default values according to datatype of the array. For eg. If the datatype of an array is int then all the bucket will be filled with 0. Or if the datatype of an array is Boolean then all the bucket will be filled with false.

### (c) Initialize Array

#### Syntax :

Arrayname[index]=value;

For eg: marks[0]=20;



### **Pgm**

```
class Array
{
    public static void main(String[] args)
    {
        int marks[]=new int[5];
        marks[0]=67;
        marks[1]=20;
        marks[2]=62;
        marks[3]=85;
        marks[4]=55;
        for(int index=0; index<5; index++)
        {
            System.out.println(marks[index]);
        }
    }
}
```

**Op – 67    20    62    85    55**

### **Length (variables of array):**

- It contains count of number of bucket present in the given array.

### **Notes :**

- If you know the size of the array and the data to be stored in the array in advance then you can declare and initialize the array in same line.
- **Syntax :**

**Datatype[] arrayname = {val1, val2....};**

### **pgm**

```
class Array1
{
    public static void main(String[] args)
    {
        String [] days={"mon", "tue", "wed", "thurs", "fri",
        "sat", "sun"};
        for (int index=0;index<days.length;index++ )
        {
            System.out.println(days[index]);
        }
    }
}
```

## ArrayIndexOutOfBoundsException

- Whenever we try to add the elements to array exceeding the size of array then JVM throws ArrayIndexOutOfBoundsException.
- If you try to retrieve the elements from the array exceeding the size of the last index of the array then JVM throws ArrayIndexOutOfBoundsException.

## Pgm

**//wap of array to find first middle and last element of an array**

```
class Array2
{
    public static void printNum(int[] a1)
    {
        int first=0;
        int last=a1.length-1;
        int mid=last/2;

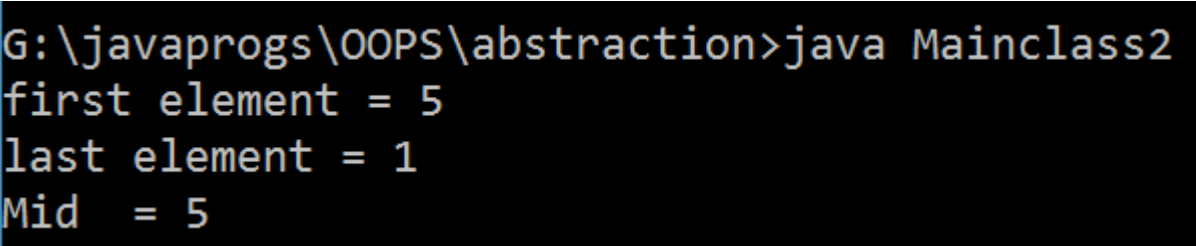
        System.out.println( "first = " +a1[first]);
        System.out.println( "last = " +a1[last]);
        System.out.println(      "mid = " +a1[mid]);
    }
    public static void main (String ar[])
    {
        int num[]={5,1,2,3,5};
        printNum(num);
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Mainclass2
first = 5
last = 5
mid = 2
```

(ii)

**//wap of array to find first middle and last element of an array**

```
class Array3
{
    public static void printNum(int[] a1)
    {
        int first=0;
        int last=a1.length-1;
        int mid=last/2;
        System.out.println( "first element = " + a1[first]);
        System.out.println( "last element = " + a1[last]);
        System.out.println("Mid = " a1[mid]);
    }
    public static void main (String ar[])
    {
        int num[]={5,1};
        printNum(num);
    }
}
```



```
G:\javaprogs\OOPS\abstraction>java Mainclass2
first element = 5
last element = 1
Mid = 5
```

**Notes :**

**First Index = 0;**

**Last Index = length -1;**

**Middle Index = last index /2;**

## CHAPTER 6 : STRING IN JAVA (BASICS)

- A string is group of characters which is written within the double quotes.
- Internally the string will be created with the help of character array.

### Methods of String

- (i) **Length()** – it returns count of number of character present in the given string.  
Eg. S1.length();
- (ii) **Equals()** – this method compares every character present in the given two strings and returns true. If they are same else it returns false.  
String s1= “jspider”;  
String s2= “jspider”;  
String s3 = “JSPIDER”;  
S1.equals(s2)- return true;  
S1.equals(s3)- return false ;
- (iii) **equalsIgnoreCase()** – this method compares every character in the given two strings by **ignoring their case**. And return true if they are same, else it returns false.  
String s1= “jspider”;  
String s2 = “JSPIDER”;  
S1.equalsIgnoreCase(s2)- return true;
- (iv) **charAt () - // charAt(index)** – This method returns the character present at the given index.  
S1.charAt(4) – return d ;
- (v) **toCharArray()** – This function or this method returns the array representation of the given string.  
Char arr[] = s1.toCharArray();

### Program

```
class StringBasic
{
    public static void main(String[] args)
    {
        String s1="jspider";
        String s2="jspider";
        String s3="JSPIDER";

        int len = s1.length();
        System.out.println("Length of String = " +len);
    }
}
```

```

        boolean res1=s1.equals(s2);
        System.out.println("Equals or not = " +res1);

        boolean res2=s1.equals(s3);
        System.out.println("Equals or not = " +res2);

        boolean res3=s1.equalsIgnoreCase(s3);
        System.out.println("Equals or not = " +res1);

        char c1=s1.charAt(4);
        System.out.println("Character at index = " +c1);

        char [] ar1=s1.toCharArray();
        for(int index=0; index<ar1.length;index++)
        {
            System.out.print(ar1[index]);
        }
    }}

```

```

G:\javaprogs\OOPS\abstraction>java Mainclass2
Length of String = 7
Equals or not = true
Equals or not = false
Equals or not = true
Character at index = d
jspider

```

# OBJECT ORIENTED PROGRAMMING SYSTEM

## CHAPTER 1 : CLASS & OBJECTS

**OBJECTS :** Any entity which has states and behaviour is called as objects. For eg. Pen, account, building.

Pen		Account		Student	
State	Behaviour	State	Behaviour	State	Behaviour
Color	Writing	a/c no.	Debit	Name	Studying
Brand	Drawing	Type of ac	Credit	Id	Exam
Shape	Pointing	Bank name	Payment	Gender	Reading
Price	.....	Balance .... etc	.....	Age	writing

### CLASS : -

- A class is blueprint of an objects.
- A class contains or defines states and behaviours of an objects.
- The states of the class are called as data members and the behaviours of the objects are called as function members.
- The data members of the class are represented by variables and the functions members of the class are represented by methods.

### JAVA NAMING CONVENTION :

#### Class :

- Any entity in java which starts with upper case declared with the keyword class is called as java class.
- Class names should be nouns, in mixed case with first letter of each internal word capitalize.  
Eg. Account, AccountName;

#### Interface :

- Any entity in java which starts with upper case and declared with the keyword interface is called java interface.
- Interface name should be capitalize like class name.
- Eg. : interface Storing, interface RunnableInterface

#### Methods :

- It defines the action which is performed on data members of the class.
- Methods should be in mixed case with the first letter lower case, with the first letter of each internal word capitalize.
- Eg : run(), getData()

**Variables :**

- Variables should be declared in mixed case with a lowercase first letter. Internal words starts with capital letters.
- Eg: i, c, myWidth;

**Constants :**

- The constants should be all uppercase with words separated by underscore (“\_”).
- Eg. MIN\_WIDTH = 5;

The members of class be classified into two types :

(a) Static Members      (b) Non-Static members

**Static members**

- Any members of the class which is declared using static keyword is called as static members.

**Static Members present in Same Class:**

- If a static method is trying to access the static members present in the same class then it can refer to them directly with the name of static member.

**Program**

```
class Demo
{
    static int v1=100;
    public static void test()
    {
        System.out.println("this is test() of demo class");
    }
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + v1);
        test();
    }
}
```

```
G:\javaprogs\OOPS\Static>javac Demo.java
```

```
G:\javaprogs\OOPS\Static>java Demo
```

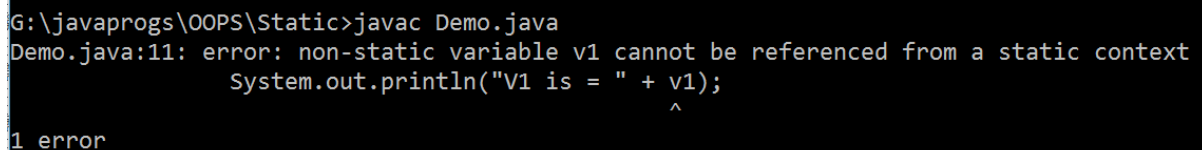
```
V1 is = 100
```

```
this is test() of demo class
```

## Program 2

class Demo

```
{
    int v1=100;
    public static void test()
    {
        System.out.println("this is test() of demo class");
    }
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + v1);
        test();
    }}
}
```



```
G:\javaprogs\OOPS\Static>javac Demo.java
Demo.java:11: error: non-static variable v1 cannot be referenced from a static context
        System.out.println("V1 is = " + v1);
                                   ^
1 error
```

**Note : - A non static variable can not be referenced by a static context.**

Note : \*\*

- Within one java program we can write any number of class.
- If a program contains multiple classes then the class which contains main method should be used as filename.

### Static Members Present in Different Class :

- We can access static members of different class using the classname with dot(.) operator followed by member name.
- Syntax :

```
className.memberName;
className.memberFunction();
```

Note : we can not use static member in different class directly. If we use it throw error.

## Program

**// static member used by static method use in different class directly.**

class Demo

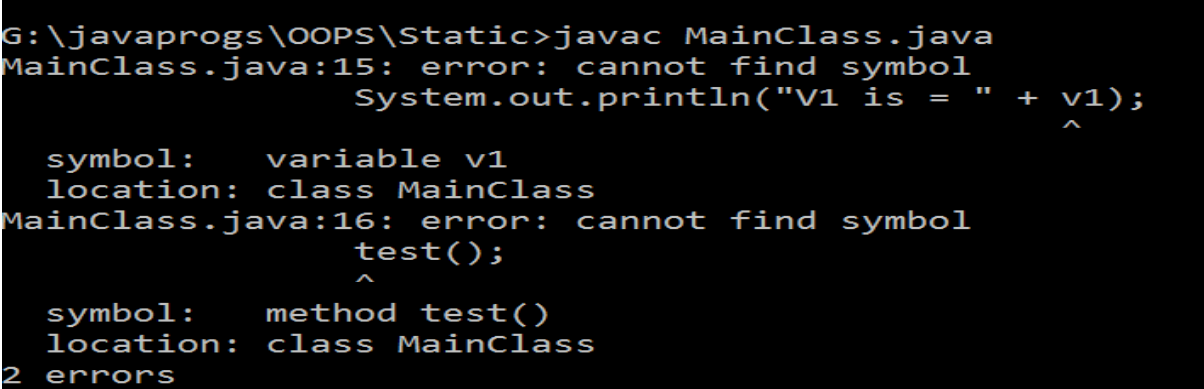
```
{
    static int v1=100;
    public static void test()
```



```

        {
            System.out.println("this is test() of demo class");
        }
    }
}
class MainClass
{
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + v1);
        test();
    }
}

```



The screenshot shows a terminal window with the following text:

```

G:\javaprogs\OOPS\Static>javac MainClass.java
MainClass.java:15: error: cannot find symbol
        System.out.println("V1 is = " + v1);
                                   ^
    symbol:   variable v1
    location: class MainClass
MainClass.java:16: error: cannot find symbol
        test();
        ^
    symbol:   method test()
    location: class MainClass
2 errors

```

### Program :

**// static member used by static method use in different class by classname.member**

```

class Demo
{
    static int v1=100;
    public static void test()
    {
        System.out.println("this is test() of demo class");
    }
}
class MainClass2
{
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + Demo.v1);
        Demo.test();
    }
}

```

## NON-STATIC MEMBERS :

- Any member of the class which is declared without using static keyword is called as non-static members.
- We can access non-static members of a class only by creating the object for the class.

### Object creation

#### Syntax :

**New className()**

Eg.

New Sample()

Here New is a keyword which creates a new object and Sample() is a **constructor call** which copy all the non-static member to object.

- A non static method can access non-static data members or non static function members present in the same class without creating any object.

### Program : calling non-static member and method from main method in a single class by not creating any object

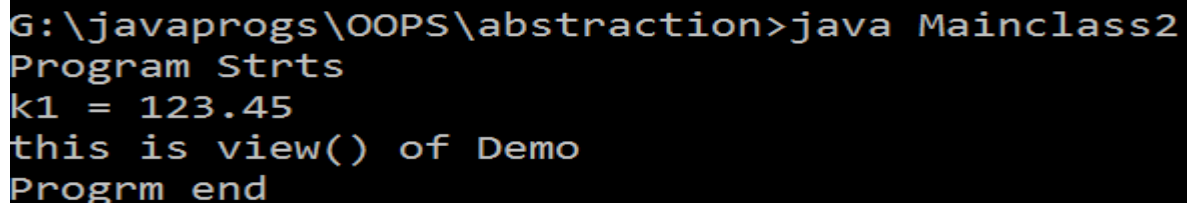
```
class Abc
{
    int z1=123;
    public void view()
    {
        System.out.println("this is view() of Abc");
        System.out.println("value of a = " + z1);
    }
    public static void main(String[] args)
    {
        System.out.println("Z1 = " + new Abc().z1);
        new Abc().view();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Mainclass2
Z1 = 123
this is view() of Abc
value of a = 123
```

**Program :**

**// use of non static members and function in other class**

```
class Mainclass
{
    double k1=123.45;
    public void count()
    {
        System.out.println("this is view() of Demo");
    }
}
class Mainclass2
{
    public static void main(String[] args)
    {
        System.out.println("Program Strts ");
        System.out.println("k1 = " + new Mainclass().k1);
        new Mainclass().count();
        System.out.println("Progrm end " );
    }
}
```



A screenshot of a terminal window showing the execution of a Java program. The command 'java Mainclass2' is entered, and the output is displayed line by line: 'Program Strts', 'k1 = 123.45', 'this is view() of Demo', and 'Progrm end'.

**Program // static member used by static method use in same class**

```
class Demo
{
    static int v1=100;
    public static void test()
    {
        System.out.println("this is test() of demo class");
    }
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + v1);
        test();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Mainclass2
V1 is = 100
this is test() of demo class
```

**Program // static member used by static method use in different class.**

```
class Demo
{
    static int v1=100;
    public static void test()
    {
        System.out.println("this is test() of demo class");
    }
}
class MainClass
{
    public static void main (String ar[])
    {
        System.out.println("V1 is = " + Demo.v1);
        Demo.test();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Mainclass2
V1 is = 100
this is test() of demo class
```

## CHAPTER 2 : REFERENCE VARIABLES

Demo d1 = new Demo ();

- It is a type of variables which is used to store the address of the objects.

```
class RefVariable
```

```
{
```

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

```
    {
```

```
        RefVariable r = new RefVariable();
```

```
        System.out.println( " Address of Variable r = " +r);
```

```
    }}
```

**// output Address of variable r = RefVariable@6073f712**

- Within a reference variables we can not store any primitive data values.

```
class RefVariable
```

```
{
```

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

```
    {
```

```
        int r = new RefVariable(); // it throw error here
```

```
        System.out.println( " Address of Variable r = " +r);
```

```
    }
```

```
}
```

- Multiple reference variable can point to same objects.

```
Demo r1 = new Demo();
```

```
Demo r2 = r1;
```

```
class RefVariable
```

```
{
```

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

```
    {
```

```
        RefVariable r = new RefVariable();
```

```
        RefVariable r2=r;
```

```
        System.out.println( " Address of Variable r = " +r);
```

```
        System.out.println( " Address of Variable r1 = " +r2);
```

```
    }
```

```
}
```

**// output Address of variable r = RefVariable@6073f712**

**// output Address of variable r2 = RefVariable@6073f712**

- If multiple reference variables are pointing to same object then changes done through one reference variables will impact other reference variables.

```
class RefVariable
{
    int r = 20;
    public static void main(String[] args)
    {
        RefVariable r1 = new RefVariable();

        System.out.println( " value of r1 = " +r1.r);
        r1.r=30; //reinitialize the value of r by using the object

        System.out.println( " Address of Variable r = " +r1.r);

        RefVariable r2 = r1;
        System.out.println("Value of r2 = " +r2.r);
    }
}
```

- Static members are called as class members because they can be accessed using the classname.
- Non static members are called as instance members because they can be accessed only by creating the object or instance.
- Static members of the class will have only one copy in the memory.

```
Static int v1 = 50;
/S.O.P (Sample.v1);
Sample.V1=50;
```

- Non-static members will have multiple copies in the memory depending on number of objects created.

```
Int v1= 100;
S.O.P ( new Simple().V1);
S.O.P (new Simple().V2);
```

## Program

```
class Account
{
    int actno=12345;
    String name = "Dinga";
```

```

String branch = "basvangudi";
double balance = 5000;
String type = "Savings";

static String bankName = "ICICI";

    public void deposit(int amt)
    {
        balance = balance+amt;
    }
public void withdraw (int amt)
{
    balance = balance-amt;
}
public void checkBalance()
{
    System.out.println(" Avaialable balance "+balance);
}
public void showAccount()
{
    System.out.println("act no = "+actno);
    System.out.println("name = "+name);
    System.out.println("brnch = "+branch);
    System.out.println("balance = "+balance);
    System.out.println("acount type = "+type);
    System.out.println("acount type = "+bankName);
}
}
class MainAccount
{
    public static void main (String ar[])
    {
        Account a1=new Account();
        a1.checkBalance();
        System.out.println();
        a1.deposit(5000);
        a1.checkBalance();
    }
}

```

```

        System.out.println();
        a1.withdraw (2000);
        System.out.println();
        a1.checkBalance();

        a1.showAccount();
    }
}

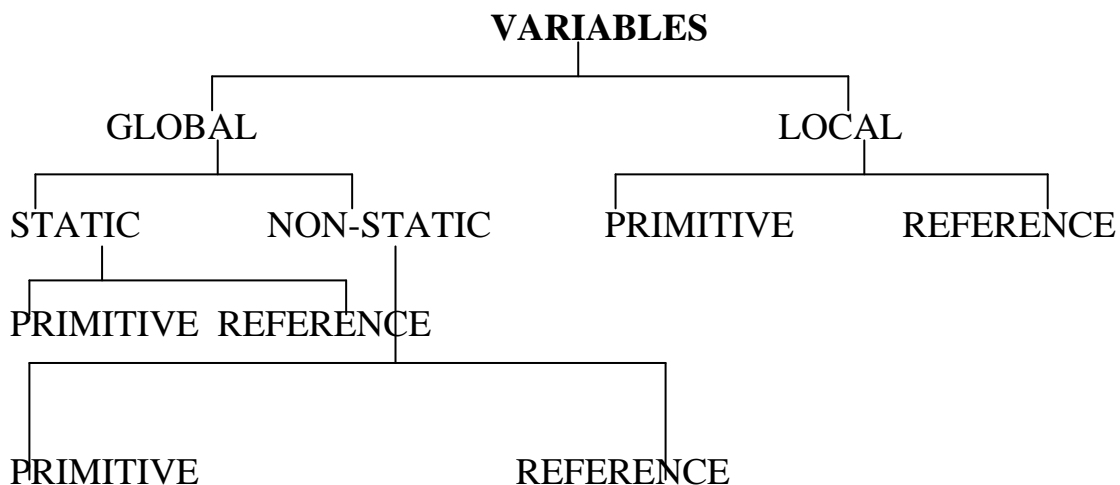
```

```

G:\javaprogs\extra>java StaticKeyword
Remaining Balance = 5000.0
Remaining Balance = 5500.0
Remaining Balance = 4500.0
Name is = Dinga
actno is = 123456
account type is = Saving
total balance = 4500.0
Bank name = ICICI

```

## GLOBAL AND LOCAL VARIABLES



## GLOBAL VARIABLES

- A variables which is declared within the scope of the class is called as global variables.



- Global variables can be accessed by all the methods present in the present in the same class.

## **LOCAL VARIABLES**

- Any variables which is declare within the method declaration or method definition are called as local variables.
- Local variables can be accessed only within the methods in which they are declared.
- Local variables can not be declared as static or non static.

### **Program : Declaration of local and global variables**

```
class Abc
{
    int z1=123; // global variables
    public static void view()
    {
        int y = 20; // local variables
        System.out.println("Value of y = " +y)
    }
    public static void main(String[] args)
    {
        System.out.println("Z1 = " + new Abc().z1);
        new Abc().view();
    }
}
```

### **Program : local variable can not used outside the method.**

```
class Abc
{
    int z1=123;
    public void view()
    {
        int a = 20;
        System.out.println("this is view() of Abc");
        System.out.println("value of a = " + z1);
        System.out.println("value of a = " + a);
    }

    public static void main(String[] args)
    {
        System.out.println("Z1 = " + new Abc().z1);
    }
}
```

```

        new Abc().view();
        System.out.println("value of a = " + a); // throws an error
    }
}

```

- If variables and local variables have same names then the compiler always give the preference to local variables.

```

class Abc
{
    static int z1=123;

    public static void main(String[] args)
    {
        int z1=400;
        System.out.println("Z1 = " + z1);
    }
}

```

**// output z1 = 400**

- If we want to use both variables together then use classname.variablesname

```

class Abc
{
    static int z1=123;

    public static void main(String[] args)
    {
        int z1=400;
        System.out.println("Z1 = " + z1);
        System.out.println("Z1 = " + Abc.z1);
    }
}

```

**// output z1 = 400**

**// output z1 = 123;**

### **Important notes :**

- Using the object of class we can access both static and non static members of the class.

- It is strictly not recommended to access static members of the class using objects.
- Global variables (both static and non-static) will be initialized by the compiler with the default values depending on the datatype.
- Local variables should be initialized by the programmer explicitly.

### Program

```
class Global2
{
    static double x1 ;

    public static void main(String[] args) {
        int z1;
        System.out.println("Value of x1 = "+x1);
        System.out.println("Value of z1 = " + z1);
        /* here z1 is not initialized so it show an error.
        but for x1 it does not show any error.
        it take default value of x1 according to theri data type.*/
    }
}
```

### DECLARING CONSTANTS :

- Final keyword is used to declare constants in java.
- If you declare any variables with final keyword then it can not be re-initialized.
- If you declare any class with final keyword then the class can not be inherited.
- If you declare any method with final keyword then it can not be overridden.

### Program

```
class Constants
{
    final double PI=3.142;

    public void area1 (int a)
    {
        double ar= PI*a*a;
        System.out.println("Area of 1st Circle" + ar);
    }
}
```

```

    public void area2 (int a)
    {
        double ar= PI*a*a;
        System.out.println("Area of 1st Circle" + ar);
    }
    public static void main(String[] args)
    {
        Constants c=new Constants();
        c.area1(5);
        c.area2(10);
    }
}

```

- Note : we can not re-initialize any constant value. If we do then it throw an error

Program

class Constants

```

{
    final double PI=3.142;

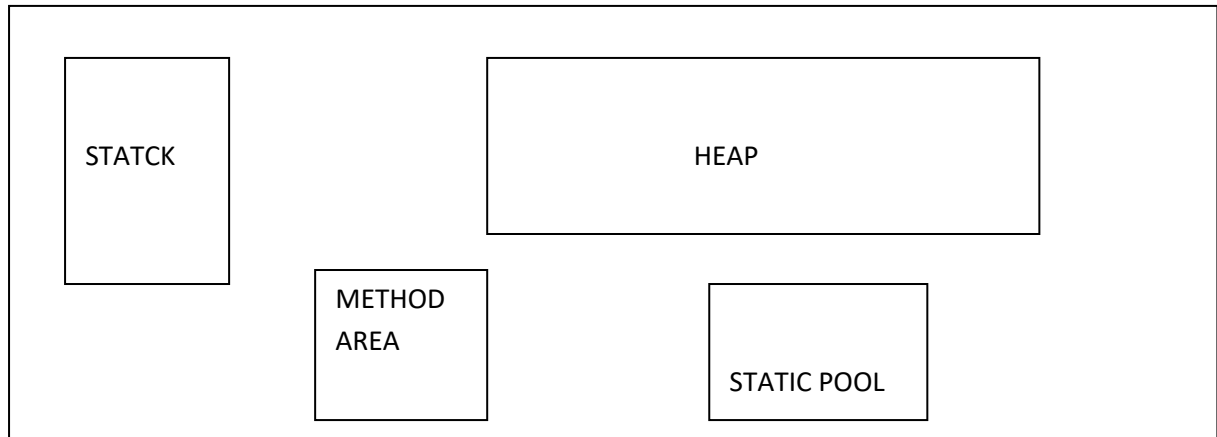
    public void area1 (int a)
    {
        PI=4525; here it throw error as cannot assign a value to
final variables.
        double ar= PI*a*a;
        System.out.println("Area of 1st Circle" + ar);
    }
    public static void main(String[] args)
    {

        Constants c=new Constants();
        c.area1(5);

    }
}

```

## JVM ARCHITECTURE



### Stack

- Any method which is in execution state will be present in stack.

### Method Area

- It contains method definition of both static and non-static methods.

### Static Pool

- It contains static members in the respective static pools of the respective classes.

### Heap

- Every object which is created using new operator will be present in heap area.

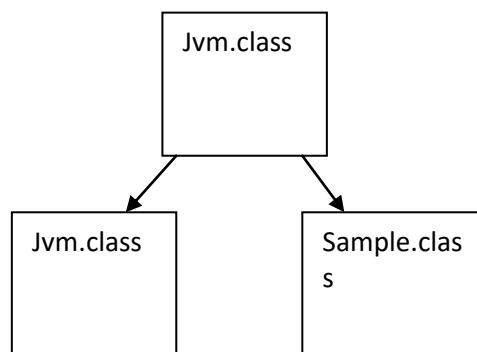
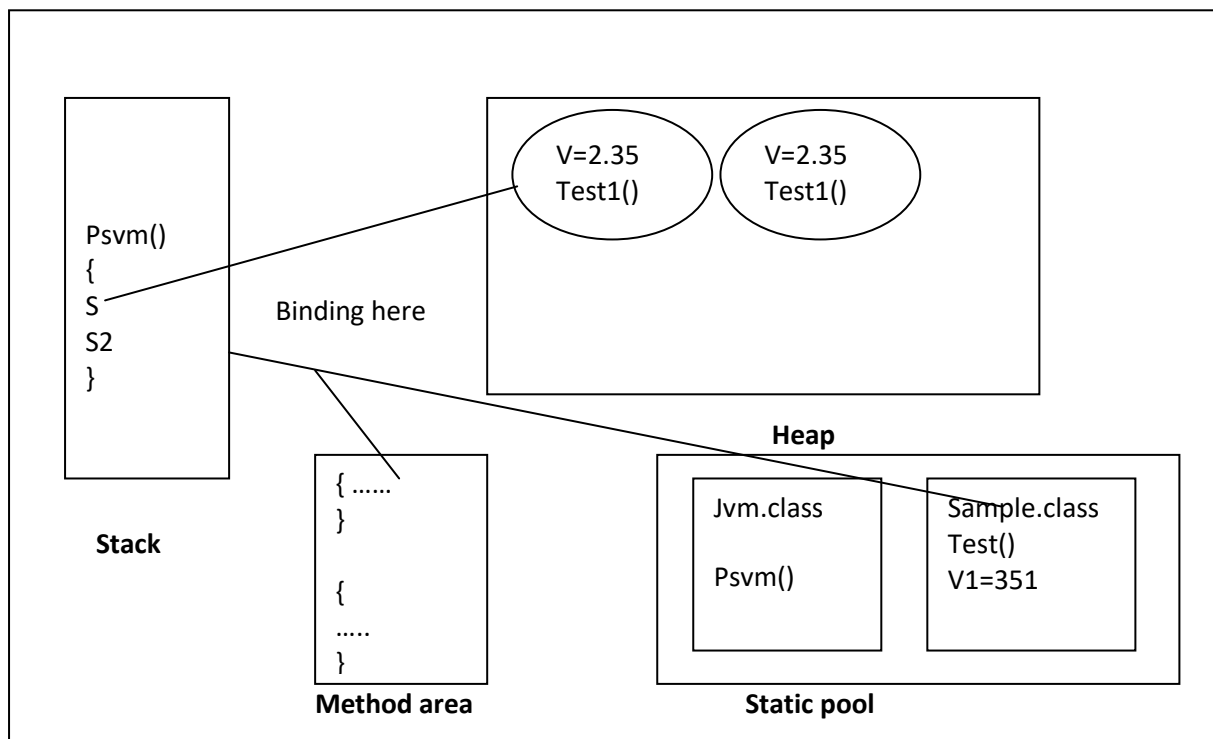
### Program

```
class Sample
{
    static int v1 = 351;
    double v2 = 2.5;
    public static void test()
    {
        System.out.println("This is test()");
        return;
    }
    public void test1()
    {
        System.out.println("this is test1()");
    }
}
```

```

class Jvm
{
    public static void main(String[] args)
    {
        System.out.println("Value of V1 = " + Sample.v1);
        Sample.test();
        Sample s = new Sample();
        System.out.println("Value of V2 " + s.v2);
        s.test1();
    }
}

```



Javac Jvm. Java

Java Jvm

## Important Notes :

- JVM starts the execution of the program by **calling classloader** to load the .class file to the memory.
- The class loader copies all the **static members** of the given class **to static pool** into the respective static pools of classes.
- Once class loader loads the class file JVM calls method for the execution.
- If the given class do not contain main method or main method not defined as **public static void main (String ar[])** then JVM will throw a runtime errors.
- JVM binds method declaration to method definition and loads it into the stack and executes the methods.
- If the main method is referring to static members of different class or if we are creating an object of a different class then the corresponding .class file of the given class will be loaded into the memory.
- Once the method completes the execution will be removed out of stack.
- Once the program completes the execution **JVM calls the classloader** to unload the .class file from the memory.
- The class loader removes all static members present in static pool.
- JVM also calls the **garbage collector** to remove all the objects present in the heap memory.

## CHAPTER 3 : CONSTRUCTOR IN JAVA

- Constructor is a special type of method which has same name as the classname.
- Every java class must and should have a constructor.
- If the programmer do not write a constructor then compiler writes default constructor implicitly.
- Constructor can not be declared as static.
- Constructor can not be declared as final.
- If you write return type for a constructor then it will be considered as a normal method.
- We can write return statement within the constructor body.
- The constructor can be executed only by creating an objects.
- Constructors can not be called explicitly using the object.
- Constructor are of two types : (a) zero or default constructor (b) parameterized constructor.

- If a programmer writes the constructor explicitly then the compiler do not write default constructor implicitly.

### **Program for simple constructor**

```
class Sample
{
    public Sample()
    {
        System.out.println("This is sample const...");
    }
}
class SimpleConstructor
{
    public static void main(String[] args)
    {
        Sample s = new Sample();
    }
}
```

**-op- This is sample const...**

**//if we dont initialize the member then it print the default value.**

```
class Sample
{
    String name;
    int id;
    double sal;
    public Sample()
    {

    }
    public void display()
    {
        System.out.println(" name is = " + name);;
        System.out.println(" id is = " + id);;
        System.out.println(" salary is = " + sal);;
    }
}
class DefaultValue
{
}
```



```

    public static void main(String[] args)
    {
        Sample s = new Sample();
        s.display();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java Mainclass2
name is = null
id is = 0
salary is = 0.0

```

Constructor are used to initialize the data members (static and non static) of the class.

### **Parameterized Constructor**

```

class Sample
{
    String name;
    int id;
    double sal;

    public Sample(String a, int b, double c)
    {
        // System.out.println("This is sample const...");
        name=a;
        id=b;
        sal=c;
        return;
    }
    public void display()
    {
        System.out.println(" name is = " + name);;
        System.out.println(" id is = " + id);;
        System.out.println(" salary is = " + sal);;
    }
}

```

```

class ParameterizedConstructor
{
    public static void main(String[] args)
    {
        Sample s = new Sample("Ajay", 101, 2000);
        s.display();
        Sample s2 = new Sample("Rohit", 102, 2500);
        s2.display();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java Mainclass2
name is = Ajay
id is = 101
salary is = 2000.0
name is = Rohit
id is = 102
salary is = 2500.0

```

## CHAPTER 4 : THIS KEYWORD AND THIS STATEMENT()

### This keyword

- It is used to differentiate between local and global variables whenever they have same names.

```

class Sample
{
    String name;
    int id;
    double sal;

    public Sample(String name, int id, double sal)
    {
        // System.out.println("This is sample const...");
        this.name=name;
        this.id=id;
    }
}

```

```

        this.sal=sal;
        return;
    }
    public void display()
    {
        System.out.println(" name is = " + name);;
        System.out.println(" id is = " + id);;
        System.out.println(" salary is = " + sal);;
    }
}
class ThisOperator
{
    public static void main(String[] args)
    {
        Sample s = new Sample("Ajay", 101, 2000);
        s.display();
        Sample s2 = new Sample("Rohit", 102, 2500);
        s2.display();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java Mainclass2
name is = Ajay
id is = 101
salary is = 2000.0
name is = Rohit
id is = 102
salary is = 2500.0

```

- This keyword is a special type of reference variables which always points to active or current instance of the class.
- This keyword can not be used within the static method.
- This keyword can be used only within the non-static methods and constructors of the class.

**// we can not like do this, it print the default values of global variables.**

```
class Demoa
{
    String name;
    int id;
    double sal;

    public Demoa(String name, int id, double sal)
    {
        name=name;
        id=id;
        sal=sal;
    }
    public void disp()
    {
        System.out.println("name is = "+this.name);
        System.out.println("id is = "+this.id);
        System.out.println("sal is = "+this.sal);
    }
    public static void main(String[] args)
    {
        Demoa a = new Demoa("ajay", 101, 1020.05);
        a.disp();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Mainclass2
name is = null
id is = 0
sal is = 0.0
```

## Program

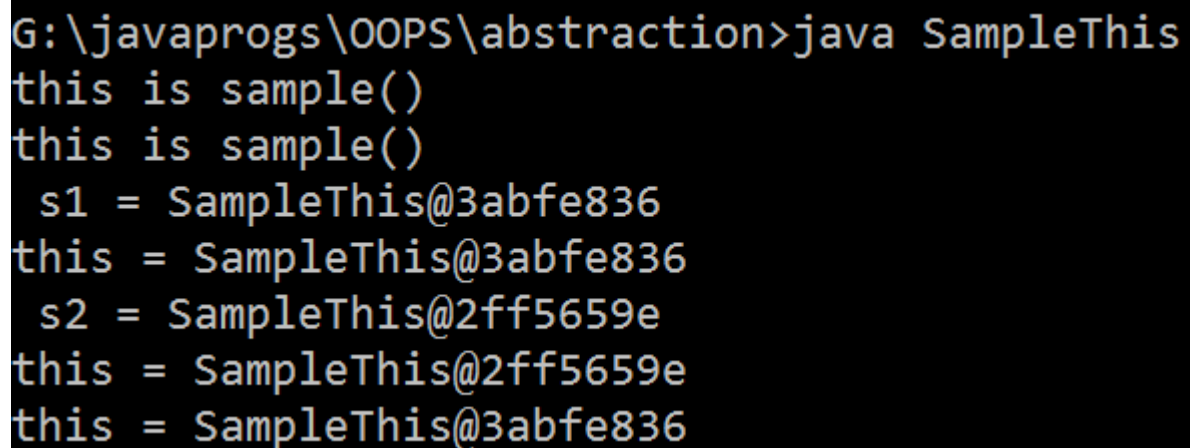
**//use of this keyword**

```
class SampleThis
{
    int x1;
    double y1;

    public SampleThis()
    {
        System.out.println("this is sample()");
    }
    public void test()
    {
        System.out.println("this = "+this);
    }

    public static void main(String[] args) {
        SampleThis s1=new SampleThis();
        SampleThis s2=new SampleThis();
        System.out.println(" s1 = "+s1);
        s1.test();
        System.out.println(" s2 = "+s2);
        s2.test();

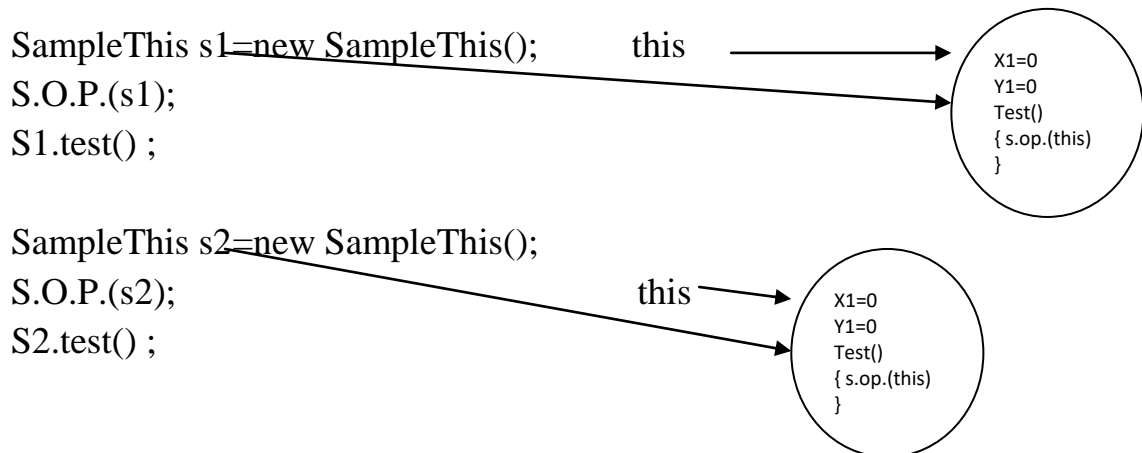
        s1.test();
    }
}
```

A screenshot of a terminal window showing the execution of a Java program. The command 'java SampleThis' is entered at the prompt. The output consists of several lines: 'this is sample()' appears twice, followed by 's1 = SampleThis@3abfe836', 'this = SampleThis@3abfe836', 's2 = SampleThis@2ff5659e', 'this = SampleThis@2ff5659e', and 'this = SampleThis@3abfe836'. The text is displayed in a monospaced font with a light blue color on a black background.

```
G:\javaprogs\OOPS\abstraction>java SampleThis
this is sample()
this is sample()
s1 = SampleThis@3abfe836
this = SampleThis@3abfe836
s2 = SampleThis@2ff5659e
this = SampleThis@2ff5659e
this = SampleThis@3abfe836
```

- This keyword differentiate between global and local variables.

For eg :



In the above example firstly this keyword references to s1 object because at that time it is active. But when s2 is active this keyword reference to s2 object.

### Constructor Overloading

- Developing multiple constructor within the same class which differ in
  - No. of arguments
  - Datatypes of arguments
  - Sequence of arguments.

Is called as constructor overloading.

### Program

```
class ConstructorOverloading
{
    public ConstructorOverloading()
    {
        System.out.println("this is zero argument constructor");
    }
    public ConstructorOverloading(int a)
    {
        System.out.println("this is int a constructor");
    }
    public ConstructorOverloading(double a)
    {
        System.out.println("this is double a constructor");
    }
}
```

```

public ConstructorOverloading(int a, double b)
{
    System.out.println("this is int a, double b");
}
public ConstructorOverloading(double b, int a)
{
    System.out.println("this is double b, int a ");
}
public static void main(String[] args)
{
    ConstructorOverloading c1=new ConstructorOverloading();
    ConstructorOverloading c2=new ConstructorOverloading(5);
    ConstructorOverloading c3=new ConstructorOverloading(10.2);
    ConstructorOverloading c4=new ConstructorOverloading(5,10.2);
    ConstructorOverloading c5=new ConstructorOverloading(2.5, 7);
}
}

```

```

G:\javaprogs\OOPS\abstraction>java ConstructorOverloading
this is zero argument constructor
this is int a constructor
this is double a constructor
this is int a, double b
this is double b, int a

```

- Constructor overloading is helpful in providing flexibility for the users to create the objects.
- Constructor overloading is the best example for compile time polymorphism.

#### **This() statement**

- It is used to call one constructor from another constructor which are present in same class.

```

class Sample
{
    public Sample()
    {
        this(10); // call the argumented constructor
    }
}

```

```

        System.out.println("this is zero -a argument const");
    }
    public Sample(int a)
    {
        System.out.println("This is int a const...");
    }
}
class ThisStatement
{
    public static void main(String[] args) {
        Sample s=new Sample();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java ThisStatement
This is int a const...
this is zero -a argument const

```

- This() statement should be written as first statement within the constructor body.

```

public Sample()
{
    this(10); // call the argumented constructor
    System.out.println("this is zero -a argument const");
    This(10); if we write like this then it throw error.
}
public Sample(int a)
{
    System.out.println("This is int a const...");
}

```

- We can not write multiple this() statement within the constructor body.

```

public Sample()
{
    this(10); // call the argumented constructor
    this(20); // if we write like this it throws an error
    System.out.println("this is zero -a argument const");
}
public Sample(int a)

```



```

    {
        System.out.println("This is int a const...");
    }

```

- The called constructor can not call back calling constructor using this() statement because it leads to recursive constructor invocation error.

```

public Sample()
{
    this(10); // call the argumented constructor
    System.out.println("this is zero -a argument const");
}
public Sample(int a)
{
    This(); // if we call like this then it throw error.
    System.out.println("This is int a const...");
}

```

- The constructor can not call itself with this statement, because it leads to recursive constructor invocation.

```

public Sample()
{
    this(); // calling itself which throws an recursive
constructor invocation
    System.out.println("this is zero -a argument const");
}
public Sample(int a)
{
    System.out.println("This is int a const...");
}

```

### **Simple this() statement program**

```

class Sample
{
    public Sample()
    {
        this(10);
        System.out.println("this is zero argument const");
    }
    public Sample(int a)

```

```

        {
            System.out.println("This is int a const...");
        }
    }
    class ThisStatement
    {
        public static void main(String[] args) {
            Sample s=new Sample();
        }
    }

```

```

G:\javaprogs\OOPS\abstraction>java ThisStatement
This is int a const...
this is zero -a argument const

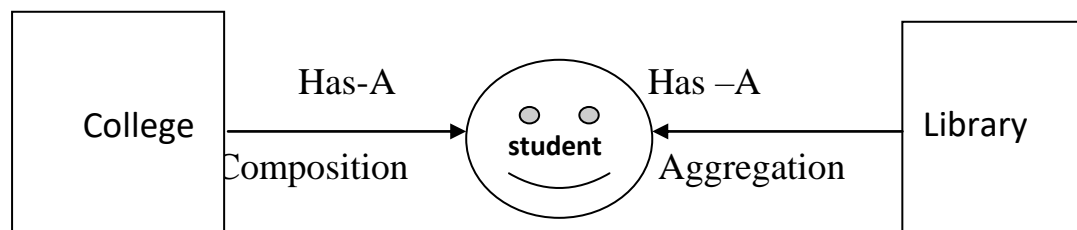
```

## RELATION BETWEEN CLASSES IN JAVA

- The relations between the classes helps in achieving two goals.
  - (i) Reduce no. of lines of code.
  - (ii) Code reusability
- There are two types of relations.
  - (i) Has-A relation
  - (ii) Is-A relation

### Has-A Relation:

- It is a type of relation which is created based on the dependency of one object on the another object.
- It is of two types : (i) Aggreation (ii) Composition



### Aggregation :

- It is a type of Has-A relation where existence of one object is not dependent on another object.
- The aggregation is achieved by creating a static reference variable pointing to the object of another class.

## Program

```
class Sample
{
    String name;
    int id;
    int nbo;
    String branch;
    public Sample(String name, int id, int nbo, String branch)
    {
        this.name=name;
        this.id=id;
        this.nbo=nbo;
        this.branch=branch;
    }
    public void showDetails()
    {
        System.out.println("Name = "+name);
        System.out.println(" id = " + id);
        System.out.println ( " no of book issued " +nbo);
        System.out.println(" branch = "+branch);
    }
}

class Library
{
    /*creating a static reference variable of class Sample where s1 points  
to object of Sample()*/
    static Sample s1 = new Sample("aatif", 101, 2, "cse");
    public static void issueBook()
    {
        System.out.println("book is issued to "+s1.name);
        s1.nbo++;
    }
}

class Program1
{
    public static void main(String[] args)
    {
//calling the methods of sample class by using the object
    }
}
```

```

        Library.s1.showDetails();
        Library.issueBook();
    }}

```

```

G:\javaprogs\OOPS\abstraction>java Program1
Name = aatif
id = 101
no of book issued 2
branch = cse
book is issued to aatif

```

Note : System.out.println() is the best example of aggregation has-A relation which helps in hide the class. Here println() method is the method of printStream class.

### Composition

- It is a type of Has-A relation where existence of one object is dependent on another object.
- The aggregation is achieved by creating a non-static reference variable pointing to the object of another class.

### Program

```

class Attachment
{
    String name;
    double size;
    String type;
    public Attachment(String name, double size, String type)
    {
        this.name=name;
        this.size=size;
        this.type=type;
    }
    public void showDetails()
    {
        System.out.println(" Attachment name = "+name);
        System.out.println("File size = "+size +"kb");
        System.out.println("File type = "+type);
    }
}

```

```

    }
    class Email
    {
        String sender;
        String subject;
        String msg;
        Attachment a1=new Attachment("file1", 2.5, "txt" );
        public Email(String sender, String subject, String msg)
        {
            this.sender=sender;
            this.subject=subject;
            this.msg=msg;
        }
        public void openMail()
        {
            System.out.println(" sender name = "+sender);
            System.out.println("subject = "+subject);
            System.out.println("msg = "+msg);
        } }
    class Composition
    {
        public static void main(String[] args)
        {
            Email e1=new Email("Abc@gmail.com", "java
language", "hello");
            e1.openMail();
            e1.a1.showDetails();
        }}

```

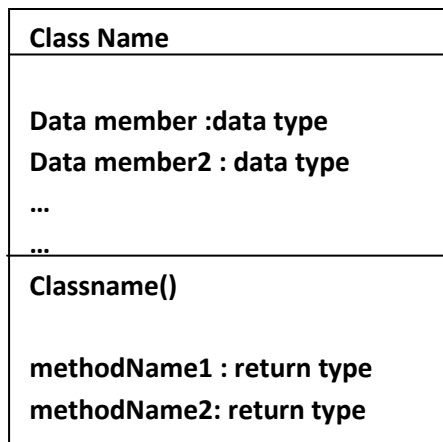
```

G:\javaprogs\OOPS\abstraction>java Composition
sender name = Abc@gmail.com
subject = java language
msg = hello
Attachment name = file1
File size = 2.5kb
File type = txt

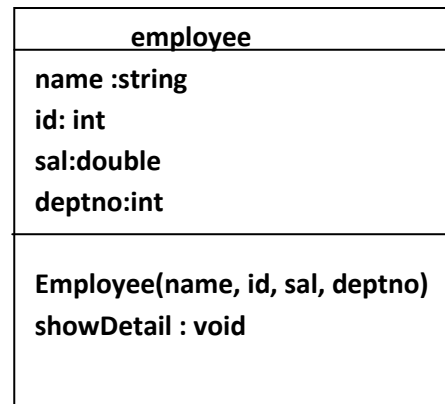
```

## Class Diagram

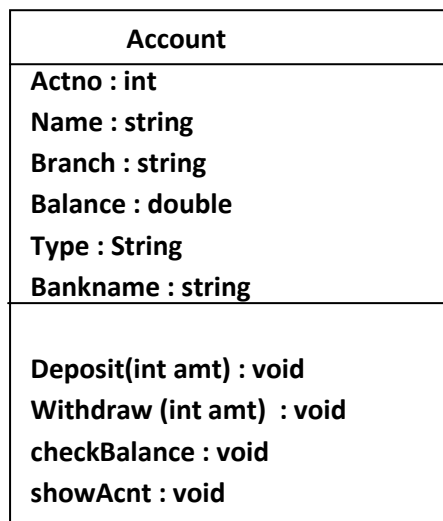
- It is a pictorial representation of a class.
- Class diagram helps in low level design (LLD) of the application.



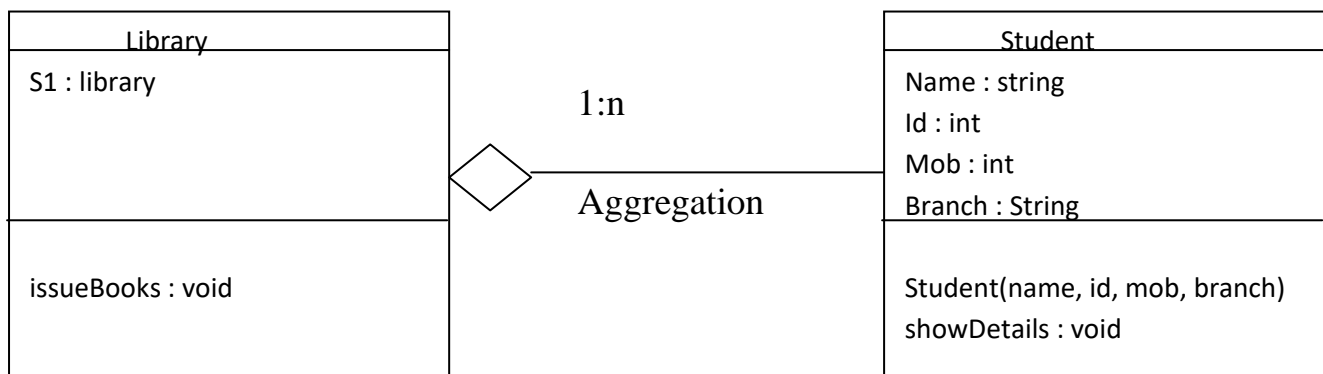
Eg:

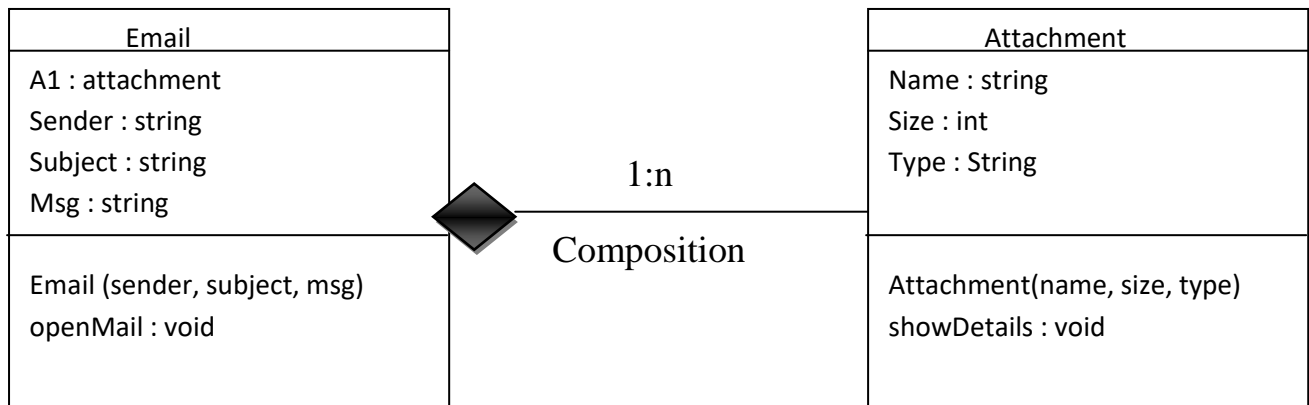


Eg of account class



## Representation of Has-A relation using class Diagram





## CHAPTER 5 : INHERITANCE IN JAVA (IS-A RELATION)

- One class acquiring the properties of another class is called as inheritance.
- The class from where the properties are inherited is called as super class, base class or parent class.
- The class which is inheriting properties from another class is called as subclass, derived class or child class.
- Inheritance between the classes is achieved with the help of **extends** keywords.
- Using the child class objects we can access properties of both child class and parent class.
- We can access both static and non-static members of the parent class from child class object.

### Simple Program

```

class Parent
{
    int x1=50;
    public void display()
    {
        System.out.println("this is display() of parent class");
    }
}
class Child extends Parent
{
    int y1=100;
  
```

```

        public void test()
        {
            System.out.println("this is test() method of child
class");
        }
    }
class SimpleInheritance
{
    public static void main(String[] args)
    {
        Child c1=new Child();
        c1.display();
        System.out.println("Value of x1 = "+c1.x1);
        c1.test();
        System.out.println("Value of y1 = "+c1.y1);
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java SimpleInheritance
this is display() of parent class
Value of x1 = 50
this is test() method of child class
Value of y1 = 100

```

- Final class can not be inherited.

```

final class Super1 // it can not be inherited
{
    int a =50;
}
class Child extends Super1
{
    int d=10;
    public void test()
    {
        System.out.println("this is child class " +d);
    }
}
class FinalClass{
    public static void main(String[] args) {
        Child c = new Child();
    }
}

```



```

        c.test();
    }}

```

```

G:\javaprogs\OOPS\abstraction>javac SimpleInheritance.java
SimpleInheritance.java:5: error: cannot inherit from final Super1
class Child extends Super1
        ^
1 error

```

- Private members, data members and functions members can not be inherited.

```

class Super1
{
    private int a=50;
}
class Child extends Super1
{
    int d=10;
    public void test()
    {
        System.out.println("this is child class " +d);
        System.out.println("this is super class" + a);
        // here it return error as a has private access in super
    }
}
class FinalClass{
    public static void main(String[] args) {
        Child c = new Child();
        c.test();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>javac SimpleInheritance.java
SimpleInheritance.java:11: error: a has private access in Super1
        System.out.println("this is super class" + a);
                                   ^
1 error

```

- Constructor of super class can not be inherited to subclass.

- Subclass can inherit final data members but it can not re-initialize them.

```
class Super1
{
    final int A =50;

class Child extends Super1
{
    int d=10;
    public void test()
    {
        System.out.println("this is child class " +d);
        System.out.println("this is super class " + A);
    }
}

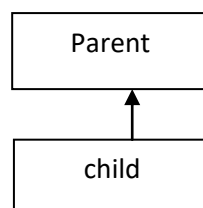
class FinalClass{
    public static void main(String[] args) {
        Child c = new Child();
        c.test();
```

```
c.A=60; /* here it throws an error that final value can not be reinitialize */
        System.out.println(c.A);
    }
}
```

- Subclass can inherit final method of super class but can not override it.
- The types of inheritance are :
  - Single Inheritance
  - Multi-level inheritance
  - Multiple inheritance
  - Hierarchical inheritance
  - Hybrid inheritance

### Single Inheritance

- One subclass acquiring or inheriting properties of one superclass is called as single inheritance.



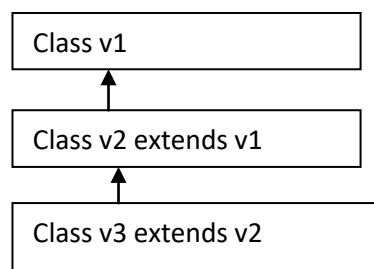
## Program

```
class Super1
{
    int a =50;
}
class Child extends Super1
{
    int d=10;
    public void test()
    {
        System.out.println("this is child class " +d);
        System.out.println("this is super class " + a);
    }
}
class FinalClass{
    public static void main(String[] args) {
        Child c = new Child();
        c.test();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java SimpleInheritance
this is child class 10
this is super class 50
```

## Multi-level inheritance

- One sub-class inheriting from one super class and that super class is inheriting from another super class is called as multilevel inheritance.



## Program

```
class WhatsappV1
{
    public void msg()
    {
```

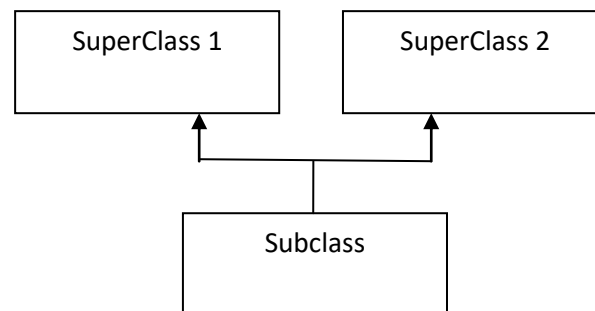
```

        System.out.println("this is msg ()");
    }
}
class WhatsappV2 extends WhatsappV1
{
    public void voiceMsg()
    {
        System.out.println("this is voice msg " );
    }
}
class WhatsappV3 extends WhatsappV2
{
    public void video()
    {
        System.out.println("This is video ()");
    }
}
class MultilevelInheritance{
    public static void main(String[] args) {
        WhatsappV3 v=new WhatsappV3();
        v.msg();
        v.voiceMsg();
        v.video();
only object creation to see which class to call which method
        WhatsappV2 v2= new WhatsappV2();
        v2.msg();
        v2.voiceMsg();
        WhatsappV1 v1= new WhatsappV1();
        v1.msg();
    }
}

```

## Multiple Inheritance

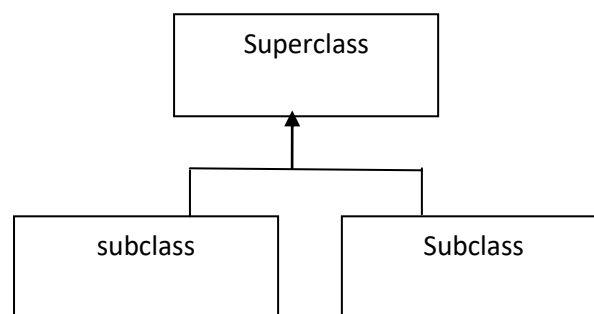
- One subclass inheriting from two or more superclass is called as multiple inheritance.



- **Java does not support multiple inheritance**

## Hierarchical inheritance :

- One superclass extended by two or more subclass is called as hierarchical inheritance.



- Using hierarchical inheritance we can achieve **generalization**.
- Defining the common methods and variables of subclass in one superclass is called as **Generalization**.
- Developing methods and variable specific to one class is called as **specialization**.

## Program

```
class Account
{
    public void createAccount()
    {
        System.out.println("Your account is created");
    }
}
class SavingAcnt extends Account
```

```

    {
        public void showSavings ()
        {
            System.out.println("Your saving account is 2000");
        }
    }
class LoanAcnt extends Account
{
    public void showLoan()
    {
        System.out.println("Your pending loan is 3000");
    }
}
class HierarchialInheritance
{
    public static void main(String[] args)
    {
        LoanAcnt l = new LoanAcnt();
        l.createAccount();
        l.showLoan();
        SavingAcnt s = new SavingAcnt();
        s.createAccount();
        s.showSavings();
    }
}

```

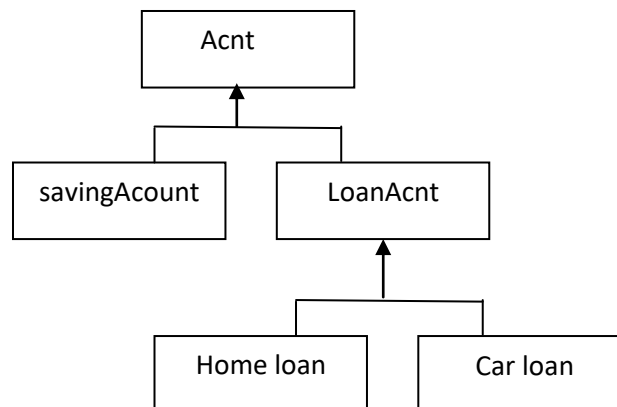
```

G:\javaprogs\OOPS\abstraction>java SimpleInheritance
Your account is created
Your pending loan is 3000
Your account is created
Your saving account is 2000

```

## Hybrid Inheritance

- In this this combination of different type of inheritance.



## SUPER(); STATEMENT

- It is used to call the super class constructor from subclass constructor.

```
class Super1
{
    public Super1()
    {
        System.out.println("this is super constructor");
    }
}
class Base extends Super
{
    public Base()
    {
        super();
        System.out.println("this is base constructor");
    }
}
class SuperStatement
{
    public static void main(String[] args) {

        Base b = new Base();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java SimpleInheritance
this is super constructor
this is base constructor
```

- Super statement can be written explicitly by the programmer or implicitly by the compiler.
- Super(); statement should be written the first line of constructor body.

```

class Super1
{
    public Super1(int a )
    {
        System.out.println("this is super constructor");
    }
}
class Base extends Super
{
    public Base()
    {
        System.out.println("this is base constructor");
        super(); // here it throws an error because super
statement is always in first line
    }
}
class SuperStatement
{
    public static void main(String[] args) {

        Base b = new Base();

    }
}

```

- Multiple super statement within the same constructor body is not allowed.

```

class Super1
{
    public Super1(int a )
    {
        System.out.println("this is super constructor");
    }
}
class Base extends Super
{

```



```

        public Base()
        {
            super();
            super(); // here it throws an error because super
statement is always in first line
            System.out.println("this is base constructor");
        }
    }
    class SuperStatement
    {
        public static void main(String[] args) {

            Base b = new Base();

        }
    }

```

- If the superclass contains only parameterized constructor then the programmer should write super statement explicitly and pass the required argument value.

Program

```

class Super1
{
    public Super1(int a )
    {
        System.out.println("this is super constructor");
    }
}
class Base extends Super1
{
    public Base()
    {
        super(10); // calling the argumented constructor
        System.out.println("this is base constructor");
    }
}
class SimpleInheritance
{
    public static void main(String[] args) {
        Base b = new Base();}
}

```

```
G:\javaprogs\OOPS\abstraction>java SimpleInheritance
this is super constructor
this is base constructor
```

- If super class have constructor overloading then to call all these methods.

### Program

```
class Super1
{
    public Super1(int a)
    {
        this(10,20); //call the constructor here
        System.out.println("this is super constructor");
    }
    public Super1(int a, int b)
    {
        System.out.println("this is ....");
    }
}
class Base extends Super1
{
    public Base()
    {
        super(10); // calling the superclass constructor
        System.out.println("this is base constructor");
    }
}
class SimpleInheritance
{
    public static void main(String[] args) {
        Base b = new Base();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java SimpleInheritance
this is ....
this is super constructor
this is base constructor
```

## Object class

- It is the supermost class in java. Every class in java directly or indirectly extends from object class.
- If the class is not extending from any class then the compiler makes the object class as superclass by writing “extends object” implicitly.

## Constructor chaining

- Subclass constructor calling super class constructor, superclass constructor calling object class constructor is called as constructor chaining.

### Note :

- within the same constructor body it is not possible to write both this() statement

```
class Super1 extends Object
{
    public Super1()
    {
        System.out.println("this is super constructor");
    }
}
class Base extends Super
{
    public Base()
    {
        this();
        super(); // here it throws an error that superl ine should
        be first line
        System.out.println("this is base constructor");
    }
}
class SuperStatement
{
    public static void main(String[] args) {

        Base b = new Base();
    }
}
```

- if the subclass and superclass have the members with same names then within the subclass methods the compiler gives preference for subclass members.

```

class Super1 extends Object
{
    int a = 50;
}
class Base extends Super
{
    double a = 60.0;
    public Base()
    {
        System.out.println("value of a = " + a);
        System.out.println("value of a = " + a);
        //output a = 60.0
    }
}
class SuperStatement
{
    public static void main(String[] args) {

        Base b = new Base();
    }
}

```

### **Super Keyword**

- It is a special type of reference variables which always points to super class object.
- Super keyword is used to differentiate between subclass and super class member whenever we try to access them in subclass method.

```

class Super1 extends Object
{
    int a = 50;
}
class Base extends Super1
{
    double a = 60.0;
    public Base()

```

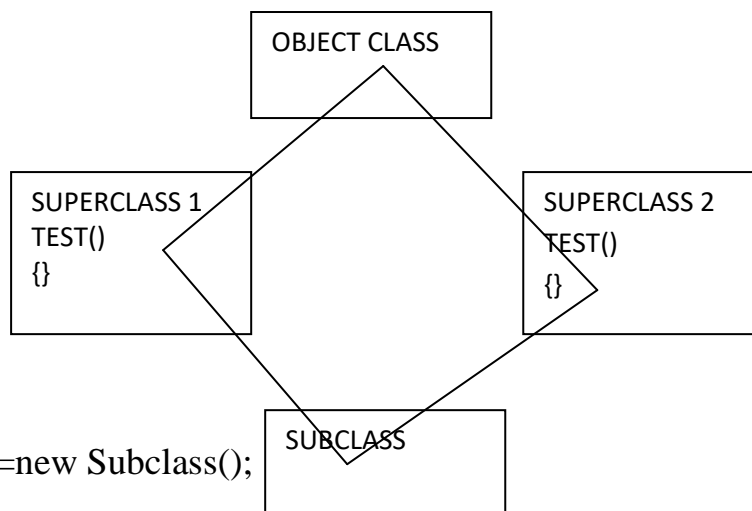
```

{
    System.out.println("value of a = " + a);
    System.out.println("value of a = " + super.a);
    // op - a = 60.0
    // op - a = 50
}
}
class SuperStatement
{
    public static void main(String[] args) {

        Base b = new Base();
    }
}

```

## WHY JAVA DOES NOT SUPPORT MULTIPLE INHERITANCE OR EXPLAIN DIAMOND PROBLEM IN JAVA



Subclass s1=new Subclass();

S1.test();

- It creates an ambiguity for the compiler to choose the path from where the properties of object class should be copied to subclass.
- To call constructor of two super-class two super(); statement are required and this is not supported in java.
- Note : multiple super statements are not allowed because super class object should be created only once

- If two super class contains method with same name and arguments and if you tried to call the method from subclass object, it results the ambiguity for the compiler which choose the method for execution.

## **CHAPTER 6 : METHOD OVERLOADING**

- Developing multiple methods with the same name within the same class which differs in
  - (i) no. of arguments
  - (ii) data type of arguments
  - (iii) sequence of arguments
 is called as method overloading.
- we can overload both static and non static methods.

```

class Graphsheet
{
    public void drawPoint()
    {
        System.out.println("drawing poitn at 0,0");
    }
    public void drawPoint(int x, char cord)
    {
        System.out.println("drawing point at "+cord + " , "+x);
    }
    public static void drawPoint(int x, int y)
    {
        System.out.println("drawing point at " +x + " , " + y);
    }
    public static void main(String[] args)
    {
        Graphsheet g=new Graphsheet();
        g.drawPoint();
        g.drawPoint(10, 'X');
        Graphsheet.drawPoint(10,20);
    }
}

```

```
G:\javaprogs\OOPS\abstraction>java Graphsheet
drawing poitn at 0,0
drawing point at X , 10
drawing point at 10 , 20
```

- Two overloaded method can have different return datatypes.

```
class Graphsheet
{
    public void drawPoint() // return type is void
    {
        System.out.println("drawing poitn at 0,0");
    }
    public int drawPoint(int x, char cord) // return type is int
    {
        System.out.println("drawing point at "+cord + " , "+x);
        return 10;
    }
    public static void main(String[] args)
    {
        Graphsheet g=new Graphsheet();
        g.drawPoint();
        g.drawPoint(10, 'X');
    }
}
```

- with the help of method overloading we can achieve code flexibility.
- Method overloading is the best example for compile time polymorphism
- The binding of method declaration to method definition is done at the compile time and hence it is called as compile time binding or early binding.
- Since the binding is done at the compile time it can not be changed at runtime and hence it is called as static binding.
- It makes easy for the programmer to remember the methods name and increase the readability of program.

### **Disadvantage of Method Overloading**

- The overloaded methods should have required java documents in order to understand the functionality of every overloaded method.

### **Application of Method Overloading**

- Whenever the project or application has same logic to be executed with different type of arguments then we go for method overloading.

### **Program**

```
class Graphsheet
{
    public void drawPoint()
    {
        System.out.println("drawing poitn at 0,0");
    }
    public void drawPoint(int x, char cord)
    {
        System.out.println("drawing point at "+cord + " , "+x);
    }
    public void drawPoint(int x, int y)
    {
        System.out.println("drawing point at " +x + " , " + y);
    }
    public void drawPoint(double x, double y)
    {
        System.out.println("drawing point at "+x+ " , " + y);
    }
    public static void main(String[] args)
    {
        Graphsheet g=new Graphsheet();
        g.drawPoint();
        g.drawPoint(10, 'X');
        g.drawPoint(10,20);
        g.drawPoint(10.2, 25.2);
    }
}
```



## CHAPTER 7 : METHOD OVERRIDING

- Subclass inheriting the method of super class and changing the method definition without changing method declaration (no. of arguments and method name should be same) according to subclass specification is called as Method Overriding.

### Program

```
class Superclass
{
    public void count(int n)
    {
        for (int i=1; i<n;i++)
        {
            System.out.println(i);
        }
    }
}
class Sublcass extends Superclass
{
    public void count(int n) // METHOD DECLATION IS SAME
    {
        for(int i=n;i>1;i--) // CHANGE THE DEFINITION
        {
            System.out.println(i);
            System.out.println();
        }
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass sup1=new Superclass();
        sup1.count(10);
        Sublcass sub1=new Sublcass();
        sub1.count(5);}}}
```

### **@override :**

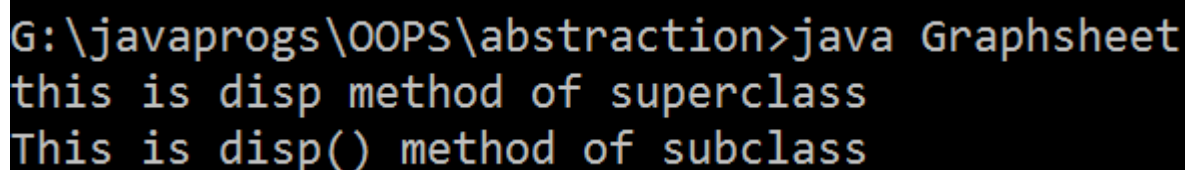
- Override annotation helps in addition compile time verification in the subclass to ensure the programmer overriding is overriding the proper method of superclass.
- It will also improve code readability.
- It is not mandatory to write @override.
- If we don't write @override then if the method name in subclass different then it treat as a another method and not doing overriding.

### **Program**

```
class Superclass
{
    public void disp()
    {
        System.out.println("this is disp method of superclass");
    }
}
class Sublcass extends Superclass
{
    @Override // Here if we write anotation then only it
    public void count()
    {
        System.out.println("This is disp() method f subclass");
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass sup1=new Superclass();
        sup1.disp();
        Sublcass sub1=new Sublcass();
        sub1.disp();
    }
}
```

**If we don't write @override annotation then it does not throw an error if superclass and subclass method is different in name.**

```
class Superclass
{
    public void disp()
    {
        System.out.println("this is disp method of superclass");
    }
}
class Sublcass extends Superclass
{
    public void count() // here method is different
    {
        System.out.println("This is disp() method f subclass");
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass sup1=new Superclass();
        sup1.disp();
        Sublcass sub1=new Sublcass();
        sub1.count();
    }
}
```



```
G:\javaprogs\OOPS\abstraction>java Graphsheet
this is disp method of superclass
This is disp() method of subclass
```

**Note :**

- Static method can not be overridden.

**Program**

```
class Superclass
{
    public static void disp()
    {
```

```

System.out.println("this is disp method of superclass");
    }
}
class Subclass extends Superclass
{
    @Override
    public static void disp()
    {
        System.out.println("This is disp() method f subclass");
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass.disp();
        Subclass.disp();
    }
}

```

**Here it throw a compile time error as method does not override or implement a method from a supertype.**

- If subclass and super class have same static methods with same declaration and same definition then it is called as method hiding.

**Program :**

```

class Superclass
{
    public static void disp()
    {
        System.out.println("this is disp() of superclass");
    }
}
class Subclass extends Superclass
{
    public static void disp()
    {
        System.out.println("This is disp() method of subclass");
    }
}

```

```

    }
    class MainOverride
    {
        public static void main(String[] args)
        {
            Subclass.disp();
        }
    }

```

```

G:\javaprogs\OOPS\abstraction>java Graphsheet
This is disp() method of subclass

```

- Final method can be inherited but cannot be overridden.

#### **Program :**

```

class Superclass
{
    public final void disp()
    {
        System.out.println("this is disp method of superclass");
    }
}
class Subclass extends Superclass
{
    @Override
    public final void disp()
    {
        System.out.println("This is disp() method f subclass");
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass s1=new Superclass() ;
        s1.disp();
        Subclass s2 = new Subclass();
        s2.disp();
    }
}

```

**Here it throw an error that overridden method is final**

```
G:\javaprogs\OOPS\abstraction>javac Graphsheet.java
Graphsheet.java:11: error: disp() in Subclass cannot override disp() in Superclass
    public final void disp()
                   ^
    overridden method is final
1 error
```

- Private method can not be inherited and cannot be overridden.

**Program :**

```
class Superclass
{
    private void disp()
    {
        System.out.println("this is disp method of superclass");
    }
}
class Subclass extends Superclass
{
    @Override
    private final void disp()
    {
        System.out.println("This is disp() method f subclass");
    }
}
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass s1=new Superclass() ;
        s1.disp();
        Subclass s2 = new Subclass();
        s2.disp();
    }
}
```

**//here it throw an error that it can not be override.**

- Method Overriding is the best example **for Runtime Polymorphism.**
- The binding of method declaration to member definition is done at runtime and hence it is called as **late binding.**

- The binding can be changed dynamically at the runtime and hence it is also called as dynamic binding.

### **Multilevel Override**

```
class Whatsapp1
{
    public void sentReport()
    {
        System.out.println("show one black tick");
    }
}
class Whatsapp2 extends Whatsapp1
{
    public void sentReport()
    {
        System.out.println("show two black tick");
    }
}
class Whatsapp3 extends Whatsapp2
{
    public void sentReport()
    {
        System.out.println("Two blue tick");
    }
}
class MultilevelOverride
{
    public static void main(String[] args)
    {
        Whatsapp1 w1= new Whatsapp1();
        w1.sentReport();
        Whatsapp2 w2 = new Whatsapp2();
        w2.sentReport();
        Whatsapp3 w3=new Whatsapp3();
        w3.sentReport();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java Graphsheet
show one black tick
show two black tick
Two blue tick
```

## **HierarchialOverride**

```
class AndroidOs
{
    public void showHomeScreen()
    {
        System.out.println("show 5 icons");
    }
}
class SamsungOs extends AndroidOs
{
    public void showHomeScreen()
    {
        System.out.println("show 2 icons");
    }
}
class RedmiOs extends AndroidOs
{
    public void showHomeScreen()
    {
        System.out.println("show 6 icons");
    }
}
class MotoOs extends AndroidOs
{
    public void showHomeScreen()
    {
        System.out.println("show 10 icons");
    }
    public void motoUpdate()
    {
        System.out.println("show Update for moto only");
    }
}
class HierarchialOverride
{
    public static void main(String[] args)
    {
```



```

        AndroidOs a1 = new AndroidOs();
        a1.showHomeScreen();
        SamsungOs s1 = new SamsungOs();
        s1.showHomeScreen();
        MotoOs m1 = new MotoOs();
        m1.showHomeScreen();
        m1.motoUpdate();
        RedmiOs r1 = new RedmiOs();
        r1.showHomeScreen();
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java Graphsheet
show 5 icons
show 2 icons
show 10 icons
show Update for moto only
show 6 icons

```

#### **Note :**

**\*\***

- if subclass and superclass contains method with same name but different arguments then the subclass will contain 2 overloaded methods.

#### **Program :**

```

class Superclass
{
    public void disp()
    {
        System.out.println("this is disp() of superclass");
    }
    public void disp(int n)
    {
        System.out.println("this is disp(int n) of superclass");
    }
}
class Subclass extends Superclass
{
    @Override

```

```

        public void disp(int n)
        {
            System.out.println("This is disp() method f subclass");
        }
    }
    class MainOverride
    {
        public static void main(String[] args)
        {
            Superclass s1=new Superclass();
            s1.disp(10);

            Subclass s2 = new Subclass();

            s2.disp();
            s2.disp(5);
        }
    }

```

```

G:\javaprogs\OOPS\abstraction>java MainOverride
this is disp(int n) of superclass
this is disp() of superclass
This is disp() method f subclass

```

- To achieve method overriding inheritance is mandatory.

**Program :**

```

class Superclass
{
    public void disp()
    {
        System.out.println("this is disp method of superclass");
    }
}
class Subclass
{
    @Override
    public final void disp()

```

```

        {
            System.out.println("This is disp() method f subclass");
        }
    }
class MainOverride
{
    public static void main(String[] args)
    {
        Superclass s1=new Superclass() ;
        s1.disp();
        Subclass s2 = new Subclass();
        s2.disp();
    }
}

```

**// here there is no inheritance between subclass and superclass so there is no overridden happen between them.**

- From the subclass object if you (programmer) try to call overridden method you (programmer) always get overridden implementation of the method.

### **Program**

```

class Superclass
{
    public void disp()
    {
        System.out.println("this is disp() of superclass");
    }
}
class Subclass extends Superclass
{
    //@Override
    public void disp()
    {
        System.out.println("This is disp() method of subclass");
    }
}
class MainOverride

```

```

{
    public static void main(String[] args)
    {
        Subclass s1=new Subclass();
        s1.disp();
    }
}

```

**//output of this program is this is disp() method of subclass, because it call always overridden method.**

- If we want to execute the overridden implementation of the method and original implementation of the method present in superclass then we can use super keyword from the subclass method to call superclass method.

### **Program**

```

class SuperClass
{
    public void view()
    {
        System.out.println("This is view() of SuperClass");
    }
    public void disp()
    {
        System.out.println("This is display() of SuperClass");
    }
}
class Subclass extends SuperClass
{
    public void view()
    {
        super.view();
        System.out.println("this is the view() of Subclass");
    }
    public void disp(int n)
    {
        System.out.println("this is the disp(int n) of subclass");
    }
}

```

```
class SuperClassOverride
{
    public static void main(String[] args)
    {
        Subclass ref1=new Subclass();
        ref1.view();
        ref1.disp();
        ref1.disp(10);
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java MainOverride
This is view() of SuperClass
this is the view() of Subclass
This is display() of SuperClass
this is the disp(int n) of subclass
```

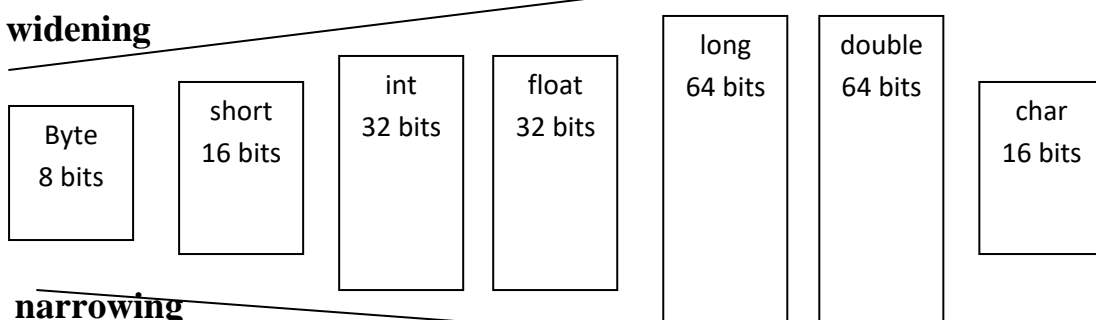
## CHAPTER 8 : TYPE CASTING

- Converting one type to another type is called as typecasting.
- It is of two type
  - (i) Primitive casting
  - (ii) Derived Casting

### Primitive Casting

- Converting one primitive type of another primitive type is called as primitive casting.

#### widening



Primitive Casting is of two type :

- (1) Widening
- (2) Narrowing

### Widening

- Converting a lower datatype value to higher datatype value is called as widening.
- Widening will be done by compiler implicitly.

### Program

```
class TypeCasting
{
    public static void main(String[] args)
    {
        int x1=20;
        double y1=x1;
        System.out.println("x1 = "+x1);
        System.out.println("y1 = "+y1);
    }
}
```

**o/p – x1=20  
y1 = 20.0**

### **Narrowing :**

- Converting a higher data type value to lower datatype value is called as Narrowing.
- Narrowing should done by the programmer explicitly by writing casting statement.
- **Type Casting Syntax :**  
**(datatype) value/variables ;**
- Narrowing always results in loss of data.

### **Program**

```
class TypeCastingLossOfData
{
    public static void main(String[] args)
    {
        int x1=127;
        byte y1=(byte) x1;
        System.out.println("x1 = "+x1);
        System.out.println("y1 = "+y1);

        int a=240;
        byte b = (byte)a;
        System.out.println("a = "+a);
        System.out.println("b = "+b);
    }
}
```

**} // in the above program while in 1<sup>st</sup> it does not loss of data because byte holds the data upto 128 bytes but in the second there is loss of data because we want o store 240 in byte, but the capacity of byte is only 128.**

**Op :**

**X1 = 127**

**Y1= 127**

**A= 240**

**B= -16 // here loss of data**

**Program :**

```
class TypeCasting
{
    public static void main(String[] args)
    {
        double x1=20;
        int y1=(int)x1;
        System.out.println("x1 = "+x1);
        System.out.println("y1 = "+y1);
    }
}
```

o/p – x1=20.0  
y1 = 20

- If we directly doing narrowing then it throws an error

**Program :**

```
class TypeCasting
{
    public static void main(String[] args)
    {
        double x1=20;
        int y1=x1;
        System.out.println("x1 = "+x1);
        System.out.println("y1 = "+y1);
    }
}
```

**//it throws an error as possible loss of precision**

**Simple typeCasing Program**

```
class TypeCasting
{
    public static void main(String[] args)
    {
        int a=20;
        double b= a; // widening
        System.out.println("a = "+a);
        System.out.println("b = "+b);

        double x1=20;
```



```

        int y1=(int) x1; // narrowing
        System.out.println("x1 = "+x1);
        System.out.println("y1 = "+y1);
    }
}

```

**Op– x1=20**  
**y1 = 20.0**  
**x1=20.0**  
**y1 = 20**

- If you try to convert a character to integer then integer variable will be containing Unicode value for the given character.

### Program

```

class CharacterTypeCasting
{
    public static void main(String[] args)
    {
        char c1 = 'A';
        int i = c1; // widening
        System.out.println("c1 = "+c1);
        System.out.println("i = " + i);

        int j = 77;
        char c2 = (char)j; // narrowing
        System.out.println(" j = " + j);
        System.out.println(" c2 = " + c2);
    }
}

```

### Program : Write a Function or method which converts any given string to lower case and print the same.

- For a method expecting primitive value we can pass the same primitive datatype value and all its lower primitive datatype value.

### Program

```

class Mainclass13
{
    public static void test(int x )
    {
        System.out.println("this is test(int x) of sample");
    }
}

```

```

    }
    public static void main(String[] args)
    {
        byte s = 10;
        test(s);
        short a = 52;
        test(a);
        int b=25;
        test(b);    }}

```

```

G:\javaprogs\OOPS\Typecasting>java Mainclass13
this is test(int x) of sample
this is test(int x) of sample
this is test(int x) of sample

```

- 
- For a method expecting primitive value then if we pass the higher primitive datatype value then it throws an error.

### Program

```

class Mainclass13
{
    public static void test(int x )
    {
        System.out.println("this is test(int x) of sample");
    }
    public static void main(String[] args)
    {
        double d = 10.5;
        test(d);
        long l = 25.3;
        test(l);
    }
}

```

```

G:\javaprogs\OOPS\Typecasting>javac Mainclass13.java
Mainclass13.java:10: error: method test in class Mainclass13 cannot be applied to given types;
        test(d);
        ^
required: int
found: double
reason: actual argument double cannot be converted to int by method invocation conversion
1 error

```

- If a class contains two overloaded methods one expecting lower datatype value, other expecting higher datatype value and if you pass a lower datatype value to call the method then always method with lower datatype value will be executed.

### Program

```
class Over
{
    public static void test(int x )
    {
        System.out.println("This is test(int x) ");
        System.out.println("print x = " + x);
    }
    public static void test(double x )
    {
        System.out.println("This is test(double x)");
        System.out.println("print x = " + x);
    }
    public static void main(String[] args)
    {
        int v1=10;
        test(v1);
    }
}

// op : this is test(int x )
//      print x = 10
```

### Derived Casting

- Converting one reference type to another reference type is called as derived casting.
- Derived Casting is of two type :
  - Upcasting
  - Downcasting

## Upcasting

- Converting subclass or childclass reference to superclass or parent class reference is called as upcasing.

## Program

```
class SuperClass
{
    public void test()
    {
        System.out.println("This is test() ");
    }
}
class Subclass extends SuperClass
{
    public void view()
    {
        System.out.println("This is view()");
    }
}
class Upcasting
{
    public static void main(String[] args)
    {
        SuperClass sup1 = new Subclass(); //Upcasting
        sup1.test();
        or we can also write like this
        Subclass sub1 = new Subclass();
        SuperClass sup2=sub1; // Upcasting
        sup2.test();
    }
}
// op : this is test
```

- Using upcasted reference variables can access only superclass or parent class property.

### Program

```
class SuperClass
{
    public void test()
    {
        System.out.println("This is test() ");
    }
}
class Subclass extends SuperClass
{
    public void view()
    {
        System.out.println("This is view()");
    }
}
class Upcasting
{
    public static void main(String[] args)
    {
        SuperClass sup1 = new Subclass();
        sup1.view(); /* here it throw an error as cannot find the
symbol because Superclass reference can only access the
superclass method not a subclass method. */
    }
}
```

- Upcasting will be done by the compiler implicitly.
- Upcasting is achieved by storing the address of child class object into parent class reference variable.

SuperClass sup1 = new Subclass();

In the above line sup1 is the reference variable which store the address of Sublclass object

## Downcasting

- Converting the upcasted reference back to subclass reference is called as downcasting.
- Downcasting should be done by the programmer explicitly by writing casting statement.

## Program

```
class SuperClass
{
    public void test()
    {
        System.out.println("This is test() ");
    }
}
class Subclass extends SuperClass
{
    public void view()
    {
        System.out.println("This is view()");
    }
}
class Upcasting
{
    public static void main(String[] args)
    {
        SuperClass sup1 = new Subclass();
        Subclass sub1 = (Subclass)sup1; // downcasting,
done by writing downcasting statment
        sub1.test();
        sub1.view();
    }
}
```

- We can downcast only upcasted reference. If we downcast directly superclass into subclass then JVM throws an exception as **ClassCastException** at runtime.

### Program

```
class SuperClass
{
    public void test()
    {
        System.out.println("This is test() ");
    }
}
class Subclass extends SuperClass
{
    public void view()
    {
        System.out.println("This is view()");
    }
}
class Upcasting
{
    public static void main(String[] args)
    {
        SuperClass sup1 = new SuperClass();
        Subclass sub1 = (Subclass)sup1; // downcast directly
        superclass without upcasting
        sub1.test();
        sub1.view();
    }
}
```

```
G:\javaprogs\OOPS\abstraction>java MainOverride
Exception in thread "main" java.lang.ClassCastException: SuperClass cannot be cast to Subclass
    at MainOverride.main(MainOverride.java:20)
```

## Derived Downcasting with Multilevel Inheritance

```
class SuperClass
{
    public void display()
    {
        System.out.println("this is display() of SuperClass");
    }
}
class SuperClass2 extends SuperClass
{
    public void click()
    {
        System.out.println("this is click() method of SuperClass2");
    }
}
class Subclass extends SuperClass2
{
    public void count()
    {
        System.out.println("this is count() of Subclass");
    }
}
class SuperClass1
{
    public static void main(String[] args)
    {
        System.out.println("Program start....");
        SuperClass2 ref1=new Subclass(); //upcasting
        ref1.click();
        ref1.display();
        System.out.println();

        SuperClass ref2 = new Subclass(); //upcsting
        ref2.display();
        System.out.println();

        Subclass ref3 = (Subclass) ref1; // downcasting
        ref3.click();
        ref3.display();
    }
}
```



```

        ref3.count();
        System.out.println();

        SuperClass2 ref4 = (SuperClass2) ref2; // downcasting
        ref4.display();
        ref4.click();
        System.out.println();
        Subclass ref5= (Subclass) ref2;
        ref5.display();
        ref5.click();
        ref5.count();
        System.out.println("program end.....");
    }}

```

```

G:\javaprogs\OOPS\abstraction>java MainOverride
Program start....
this is click() method of SuperClass2
this is display() of SuperClass

this is display() of SuperClass

this is click() method of SuperClass2
this is display() of SuperClass
this is count() of Subclass

this is display() of SuperClass
this is click() method of SuperClass2

this is display() of SuperClass
this is click() method of SuperClass2
this is count() of Subclass
program end.....

```

- If you try to downcast a reference to the class type which do not contain the properties of given class then JVM throws **ClassCastException** at runtime.

### **Program**

```
class Mouse
{
    public void click()
    {
        System.out.println("mouse button clicked");
    }
}
class Pendrive
{
    public void read()
    {
        System.out.println("reading data from pendrive");
    }
    public void write()
    {
        System.out.println("Write data to pendrive");
    }
}
class Usb_Port
{
    public static void connect(Object obj)
    {
        Mouse ref1=(Mouse)obj;
        ref1.click();
    }
}
class MainClass5
{
    public static void main(String[] args)
    {
        Mouse m1= new Mouse();
        Pendrive p1=new Pendrive();
        Usb_Port.connect(m1);
    }
}
```

```
        Usb_Port.connect(p1); // here it throw runtime error as
        ClassCastException as Pendrive class can not cast to Mouse Class
    }
}
```

- We can avoid ClassCastExcpetion by using **instanceof operator**.
- Instanceof operator checks if the given reference contains the property of given class and return true if the properties are present else it return false.

### **Program**

```
class Mouse
{
    public void click()
    {
        System.out.println("mouse button clicked");
    }
}
class Pendrive
{
    public void read()
    {
        System.out.println("reading data from pendrive");
    }
    public void write()
    {
        System.out.println("Write data to pendrive");
    }
}
class Usb_Port
{
    public static void connect(Object obj)
    {
// to overcome ClassCastException exception use instanceof
Operator
        if(obj instanceof Mouse == true)
        {
            Mouse ref1=(Mouse)obj;
            ref1.click();
        }
    }
}
```

```

        else
        {
            Pendrive ref = (Pendrive)obj;
            ref.read();
            ref.write();
        }
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Mouse m1= new Mouse();
        Pendrive p1=new Pendrive();
        Usb_Port.connect(m1);
        Usb_Port.connect(p1);
    }
}

```

```

G:\javaprogs\OOPS\abstraction>java MainOverride
mouse button clicked
reading data from pendrive
Write data to pendrive

```

- If you want to relate to unrelated classes then you can use **Object Class** as the common superclass to perform derived casting, achieve runtime polymorphism

#### **Program**

```

public static void connect(Object obj)
// here Object is the SuperClass of Mouse and Pendrive Class.

```

- If a method expecting superclass reference then for the same method you can pass reference of superclass and all its subclass.

### Program

```

class Mouse
{
    public void click()
    {
        System.out.println("mouse button clicked");
    }
}
class Pendrive
{
    public void read()
    {
        System.out.println("reading data from pendrive");
    }
    public void write()
    {
        System.out.println("Write data to pendrive");
    }
}
class Usb_Port
{
    public static void connect(Object obj)
    {
        System.out.println("this is object obj of Usb_Port1");
    }
}
class Mainclass15
{
    public static void main(String[] args)
    {
        Mouse m1= new Mouse();
        Usb_Port.connect(m1); // pass the subclass reference to connect method which expecting superclass reference
        Pendrive p1 = new Pendrive();// pass the subclass reference to connect method which expecting superclass reference
        Usb_Port.connect(p1);
    }
}

```

```
Object obj=new Object();
    Usb_Port.connect(obj); // pass the superclass reference to
connect method which expecting superclass reference
}}
```

```
G:\javaprogs\OOPS\Typecasting>java Mainclass15
this is object obj of Usb_Port1
this is object obj of Usb_Port1
this is object obj of Usb_Port1
```

- If there are two overloaded methods in a class one expecting superclass reference and the other method expecting subclass reference and if you pass subclass reference to call the method then always method with subclass reference will be executed.

### Program

```
class Mouse
{
    public void click()
    {
        System.out.println("mouse button clicked");
    }
}
class Pendrive
{
    public void read()
    {
        System.out.println("reading data from pendrive");
    }
    public void write()
    {
        System.out.println("Write data to pendrive");
    }
}
class Usb_Port
{
    public static void connect (Mouse obj)
    {
```

```

        System.out.println("this is connect (Mouse obj)");
    }
    public static void connect(Object obj)
    {
        System.out.println("this is connect (Object obj)");
    }
}
class MainClass
{
    public static void main(String[] args)
    {
        Mouse m1= new Mouse();
        Usb_Port.connect(m1);
    }
}
// op - this is connect (Mouse Obj)

```

## CHAPTER 9 : ABSTRACT CLASS AND ABSTRACT METHOD

- A method which has only declaration and no definition is called as abstract method.
- A method which has declaration as well as definition is called as **concrete method**.
- If a method is abstract then it should be declared using abstract keyword.
- A class which is declared with abstract keyword is called as abstract class.
- If a class contains atleast one abstract method then the class must be declared as abstract.

### Program

abstract class Sample // **if we don't declare class as abstract then it throw an error as class Sample is not abstract and not override abstract method test().**

```
{
    abstract public void test(); // this is an abstract method
}
class Mainclass
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        System.out.println("Program end ");
    }
}
```

- Abstract class can contain :
  - (i) Only abstract method or only concrete methods or both abstract and concrete method.

```
abstract class Sample
{
    abstract public void test(); // abstract method
    public void count() // concrete method
    {
        System.out.println("this is count() of sample");
    }
}
```



```

    }
    class Mainclass2
    {
        public static void main(String[] args)
        {
            System.out.println("program start ");
        }
    }

```

- It is impossible to create objects for abstract class.

### Program

```

abstract class Sample
{
    abstract public void test(); //abstract method
    public void count() // concrete method
    {
        System.out.println("this is count() of sample");
    }
}
class Mainclass2
{
    public static void main(String[] args)
    {
        System.out.println("program start ");
        Sample s= new Sample(); // here object is created but it
        throw an error
        s.count();
    }
}

```

```

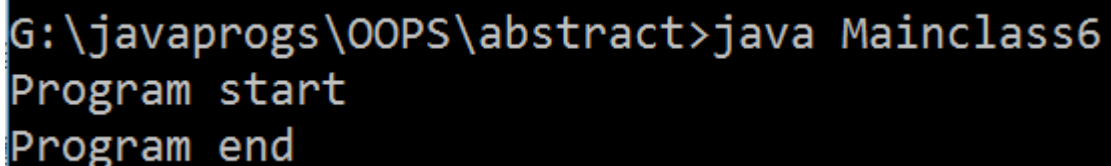
G:\javaprogs\OOPS\abstract>javac Mainclass2.java
Mainclass2.java:15: error: Sample is abstract; cannot be instantiated
        Sample s= new Sample();
                        ^
1 error

```

- The concrete methods of abstract class can be static or non static.

#### Program

```
abstract class Sample
{
    abstract public void test();
    public static void count() // static concrete method
    {
        System.out.println("this is count() of sample");
    }
    public void disp() // non-static concrete method
    {
        System.out.println("this is disp() of sample");
    }
}
class Mainclass6
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        System.out.println("Program end ");
    }
}
```



```
G:\javaprogs\OOPS\abstract>java Mainclass6
Program start
Program end
```

- The data members of abstract class can be static or non-static.

#### Program

```
abstract class Sample
{
    static int x = 50; //static data members
    int b = 20; // non static data members
    abstract public void test();
}
class Mainclass6
{
}
```

```

        public static void main(String[] args)
        {
            System.out.println("Program start");
            System.out.println("Program end ");
        }
    }

```

### **Op – Program Start**

### **Program end**

- If a class is extending an abstract class the subclass should override all the abstract methods of superclass.

### **Program**

**//to access non static member of abstract class 2nd solution**

**abstract class Sample**

```

{
    int x = 50;
    abstract public void test();
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}

class Demo extends Sample
{
    @Override
    public void test() // override the method of abstract class
    {
        System.out.println("Overriding test method of Sample in Demo");
    }
}

```

**class Mainclass8**

```

{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        Demo d1 = new Demo();
        d1.count();
        System.out.println("value of x = "+d1.x);
        System.out.println("Program end ");
    }
}

```

```

    }

G:\javaprogs\OOPS\abstract>java Mainclass8
Program start
this is count() of sample
value of x = 50
Program end

```

- If the class do not override the abstract methods of super class then the class should be declared as abstract.

### Program

```

abstract class Super
{
    abstract public void test();
}
abstract class Sub extends Super
{

}
class ab extends Sub
{
    public void test()
    {
        System.out.println("this is test()");
    }
}
class Practice
{
    public static void main(String[] args) {
        ab a = new ab();
        a.test();
    }
}

```

- Abstract class do not declared as final. It throw a compile time error.

### Program

```
// can not be define as final
final abstract class Sample
{
    abstract public void test();
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}
class Mainclass9
{
    public static void main(String[] args)
    {
        System.out.println("Program start");

        System.out.println("Program end ");
    }
}
```

```
G:\javaprogs\OOPS\abstract>javac Mainclass9.java
Mainclass9.java:2: error: illegal combination of modifiers: abstract and final
final abstract class Sample
    ^
1 error
```

- Abstract method can not be declared as static, because static method can not be overridden. It throw a compile time error.

### Program

```
abstract class Sample
{
    abstract public static void test(); // can not declare abstract
    method as static
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}
class Mainclass9
{
}
```

```

        public static void main(String[] args)
        {
            System.out.println("Program start");

            System.out.println("Program end ");
        }
    }

```

```

G:\javaprogs\OOPS\abstract>javac Mainclass9.java
Mainclass9.java:4: error: illegal combination of modifiers: abstract and static
    abstract public static void test();
                ^
1 error

```

- abstract method can not be declared as final, because final method can not be overridden.

### Program

```

abstract class Sample
{
    abstract public final void test(); // can't define abstract method as final it throw a compile time error
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}

class Mainclass9
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        System.out.println("Program end ");
    }
}

```

```

G:\javaprogs\OOPS\abstract>javac Mainclass9.java
Mainclass9.java:4: error: illegal combination of modifiers: abstract and final
    abstract public final void test();
                ^
1 error

```

- abstract methods can not be declared as private because private method can not be overridden, or cant be inherited.

### Program

```
abstract class Sample
{
    abstract private void test();
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}
class Mainclass9
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        System.out.println("Program end ");
    }
}
```

```
G:\javaprogs\OOPS\abstract>javac Mainclass9.java
Mainclass9.java:4: error: illegal combination of modifiers: abstract and private
    abstract private void test();
                    ^
1 error
```

- Abstract class can not be declared as private.

### Program

```
private abstract class Sample
{
    abstract public void test();
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}
class Mainclass9
{
    public static void main(String[] args)
    {
        System.out.println("Program start");
        System.out.println("Program end ");
    }
}
```

```
G:\javaprogs\OOPS\abstract>javac Mainclass9.java
Mainclass9.java:2: error: modifier private not allowed here
private abstract class Sample
      ^
1 error
```

- the static member of abstract class directly call by using classname.datamember name

### Program

```
abstract class Sample
{
    static int x = 50;
    abstract public void test();
}
class Mainclass12
{
    public static void main(String[] args)
    {
        System.out.println("value of x " + Sample.x);
    }
}
```

```
G:\javaprogs\OOPS\abstract>java Mainclass12
value of x 50
```

- The non-static members of abstract class can be access only by creating the object of subclass.

### Program

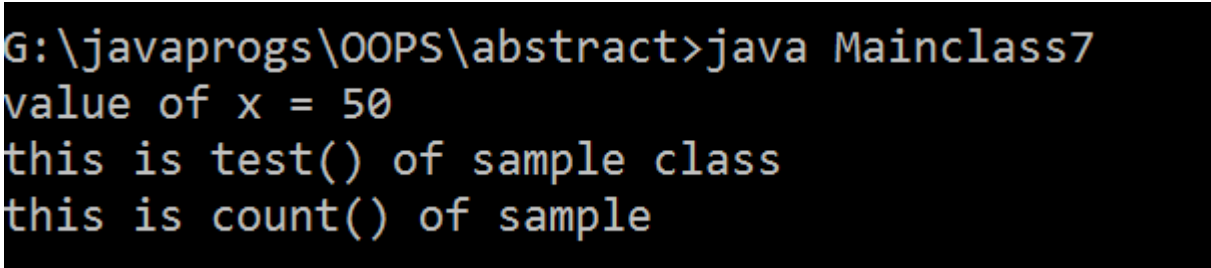
```
abstract class Sample
{
    static int x = 50;
    abstract public void test();
    public void count()
    {
        System.out.println("this is count() of sample");
    }
}
class Demo extends Sample
{
    @Override
```



```

        public void test()
        {
            System.out.println("this is test() of sample class ");
        }
    }
class Mainclass7
{
    public static void main(String[] args)
    {
        System.out.println("value of x = "+Sample.x);
        Demo d=new Demo();
        d.test();
        d.count();
    }
}

```



```

G:\javaprogs\OOPS\abstract>java Mainclass7
value of x = 50
this is test() of sample class
this is count() of sample

```

- we can create a reference variable for an abstract class.

### Program

```

abstract class RefVar
{
    abstract public void test();
    public static void main(String[] args)
    {
        System.out.println("program start");
        RefVar r ; // reference variable.
    }
}

```

- the abstract class reference variables are used to achieve
  - Derived Casting
  - Runtime Polymorphism

## Multilevel Abstract Program

```
abstract class Account
{
    public void createAccount()
    {
        System.out.println("Account is created");
    }
    abstract public void getAcntSmt();
}
class Saving extends Account{
    @Override
    public void getAcntSmt()
    {
        System.out.println("statement of saving account");
    }
}
class Loan extends Account
{
    public void getAcntSmt()
    {
        System.out.println("Statment of loan account");
    }
}
class Fd extends Account
{
    public void getAcntSmt()
    {
        System.out.println("Statment of FD account");
    }
}
class MultilevelInheritance
{
    public static void main(String[] args) {

        Saving s = new Saving();
        s.createAccount();
        s.getAcntSmt();
    }
}
```

```

        Loan l = new Loan();
        l.createAccount();
        l.getAcntSmt();
        Fd f = new Fd();
        f.createAccount();
        f.getAcntSmt();
    }
}

```

```

G:\javaprogs\OOPS\abstract>java MultilevelInheritance
Account is created
statement of saving account
Account is created
Statment of loan account
Account is created
Statment of FD account

```

## CHAPTER 10 : INTERFACE

- It is a type of class where the methods are by default abstract and the datamembers are by default static and final.

(i) methods are default abstract

### Program

```

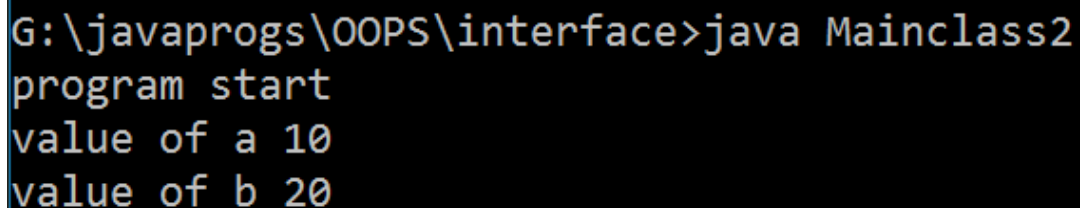
interface Run
{
    public void test();
}
class Mainclass2 implements Run
{
    public void test()
    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
        System.out.println("program start");} }

```

- (ii) data members are default static

### Program

```
interface Run
{
    int a=10; //default take as static
    static int b = 20;
    public void test();
}
class Mainclass2 implements Run
{
    public void test()
    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
        System.out.println("program start");
        System.out.println("value of a " + Run.a);
        System.out.println("value of b " + Run.b);
    }
}
```



```
G:\javaprogs\OOPS\interface>java Mainclass2
program start
value of a 10
value of b 20
```

- (iii) data members are default final

### Program

```
interface Run
{
    int a=10; //default take as static
    static int b = 20;
    public void test();
}
class Mainclass2 implements Run
{
    public void test()
```

```

    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
        System.out.println("program start");
        Run.a=20; // here it throw an error
        System.out.println("value of a " + a);
        System.out.println("value of b " + b);
    }
}

```

```

G:\javaprogs\OOPS\interface>javac Mainclass2.java
Mainclass2.java:15: error: cannot assign a value to final variable a
        Run.a=20;
            ^
1 error

```

- It is imposible to create object for interface.

### Program

```

interface Run
{
    public void test();
}
class Mainclass2 implements Run
{
    public void test()
    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
        System.out.println("program start");
        Run r = new Run(); //creating the object
    }
}

```

```

G:\javaprogs\OOPS\interface>javac Mainclass2.java
Mainclass2.java:13: error: Run is abstract; cannot be instantiated
        Run r = new Run();
            ^
1 error

```

- we can create reference variable or interface which is used to achieve
  - (i) runtime polymorphism
  - (ii) Derived Casting

### Program

```
interface Run
{
    public void test();
}
class Mainclass2 implements Run
{
    public void test()
    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
        System.out.println("program start");
        Run r ; // creating a reference variable
    }
} op – hello
```

- a class can inherit from an interface by using **implements** keyword only.

### Program

```
interface Run
{
    public void test();
}
class Mainclass2 implements Run // use the interface here
{
    public void test()
    {
        System.out.println("hello");
    }
    public static void main(String[] args) {
    }}
}}
```

- if a class implements from an interface then the class should override every abstract method of the interface.

**(i) override every abstract method**

interface Run

```
{  
    public void disp();  
    public void test();  
}
```

class Mainclass2 implements Run

```
{  
    public void disp()  
    {  
        System.out.println("this is disp");  
    }  
    public void test()  
    {  
        System.out.println("this is test");  
    }  
    public static void main(String[] args) {  
        Mainclass2 m = new Mainclass2();  
        m.disp();  
        m.test();  
    }  
}
```

**op- this is disp**

**This is test**

**(ii) if don't override every abstract method**

interface Run

```
{  
    public void disp();  
    public void test();  
}
```

class Mainclass2 implements Run

```
{  
    public void disp()  
    {  
        System.out.println("this is disp");  
    }  
}
```

```

    }
    public static void main(String[] args) {
        Mainclass2 m = new Mainclass2();
        m.disp();
    }
}

```

```

G:\javaprogs\OOPS\interface>javac Mainclass2.java
Mainclass2.java:6: error: Mainclass2 is not abstract and does not override
un
    class Mainclass2 implements Run
    ^
1 error

```

- Till j.D.k. 1.7 it was not possible to write concrete method inside the interface.

### Program

```

interface Run
{
    public void disp() // it throws an error
    {

    }
}
class Mainclass2 implements Run
{
    public static void main(String[] args) {

    }
}

```

```

G:\javaprogs\OOPS\interface>javac Mainclass2.java
Mainclass2.java:5: error: interface methods cannot have body
    {
    ^

```

- interface do not extends from any class (not even object class)
- interface do not have any constructor.

### Program

```

interface Run
{
    Run() // interface dont have a constructor
    {

    }
}

```



```

    }
}
class Sub implements Run
{

}
class Mainclass7
{
public static void main(String[] args) {
}}

```

- one interface can only extends another interface but can not implements it.

### **Program**

```

interface Run
{
}
interface Run1 extends Run{

}
class Sub implements Run, Run1
{

}
class Mainclass7
{
public static void main(String[] args) {
    System.out.println("Program Start");
}}

```

### **Op- Program Start**

- if we write implements instead of extends then it throw an error

### **Program**

```

interface Run
{
}
interface Run1 implements Run{

}

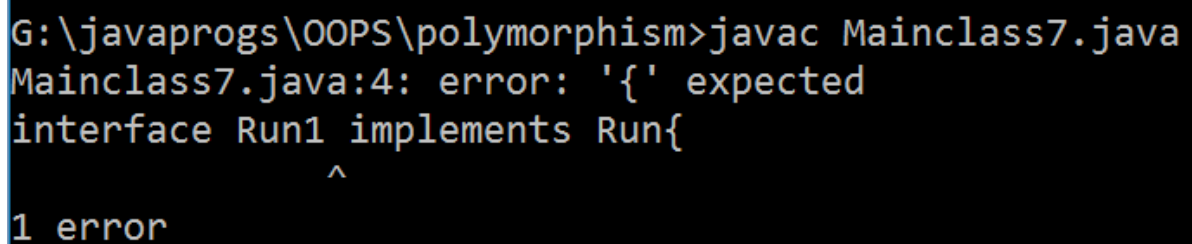
```

```

class Sub implements Run, Run1
{

}
class Mainclass7
{
public static void main(String[] args) {
    System.out.println("Program Start");
}}

```



```

G:\javaprogs\OOPS\polymorphism>javac Mainclass7.java
Mainclass7.java:4: error: '{' expected
interface Run1 implements Run{
                        ^
1 error

```

- A class can extends one class and implements an interface at the same time.

Program

```

interface Run
{
}
class Super
{

}
class Sub extends Super implements Run //here extending a class and implementing a interface at same time
{

}
class Mainclass7
{
public static void main(String[] args) {
    System.out.println("Program Start");
}}

```

**Op- Program Start**

- One class implements any number of interfaces.

### Program

```
interface Run
{
}
interface Run1
{}
interface Run2
{}
class Sub implements Run, Run1, Run2 // implementing morethan one interface at a time
{

}
class Mainclass7
{
public static void main(String[] args) {
    System.out.println("Program Start");
}}
```

### Op- Program Start

- One interface can extends any number of interface.

### Program

```
interface Run
{
}
interface Run1
{}
interface Run2 extends Run, Run1 // interface extending morethan one interface
{}
class Sub implements Run2
{

}
class Mainclass7
{
public static void main(String[] args) {
```

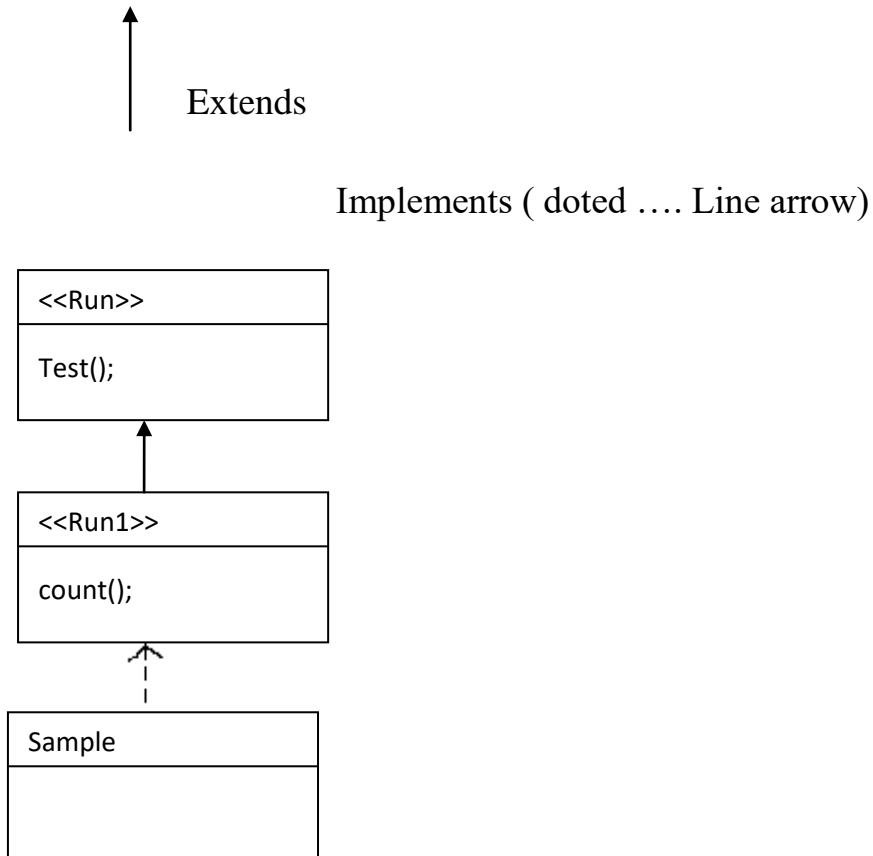
```

        System.out.println("Program Start");
    }}

```

## Op – Program Start

Class Diagram of interface



## Program

```

interface Run
{
    public void test();
}
interface Run1 extends Run
{
    public void disp();
}
class Sample implements Run1
{
    public void test()
    {

```

```

        System.out.println("This is test method");
    }
    public void disp()
    {
        System.out.println("This is disp method");
    }
}
class Mainclass8
{
    public static void main(String[] args) {
        Sample s = new Sample();
        s.test();
        s.disp();
    }
}

```

**Op – this is test method**  
**This is disp method**

### **Multiple inheritance using interface**

#### **Program**

```

interface Run
{
    public void test();
}
interface Run1
{
    public void disp();
}
class Sample implements Run1, Run
{
    public void test()
    {
        System.out.println("This is test method");
    }
    public void disp()
    {
        System.out.println("This is disp method");
    }
}

```

```

    }
    class Mainclass8
    {
        public static void main(String[] args) {
            Sample s = new Sample();
            s.test();
            s.disp();
        }
    }

```

**Op – this is test method**

**This is disp method**

### **Marker interface**

- An interface which do not contain any methods is called as master interface

### **Program**

```

interface Run
{

}

```

### **Functional interface**

- An interface which contains only one abstract method is called as functional interface.

### **Program**

```

interface Run
{
    public void disp();
}

```

## CHAPTER 11 : POLYMORPHISM

- One entity showing different behaviours at different places is called as polymorphism.
- Polymorphism helps in code flexibility
- There are two type of polymorphism.
  - (i) Compile time polymorphism
  - (ii) Runtime polymorphism

### Compile time polymorphism

- Binding the method declaration to method definition by the compiler at compile time based on the arguments is called as compile time polymorphism.
- It is also called as static binding.
- Eg. Method overloading and constructor overloading

### Runtime polymorphism

- Binding the method declaration to method definition by the jvm at run time based on the object is called as runtime polymorphism.
- It is also called as dynamic binding.
- Eg. Method overriding.

### Note :

**Using an upcasted reference if you call an overridden method then you always get overridden implementation on subclass implementation.**

- To achieve runtime polymorphism we have to follow three steps :
  - (i) Inheritance
  - (ii) Method overriding
  - (iii) Upcasting

### Program

```
class Superclass
{
    public void test()
    {
        System.out.println("this is test()");
    }
}
class Subclass extends Superclass // inheritance
{
    public void test() // method overriding
    {
```

```

        System.out.println("overriding of test () in subclass");
    }
}
class Mainclass1
{
    public static void main(String[] args) {

        Superclass sup1= new Subclass(); // upcasting
        sup1.test();
    }
}

```

### **Op – overriding of test() in subclass**

- **To show code flexibility using polymorphism**

### **Program**

```

interface card
{
    public void makePayment();
}
class Debit implements card
{
    public void makePayment()
    {
        System.out.println("payment done by Debit");
    }
}
class Credit implements card
{
    public void makePayment()
    {
        System.out.println("payment done by credit");
    }
}
class Swipemachine
{
    public static void swipe(card c)
    {

```



```

        c.makePayment();
    }
}
class Mainclass10
{
    public static void main(String[] args) {
        Credit c = new Credit();
        Swipemachine.swipe(c);
        Debit d = new Debit();
        Swipemachine.swipe(d);}}

```

```

G:\javaprogs\OOPS\polymorphism>java Mainclass10
payment done by credit
payment done by Debit

```

## 11. ABSTRACTION

- Hiding the implementation details of the class and exposing only the behaviours or services is called as abstraction.
- To achieve abstraction we have to follow three steps :
  - (i) Generalize the methods of implementation class and store them inside the interface.

### Program

```

interface Account
{
    public void deposit();
    public void withdraw();
}
class Saving implements Account
{
    public void deposit()
    {
        System.out.println("Money is deposit in saving account");
    }
    public void withdraw()
    {

```

```

        System.out.println("money is withdraw from saving
account");
    }
}
class Fd implements Account
{
    public void deposit()
    {
        System.out.println("money is deposit in FD account");
    }
    public void withdraw()
    {
        System.out.println("money is withdraw from FD account");
    }
}

```

- (ii) Create the object of implementation class and store the address in interface reference variable.

### Program

```

class AcntMgr
{
    public static Account createAcnt(char type)
    {
        Account a1; // interface reference
        if(type=='S')
        {
            a1 = new Saving(); //create the object of
implementation class and store the address in interface reference
            return a1;
        }
        else
        {
            a1 = new Fd();
            return a1;
        }
    }
}

```

- (iii) Use the interface reference variables to access the methods of implementation class.

### Program

```
Account a1 = AcntMgr.createAcnt('S'); //access the method of
implementation class using interface reference
    a1.deposit();
    a1.withdraw();    }}
```

- Using abstraction we can developed APIs(Application Programming Interface)
- Using abstraction we can achieve **Loose Coupling**.
- To achieve loose coupling we have to create three layers :
  - (i) Object implementation layer
  - (ii) Object creation layer
  - (iii) Object utilization layer.

### Program

```
interface Account
{
    public void deposit();
    public void withdraw();
}
class Saving implements Account
{
    public void deposit()
    {
        System.out.println("Money is deposit in saving account");
    }
    public void withdraw()
    {
        System.out.println("money is withdraw from saving
account");
    }
}
class Fd implements Account
{

```

```

        public void deposit()
        {
            System.out.println("money is deposit in FD account");
        }
        public void withdraw()
        {
            System.out.println("money is withdraw from FD account");
        }
    }
    class AcntMgr
    {
        public static Account createAcnt(char type)
        {
            Account a1;
            if(type=='S')
            {
                a1 = new Saving();
                return a1;
            }
            else
            {
                a1 = new Fd();
                return a1;
            }
        }
    }
    class Mainclass1
    {
        public static void main(String[] args)
        {
            Account a1 = AcntMgr.createAcnt('S');
            a1.deposit();
            a1.withdraw();    }}

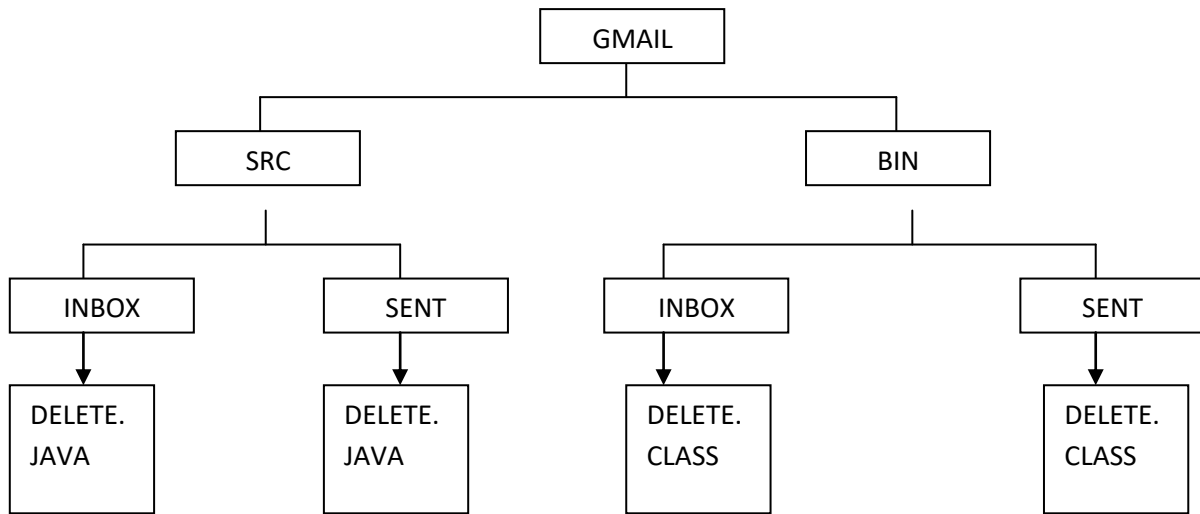
```

```

G:\javaprogs\OOPS\abstraction>java Mainclass1
Money is deposit in saving account
money is withdraw from saving account

```

## 12. JAVA PACKAGES



- A package is a group of classes and interface which is specific to one module or features the project.

### Advantage:

- Maintenance of the application source code is easy.
- It avoids naming collision.
- It provides security for the class and its members with the help of access specifiers.

### Note –

- The package name should be in the reverse order of domain .
- **Syntax :**
  - Domainname.appname.modulename
  - Domainname.companyname.appname.modulename
- For eg :
  - Com.gmail.inbox;
- Package names should be always written in lower case.
- To access the properties of a class or to create the object of the class present in different package we can use two different approaches.
  - Using fully qualified classname
  - Using import statement

### Fully Qualified classname

- A classname which is written with its package name is called as fully qualified classname
- Syntax :

Domain.app.module.classname

Eg: Com.gmail.inbox.demo

Program 1<sup>st</sup> package

```
package com.gmail.inbox;
```

```
public class Demo {
```

```
    public static void test()
```

```
    {
```

```
        System.out.println("test() from com.gmail.inbox; ");
```

```
    }}
```

**Program 2nd package**

```
package com.gmail.sent;
```

```
public class Demo {
```

```
    public static void test()
```

```
    {
```

```
        System.out.println("test() from com.gmail.inbox; ");
```

```
    }
```

```
}
```

**Program 3<sup>rd</sup> to use both package**

```
package com.gmail.run;
```

```
import com.gmail.inbox.Demo;
```

```
public class Mainclass {
```

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

```
        Demo.test();
```

```
        com.gmail.sent.Demo.test();
```

```
    }
```

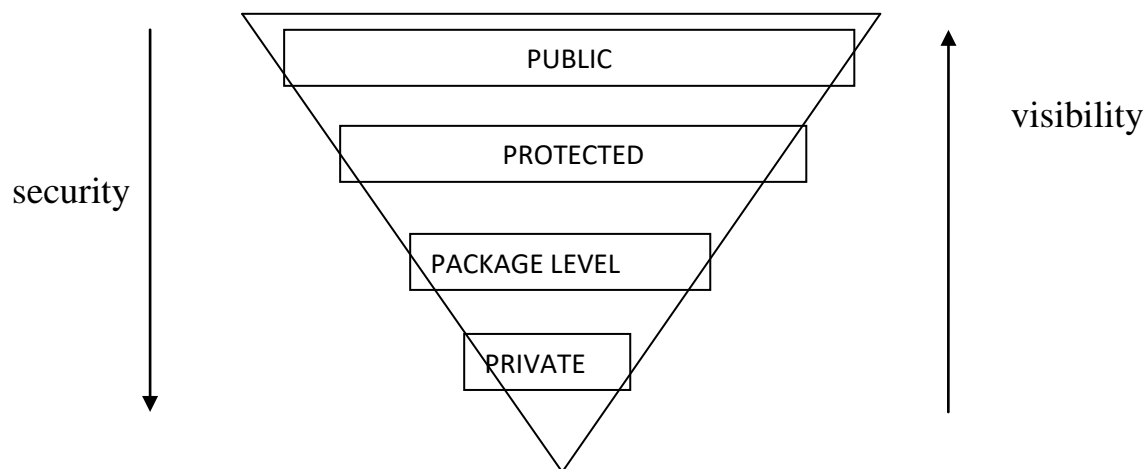
```
}
```

**Op - test() from com.gmail.inbox;**

**test () of com.gmail.sent;**

- If we have two packages with same name and same classname, in this case we have to use both import statement and fully qualified classname together. If we only use two import statement then it show a compile time error.

### 13. ACCESS SPECIFIER



- Access specifier are use to provide security for the classes and its members by controlling visibility.

#### **Public :**

- If you declare any entity with public access specifier then it can be accessed from the classes present in same package or different package.
- Public classes and its members will have highest visilibty and lowest security.
- **Used in same package .**

```
package com.jspider.pkg1;
public class Demo
{
    public int v1=100;
    public void test()
    {
        System.out.println( "this is public void test()");
    }
}
```

## 2<sup>nd</sup> Program which use the first package

package com.jspider.pkg1; // if we use public datamembers or methods in same package and in different class then there is no need of using any import statement or fully qualified classname. We can directly use

```
public class Sample {  
    public static void main(String[] args) {  
        Demo d = new Demo();  
        System.out.println("v1 = " + d.v1);  
        d.test();  
    }  
}
```

**Op- v1 = 100**

**this is public void test()**

### - Used in different package

- (i) It is mandatory to use import or fully qualified classname when we want to access the members and methods of different packages. If we don't do it, then it throw a compile time error.

```
package com.jspider.pkg2;  
import com.jspider.pkg1.Demo;  
public class Run {  
    public static void main (String ar[])  
    {  
        Demo d1 = new Demo();  
        d1.test();  
        System.out.println("v1 = " + d1.v1);  
    }  
}
```

**Op- v1 = 100**

**this is public void test()**



### **Protected :**

- If you declare any entity with protected access specifier then it can be accessed from the classes present in the same package.
- It is partially visible to another packages.
- **Used in same packages**

#### **Program 1**

```
package com.jspider.pkg3;
public class Demo1 {
    protected int a = 10;
    protected void test()
    {
        System.out.println("protected test() from com.jspider.pkg3; ");
    }
}
```

#### **Program 2**

```
package com.jspider.pkg3;
public class Sample1 {

    public static void main(String[] args) {
        Demo1 d1= new Demo1();
        System.out.println("v1 = " +d1.a);
        d1.test();
    }
}
```

**Op - v1 = 10**

**protected test() from com.jspider.pkg3;**

- **Used in different package without inheritance**

```
package com.jspider.pkg4;
import com.jspider.pkg3.Demo1;
public class Run1 {
    public static void main(String[] args) {
        System.out.println("a = " + d1.a); /throw an error
        d1.test(); // here it throw an error because protected
members and methods can't not be accessed directly by another
packages. }}
```

- The class present in different packages can accessed protected members only by inheritance and creating the object of subclass.

### Program

```
package com.jspider.pkg4;
import com.jspider.pkg3.Demo1;
public class Run1 extends Demo1{
    public static void main(String[] args) {
        Run1 d1= new Run1();
        System.out.println("a = " + d1.a);
        d1.test();
    }
}
```

**Op - a = 10**

**protected test() from com.jspider.pkg3;**

### Package-level :

- If you declare any entity without any access specifier keyword then those members are considered as package level members.
- Package level members can be accessed only by classes present in same packages.
- **Used in same packages**

### Program

```
package com.jspider.pkglevel;
public class PkgLevel1 {
    int v=20;
    void test()
    {
        System.out.println("this is pkglevel test()");
    }
}
```

### Program 2

```
package com.jspider.pkglevel;
public class Pkglevel2 {
    public static void main(String[] args) {
        System.out.println("a = " +PkgLevel1.v);
        PkgLevel1.test();
    } } op a = 20
this is pkglevel test()
```

- **Used in different packages**

Program

```
package com.jspider.pkglevel2;
import com.jspider.pkglevel.PkgLevel1;
public class Run2 {
    public static void main(String[] args) {
        PkgLevel1 pk = new PkgLevel1();
        System.out.println("a = " + PkgLevel1.a);
        PkgLevel1.test();
    }
}
```

**// in this program it throw an error because a datamember or methods defined with package-level specifier can not be accessed from different packages.**

## **Private**

- If you declare any entity as private specifier then it can be accessed only within the class in which the members are declared.
- Compared to other access specifier private members will have highest security and lowest visibility.
- **Used in same class**

Program

```
package com.jspider.privates;
public class Demo1 {
    private int a = 20;
    private void test()
    {
        System.out.println("a = " + a);
    }
    public static void main(String[] args) {
        Demo1 d= new Demo1();
        d.test();
    }
}
```

**Op – a = 20**

- **Used in same package in different class**

Program

```
package com.jspider.privates;  
public class Sample1 {  
    public static void main(String[] args) {  
  
        Demo1 d = new Demo1();  
        System.out.println("a = " +a);  
        a.test();  
    }  
}
```

**} // here it throw a compile time error, because private members and methods of a class can not accessed in another class in same packages also.**

- **Used in different packages**

Program

```
package com.jspider.privates2;  
import com.jspider.privates.Demo1;  
public class Run3 {  
    public static void main(String[] args) {  
        Demo1 d = new Demo1();  
        System.out.println("a = " +a);  
        a.test();  
    }  
}
```

**// here it throw a compile time error, because private members and methods of a class can not accessed in another class in different packages also.**

- The subclass can not reduce the visibility of inherited method from superclass. the superclass can only increase the visibility of inherited method from superclass.

superclass	subclass
public	public
protected	public protected
pkg-level	pkg-level public protected
private	can't be inherited

**(a) if try to reduce the visibility then it throw an error**

### Program

```

package com.jspider.pkg3;
class Base
{
    public void Base()
    {
        System.out.println("this is const");
    }
}
class Child extends Base
{
    protected void Base() // here it throw an error
    {
        System.out.println("this is overriding");
    }
}
public class Demo1 {
    public static void main(String[] args) {

        Child b = new Child();
        b.Base();
    }
}

```

**(b)to increase the visilibyt of method**

```
package com.jspider.pkg3;
class Base
{
    void Base()
    {
        System.out.println("this is const");
    }
}
class Child extends Base
{
    protected void Base()
    {
        System.out.println("this is overriding");
    }
}
public class Demo1 {
    public static void main(String[] args) {

        Child b = new Child();
        b.Base();
    }
}
```

**op – this is overriding**

## 14. ENCAPSULATION

- Providing the security for data members and declaring them as private and provided access through getters and setters (public method) is called as encapsulation.

### Program

```
class Account
{
    private int actno = 1234; // data members are private
    private double bal;
}
```

- Encapsulation is used to provide security for the data members against invalid data.
  - (i) **It means that in a scenario where we want to enter the amount in bank account. So it can not be in negative value. To overcome this challenge encapsulation helps**
- The set method is used to **initialize or re-initialize** the data members. Usually set methods do not return any value and hence the return type of the set method is usually void.

### Program

```
public void setBal(double amt)
{
    if(amt>0)
    {
        bal=bal+amt; // initialize and re-initialize data members
    }
    else
    {
        System.out.println("invalid amount");
    }
}
```

- getMethod should return the value of the data members. The return type of get method depends on datatype of the data members whose value is returned from getMethod.

### Program

```
public double getBal(int actno)
{
    if(actno==1234)
    {
```

```

        return bal;
    }
    else
    {
        return -777;
    }
}

```

- Using encapsulation the control over the data members of the class remains in the same class.

### Program

```

package com.jsp.encapsulation;

class Account
{
    private int actno = 1234;
    private double bal;
    public void setBal(double amt)
    {
        if(amt>0)
        {
            bal=bal+amt;
        }
        else
        {
            System.out.println("invalid amount");
        }
    }
    public double getBal(int actno)
    {
        if(actno==1234)
        {
            return bal;
        }
        else
        {
            return -777;
        }
    }
    public int showAcnt(int pin)
    {
        if(pin==1234)

```



```

        {
            return actno;
        }
        else
        {
            return -777;
        }
    }
}

public class Mainclass {

    public static void main(String[] args) {
        Account a = new Account();
        System.out.println("For valid value");
        a.setBal(2000);
        double d1= a.getBal(1234);
        System.out.println("balance = "+d1);
        int i = a.showAcnt(1234);
        System.out.println("Acount no = " +i);
        System.out.println("-----");
        System.out.println("For invalid value");
        a.setBal(-2000);
        double d2= a.getBal(12345);
        System.out.println("balance = "+d2);
        int i1 = a.showAcnt(12345);
        System.out.println("Acount no = " +i1);
    }
}
Op-

```

```

For valid value
balance = 2000.0
Acount no = 1234
-----
For invalid value
invalid amount
balance = -777.0
Acount no = -777

```

## 15. NESTED CLASS

- A class which is written within the body of another class is called as nested class.

### Program

```
class Outerclass
{
    class Nested // class inside a class
    {
    }
}
```

- Nested class are of two types :  
(A)Static Nested Class  
(B)Inner class
- A nested class which is declared with static keyword is called as static nested class.

### Program

```
class Outerclass
{
    static class Nested // static nested class
    {
    }
}
```

- A nested class which is declared without static keyword is called as inner class.

### Program

```
class Outerclass
{
    class InnerClass
    {
    }
}
```

### Static Nested Class

- If you want to access the members of static nested classes or you want to create object of static nested class then there are two approaches :
  - (i) Using fully qualified classname
  - (ii) Using import statement

(i) by using fully qualified classname

### Program

```
package com.jsp.innerclass;
class OuterClass
{
    static int v1 = 10;
    static class StaticNestedClass
    {
        static double z1 = 2.142;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("V1 = "+v1);
            System.out.println("z1 = " +z1);
        }
    }
}

public class Mainclass1 {

    public static void main(String[] args) {

        System.out.println(com.jsp.innerclass.OuterClass.StaticNestedClass.z1);
        // 1st execution from here 2.142

        com.jsp.innerclass.OuterClass.StaticNestedClass ref1 =
        new com.jsp.innerclass.OuterClass.StaticNestedClass();
        ref1.count(); //print all the statement
    }
} op –
```

**2.142**  
**this is count() of StaticNestedClass**  
**V1 = 10**  
**z1 = 2.142**

(ii) by using import statement

### Program

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static int v1 = 10;
    static class StaticNestedClass
    {
        static double z1 = 2.142;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("V1 = "+v1);
            System.out.println("z1 = " +z1);
        }
    }
}

public class Mainclass1 {

    public static void main(String[] args) {

        System.out.println(StaticNestedClass.z1);

        StaticNestedClass ref1 = new StaticNestedClass();
        ref1.count();
    }
}
```

op –

**2.142**

**this is count() of StaticNestedClass**

**V1 = 10**

**z1 = 2.142**

- The outerclass behaves like package for static nested class.

### Program

```
com.jsp.innerclass.OuterClass.StaticNestedClass;
```

- Members of static nested class can be of two types.
  - static members
  - non- static members

### Program

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static class StaticNestedClass
    {
        static double z1 = 2.142; //static data members
        int a = 20; // non static data members
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("a = "+a);
            System.out.println("z1 = " +z1);
        }
    }
}

public class Mainclass1 {

    public static void main(String[] args) {
        StaticNestedClass ref1 = new StaticNestedClass();
        ref1.count();
    }
}

op - this is count() of StaticNestedClass
V1 = 10
z1 = 2.142
```

- The static members of static nested class can be accessed by the classname.
- The non-static members of static nested class can be accessed by creating the object.

### Program

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static class StaticNestedClass
    {
        static double z1 = 2.142;
        int a = 20;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("a = "+a);
            System.out.println("z1 = " +z1);
        }
    }
}

public class Mainclass1 {
    public static void main(String[] args) {
        StaticNestedClass ref1 = new StaticNestedClass();
        ref1.count(); // calling the non-static method using object
        System.out.println("static data members " +StaticNestedClass.z1);
        //calling the static data members
        System.out.println(ref1.a); // calling the non-static data members
    }
}
```

**op-**

```
this is count() of StaticNestedClass
V1 = 10
z1 = 2.142
static data members 2.142
10 // why it throw exception
```

- A static nested class can access all the properties of outerclass directly and the outerclass can access all the properties of static nested class by using classname and creating the object.
- (i) **A static nested class can access all the properties of outerclass directly**

### Program

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static int v1 = 10; //static data members
    int b=20; // non static data members
    static class StaticNestedClass
    {
        static double z1 = 2.142;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("V1 = "+v1); // directly used static v1 of outerclass
            System.out.println("z1 = " +z1);
            System.out.println("b = " +new OuterClass().b); //using non-static b of
            outerclass by creating object
        }
    }
}

public class Mainclass {
    public static void main(String[] args) {
        StaticNestedClass ref1 = new StaticNestedClass();
        ref1.count();
    }
}

op-
this is count() of StaticNestedClass
V1 = 10
z1 = 2.142
b = 20
```

- (ii) **outerclass can access all the properties of static nested class by using classname and creating the object.**

**Program**

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static int v1 = 10;
    int b=20;
    public static void test()
    {
        System.out.println("this is test() of outerclass");
        System.out.println("V1 = "+v1);
        System.out.println("z1 = " + StaticNestedClass.z1); //static data
        StaticNestedClass sn1 = new StaticNestedClass();
        sn1.count(); // non static methods of staticNestedClass accessed
        by using object by outerclass }
    static class StaticNestedClass
    {
        static double z1 = 2.142;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
            System.out.println("z1 = " +z1);
        }
    }
}
public class Mainclass {

    public static void main(String[] args) {
        OuterClass.test();
    }
}
```

op –

**this is test() of outerclass**  
**V1 = 10**  
**z1 = 2.142**  
**this is count() of StaticNestedClass**  
**z1 = 2.142**



- Using outerclass object we can not access the properties of static nested class.
- Using static nested class object, it is not possible to access the properties of outerclass.

### Program

```
package com.jsp.innerclass;
import com.jsp.innerclass.OuterClass.StaticNestedClass;
class OuterClass
{
    static int v1 = 10;
    int b=20;
    public static void test()
    {
        System.out.println("this is test() of outerclass");
    }

    static class StaticNestedClass
    {
        double z1 = 2.142;
        public void count()
        {
            System.out.println("this is count() of StaticNestedClass");
        }
    }
}

public class Mainclass {
    public static void main(String[] args) {
        StaticNestedClass ref1 = new StaticNestedClass();
        ref1.test(); // here it throw an error
        System.out.println(ref1.b); // here it throw an error

        OuterClass ref2 = new OuterClass();
        ref2.count(); // here it throw an error
        System.out.println(ref2.z1); // here it throw an error
    }
}
```

## Innerclass

- A nested class which is declared without using static keyword is called as innerclass.

## Program

```
class Outerclass
{
    class InnerClass
    {
    }
}
```

- The datamembers of innerclass can be declared only non-static can not be declared as static.

## Program

```
class Outerclass
{
    class Innerclass
    {
        int a = 20;
        static int b=30; // here it throw an error
    }
}
```

- If you want to declare any static data members within the innerclass then it must be declared as final.

## Program

```
class Outerclass
{
    class Innerclass
    {
        int a = 20;
        static final int b=30; // here it don't throw an error
    }
}
```

- The function member of innerclass can never be declared as static.

## Program

```
class Outerclass
{
    class Innerclass
```

```

{
    public void disp()
    {
    }
    public static void test() // here it throw an error
    {
    }
}

```

- The outerclass and innerclass can accessed each other property.
- The outerclass can access innerclass static member using classname and non-static members by creating the object.

### Program

```

package com.jsp.innerclass;
class SubOuterClass
{
    int b = 30;
    public void count()
    {
        System.out.println("b = "+b);
        System.out.println("b = "+Innerclass.c); // calling the static data
members of innerclass by using classname
        Innerclass i = new Innerclass();
        System.out.println("a = " + i.a);
        i.disp(); // calling the non-static method of innerclass by creating the
object of innerclass
    }
    class Innerclass
    {
        int a = 20;
        static final int c=50;
        public void disp()
        {
            System.out.println("this is innerclass disp");
            System.out.println("a = "+a);
            System.out.println("c = "+c);
        }
    }
}

```

```

public class Sample {

    public static void main(String[] args) {

        SubOuterClass ref1 = new SubOuterClass();
        ref1.count();
    }
}

```

```

op
b = 30
b = 50
a = 20
this is innerclass disp
a = 20
c = 50

```

- The innerclass object can be created only after creating the object of outerclass because the constructor of inner class can be accessed only through the object of outerclass.
- To create the object of innerclass or to accessed static members of innerclass we have two approaches.
  - (i) fully qualified classname
  - (ii) using import statement

### **fully qualified classname**

### **Using import statement**

#### **Program**

```

package com.jsp.innerclass;
import com.jsp.innerclass.SubOuterClass.Innerclass;
class SubOuterClass
{
    int b = 30;
    class Innerclass
    {
        int a = 20;
        static final int c=50;
        public void disp()
        {
            System.out.println("this is innerclass disp");
        }
    }
}

```

```

        System.out.println("a = "+a);
        System.out.println("c = "+c);
        System.out.println("b = " +b);
    }
    }}
public class Sample
{
    public static void main(String[] args) {
        SubOuterClass ref1 = new SubOuterClass(); // creating the object
of outerclass
        Innerclass ref2 = ref1.new Innerclass(); // by using the reference
variable of outerclass create the object of innerclass
        ref2.disp();
    } } op –
        this is innerclass disp
        a = 20
        c = 50
        b = 30

```

## Anonymous Inner Class

- It is a type of innerclass while exists or created without any name.
- The name of anonymous innerclass is decided by the compiler at the compiler time.
- The instance or object of anonymous innerclass will be created only once at the runtime.

## Program

```
package com.jspider.pkg1;
class Superclass
{
    public void count(int num)
    {
        for (int i = 1; i<=num; i++)
        {
            System.out.println(i);
        }
    }
}
public class Mainclass4 {
    public Mainclass4() {
        Superclass ref1= new Superclass()
        {
            int d1=13;
            @Override
            public void count(int num)
            {
                for (int i=num; i>=1;i--)
                {
                    System.out.print(i);
                }
            }
        };
        ref1.count(5);
    }
}
```

**op- 5 4 3 2 1**

## Using interface

### Program

```
package com.jspider.pkg1;
interface Run
{
    public void disp();
}
public class Anonymous {
    public static void main (String ar[])
    {
        Run r = new Run()
        {
            @Override
            public void disp()
            {
                System.out.println("hi");
            }
        };
        r.disp();}} op – hi
```

## Using interface

### Program

```
package com.jspider.pkg1;
interface Run
{
    public void disp(); }
public class Anonymous {
    public static void main (String ar[])
    {
        new Run()
        { @Override
        public void disp()
        {
            System.out.println("hi");
        }
        }.disp();    }} op - hi
```

## JDK 1.8-9 FEATURES

### Interface Enhancement

- As part of JDK 1.8 enhancement interfaces supports concrete methods.
- The concrete methods of interface can be of only two types :
  - (i) static
  - (ii) Default

### Program

```
interface Run
{
    public static void disp()
    {
    }
    public default void test()
    {
    }
}

public class Sample {
    public static void main(String[] args)
    {
    }
}
```

- we can call static methods of interfaces by using interface name with dot operator.

### Program

```
package com.jspider.jdk8;

interface Run
{
    public static void disp()
    {
        System.out.println("this is disp");
    }
    public default void test()
    {
    }
}

public class Sample {
    public static void main(String[] args) {
        Run.disp(); // to call the static method
    }
    op – this is disp
}
```



- Static methods of interface can not be accessed from the object of implementation class.
- To access default method of interface firstly any class should have override all the method of interface. By creating the implementation class object only we can accessed the default method of interface.

### Program

```
package com.jspider.jdk8;
interface Run
{
    public static void disp()
    {
        System.out.println("this is disp");
    }
    public default void test()
    {
        System.out.println("this is test");
    }
}
class Demo implements Run
{
}
public class Sample {

    public static void main(String[] args) {
        Run.disp();
        Demo d = new Demo();
        d.test(); access the default method of interface by object
        d.disp(); // it throw an error because we can not access the
static method of interface by using object.
    }}
}
```

- If a class is implementing two interfaces and if both interfaces contains same abstract method with same name and arguments then the implementation class should override the abstract method only once.

### Program

```
package com.jspider.jdk8;
interface Run
{
    public void disp();
}
interface Run1
{
    public void disp();
}
class Demo implements Run, Run1
{
    @Override
    public void disp()
    {
        System.out.println("this is overriding method");
    }
}
public class Sample {

    public static void main(String[] args) {
        Demo d = new Demo();
        d.disp();
    }
} op –this is overriding method
```

- If a class implements two interfaces which contains same default method which have same name and same arguments then the implementation class should override the default method to resolve the ambiguity.

### Program

**same as it last program**

- if you want to call the super interface implementation of a default method then you can call the method with following syntax.

syntax :

```
interfacename.super.methodname;
```

Program

```
package practice;
```

```
interface Run
```

```
{
```

```
    public default void disp()
```

```
    {
```

```
        System.out.println("hello");
```

```
    }
```

```
}
```

```
interface Run1
```

```
{
```

```
    public default void disp()
```

```
    {
```

```
        System.out.println("hi");
```

```
    }
```

```
}
```

```
class C implements Run, Run1
```

```
{
```

```
    public void disp()
```

```
    {
```

```
        Run.super.disp(); // calling the super interface
```

```
        Run1.super.disp();
```

```
    }}
```

```
    public class Test {
```

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

```
            C c = new C ();
```

```
            c.disp();
```

```
        }} op – hello
```

```
            hi
```

- If a class implements two interfaces which contains same static methods which have same name and arguments then we can differentiate with the help of interface name to call those methods.

### Program

```
package practice;
interface Run
{
    public static void disp()
    {
        System.out.println("hello");
    }
}
interface Run1
{
    public static void disp()
    {
        System.out.println("hi");
    }
}
public class Test {
    public static void main(String[] args) {
        Run.disp();
        Run1.disp(); // calling static method of interface

    }
}
```

- we can override the **static method of interface**. When calling from object of implementation class call the method.

### FUNCTIONAL PROGRAMMING

- Passing a function as the argument to another function is called as functional programming.
- functional programming in java can be done only with the help of functional interface.

## LAMBDA FUNCTION/EXPRESSION

- It is an anonymous function which written to give the implementation for abstract method class present in functional interface.

- Syntax :

```
interface_name refvar = (arglist) ->
{   };
```

**-> - it is called as lambda operator**

### Program

```
package practice;
interface Run
{
    public void disp();
}
public class Test {
    public static void main(String[] args) {
        Run r = ()->
        {
            System.out.println("this is lambda function");
        };
        r.disp();
    } } op – this is lambda function
```

### Lambda Function with return value

#### Program

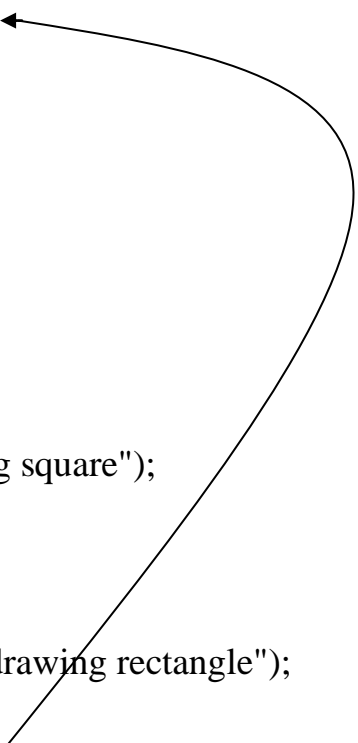
```
package practice;
interface Run
{
    public int add(int a, int b);
}
public class Test {
    public static void main(String[] args) {
        Run r = (int a, int b)-> //passing the arguments
        {
            int c= a+b;
            return c;
        };
        int res = r.add(10,20); // return a value and save it in a variable
        System.out.println("res = "+res);    } }
```

**op – res = 30**

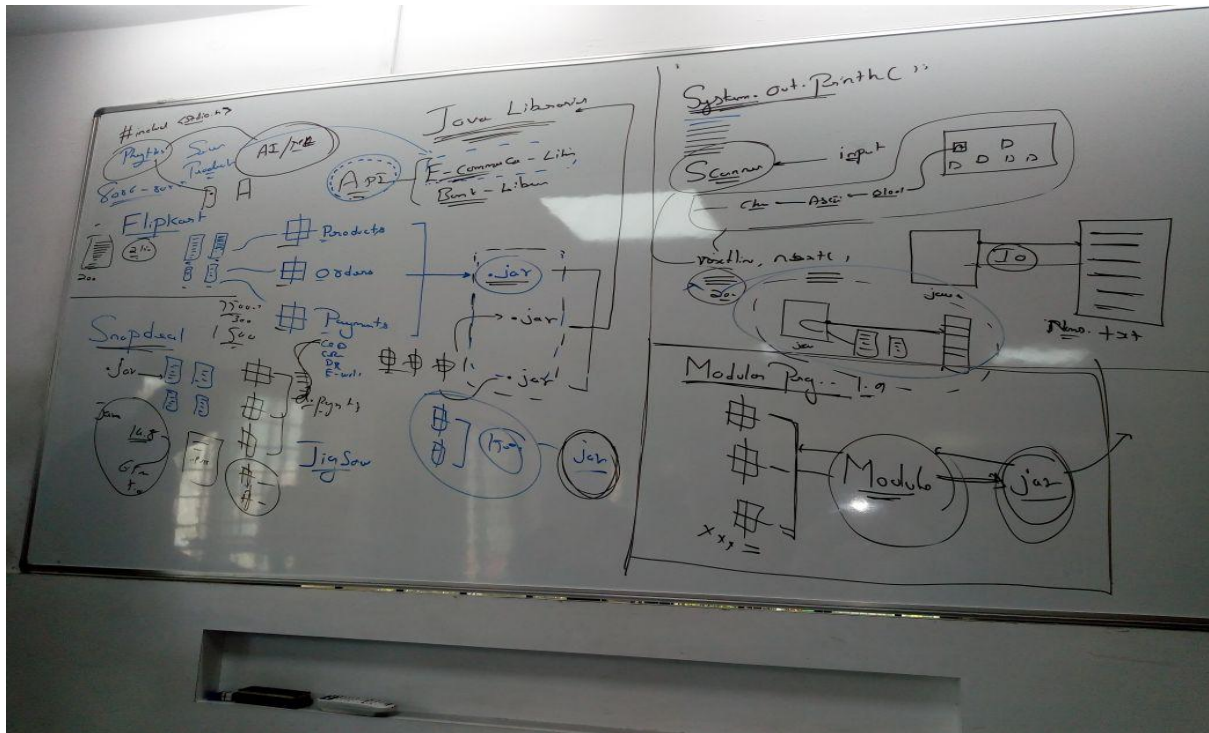
## Functional Programming with Lambda

```
package practice;
interface Run
{
    public void draw();
}
class ShapeDrawer
{
    public static void drawAnything(Run r)
    {
        r.draw();
    }
}
public class Test {
    public static void main(String[] args) {
        Run r = ()->
        {
            System.out.println("drawing square");
        };
        Run r1 = ()->
        {
            System.out.println("drawing rectangle");
        };
        ShapeDrawer.drawAnything(r); // passing the reference
        ShapeDrawer.drawAnything(r1);
    }
}
```

**op – drawing square**  
**drawing rectangle**



# JAVA LIBRARY



- It is a group of jar file(API) which consists of classes which are used to perform basic or advance level operation in java program
- As part of JDK-9 java introduces modular programming.

## Module

- A module is logical group of packages related to one jar file.
- By using modular we can eliminate the unnecessary packages from the jar files.
- The modular programming JDK-9 was introduced as part of **Jigsaw project**

## **Inbuilt Packages**

### **Java.lang.package**

- java lang package contains the classes and interfaces which are required to write simple java program or to develop complicated java application

### **Object Class**

- It is the supermost class in java
- Every class directly or indirectly inherits from object class.
- Object class is present in java.lang.package.
- Every methods of Object class is non-static.

#### **Methods of Object Class :**

- |                             |   |         |        |
|-----------------------------|---|---------|--------|
| - hashCode()                | - | int     |        |
| - toString()                | - | String  |        |
| - equals(Object ref)        | - | Boolean |        |
| - wait()                    | - | void    | -final |
| - wait(long mills)          | - | void    | final  |
| - wait(long mills, int num) | - | void    | final  |
| - notify()                  | - | void    | final  |
| - notifyAll()               | - | void    | final  |
| - finalize()                | - | void    |        |
| - clone()                   | - | Object  |        |
| - getClass()                | - | Object  | final  |

#### **hashCode()**

- This method returns hashCode value of the given object
- hashCode value is an unique integer value which is generated based on the address of the object.

#### **Program**

```
package com.inbuiltpackages.library;
public class Mainclass {
    public static void main(String[] args) {
        Object obj1 = new Object();
        Object obj2 = new Object();
        int h1=obj1.hashCode();
        int h2 = obj2.hashCode();
        System.out.println("h1 = " +h1);
        System.out.println("h2 = "+h2);
    }
}
```



**}} - op - h1 = 1603195447  
h2 = 792791759**

**toString() –**

- This methods returns string representation of the object.
- The string representation of object contains :
  - (i) fully qualified classname
  - (ii) @ character
  - (iii) hexadecimal value of hashCode.

**syntax**

fullyqualifiedclassname@hexadecimal of hashCode

eg: java.lang.Object@4352625

**Program**

```
package com.inbuiltpackages.library;
```

```
public class Mainclass {
```

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

```
        Object obj1 = new Object();
```

```
        String str = obj1.toString();
```

```
        System.out.println("str = " +str);
```

```
        System.out.println(obj1);
```

```
    }
```

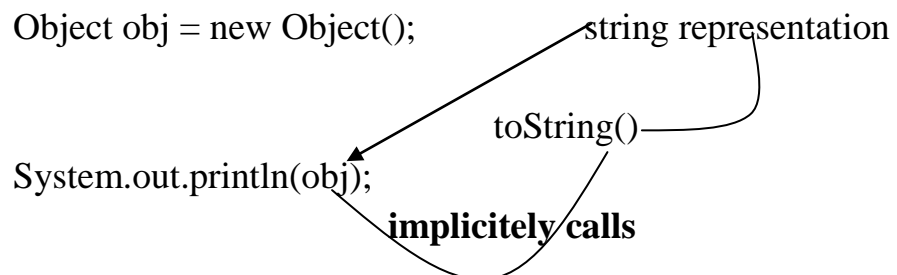
```
}
```

**op - str = java.lang.Object@5f8ed237**

- **Question :** What happens if you try to print a reference variables?
- **Ans :** println() methods implicitly calls toString() method and prints the returned string values.

```
Object obj = new Object();
```

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



## Program

```
package com.inbuiltpackages.library;
public class Mainclass {
    public static void main(String[] args) {
        Object obj1 = new Object();
        Object obj2 = new Object();
        System.out.println("obj1 = " +obj1);
        System.out.println("obj2 = " +obj2);
    }
}
```

**op - obj1 = java.lang.Object@5f8ed237**

**obj2 = java.lang.Object@2f410acf**

## **equals(Object ref)**

- This method compares hashCode value of given two object and returns true if they are equals else it returns false it depends on hashCode.

## Program

```
package com.inbuiltpackages.library;

public class Mainclass {

    public static void main(String[] args) {

        Object obj1 = new Object();
        Object obj2 = new Object();
        boolean b1 = obj1.equals(obj2);
        System.out.println(b1);
        boolean b2 = obj1.equals(obj1);
        System.out.println(b2);
    }
}
```

## STRING CLASS (JDK 1.0)

- String is a class which is used to store string values within the string object.
- Note : String value are internally stored in character Array.
- String class is immediate subclass of Object class.

### Program

```
java.lang.Object
java.lang.String
```

- String class is final class and it can not be inherited.
- String class implements :
  - (i) Serializable
  - (ii) comparable
  - (iii) charSequence interfaces.

### Program

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

- **hashCode()** method of object class is overridden in String class which returns an integer value i.e. generated based on ASCII value of characters present in given String values.

### Program

```
package com.jsp.string;
public class Mainclass {
    public static void main(String[] args) {
        String str = new String("hello");
        String str2 = new String("hello");
        int h1 = str.hashCode();
        int h2 = str.hashCode();
        System.out.println("h1 = "+h1);
        System.out.println("h1 = "+h2);
    }
}
```

**op –**

**h1 = 99162322**

**h2 = 99162322**

- toString() method of object class is overridden in String class (by the developer of String class) which returns string value present in given string object.

#### Program

```
package com.jsp.string;
public class Mainclass {

    public static void main(String[] args) {

        String str = new String("hello");
        String str2 = new String("hello");
        String s1 = str.toString();
        String s2 = str2.toString();
        System.out.println("s1 = "+s1);
        System.out.println("s2 = "+s2);
        System.out.println("str = "+str);
    }
}
```

**op –**

```
s1 = hello
s2 = hello
str = hello
```

- equals() method of object class is overridden in string class which returns true if both string contains same characters else return false.

#### Program

```
package com.jsp.string;
public class Mainclass {

    public static void main(String[] args) {
        String str = new String("hello");
        String str2 = new String("hello");
        String str3 = new String("hey");
        boolean b1 = str.equals(str2);
        boolean b2 = str2.equals(str3);
        System.out.println("b1 = " + b1);
        System.out.println("b2 = " + b2);
    }
} op -    b1 = true
b2 = false
```

- In java string object can be created in two ways :
  - (i) Using new Operator
  - (ii) Without using new operator
  - (iii) String object can be also created using concatenation Operator.

```
String str = new String("hello");
String s1 = "java";
String s2 = "java" + str;
```

- String object are stored in a special memory area called String pool.
- The string pool has two parts :
  - (i) constant pool
  - (ii) non-constant pool
- The String object which are created using new operator are stored in non-constant pool.
- Within non-constant pool duplicated are allowed.
- String object which are created without using new operator will be stored in constant pool.
- Within constant pool duplicated are **not allowed**

```
var + var;
var + " ";
" " + var;
```
- Using any one of the above combination the resultant string created will be stored in non-constant pool.
- " " + " "; this is stored in constant pool.

## Program

```
package com.jsp.string;
public class Mainclass2 {
    public static void main(String[] args) {

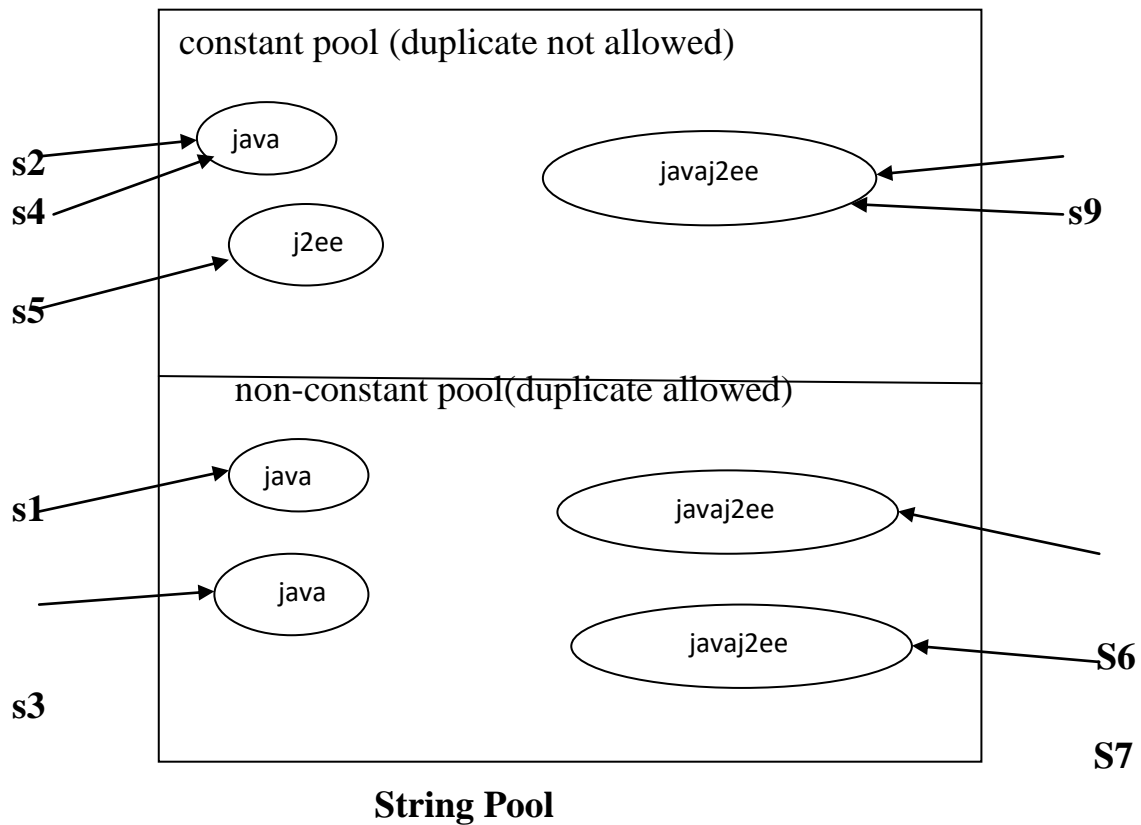
        String str1 = new String("java"); //creating string object
        String str2 = " java";           //creating string object
        String str3 = "java"+str2;       //creating string object

        String s1 = new String("java");
        String s2 = "java";
        String s3 = new String("java");
```

```

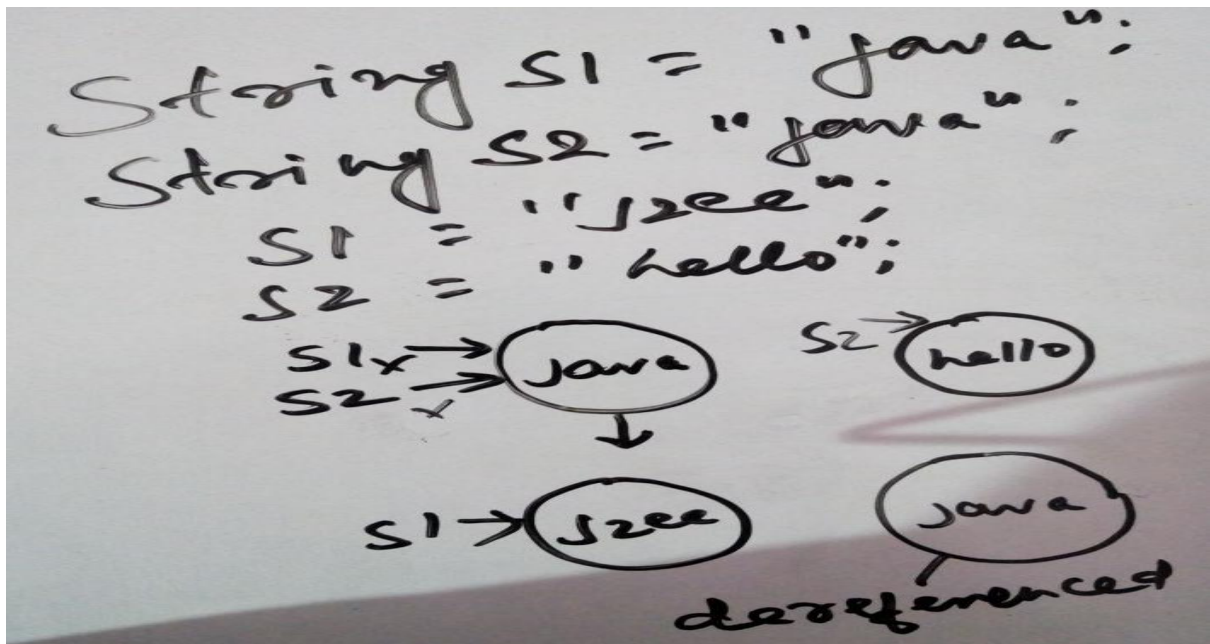
String s4= "java";
String s5 = "j2ee";
String s6= s1+s5;
String s7 = "java"+s5;
String s8 = "java" + "j2ee";
String s9 = "javaj2ee";
System.out.println(s1==s2); //false
System.out.println(s1==s3); //false
System.out.println(s2==s4); //true
System.out.println(s6==s7); // false
System.out.println(s8==s9); // true
    }
}

```



### Question : String class is immutable class explain ?

- if you try to change the existing string object, instead of changing the given object a new object will be created with the changes. The reference variable stops pointing to new object.
- The given object now becomes dereferenced.
- only the object is created without using new keyword is immutable.



### Advantages :

- If you referenced variables pointing to same string object then changes done from one reference variable will not effect other reference variables.

### Disadvantages :

- The dereferenced object will be present in the heap memory for the future use but these referenced object if not controlled results in **OutOfMemoryError**

### Program

```
package com.jsp.string;  
public class StringBuffer1 {  
    public static String firstHalf(String str)  
    {  
        int mid = (str.length()-1)/2;  
        String res=" ";  
        for(int i =0; i<=mid; i++)  
        {
```

```

        res = res+str.charAt(i);
        System.out.println(res);
        System.out.println(i);
    }
    return res;
}
public static void main(String[] args)
{
    String s1 = new String("aaaaaabbmbmbmbmbmbmbmb");
    firstHalf(s1);
}

```

**// here 8 dereferenced object are created.**

- To overcome disadvantages of immutable property of string class java introduced two new classes.
  - (i) StringBuffer
  - (ii) StringBuilder

## **String Function**

### **Program**

```
package strings;
```

```

public class Mainclass3 {

    public static void main(String[] args) {
        String a = "hello";
        String b = "hello";
        String c = new String("hello");
        String d = "HELLO";
        String e = "Jspider java class";

        System.out.println("length = "+a.length()); //5
        System.out.println(a==b); // true
        System.out.println(a==c); // false
        System.out.println(a.equals(b)); // true
        System.out.println(a.equals(c)); //true
        System.out.println(a.equals(d)); // false
        System.out.println(a.equalsIgnoreCase(d)); //true
    }
}

```



```
System.out.println(a.charAt(2)); //1
System.out.println(a.concat(" " +b)); //hello hello
System.out.println(a+" " + " "+ b); //hello hello
```

```
System.out.println(a.indexOf("l")); //2
System.out.println(a.indexOf("l", 1)); //2
System.out.println(a.indexOf("l", 3)); //3
```

```
System.out.println(a.lastIndexOf("l")); //3
```

```
System.out.println(a.toUpperCase()); //HELLO
System.out.println(d.toLowerCase()); //hello
```

```
System.out.println(a.substring(3)); //lo
System.out.println(e.substring(2,10)); //pider ja
```

```
System.out.println(a.compareTo(d)); // 32
System.out.println(a.compareToIgnoreCase(b)); //0
```

```
String f = " hello ";
System.out.println(f.trim()); //hello
```

```
System.out.println(a.replace("h", "w")); //wello
```

```
}}
```

### **StringBuffer(JDK 1.0 )**

- It is the immediate subclass of object class.
- StringBuffer class is present in java.lang.package
- StringBuffer implements serializable, appendable and charSequence interface.
- StringBuffer class is declared as final and it can not be inherited.
- StringBuffer is mutable class.
- StringBuffer objects can be created only by using new operator.
- StringBuffer is thread-safe class.
- Every method of StringBuffer is synchronized.

## StringBuffer Function

```
package strings;
public class Mainclass4 {
    public static void main(String[] args) {

        StringBuffer s = new StringBuffer("javayou");
        System.out.println(s.length()); //7
        System.out.println(s.capacity()); //23

        s.append("friend");
        System.out.println(s); //javayoufriend

        s.insert(4, "for");
        System.out.println(s); //javaforyoufriend

        char c [] = {'a', 'a', 't', 'i', 'f'};
        s.insert(16, c);
        System.out.println(s); //javaforyoufriendaatif

        System.out.println(s.charAt(2)); //v

        StringBuffer sb = new StringBuffer("javaworld");
        sb.delete(3, 5);
        System.out.println(sb); //javorld

        StringBuffer sb1 = new StringBuffer("javaworld");
        sb1.deleteCharAt(4);

        System.out.println(sb1); //javaorld

        StringBuffer sb2 = new StringBuffer("java");
        System.out.println(sb2.reverse()); //avaj

    }
}
```

**toString()** of object class is overridden in StringBuffer class which returns the string value present in the given StringBuffer Object.

**//inheriting property of object class**

```
package com.jsp.string;
public class Mainclass4 {
    public static void main(String[] args) {
        StringBuffer sb1 = new StringBuffer("hello");
        StringBuffer sb2 = new StringBuffer("hello");
        int h1 = sb1.hashCode(); // not overridden
        int h2 = sb2.hashCode();
        System.out.println("h1 = "+h1);
        System.out.println("h2 = "+h2);
        String str = sb1.toString(); // overridden
        System.out.println(sb1);
        boolean b1 = sb1.equals(sb2); // not overridden
        System.out.println(b1);
    }
}
```

**op –**

**h1 = 792791759**

**h2 = 1191747167**

**hello**

**false**

### **StringBuilder (jdk 1.5)**

- StringBuilder is not thread safe class.
- The methods of StringBuilder class are not synchronize.
- It is the immediate subclass of object class.
- StringBuilder class is present in java.lang.package
- StringBuilder implements serializable, appendable and charSequence interface.
- StringBuilder class is declared as final and it can not be inherited.
- StringBuilder is mutable class.
- StringBuilder objects can be created only by using new operator.
- StringBuilder is thread-safe class.
- Every method of StringBuilder is **synchronized**.

**toString()** of object class is overridden in **StringBuilder** class which returns the string value present in the given **StringBuffer** Object.

**//inheriting property of object class**

```
package com.jsp.string;
public class Mainclass4 {
    public static void main(String[] args) {

        StringBuilder sb1 = new StringBuilder("hello");
        StringBuilder sb2 = new StringBuilder("hello");
        int h1 = sb1.hashCode(); // not overridden
        int h2 = sb2.hashCode();
        System.out.println("h1 = "+h1);
        System.out.println("h2 = "+h2);

        String str = sb1.toString(); // overriden
        System.out.println(sb1);

        boolean b1 = sb1.equals(sb2); // not overridden
        System.out.println(b1);
    }
} op -
```

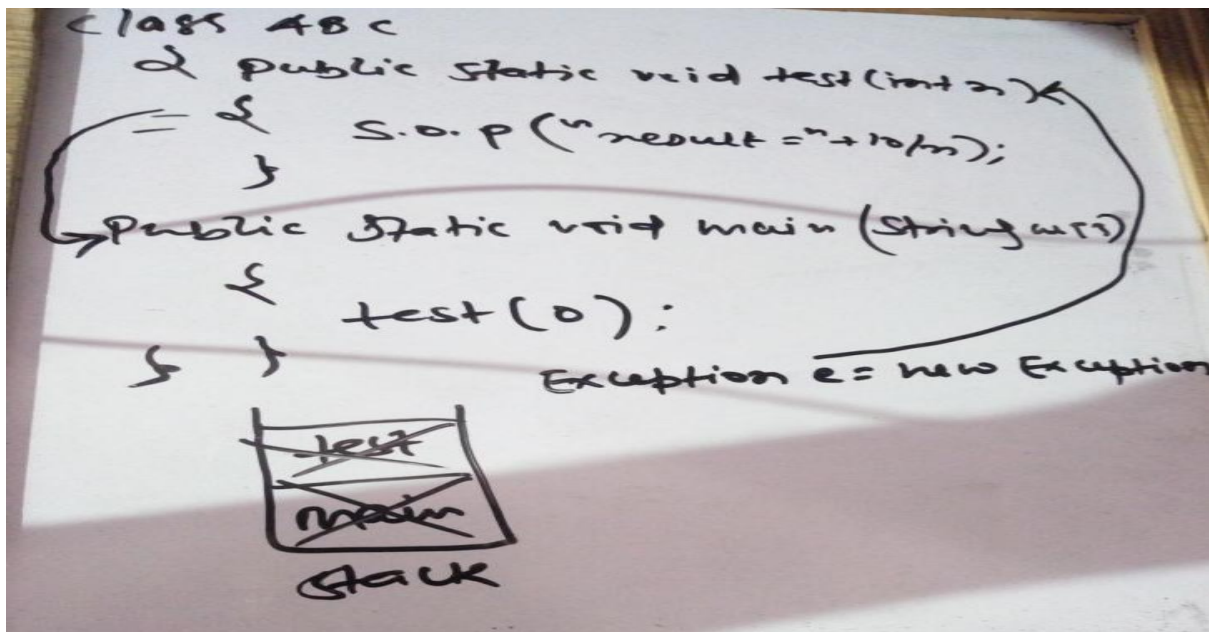
**h1 = 1603195447**  
**h2 = 792791759**  
**hello**  
**false**

**Differences**

<b>String</b>	<b>StringBuffer</b>	<b>StringBuilder</b>
immutable class	Mutable class	Mutable class
Not thread safe class	thread safe class	not thread safe
methods are not synchronized	methods are synchronized	methods are not synchronized
Objects can be created with or without using new operator	objects can be created only by using new Operator	objects can be created only by using new Operator
hashCode(), toString(), equals() of object class are overridden	only toString() of object class is overridden	only toString() of object class is overridden

## EXCEPTION HANDLING

- Exception is an unexpected event which occurs at runtime due to unexpected operations performed by a line of code.
- Exceptions occur only at runtime.
  - (i) checked Exception
  - (ii) Unchecked Exception
- Whenever there is an exception JVM creates the objects of the corresponding exception class.
- JVM will **throw** the exception to the method which created the exception. If the method is not able to handle the exception then JVM will terminate the method and remove it from the stack.
- JVM will now pass the exception object to the calling method and if the calling method is not able to handle the exception objects then it will be terminated and removed from the stack.



- JVM calls default exception handler to handle an unhandled exception.
- **Default Exception handler** will handle the exception and prints three messages on the screen.
  - (i) Name of the exception class
  - (ii) Extra Message
  - (iii) Stack Trace

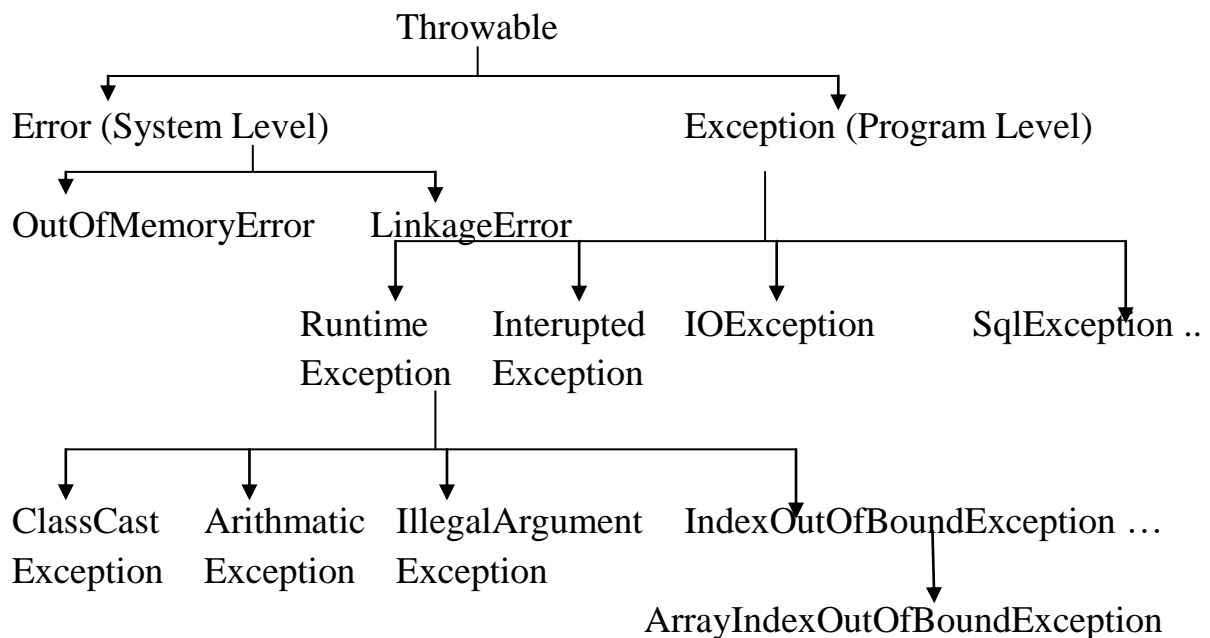
Exception in thread "main" **java.lang.ArithmeticException: / by zero**  
at com.object.exception.Mainclass4.test(Mainclass4.java:7)  
at com.object.exception.Mainclass4.main(Mainclass4.java:12)

## Program

```
package com.object.exception;
public class Mainclass4 {
    public static void test(int n)
    {
        System.out.println(10/n);
    }
    public static void main(String[] args)
    {
        System.out.println("program start");
        test(0);
        System.out.println("program end");
    }
}
```

**op -** Exception in thread "main" **java.lang.ArithmeticException: / by zero**  
at com.object.exception.Mainclass4.test(Mainclass4.java:7)  
at com.object.exception.Mainclass4.main(Mainclass4.java:12)

## Exception Hierarchy



## try and catch block

```
try
{
    risky codes;
}
catch (Exception ref_var)
{
    Alternate codes;
}
```

- try block contains all risky line of codes which may throw an exception.
- catch block contains all the alternate codes that should be executed on the event of exception.

## Program

```
package com.object.exception;
public class Mainclass {
    public static void test(int n)
    {
        System.out.println("enter test()");
        try
        {
            System.out.println("result : "+10/n); // risky line of code
        }
        catch(ArithmeticException ref)
        {
            System.out.println("caught Arithmetic Exception");
            System.out.println("invalid number for division");
            //alternate code
        }
        System.out.println("exit test()");
    }
    public static void main(String[] args) {
        System.out.println("Program start");
        test(0);
        System.out.println("Program ends");
    }
}
```

**op - Program start**

**enter test()**

**caught Arithmetic Exception**

**invalid number for division**

**exit test()**

**Program ends**

- If a line of code throws an exception within the try block then the rest of codes within the try block will be invalid and does not execute.

### Program

```
package com.object.exception;
public class Mainclass6 {
    public static void test(int n)
    {
        System.out.println("enter test");
        try
        {
            System.out.println("result : "+10/n);
            System.out.println("hello"); // this line of code is not printed
        }
        catch(ArithmeticException ref)
        {
            System.out.println("caught Arithmetic Exception");
            System.out.println("invalid number for division");
        }
        System.out.println("exit test()");
    }
    public static void main(String[] args) {

        System.out.println("Program start");
        test(0);
        System.out.println("Program ends");
    }
}
```

op-

```
Program start
enter test
caught Arithmetic Exception
invalid number for division
exit test()
Program ends
```



## try with multiple catch block

- for a single try block we can write any number of catch blocks.

### Program

```
package com.object.exception;
public class Mainclass {
    public static void test(int n)
    {
        int a1 [] = {10,20,30};
        System.out.println("enter test");
        try // one try block
        {
            System.out.println("result : "+10/n);
            System.out.println("array element : "+a1[n]);
        }
        catch(ArithmeticException ref) // multiple catch statement
        {
            System.out.println("caught Arithmetic Exception");
            System.out.println("invalid number for division");
        }
        catch(ArrayIndexOutOfBoundsException ref2) // multiple catch statement
        {
            System.out.println("ArrayIndexOutOfBoundsException");
            System.out.println("invalid index");
            System.out.println("exit test()");
        }
        public static void main(String[] args) {
            System.out.println("Program start");
            test(5);
            System.out.println("Program ends");
        }
    }
}
```

**op - Program start**

**enter test**

**result : 2**

**ArrayIndexOutOfBoundsException**

**invalid index**

**exit test()**

**Program ends**

## Generic try-catch block

### Program

```
package com.object.exception;
public class Mainclass2 {
    public static void test(int n)
    {
        System.out.println("enter test");
        int a1 [] = { 10,20,30};
        String s1 = "hello";
        Object obj = s1; // upcasting of string into object class
        try
        {
            System.out.println("result : "+10/n);
            System.out.println("array element : "+a1[n]);
            StringBuffer sb1 = (StringBuffer)obj; // direct downcasting of object class into
StringBuffer throw ClassCastException
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        System.out.println("Program start");
        test(2);
        System.out.println("Program ends");
    }
}
```

### Program start

enter test

result : 5

array element : 30

**java.lang.ClassCastException: java.base/java.lang.String cannot be  
cast to java.base/java.lang.StringBuffer**

**at com.object.exception.Mainclass2.test(Mainclass2.java:16)**

**at com.object.exception.Mainclass2.main(Mainclass2.java:27)**

**Program ends**

## **printStackTrace()**

- This methods print three msg.
  - (i) Name of the exception class.
  - (ii) Extra Message
  - (iii) Stack Trace.

## **Program**

```
catch(Exception e)
{
    e.printStackTrace();
}
```

## **Checked Exception**

- Exceptions which are checked by the compiler at compile time are called as checked exceptions.
- All the exceptions which are immediate subclass (ignore Runtime Exception) are checked Exceptions.
- eg. InterruptedException, IOException, SQLException

## **Program**

```
package com.object.exception;
public class Mainclass4 {
    public static void test(int n)
    {
        Thread.sleep(1000); // checked Exception at compiletime
    }
    public static void main(String[] args)
    {
        System.out.println("program start");
        test(0);
        System.out.println("program end");
    }
}
```

**op-** Exception in thread "main" program start

java.lang.Error: Unresolved compilation problem:

Unhandled exception type **InterruptedException**

at com.object.exception.Mainclass4.test(Mainclass4.java:7)

at com.object.exception.Mainclass4.main(Mainclass4.java:12)

## Unchecked Exception

- Exceptions which are not checked by the compiler at compile time are called as unchecked Exceptions.
- All the exceptions which are subclass of RuntimeExceptionClass are unchecked exception.

## Exceptions Propagation:

- Passing the exceptions object from called method to calling method is known as Exception Propagation.
- Unchecked exceptions will be propagated implicitly by JVM.
- we can handle the exceptions of called methods in calling method.

## Program

```
package com.object.exception;
public class Mainclass4 {
    public static void test(int n) // called method
    {
        System.out.println("result = "+10/n);
    }
    public static void main(String[] args)
    {
        System.out.println("program start");
        try // handle the exception in calling method
        {
            test(0);
        }
        catch (ArithmeticException ae)
        {
            ae.printStackTrace();
        }
        System.out.println("program end");
    }
}
```

## Op

program start

java.lang.ArithmeticException: / by zero

at com.object.exception.Mainclass4.test(Mainclass4.java:7)

at com.object.exception.Mainclass4.main(Mainclass4.java:14)

program end

**throws keyword :**

- throws keyword is used to propagate checked exceptions from called method to calling method explicitly.
- throws keyword should be written with method declaration.
- Using throws keyword we can propagate both checked and unchecked exceptions.

**Program**

```
package com.object.exception;
public class Mainclass3 {
    public static void test(int n) throws Exception
    {
        System.out.println("enter test");
        System.out.println("result : "+10/n);
        System.out.println("exit test");
        Thread.sleep(2000); //checked exception
    }
    public static void main(String[] args) {
        System.out.println("Program Start");
        try
        {
            test(0);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        System.out.println("Program end");
    }
}
```

**object :**

java.lang.ArithmeticException: / by zero

Program Start

enter test

at com.object.exception.Mainclass3.test(Mainclass3.java:8)

at com.object.exception.Mainclass3.main(Mainclass3.java:18)

Program end

- we can also call number of methods from calling propagated try block

### Program

```
package practice;
public class Abc
{
    public static void test(int x)
    {
        System.out.println("result =" + 10/x);
    }
    public static void count(int y)
    {
        int a[] = { 10, 20, 30 };
        System.out.println(a[y]);
    }
    public static void main(String[] args) {

        try
        {
            test(10); //calling the method
            count(5); //calling the method
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

op –

```
result =1
java.lang.ArrayIndexOutOfBoundsException: 5
    at practice.Abc.count(Abc.java:12)
    at practice.Abc.main(Abc.java:20)
```

### **throw keyword**

- throw keyword is used to throw the exceptions explicitly.
- throw keyword is written within the method definition.

### **Program**

```
public double getBal(int actno)
{
    if(this.actno==actno)
    {
        return bal;
    }
    else
    {
        String msg = "invalid actno";
        IllegalArgumentException ie = new IllegalArgumentException(msg);
        throw ie; // throw keyword inside a method
    }
}
```

- After throw keyword it is not possible to write any other line of codes.

### **Program**

```
public double getBal(int actno)
{
    if(this.actno==actno)
    {
        return bal;
    }
    else
    {
        String msg = "invalid actno";
        IllegalArgumentException ie = new IllegalArgumentException(msg);
        throw ie;
        System.out.println("hello"); // throw a compile time error
    }
}
```

- we can pass extra message about the exception with the constructor of exception class.

### **Program**

```
String msg = "invalid actno";
IllegalArgumentException ie = new IllegalArgumentException(msg);
```

## Program

```
package com.object.exception;
class Account1
{
    private int actno=1234;;
    private double bal=520.5;

    public double getBal(int actno)
    {
        if(this.actno==actno)
        {
            return bal;
        }
        else
        {
            String msg = "invalid actno";
            IllegalArgumentException ie = new IllegalArgumentException(msg);
            throw ie;
        }
    }
}

public class Mainclass8
{
    public static void main (String ar[])
    {
        Account a = new Account();
        double b = a.getBal(1234);
        System.out.println("balance =" +b);
    }
}

output – balance = 520.5
```



## **DATA STRUCTURE**

- It is a particular way of arranging the data to use them efficiently in the programs.
- Data Structure are of two types :
  - (i) Linear Data Structure
  - (ii) Non Linear Data Structure

### **Linear Data Structure**

- It is a type of data structure where the data is stored one next to another in a linear order or sequences. eg, stack, queue, linked list

### **Non- Linear Data Structure**

- It is a type of data structure where the data is stored according to some relation. eg, tree, graph

### **Linear Data Structure**

#### **Array**

- Array is an object in java.
- To create an array we always use new operator.
- The array variable is actually a reference variable which always points to array object.
- The size of the array can be specified using byte, short and int.
- If you specify an array size with -ve number then jvm throws negative array size exception at runtime.

#### **Program**

```
public class Mainclass {  
    public static void main(String[] args) {  
  
        int s1= -2; // array can not be in negative size  
        int [] a2=new int[s1];  
        System.out.println(a2.length);  
    }  
}
```

**op - Exception in thread "main" java.lang.NegativeArraySizeException  
at com.ds.arrays.Mainclass.main(Mainclass.java:11)**

- An array can be declared as zero. This array have zero bucket and it is not possible to store any data inside this array.

### Program

```
package com.ds.arrays;
public class Mainclass {
    public static void main(String[] args) {
        int s3= 0; // array can be declared with 0
        int [] a3=new int[s3];
        a3[0]= 10; // but we can not store any value
        System.out.println(a3.length);
    }
}
op-Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at com.ds.arrays.Mainclass.main(Mainclass.java:16)
```

### Stack

#### Program

```
package com.ds.stack;
class MyStack {

    private final int DEFAULT_SIZE = 5;
    private int [] stack ;
    private int size = 0;
    private int top = -1;
    private int capacity=0; // to display the holded element_user used

    public MyStack()
    {
        stack = new int [DEFAULT_SIZE];
        capacity = DEFAULT_SIZE;
    }
    public MyStack(int user_size)
    {
        stack = new int [user_size];
        capacity = user_size;
    }
    public boolean push (int value)
    {
        if (size != stack.length)
        {
            stack [++top]= value;
            size ++; //
```

```

        return true;
    }
    else
    {
        return false;
    }
}
public int pop()
{
    if(size !=0)
    {
        size--;
        return stack [top--];
    }
    else
    {
        String msg = "stack is empty";
        IndexOutOfBoundsException e1 = new IndexOutOfBoundsException(msg);
        throw e1;
    }
}

public int getsize ()
{
    return size;
}
public int getCapacity()
{
    return capacity;
}
}
public class Test {
    public static void main(String[] args)
    {
        Mainclass m = new Mainclass();
        boolean b;
        for (int i =0; i<m.getCapacity();i++)
        {
            b=m.push(225);
            System.out.println(b);
        }
        for(int i=0;i<m.getCapacity();i++ )
        {
            int a =m.pop();

```

```

        System.out.println(a);
    }
}

```

/\* in the case of pop element if you use getsize() then in this case every element is not printed

\* because at a when i becomes greater than size because every time size get decremented

\* and for loop terminated, thats y we use m.getCapacity() because capacity remains same

\* as the size and does not change.

\* \*/

**op –**

**true**

**true**

**true**

**true**

**true**

**225**

**225**

**225**

**225**

**225**

## Queue

**package** com.ds.queue;

**class** MyQueue1 {

**private final int** DEFAULT\_SIZE = 5;

**private int[]** queue;

**private int** size = 0;

**private int** start = -1;

**private int** capacity = 0;

**private int** end = -1;

**public** MyQueue1()

{

queue= **new int** [DEFAULT\_SIZE]; //initialize the array

capacity = DEFAULT\_SIZE;// store default size into capacity

}

```

public MyQueue1(int user_size)
{
    queue = new int [user_size]; // initialize the array
    capacity = user_size; // store user size into capacity
}

public boolean enqueue(int value)
{
    if(size!=queue.length)//check the size of queue
    {
        queue[++end]=value; // add the element into queue
        size++; // increment the size
        return true; //
    }
    else
    {
        return false;
    }
}

public int dequeue()
{
    if(size !=0)// check the size of queue
    {
        int v1 = queue[++start]; // dequeue element
        size--; // decrement the size of queue
        return v1;
    }
    else
    {
        String msg = "queue is empty";
        IndexOutOfBoundsException e1 = new IndexOutOfBoundsException(msg);
        throw e1;
    }
}

public int size() // to get access of private data members
{
    return size;
}

public int getCapacity()
{
    return capacity;
}
}

```

```

public class test {

    public static void main(String[] args) {

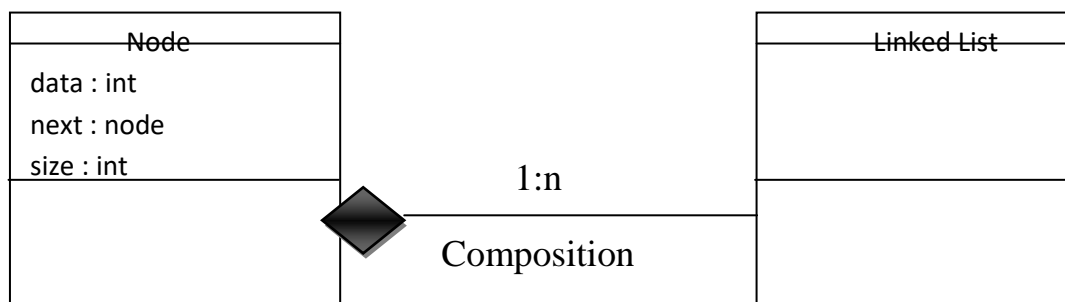
        MyQueue m1 = new MyQueue();
        boolean b1;
        for(int i=0; i<=m1.getCapacity(); i++)
        {
            b1=m1.enqueue(10);
            b1=m1.enqueue(20);
            b1=m1.enqueue(30);
            System.out.println(b1);
        }
        for(int i=0; i<m1.getCapacity();i++)
        {
            System.out.println(m1.dequeue());
        }
    }
}

```

op-  
 true  
 false  
 false  
 false  
 false  
 false  
 10  
 20  
 30  
 10  
 20

## LinkedList

Nodes ————— Linked List  
 Has-A



- A Node can be considered as an object which contains two parts.
  - (i) Data
  - (ii) Address of next node

Data	Address of next node
------	----------------------

## Program

**package** com.ds.linkedlist;

```

class Node1 {
    int data;
    Node next;
    public Node1(int data)
    {
        this.data=data;
    }
    public void setNext(Node next)
    {
        this.next=next;
    }
    public Node getNext()
    {
        return next;
    }
}

class MyLinkedList1 {
    private int size=0;
    private Node first;
    private Node last;
    public MyLinkedList1()
    {
        first=null;
        last=null;
    }
    public boolean addToLast(int value)
    {
        Node n1= new Node(value); // create the object of node class
        size++; // update size whenever an object is created
        if(first==null) // if the size is null
        {
            first=n1; //
            last=n1;
        }
    }
}

```

```

    }
    else
    {
        last.setNext(n1);
        last=n1;
    }
    return true;
}
public int getValue(int position)
{
    if(position<=size)
    {
        if(position==1)
        {
            return first.data;
        }
        else if (position==size)
        {
            return last.data;
        }
        else
        {
            Node temp =first.getNext();
            for(int i=2;i<position;i++)
            {
                temp=temp.getNext();
            }
            return temp.data;
        }
    }
    else
    {
        String msg = "invalid position";
        IllegalArgumentException ie = new IllegalArgumentException(msg);
        throw ie;
    }
}
public int size()
{
    return size;
}
}

```



```

public class Test {
    public static void main(String[] args) {

        MyLinkedList m1 = new MyLinkedList();
        m1.addToLast(20);
        m1.addToLast(40);
        m1.addToLast(60);
        m1.addToLast(80);
        m1.addToLast(100);
        m1.addToLast(120);
        m1.addToLast(140);
        int v1 = m1.getValue(2);
        System.out.println(v1);

        System.out.println("-----");
        int sz = m1.size();
        for(int i =1; i<=sz;i++)
        {
            int v2 = m1.getValue(i);
            System.out.println(v2);
        }
    }
}

```

op –

40

-----

20

40

60

80

100

120

140

## OBJECT ARRAY

- It is a type of array where every bucket of the array represents a reference variable of the given class or interface.

- Syntax :

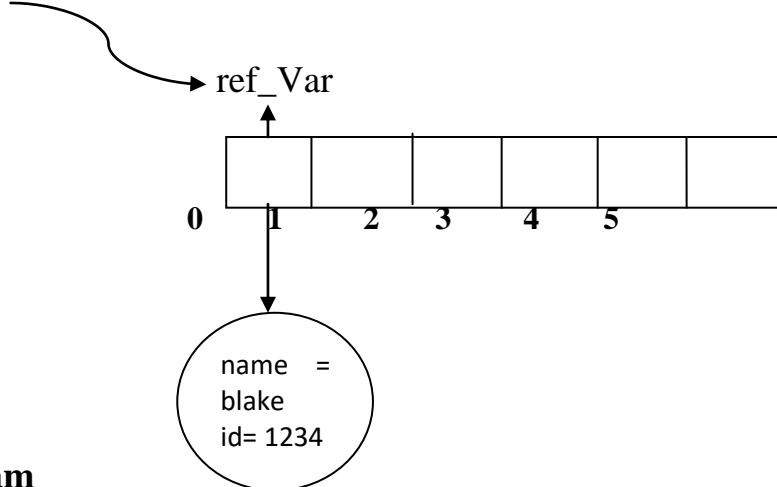
classname [] ref\_var = new classname[size];

eg :

```
Employee [] e1 = new Employee[5];
```

```
e1[0]=new Employee("blake", 1234);
```

```
e1[0].name;
```



### Program

```
package com.array.objectarray;
```

```
class Employee
```

```
{
```

```
    String name;
```

```
    int id;
```

```
    double bal;
```

```
    Employee(String name, int id, double bal)
```

```
    {
```

```
        this.name=name;
```

```
        this.id=id;
```

```
        this.bal=bal;
```

```
    }}
```

```
public class Mainclass {
```

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

```
        Employee[] emplist = new Employee[2];
```

```
        emplist[0]= new Employee("blake", 142,252.5);
```

```
        emplist[1]= new Employee("Martin", 122,525.5);
```

```

        processSalary(emplist);
    }
    public static void processSalary(Employee[] emp)
    {
        for(int i=0; i<emp.length;i++)
        {
            System.out.println("name = "+emp[i].name);
            System.out.println("Id = "+emp[i].id);
            System.out.println("-----");
        }
    }
}

```

**eg :**

```

name = blake
Id = 142
-----
name = Martin
Id = 122
-----

```

## **Program 2**

```

package com.arry.objectarray;
class VehicleCar
{
    String model;
    String name;
    double price;
    VehicleCar(String model, String name, double price)
    {
        this.model=model;
        this.name=name;
        this.price=price;
    }
    @Override
    public String toString()
    {
        return "Model = " +model + "Name = "+ name + " price = " +price ;
    }
}
class VehicleBike
{

```

```

String model;
String name;
double price;
VehicleBike(String model, String name, double price)
{
    this.model=model;
    this.name=name;
    this.price=price;
}
@Override
public String toString()
{
return "Model = " + model + " Name = "+name + " price = " +price ;
}
}
class Mainclass
{
    public static void main(String[] args) {

        Object [] vList = new Object[4];
        vList[0]= new VehicleCar("honda", " city", 25.2);
        vList[1]= new VehicleCar("hyundai", "i20", 8.20);
        vList[2]= new VehicleBike("KTM", "Duke", 1.75);
        vList[3]= new VehicleBike("Royal", "Classic", 2.75);
        showVehicleDetails(vList);
    }
    public static void showVehicleDetails(Object[] ref)
    {
        for(int i = 0; i<ref.length;i++)
            System.out.println(ref[i]);
    }
}

```

### **output**

```

Model = hondaName =  city price = 25.2
Model = hyundaiName = i20 price = 8.2
Model = KTM Name = Duke price = 1.75
Model = Royal Name = Classic price = 2.75

```

-----

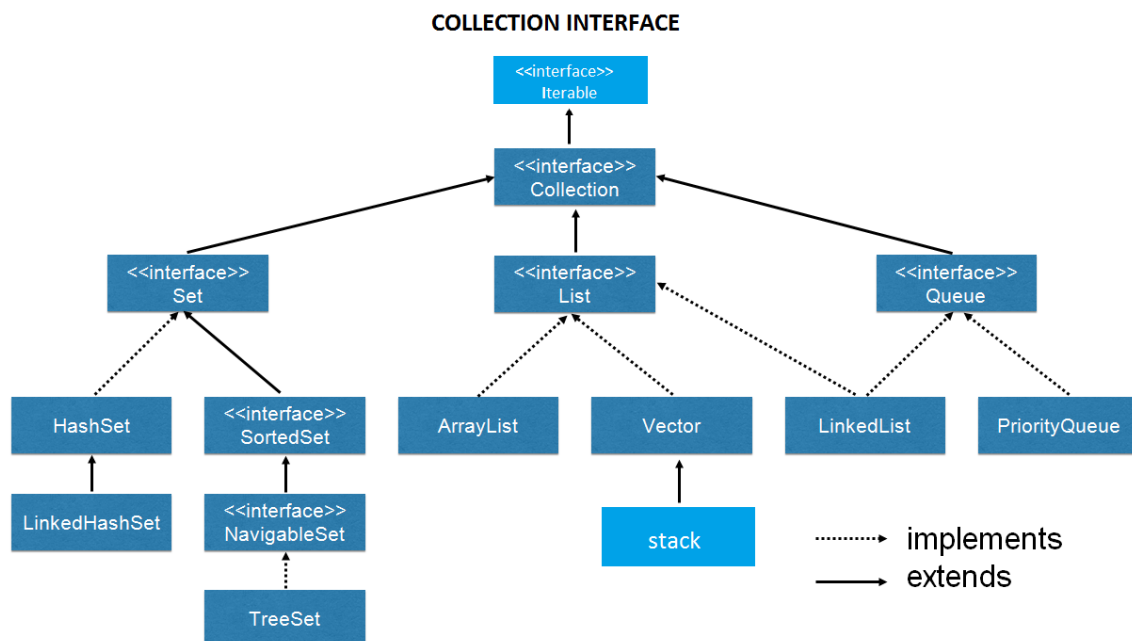
# COLLECTION FRAMEWORK

## Collection

- A collection is group of references which is represented as one entity.
- collections are growable in nature and the size of collection can be changed at runtime.
- Collections support heterogeneous data.
- Every collection will have at least one underlying data structure.
- Every collection will have inbuilt methods which are useful to perform basic or advanced operation on the data present in the collections.
- Collection framework is built in Java to help the programmers and provide the API which will abstract the effort of building the data structure.

## Collection Framework

- It is a group of class and interfaces which is present in java.util packages to perform operation on dynamic data



## LIST

### Types of List

- ArrayList
- Linked List
- Stack
- Vector

## Features of List

- Duplicates are allowed
- Indexed type collection
- Multiple Null values are allowed.
- Insertion order is preserved.

### ArrayList (JDK 1.2)

- ArrayList implements list, collection, iterable, serializable, cloneable and random access interfaces.
- The underlying data structure for ArrayList is resizable array.
- The time required to retrieve first element or last element of ArrayList is always constant or same.

## Constructor

- **ArrayList()**  
Construct an empty list with an initial capacity of ten.
- **ArrayList(int initial capacity)**  
Construct an empty list with the specified initial capacity.
- **ArrayList(Collection c)**  
This Constructor creates a list by converting given collection.

## Program 1

```
package com.ds.collectionframework;
class Employee
{
    String name;
    int id;
    double sal;
    Employee(String name, int id, double sal)
    { this.name=name;
      this.id=id;
      this.sal=sal;
    }
}
```

## Program 2 to utilize it

```
//ArrayList Program
package com.ds.collectionframework;
import java.util.ArrayList;
public class Mainclass {
    public static void main(String[] args) {
        ArrayList emplist = new ArrayList();
    }
}
```

```

        emplist.add(new Employee ("Blake", 1234, 2341.3));
        emplist.add(new Employee ("Sundar", 1274, 2346.3));
        processSalary(emplist);
    }

```

```

public static void processSalary(ArrayList ref)
{
    for (int i =0; i<ref.size(); i++)
    {
        Employee e1 = (Employee) ref.get(i);
        System.out.println("name = "+e1.name);
        System.out.println("salary = "+e1.sal);
    }
}

```

### Functions of ArrayList

```
package com.jsp.arraylist;
```

```

import java.util.*;

public class Mainclass2 {
    public static void main(String[] args) {

        ArrayList a = new ArrayList();
        System.out.println("initial size = "+a.size());
        //initial size = 0
        a.add("aatif");
        a.add("salman");
        System.out.println("size "+a.size());
        // size = 2
        System.out.println("elements present "+a);
        //elements present [aatif, salman]
        a.remove("salman");
        System.out.println("elements present "+a);
        //elements present [aatif]
        a.add("salman");
        System.out.println("elements present "+a);
        //elements present [aatif, salman]
        a.add(0,"satya");
        System.out.println("elements present "+a);
    }
}

```

```

//elements present [satya, aatif, salman]
a.remove(0);
System.out.println("elements present "+a);
//elements present [aatif, salman]

a.set(1, "sadaf");
System.out.println("elements present "+a);
//elements present [aatif, sadaf]
int pos= a.indexOf("aatif");
System.out.println("position of = "+pos);
//posiition of = 0
int pos1= a.indexOf("preeti");
System.out.println("position of = "+pos1);
//position of = -1
String sb = (String) a.get(1); //upcasted to object type
System.out.println("at index = " +sb);
//at index = sadaf
Boolean b = a.contains("salman");
System.out.println(b); // returns false
a.clear();
System.out.println("Elements after clear " +a);
//elements after clear = []; empty
}}

```

- **When there is two type of data stored in ArrayList**

#### **Program**

```

package com.ds.collectionframework;
import java.util.*;
abstract class A
{
    String model;
    String name;
    double price;
    public abstract String toString();
}
class Car extends A
{

```



```

    public Car(String model, String name, double price)
    {
        this.model=model;
        this.name=name;
        this.price=price;
    }
    @Override
    public String toString()
    {
return "Model = " + model + " Name = "+name + " price = " +price ;
    }
}
class Bike extends A
{
    public Bike(String model, String name, double price)
    {
        this.model=model;
        this.name=name;
        this.price=price;
    }
    @Override
    public String toString()
    {
return "Model = " + model + " Name = "+name + " price = " +price ;
    }
}
public class Mainclass3 {
    public static void main(String[] args) {

        ArrayList vlist = new ArrayList();
        vlist.add(new VehicleCar("honda", " city", 25.2));
        vlist.add(new VehicleCar("hyundai", "i20", 8.20));
        vlist.add(new VehicleBike("KTM", "Duke", 1.75));
        vlist.add(new VehicleBike("Royal", "Classic", 2.75));

        showVehicleDetails(vlist);
    }
}

```

```

public static void showVehicleDetails(ArrayList ref)
{
    for(int i=0; i<ref.size();i++)
    {
        if (ref.get(i) instanceof VehicleCar)
        {
            VehicleCar c1 = (VehicleCar)ref.get(i);
            System.out.println(c1);
        }
        else
        {
            VehicleBike b1 = (VehicleBike)ref.get(i);
            System.out.println(b1);
        }
    }
}

```

op-

```

Model = hondaName = city price = 25.2
Model = hyundaiName = i20 price = 8.2
Model = KTM Name = Duke price = 1.75
Model = Royal Name = Classic price = 2.75

```

### or Program 1<sup>st</sup> program

```

package com.ds.collectionframework;
class VehicleCar
{
    String model;
    String name;
    double price;
    VehicleCar(String model, String name, double price)
    {
        this.model=model;
        this.name=name;
        this.price=price;    }
    @Override
    public String toString()
    {
        return "Model = " +model + "Name = "+ name + " price = " +price ;
    }
}

```

```

class VehicleBike
{
    String model;
    String name;
    double price;
    VehicleBike(String model, String name, double price)
    {
        this.model=model;
        this.name=name;
        this.price=price;
    }
    @Override
    public String toString()
    {
        return "Model = " + model + " Name = "+name + " price = " +price ;
    }
}

```

### **Program 2 to implement it**

```

package com.ds.collectionframework;
import java.util.*;
public class Mainclass7 {
    public static void main(String[] args) {
        ArrayList vlist = new ArrayList();
        vlist.add(new VehicleCar("honda", " city", 25.2));
        vlist.add(new VehicleCar("hyundai", "i20", 8.20));
        vlist.add(new VehicleBike("KTM", "Duke", 1.752));
        vlist.add(new VehicleBike("Royal", "Classic", 2.75));
        showVehicleDetails(vlist);
    }
    public static void showVehicleDetails(ArrayList ref)
    {
        for(int i=0; i<ref.size();i++)
        {
            Object obj= ref.get(i);
            System.out.println(ref.get(i));
            System.out.println("-----");
        }
    }
}

```

op –

**Model = hondaName = city price = 25.2**

-----

**Model = hyundaiName = i20 price = 8.2**

-----

**Model = KTM Name = Duke price = 1.752**

-----

**Model = Royal Name = Classic price = 2.75**

-----

## Generics

- Generics in collection are used to achieve two goals.
  - (i) type-safety by writing : collection<classname>
  - (ii) Avoid unnecessary downcasting of object.

## Program

```
package com.ds.collectionframework;
import java.util.ArrayList;
public class Mainclass6 {
    public static void main(String[] args) {
        ArrayList<Employee> emplist = new ArrayList<Employee>();
        emplist.add(new Employee ("Blake", 1234, 2341.3));
        emplist.add(new Employee ("Sundar", 1274, 2346.3));
        processSalary(emplist);
    }
    public static void processSalary(ArrayList<Employee> ref)
    {
        for (int i =0; i<ref.size(); i++)
        {
            Employee e1 = (Employee) ref.get(i);
            System.out.println("name = "+e1.name);
            System.out.println("salary = "+e1.sal);
        }
    }
}
```

**op-    name = Blake**  
**salary = 2341.3**  
**name = Sundar**  
**salary = 2346.3**

## Functions of Stack

```
import java.util.Stack;
public class A {

    public static void main(String[] args) {

        Stack s = new Stack();
        System.out.println("initial size = "+s.size());
        //initial size = 0
        s.add("aatif");
        s.add("salman");
        s.add("preeti");
        System.out.println("elements present = "+s);
        //elements present = [aatif, salman, preeti]
        s.remove("preeti");
        System.out.println("elements present = "+s);
        //elements present = [aatif, salman]
        s.add(0, "satya");
        System.out.println("elements present = "+s);
        //elements present = [satya, aatif, salman]
        s.set(0, "sada");

        int pos = s.indexOf("aatif");
        System.out.println(pos);
        //1
        int pos1 = s.indexOf("bholu");
        System.out.println(pos1);
        //-1
        String sb = (String)s.get(1);
        System.out.println("At index =" + sb);
        //At index =aatif
        Boolean b = s.contains("aatif");
        System.out.println(b);
        //true
        Boolean b1 = s.contains("jeet");
        System.out.println(b1);
        //false
        s.clear();
        System.out.println("elements present = "+s);
        //elements present = []
    }
}
```

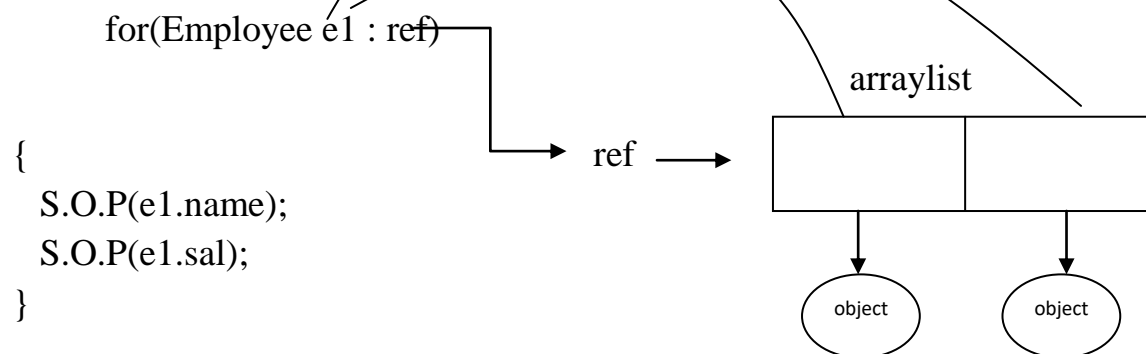
### for each loop / enhanced loop/advance loop

- for each loop helps the programmer to iterate collections and arrays without worrying about index, size and type of the arrays & collections.

### Syntax

for(classname ref\_Var : ref\_var of collection & Array)

for eg :



### Program

//arraylist Program

```
package com.ds.collectionframework;
```

```
import java.util.ArrayList;
```

```
public class Mainclass6 {
```

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

```
    ArrayList<Employee> emplist = new ArrayList<Employee>();
```

```
    emplist.add(new Employee ("Blake", 1234, 2341.3));
```

```
    emplist.add(new Employee ("Sundar", 1274, 2346.3));
```

```
    processSalary(emplist);
```

```
  }
```

```
  public static void processSalary(ArrayList<Employee> ref)
```

```
  {
```

```
    for(Employee e1 : ref)
```

```
    {
```

```
        System.out.println(e1.name);
```

```
        System.out.println(e1.sal);
```

```
    }}}
```

**op - Blake**

**2341.3**

**Sundar**

**2346.3**

## **for-each Method()**

- This method is introduced from JDK 1.8 which helps the programmer to iterate collections.
- For-each method accepts consumer(interface) type references.

## **Consumer interface**

- It is a functions interface which contains only one abstract method by the name Accept(T t)

## **Program**

```
package com.ds.collectionframework;
//use of for each method
import java.util.ArrayList;
import java.util.function.Consumer;
public class Mainclass9 {
    public static void main(String[] args) {
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add(new String("hello"));
        a1.add(new String("java"));
        a1.add(new String("Android"));
        for(int i=0; i<a1.size();i++)
        {
            String str = a1.get(i);
            System.out.println(str.length());
        }
        //System.out.println(a1.get(i).length()); we can also write like that }
        /we can also write like this lambda function
        Consumer<String> ref = (String s1)->
        {
            System.out.println(s1.length()); };
        a1.forEach(ref);
        //we can also write like this lambda function
        Consumer <String> ref1= (String s2) -> System.out.println(s2.length());
        a1.forEach(ref1);
        //we can also write like this lambda function
        a1.forEach((String s3)-> System.out.println(s3.length()));
        //we can also write like this lambda function
        a1.forEach(s4-> System.out.println(s4.length()));

    }}
}
```

## Vector (JDK 1.0)

- All the methods are synchronized
- Vector is thread safe class.
- Vector implements list, collections, iterable.
- Vector implements list, collection, iterable seralizable, cloneable and random access interfaces.
- The underline data structure for Vector is resizable array.
- The time required to retrieve first element or last element of Vector is always constant or same.

## Constructor

- **Vector()**  
Construct an empty list with an initial capacity of ten.
- **Vector(int initial capacity)**  
Construct an empty list with the specified initial capacity.
- **Vector(Collection c)**  
This Constructor creates a list by converting given collection.

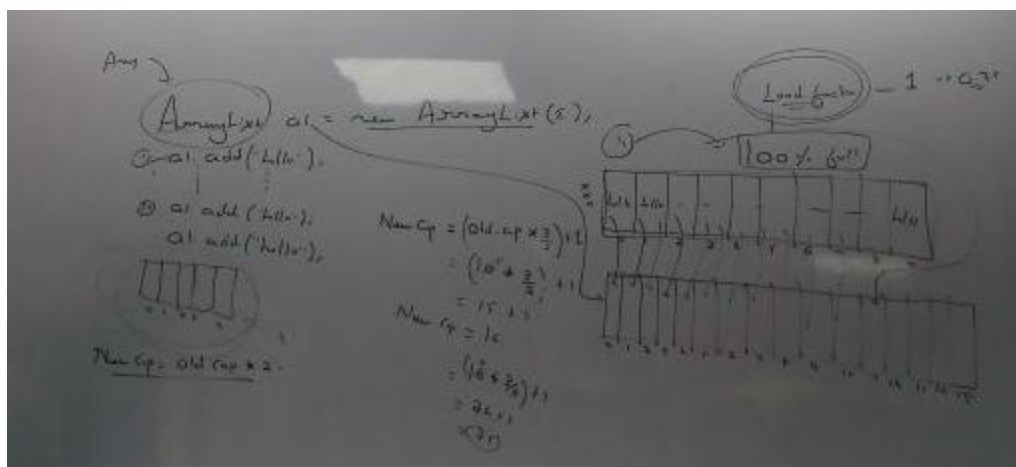
**Note :**

- The load factor decides when should be new collection created to add new elements.
- The load factor of arraylist and vector is 100% or 1.
- The new capacity of arraylist is calculated using the below formulae.

$$\text{New Capacity} = (\text{old capacity} * 3/2) + 1$$

- New Capacity of vector is calculated with below formulae.

$$\text{New Capacity} = \text{old capacity} * 2$$





## SET

- Set do now allow duplicates.
- Set do not have index.
- only one null value can be stored.
- insertion order is not preserved.

### (a) Set do not allow duplicates

#### Program

```
package com.cf.set;
import java.util.HashSet;
public class Mainclass1 {
    public static void main(String[] args) {

        HashSet<String> h = new HashSet<String>();
        h.add(new String("hello"));
        h.add(new String("hello"));
        h.forEach(s4-> System.out.println(s4));
    }
}
```

**op – hello**

- Set do not have index.

#### Program

```
import java.util.HashSet;
import java.util.function.Consumer;
public class Mainclass1 {

    public static void main(String[] args) {

        HashSet<String> h = new HashSet<String>();
        h.add(new String("hello"));
        h.add(new String("hi"));
        h.add(new String("java"));

        h.forEach(s4-> System.out.println(s4));
    }
}
```

**op - hi  
java  
hello**

- **only one null value can be stored.**

### **Program**

```
import java.util.HashSet;

public class Mainclass1 {
    public static void main(String[] args) {
        HashSet<String> h = new HashSet<String>();
        h.add(new String("hello"));
        String s=null;
        String b = null;
        h.add(s);
        h.add(b);
        h.add(new String("java"));
        h.forEach(s4-> System.out.println(s4));
    }
}
```

**op –**  
**null**  
**java**  
**hello**

- **insertion order is not preserved.**

### **Program**

```
import java.util.HashSet;
import java.util.function.Consumer;

public class Mainclass1 {
    public static void main(String[] args) {

        HashSet<String> h = new HashSet<String>();
        h.add(new String("hello"));
        h.add(new String("hi"));
        h.add(new String("java"));

        h.forEach(s4-> System.out.println(s4));
    }
}
```

**op - hi**  
**java**  
**hello**

## **hashSet**

- hashSet implements iterable, collection, set, clonable and serializable interfaces.
- The underline data structure for hashset is hash table.
- hashSet preserve uniqueness by compairing hashcode value of given objects.

### **Constructors of hashSet:**

- (i) public hashSet()  
construct a new empty set with default initial capacity(16) and load factors (0.75)
  - (ii) public hashSet(int initialcapacity)  
construct a new empty set with given initial capacity and load factor(0.75)
  - (iii) public hashset(int initialcapaicty, float loadfactor)  
construct a new empty set with given initial capacity and load factor.
  - (iv) public hashSet(Collection c)  
construct a new set with given collection.
- if you try to add duplicates value, we don't get any compile time error and runtime exceptions.

### **Adding unique user defined object :**

- if you want to add unique user defined object to the set then you should override hashcode method and equals method in the given user defined class.

### **Program**

```
package com.cf.set;
public class Employee {
    String name;
    int id;
    double sal;
    Employee(String name, int id, double sal)
    { this.name=name;
      this.id=id;
      this.sal=sal;
    }
}
```

```

@Override
public boolean equals(Object obj)
{
    Employee emp=(Employee)obj;
    if(this.hashCode()==emp.hashCode())
    {
        return true;    }
    else
    {
        return false;
    }
}

@Override
public int hashCode()
{
    return id;
}

@Override
public String toString()
{
    String info = name + " " +id + " "+ sal;
    return info;
}

```

## 2<sup>nd</sup> Program

```

package com.cf.set;
import java.util.HashSet;
public class Mainclass3 {
    public static void main(String[] args) {
        HashSet<Employee> hs = new HashSet<Employee>();
        hs.add(new Employee("Blake",101,2012.2));
        hs.add(new Employee("Rohit",102,2052.2));
        hs.add(new Employee("Blake",101,2012.2));
        hs.forEach(emp -> System.out.println(emp));
    }
}

```

**op - Blake 101 2012.2**  
**Rohit 102 2052.2**

## LinkedHashSet

- The underline data structure is hashtable and linkedList.
- LinkedHashSet preserve insertion order

## Program

```
import java.util.HashSet;
import java.util.LinkedHashSet;

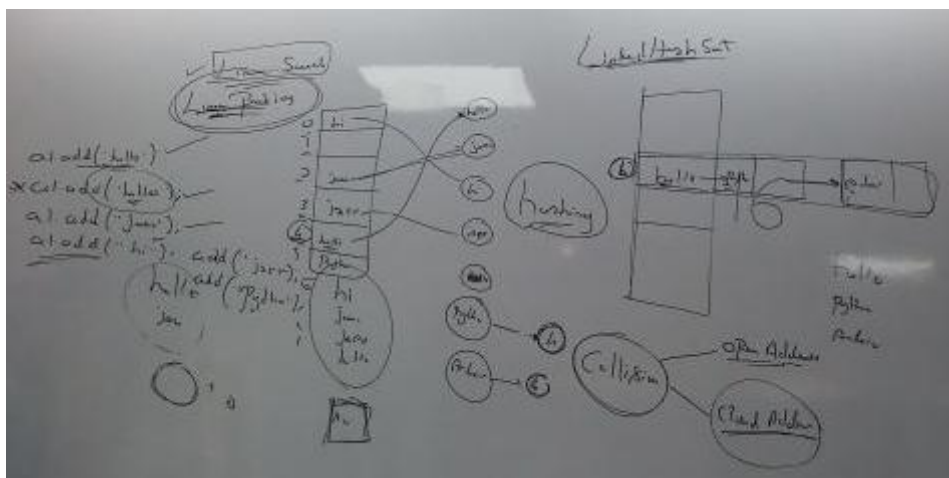
public class Mainclass5 {

    public static void main(String[] args) {

        LinkedHashSet<String> hs = new LinkedHashSet<String>();
        hs.add(new String("hello"));
        hs.add(new String("java"));
        hs.add(new String("android"));

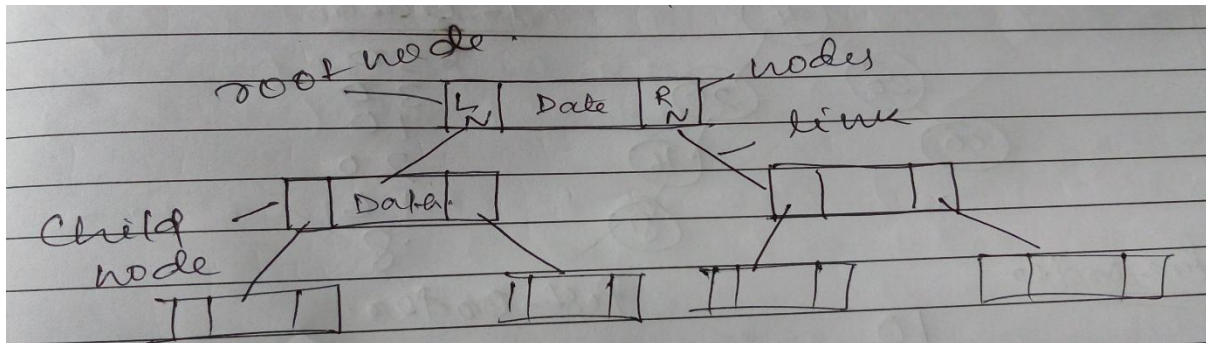
        hs.forEach(str -> System.out.println(str));
    }
}
```

**op –**  
**hello**  
**java**  
**android**



## TREE DATA STRUCTURE

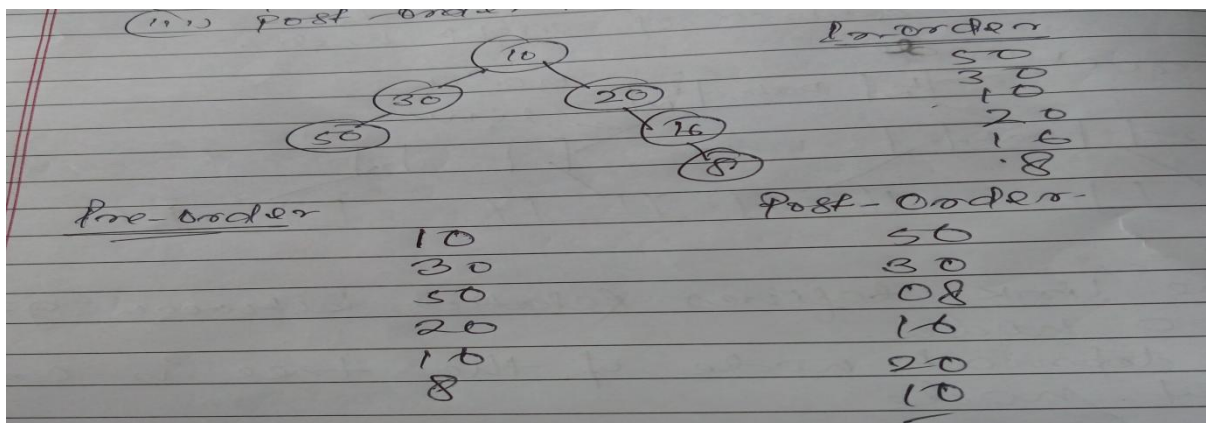
- A tree is a **non-linear data structure** which contains group of nodes and links.
- Nodes: The node of tree consists of three parts :
  - (i) Data
  - (ii) Address of left node
  - (iii) Address of right node



- **Link** – The link defines relation between given two nodes.
- The topmost node of the tree is called root node.

### Binary Tree

- It is a type of tree where every parent node can have minimum zero child nodes or maximum 2 child nodes.
- In Order to retrieve the elements from the tree we have to traverse through every nodes of the given tree.
- we can traverse through the tree using three different order
  - (i) inorder- left, parent, right
  - (ii) pre-order – Parent, left, right
  - (iii) post-order – left, right, parent



## Tree Collection

- TreeSet do not allow duplicates.
- TreeSet do not have any index, **but return sorted result.**
- TreeSet do not allow null values.
- TreeSet do not preserve insertion order.
- All elements inserted into the set must implements the comparable interfaces.
- The elements added to the treeSet can be ordered(sorted) in two ways:
  - (i) Natural Ordering
  - (ii) Customized Ordering

### (i) TreeSet do not allow duplicates.

#### Program

```
import java.util.TreeSet;
public class Mainclass {
    public static void main(String[] args) {

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("java");
        ts.add("android");
        ts.add("java");

        ts.forEach(str -> System.out.println(str));

    } } op – android
java
```

### (ii) TreeSet do not have any index, but return sorted result.

#### Program

```
import java.util.TreeSet;
public class Mainclass {
    public static void main(String[] args) {

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("java");
        ts.add("android");
        ts.add("java");
        ts.add("bigdata");
```

```

        ts.add("python");
        ts.add("eclipse");
        ts.add("class");
    ts.forEach(str -> System.out.println(str));
}

```

**op -**  
**android**  
**bigdata**  
**class**  
**eclipse**  
**java**  
**python**

**(iii) TreeSet do not allow null values.**

```

package com.jsp.treeset;

```

```

import java.util.TreeSet;
public class Mainclass {
    public static void main(String[] args) {

```

```

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("java");
        ts.add("android");
        ts.add(null);
        ts.forEach(str -> System.out.println(str));

```

```

    }
}

```

**op – here it throw a runtime exception as**

**Exception in thread "main" java.lang.NullPointerException**  
**at java.base/java.util.TreeMap.put(Unknown Source)**  
**at java.base/java.util.TreeSet.add(Unknown Source)**  
**at com.jsp.treeset.Mainclass.main(Mainclass.java:10)**



**(i) Insertion order is not preserved**

**Program**

```
import java.util.TreeSet;
public class Mainclass {
    public static void main(String[] args) {

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("java");
        ts.add("android");
        ts.add("java");
        ts.add("bigdata");
        ts.add("python");
        ts.add("eclipse");
        ts.add("class");
        ts.forEach(str -> System.out.println(str));
    }
}
```

**op -**  
**android**  
**bigdata**  
**class**  
**eclipse**  
**java**  
**python**

**Comparable Interface**

- This interface helps in comparing the java object with the user defined order.
- Comparable interface contains one abstract method by the name compareTo(Object ref).
- compareTo method returns positive number, if given object is greater than another object.
- compareTo method return negative number if given object is smaller or lesser than another object.
- compareTo method returns zero if given object is equal to another object or having same value.

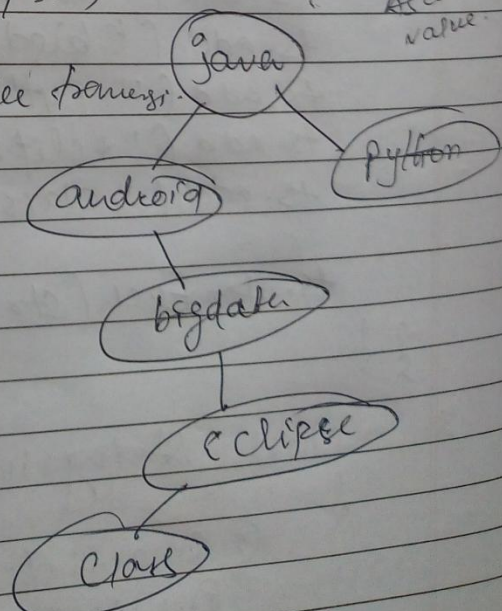
compareTo() return +, -, 0, Integer.  
 TreeSet ts = new TreeSet<String>();

ts.add("java"); android.compareTo("java"), -  
 ts.add("android"); bigdata.compareTo("java"), -  
 ts.add("bigdata"); bigdata.compareTo("android"), +  
 ts.add("python"); python.compareTo("java"), +  
 ts.add("eclipse"); eclipse.compareTo("java"), -  
 eclipse.compareTo("android"), +  
 eclipse.compareTo("bigdata"), +

ts.add("class"); class.compareTo("java"), =  
 class.compareTo("android") +  
 class.compareTo("bigdata") +  
 class.compareTo("eclipse") =

It only preserve the order of elements.

Android  
 bigdata  
 class  
 eclipse  
 java  
 python



## Saving User Defined Class in TreeSet

### Program 1<sup>st</sup>

```
package com.jsp.treeset;
class Employee implements Comparable<Employee>
{
    String name;
    int id;
    double sal;
    Employee(String name, int id, double sal)
    { this.name=name;
      this.id=id;
      this.sal=sal;
    }
    @Override
    public String toString()
    {
        String info = name + " " +id + " "+ sal;
        return info;
    }
    /* @Override // sorted on the base of id
    public int compareTo(Employee emp) {

        int val=this.id-emp.id;
        return val;
    } */
    /*@Override // sorted on the base of salary
    public int compareTo(Employee emp) {
        int val=(int)(this.sal-emp.sal);
        return val;
    } */
    @Override // sorted on the base of name
    public int compareTo(Employee emp) {
        String n1=this.name;
        String n2=emp.name;
        int val = n1.compareTo(n2);
        return val;
    }
}}
```

## **2<sup>nd</sup> Program to implement it**

```
package com.jsp.treeset;
import java.util.TreeSet;
public class Mainclass1
{
    public static void main(String[] args) {

        TreeSet<Employee> emplist = new TreeSet<Employee>();

        emplist.add(new Employee ("Blake", 1234, 2341.3));
        emplist.add(new Employee ("Sundar", 1274, 2346.3));
        emplist.add(new Employee ("Satya", 1525, 9684.3));
        emplist.add(new Employee ("Aatif", 1895, 2358.3));
        emplist.add(new Employee ("aatif", 1895, 2358.3));
        emplist.forEach(emp -> System.out.println(emp));

    }
}
```

**op- It returns output on the basis of sorting of name in ascending order**

```
Aatif 1895 2358.3
Blake 1234 2341.3
Satya 1525 9684.3
Sundar 1274 2346.3
aatif 1895 2358.3
```

## Customized Sorting

- Sorting of inbuilt classes such as String according to programmer choice.
- Comparator interface is used for Customized Sorting.
- Comparator interface is a part of java.util. package.
- Comparator interface is a functional interface having one abstract method as compare(T o1, T o2)

## Program

```
package com.jsp.treeset;
import java.util.Comparator;
import java.util.TreeSet;
public class Mainclass2 {
    public static void main(String[] args) {
// using lambda function here because Comparator is an functional
interface
```

```
        Comparator<String> c1= (String s1, String s2)->
        {
            int v1 = s2.compareTo(s1);
            return v1;
        };
    }
```

## passing the reference of interface to the constructor of TreeSet

```
        TreeSet<String> ts = new TreeSet<String>(c1);
        ts.add("java");
        ts.add("android");
        ts.add("bigdata");
        ts.add("python");
        ts.add("eclipse");
        ts.add("class");

        ts.forEach(str -> System.out.println(str));
    }
```

**op – in descending order**

**python**

**java**

**eclipse**

**class**

**bigdata**

**android**

## Comparator interface

- It is a functional interface which is used to perform customized sorting.
- If the programmer want to use customized sorting and do not want to use the natural ordering then we use comparator interface.
- Using comparator we can sort both comparable(eg. String) and non-comparable (eg, StringBuffer) object.
- Comparator contains only one abstract method by the name compare.
- compare() method returns positive number, if given object is greater than another object.
- compare() method return negative number if given object is smaller or lesser than another object.
- compare() method returns zero if given object is equal to another object or having same value.

## Use of Comparator interface in StringBuffer

### Program

```
package com.jsp.treeset;
import java.util.Comparator;
import java.util.TreeSet;
public class Mainclass3 {
    public static void main(String[] args) {

        Comparator<StringBuffer> c1= (StringBuffer sb1, StringBuffer
sb2)->
        {
            String str1=sb1.toString();
            String str2=sb2.toString();
            int v1 = str1.compareTo(str2);
            return v1;
        };

        TreeSet<StringBuffer> ts = new TreeSet<StringBuffer>(c1);
        ts.add(new StringBuffer("java"));
        ts.add(new StringBuffer("android"));
        ts.add(new StringBuffer("bigdata"));
        ts.add(new StringBuffer("python"));
        ts.add(new StringBuffer("eclipse"));
        ts.add(new StringBuffer("class"));

        ts.forEach(str -> System.out.println(str));

    }
}
```

**op-  
android  
bigdata  
class  
eclipse  
java  
python**

## **QUEUE TYPE OF COLLECTION**

- Duplicates are allowed.
- Index is present
- Insertion Order is preserve.
- Multiple null values are allowed.

### **(i) Duplicates are allowed**

#### **Program**

```
package com.ds.queuecollection;
```

```
import java.util.LinkedList;
```

```
public class Mainclass {
```

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

```
        LinkedList<String> list = new LinkedList<String>();
```

```
        list.add("java");
```

```
        list.add("java");
```

```
        list.add("java");
```

```
        list.forEach(str -> System.out.println(str));
```

```
    }}
```

**op-  
java  
java  
java**

(ii) **Index is present**

**Program**

```
package com.ds.queuecollection;
```

```
import java.util.LinkedList;
```

```
public class Mainclass {
```

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

```
        LinkedList<String> list = new LinkedList<String>();
```

```
        list.add("java");
```

```
        list.add("android");
```

```
        list.add("bigdata");
```

```
        String a= list.get(2);
```

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

```
    }}
```

**op-**

**bigdata**

(iii) **Insertion Order is preserve**

**Program**

```
package com.ds.queuecollection;
```

```
import java.util.LinkedList;
```

```
public class Mainclass {
```

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

```
        LinkedList<String> list = new LinkedList<String>();
```

```
        list.add("java");
```

```
        list.add("android");
```

```
        list.add("bigdata");
```

```
        list.forEach(str -> System.out.println(str));
```

```
    }}
```

**op-**

**java**

**android**

**bigdata**



- (iv) Multiple Null values are allowed.

### Program

```
package com.ds.queuecollection;
import java.util.LinkedList;
public class Mainclass {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();
        list.add(null);
        list.add(null);
        list.add(null);
        list.forEach(str -> System.out.println(str));
    }
}
```

**op-**  
**null**  
**null**  
**null**

### LinkedList Collection

- The underline data structure for linked list collection is LinkedList data structure.
- LinkedList implements List and Queue interface.
- LinkedList can be used as a list and as well as queue.
- We can retrieve the elements from the linkedList using two method.
  - (i) get(index)
  - (ii) poll()

### Constructor of LinkedList :

- (i) **public LinkedList()**  
construct an empty list.
- (ii) **public LinkedList(Collection c)**  
Construct a list from the given collection.

### Poll()

- It returns the head elements from the queue and also removes it from the queue.

### Peek()

- It returns head elements from the queue but doesn't remove it from queue.

## Use of Poll method

### Program

```
package com.ds.queuecollection;
```

```
import java.util.LinkedList;
```

```
public class Mainclass {
```

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

```
        LinkedList<String> list = new LinkedList<String>();
```

```
        list.add("java");
```

```
        list.add("android");
```

```
        list.add("bigdata");
```

```
        list.add("python");
```

```
        list.add("eclipse");
```

```
        list.add("class");
```

```
        System.out.println("size =" + list.size());
```

```
        int s1=list.size(); // we declare list size here because if we directly  
        use size() method inside for loop then it does not retrieve every  
        element.
```

```
        for(int i=0; i<s1;i++)
```

```
        {
```

```
            String s2 = list.poll();
```

```
            System.out.println(s2);
```

```
        }
```

```
        System.out.println("size=" +list.size());
```

```
    }
```

```
}
```

**op-**

**size =6**

**java**

**android**

**bigdata**

**python**

**eclipse**

**class**

**size =0**

## Directly using size() method inside for loop

### Program

```
package com.ds.queuecollection;
import java.util.LinkedList;
public class Mainclass {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("java");
        list.add("android");
        list.add("bigdata");
        list.add("python");
        list.add("eclipse");
        list.add("class");
        System.out.println("size =" + list.size());
        for(int i=0; i<list.size();i++)
        {
            String s2 = list.poll();
            System.out.println(s2);
            System.out.println("size =" +list.size());
        }
    }
}
```

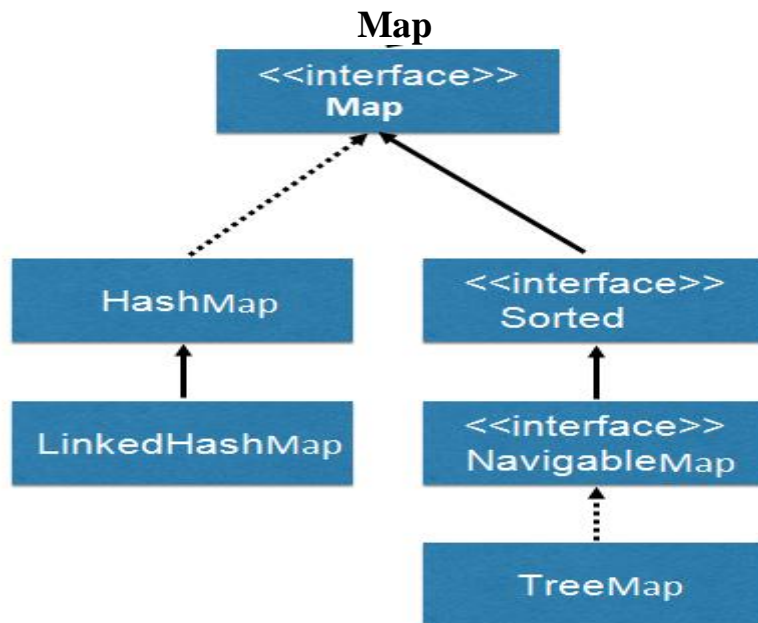
op-

```
size =6
java
android
bigdata
size =3
```

### Use of peek ()

```
import java.util.LinkedList;
public class Mainclass {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<String>();
        list.add("java");
        list.add("android");
        list.add("bigdata");
        System.out.println("size =" + list.size());
        for(int i=0; i<list.size();i++)
        {
            String s2 = list.peek();
            System.out.println(s2);
            System.out.println("size =" +list.size());
        }
    }
}
```

```
size =3
java
java
java
size =3
```



key	value
name	phone
a	1425
b	525
a	2536

- Within the map data is stored in key value format.
- To add the element we can use put (key, value) method and to retrieve the element we have to use get(key) method.

### Program

```

package com.ds.map;
import java.util.HashMap;
import java.util.Set;
import java.util.function.BiConsumer;

public class Mainclass {

    public static void main(String[] args)
    {
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();
        hm1.put("aa", 1234); // use of put method
        hm1.put("bb", 1425);
        hm1.put("cc", 4528);
        hm1.put("dd", 4568);
    }
}
  
```

```
System.out.println("-----");  
Set <String> keys=hm1.keySet(); // to store all keys inside a set
```

```
for(String str : keys) //using of advance for loop  
{  
System.out.println("key =" + str + " " + "value = " + hm1.get(str));  
}}}
```

**op –**

**key =aa value = 1234**

**key =bb value = 1425**

**key =cc value = 4528**

**key =dd value = 4568**

- If you pass an invalid key for get() then get() returns null values.

Program

```
package com.ds.map;  
import java.util.HashMap;  
public class Mainclass {  
    public static void main(String[] args)  
    {  
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();  
        hm1.put("aa", 1234);  
        hm1.put("bb", 1425);  
        hm1.put("cc", 4528);  
        hm1.put("dd", 4568);  
        System.out.println(hm1.get("aaa"));  
    }  
}
```

**op- null**

- Map do not allow duplicates. If you try to add duplicates, then the old value will be replaced with new value for the given key.

Program

```
package com.ds.map;  
import java.util.HashMap;  
public class Mainclass {  
    public static void main(String[] args)  
    {  
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();  
        hm1.put("aa", 1234);  
        hm1.put("aa", 124252);  
        System.out.println(hm1.get("aa"));  
    }  
} op – 124252
```

- Map do not have an index.

### Program

```
package com.ds.map;
import java.util.HashMap;
import java.util.function.BiConsumer;
public class Mainclass {
    public static void main(String[] args)
    {
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();
        hm1.put("ab", 1234);
        hm1.put("bdsa", 124252);
        hm1.put("ccd", 1242);
        hm1.put("dd", 14252);
        hm1.put("edf", 21454252);
        BiConsumer<String, Integer> c1= (String key, Integer value) ->
        {
            System.out.println("key="+ key+" "+"value = "+value);
        };
        hm1.forEach(c1);    }}
```

op -                   key =dd value = 14252  
                           key =ab value = 1234  
                           key =ccd value = 1242  
                           key =edf value = 21454252  
                           key =bdsa value = 124252

- Key can not be duplicates.
- Values can be duplicates.

```
import java.util.HashMap;
import java.util.function.BiConsumer;
public class Mainclass {
    public static void main(String[] args)
    {HashMap<String, Integer> hm1 = new HashMap<String, Integer>();
        hm1.put("ab", 1234);
        hm1.put("bb", 1234);
        BiConsumer<String, Integer> c1= (String key, Integer value) ->
        {
            System.out.println("key="+ key+" "+"value = "+value);
        };
        hm1.forEach(c1);
    }}
```

op -   key =bb value = 1234  
           key =ab value = 1234

- null can be used as key and values both.

Program

```
package com.ds.map;
import java.util.HashMap;
import java.util.function.BiConsumer;
public class Mainclass {
    public static void main(String[] args)
    {
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();
        hm1.put("a", null);
        hm1.put(null, 1425);
        System.out.println(hm1.get("a"));
        System.out.println(hm1.get(null));
    }
}
```

**op- values = null**  
**values = 1425**

Simple Program

```
package com.ds.map;
import java.util.HashMap;
import java.util.Set;
import java.util.function.BiConsumer;
public class Mainclass {
    public static void main(String[] args)
    {
        HashMap<String, Integer> hm1 = new HashMap<String, Integer>();
        hm1.put("aa", 1234);
        hm1.put("bb", 1425);
        hm1.put("cc", 4528);
        hm1.put("dd", 4568);
        BiConsumer<String, Integer> c1 = (String key, Integer value) ->
        {
            System.out.println("key =" + key + " " + "value = " + value);
        };
        hm1.forEach(c1);
        System.out.println("-----");
        //or we can also do like

        hm1.forEach( (String key, Integer value) ->
        System.out.println("key =" + key + " " + "value = " + value));

        //or we can also do like
```

```

System.out.println("-----");
Set <String> keys=hm1.keySet();

for(String str : keys)
{
    //      System.out.println(hm1.get(str));
    System.out.println("key="+ str+" "+"value = "+hm1.get(str));
}
}

```

**op-**

**key =aa value = 1234**

**key =bb value = 1425**

**key =cc value = 4528**

**key =dd value = 4568**

-----

**key =aa value = 1234**

**key =bb value = 1425**

**key =cc value = 4528**

**key =dd value = 4568**

-----

**key =aa value = 1234**

**key =bb value = 1425**

**key =cc value = 4528**

**key =dd value = 4568**



## WRAPPER CLASSES

- Wrapper classes are used to convert primitive to java objects.
- Wrapper classes provides many inbuilt methods to perform basic operations on the given primitive values.
- For every primitive datatype one corresponding java class is present at java.lang package.

### Boxing

- Converting a primitive value to wrapper class object using new operator is called as Boxing.

Program

```
package com.jsp.wrapperclasses;
```

```
public class Mainclass {
```

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

```
        int x1=10;
```

```
        double y1=22.13;
```

```
        Integer i1=new Integer(x1); // boxing
```

```
        Double d1=new Double(y1); // boxing
```

```
        System.out.println("x1 = "+x1);
```

```
        System.out.println("i1="+i1);
```

```
        System.out.println("y1 = "+y1);
```

```
        System.out.println("d1 = "+d1);
```

```
    }
```

```
}
```

**op –**

**x1 = 10**

**i1=10**

**y1 = 22.13**

**d1 = 22.13**

## Auto-Boxing

- Converting a primitive value to wrapper class object without using new operator is called auto-boxing.

### Program

```
public class Mainclass {  
    public static void main(String[] args) {  
        int x1=10;  
        double y1=22.13;  
        Integer i1=x1; // auto-boxing  
        Double d1=y1; // auto-boxing  
        System.out.println("x1 = "+x1);  
        System.out.println("i1="+i1);  
        System.out.println("y1 = "+y1);  
        System.out.println("d1 = "+d1);  
    }  
}
```

op –

**x1 = 10**

**i1=10**

**y1 = 22.13**

**d1 = 22.13**

## Unboxing

- Converting wrapper class object back to primitive value using value methods is called as unboxing.

### Program

```
public class Mainclass {  
    public static void main(String[] args) {  
        int x1=10;  
        Integer i1=x1; // auto-boxing  
        int x3=i1.intValue(); // unboxing  
        System.out.println("x1 = "+x1);  
        System.out.println("i1="+i1);  
        System.out.println("x3 = "+x3);  
    }  
}
```

op-

**x1 = 10**

**i1=10**

**x3 = 10**

## Auto-unboxing

- Converting wrapper class object to primitive value without using value methods is called as auto-boxing.

## Program

```
package com.jsp.wrapperclasses;
```

```
public class Mainclass {  
    public static void main(String[] args) {  
  
        double y1=22.13;  
        Double d1=y1; // auto-boxing  
        Double d4=d1; // auto-unboxing without using new operator  
  
        System.out.println("y1 = "+y1);  
        System.out.println("d1 = "+d1);  
        System.out.println("d4 = "+d4);  
    }  
}
```

```
op-  
y1 = 22.13  
d1 = 22.13  
d4 = 22.13
```

- toString() of object class is overridden in every wrapper class which returns the value present in the given object.
- If you try to assign a primitive value to object class reference then jvm performs two operations –
  - (i) It creates an wrapper class object for the given primitive value (auto-boxing)
  - (ii) The wrapper class object will be upcasted to object class reference.

## Program

```
public class Mainclass2 {  
    public static void add(Object i1, Object i2)  
    {  
        System.out.println("this is add method");  
    }  
    public static void disp(Integer i1, Integer i2)  
    {  
        System.out.println("this is disp()");  
    }  
}
```

```

public static void main(String[] args) {

    // Integer i1 = new Integer(10);
    // Object obj=i1; // upcasting // we can also write like that

    Integer in1=new Integer(10);
    Integer in2 = new Integer(20);
    add(in1,in2);
    disp(in1,in2);

}
}

```

**op-**

**this is add method**

**this is disp()**

**Question : Can we store primitive values within the collection.**

**Ans :** No, if you try to add primitive value to the collection then JVM

- creates a wrapper class object for the given primitive.
- Upcast that wrapper class object to object class references and this upcasted references will be stored within the collection.

**Program**

```

public class Mainclass2 {
    public static void disp(Integer i1, Integer i2)
    {
        System.out.println("this is disp()");
    }
    public static void main(String[] args) {

        disp(10,20); // here we pass the primitive to wrapper class

    }
}

```

**op – this is disp()**

- Using a wrapper class we can convert given string value to primitive value with the help of parse methods.

#### Program

```
package com.jsp.wrapperclasses;
```

```
public class Mainclass3 {
```

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

```
        String n1="100";
```

```
        String n2= "200";
```

```
        System.out.println("res = " +n1+n2);
```

```
        int i1=Integer.parseInt(n1);
```

```
        int i2 = Integer.parseInt(n2);
```

```
        System.out.println("res = " +(i1+i2));
```

```
    }}
```

**op-**

**res = 100200**

**res = 300**

- String class provides a static methods called valueOf() which converts any given primitive value to String object.

#### Program

```
package com.jsp.wrapperclasses;
```

```
public class Mainclass3 {
```

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

```
        int a = 150;
```

```
        int b = 150;
```

```
        System.out.println("res = " +(a+b));
```

```
        String s1 = String.valueOf(a);
```

```
        String s2 = String.valueOf(b);
```

```
        System.out.println("res = " +s1+s2);
```

```
    }
```

```
}
```

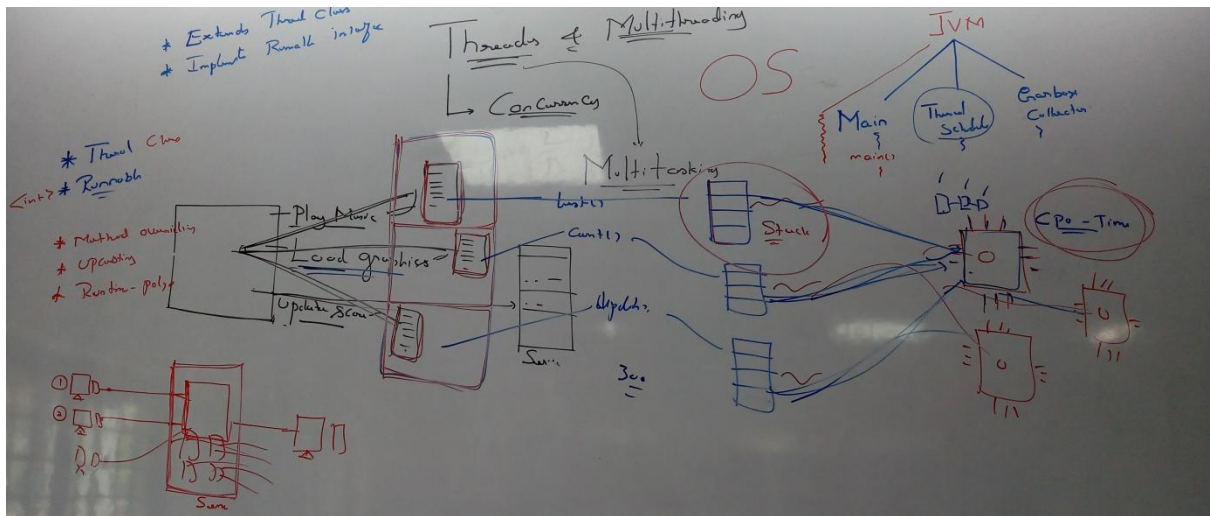
**op –**

**res = 300**

**res = 150150**

# Thread & MultiThreading

- Thread is an independent part of same program which gets its own stack and cpu-time for the execution.
- Threads are used to achieve program level multitasking and concurrency.
- Whenever jvm starts the execution it will create three threads.
  - (i) Main Thread
  - (ii) Thread Scheduler
  - (iii) Garbage collector



- By default all the programs in java will be executed in main thread.
- To create threads in java, we have two approaches.
  - i) Extending thread class.
  - ii) implementing runnable interface.

## Extending Thread Class

- Create a class extending thread class.
- Override run method of thread class in subclass and write the business logics that has to be executed concurrently.
- create the object of subclass and using the subclass object call **Start** method.
- If you call run method from the subclass object then you always execute all the business logics in run method within the main thread.
- Start method creates a new thread and calls run method implicitly.

## Program

```
package com.threads;
```

**//Create a class extending thread class.**

```
class ThreadOne extends Thread
```

```
{
```

**//Override run method of thread class in subclass and write the business logics that has to be executed concurrently.**

```
    @Override
```

```
    public void run() {
```

```
        System.out.println("executing task one ");
```

```
    }
```

```
}
```

```
class ThreadTwo extends Thread
```

```
{
```

```
    @Override
```

```
    public void run() {
```

```
        System.out.println("executing task two");
```

```
    }
```

```
}
```

```
public class Mainclass {
```

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

```
        System.out.println("Program starts");
```

**// create the object of subclass and using the subclass object call Start method.**

```
        ThreadOne t1 = new ThreadOne();
```

```
        ThreadTwo t2 = new ThreadTwo();
```

```
        t1.start(); // start method internally call run method and execute business logic
```

```
        t2.start();
```

```
        System.out.println("Program end");
```

```
    }}
```

**op –**

**Program starts**

**Program end**

**executing task one**

**executing task two**

**note – here output change every time, because start method provide a cpu time to every thread, in which time cpu is free the thread is executed.**

## Implementing Runnable Interface

- Runnable is a functional interface and supports lambda function.
- create a lambda function for runnable interface.
  - i) create the object of thread class and pass the lambda function references to the constructor of thread class.
  - ii) use the thread object and call start method.
  - iii) Start method is the method of Thread class only.

### Program

```
package com.threads;
public class Mainclass2 {
    public static void main(String[] args) {

        System.out.println("program start");
        //create a lambda function for runnable interface.
        Runnable r1 = () ->
        {
            System.out.println("executing task one ");
        };
        Runnable r2 = () ->
        {
            System.out.println("executing task two ");
        };

        create the object of thread class and pass the lambda function references to
        the constructor of thread class.

        Thread t1 = new Thread(r1);
        t1.start(); use the thread object and call start method.
        Thread t2 = new Thread(r2);
        t2.start();
        System.out.println("Program end");

    }
}
```

op –  
Program starts  
Program end  
executing task one  
executing task two

note – here output change every time, because start method provide a cpu time to every thread, in which time cpu is free the thread is executed.



- **Question : Why runnable interface was introduced in java.**

Ans : - If a class is already extending another super class then it is not possible to extend thread class because it leads to multiple inheritance and multiple inheritance is not supported in java.

### **Every Thread will have three property**

- Name
- id
- priority

### **Name :**

- The name of the thread can be assign by the programmer to identify every thread uniquely by the programmer.

Program

```
package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {
        System.out.println("program start");
        Runnable r1 = () ->
        {
            System.out.println("executing task one ");
        };
        Runnable r2 = () ->
        {
            System.out.println("executing task two ");
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        String t1Name = t1.getName();
        System.out.println("t1 name default name = " +t1Name);

        String t2Name = t2.getName();
        System.out.println("t2 name default name = " +t2Name);

        t1.setName("task 1 ");
        String t3Name = t1.getName();
        System.out.println("t3 name user define name = " +t3Name);

        t2.setName("tast 2");
        String t4Name = t2.getName();
        System.out.println("t4 name user definmed name = " +t4Name);
    }
}
```

```

        t1.start();
        t2.start();
    }}

```

**op-**  
**program start**  
**t1 name default name = Thread-0**  
**t2 name default name = Thread-1**  
**t3 name user define name = task 1**  
**t4 name user definmed name = tast 2**  
**executing task one**  
**executing task two**

**Id :**

- The id of the thread will be assigned by the thread scheduler and hence it can not be changed by the programmer.

Program

```

package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {

        Runnable r1 = () ->
        {
        };
        Runnable r2 = () ->
        {
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        long t1Id = t1.getId();
        System.out.println("t1 id = " +t1Id);

        long t2Id = t2.getId();
        System.out.println("t2 id = " +t2Id);

        t1.start();
        t2.start();
    }}

    op -
    t1 id = 12
    t2 id = 13

```

## Priority :

- The priority of the thread helps the thread scheduler to order the execution of the threads.
- The priority of the thread can be changed by using setPriority method.
- The priority of the thread should be always within the range of 1 to 10, where 1 is the lowest priority, 5 is the normal priority and 10 is the highest priority.

Program

```
package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {
        System.out.println("program start");

        Runnable r1 = () ->
        {
            System.out.println("executing task one ");
        };
        Runnable r2 = () ->
        {
            System.out.println("executing task two ");
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        int t1P = t1.getPriority(); // default priority is 5
        System.out.println("t1 Prioiry = " +t1P);

        int t2P = t2.getPriority();
        System.out.println("t2 Prioiry = " +t2P);

        t2.setPriority(8); // t2 executed first
        t1.start();
        t2.start();
    }
}
```

op –

program start

t1 Prioiry = 5

t2 Prioiry = 5

executing task two

executing task one

- To Pause the execution of thread we have three methods in thread class.
  - yield()
  - sleep()
  - join()

### yield()

- yield() will pause the execution of current thread and gives the chance for another thread which is having same or higher priority.
- It is a static method calling using **Thread.yield()**.
- we use when we don't know the time.

### Program

```
package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {
        System.out.println("program start");
        Runnable r1 = () ->
        {
            Thread.yield();
            for(int i =1; i<3;i++)
            {
                System.out.println("executing task one ");
            }
        };
        Runnable r2 = () ->
        {
            for(int i =1; i<3;i++)
            {
                System.out.println("executing task two ");
            }
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}
```

op –  
 program start  
 executing task two  
 executing task two  
 executing task one  
 executing task one

### sleep() :

- sleep() will pause the execution of given thread for the specified time limit in millisecond and gives the chance for the other thread which is having same or higher priority.
- here in this program there is a limit of 1 second.

### Program

```
package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {
        System.out.println("program start");
        Runnable r1 = () ->
        {
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            for(int i =1; i<3;i++)
            {
                System.out.println("executing task one ");
            }
        };
        Runnable r2 = () ->
        {
            for(int i =1; i<3;i++)
            {
                System.out.println("executing task two ");
            }
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

op –  
program start  
executing task two  
executing task two  
executing task one  
executing task one

### join() :

- join() will pause the execution of all threads until the given thread completes the execution.
- join() is a non static method of thread class and it should be called only from the thread object.
- join() is used after the start method.

### Program

```
package com.threads;
public class Mainclass4 {
    public static void main(String[] args) {
        System.out.println("program start");

        Runnable r1 = () ->
        {
            for(int i=1; i<3;i++)
            {
                System.out.println("executing task one ");
            }
        };
        Runnable r2 = () ->
        {
            for(int i=1; i<3;i++)
            {
                System.out.println("executing task two ");
            }
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

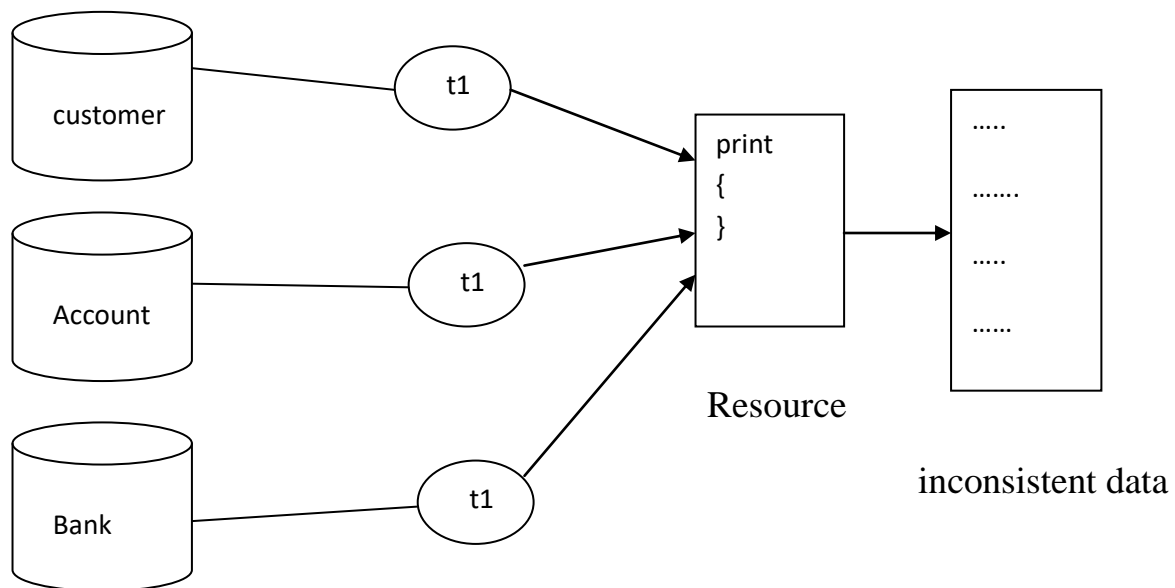
        t1.start();
        t2.start();

        try {
            t1.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Program end");
    }
}
```

op -                      program start  
                               executing task one  
                               executing task two  
                               executing task two  
                               executing task one  
                               Program end

output changes

### Race Condition

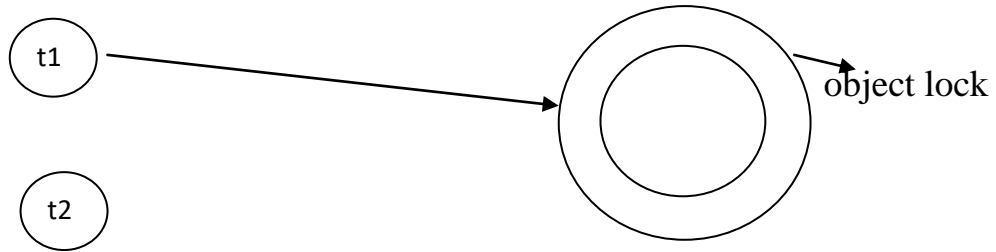


### Race Condition:

- Multiple threads trying to access same resources at the same time is called as race condition.
- It leads data inconsistency.
- Race condition can be avoided with the help of synchronized keyword.

### Thread Synchronization:

- If a method is declared with synchronized keyword then the given method can be accessed by only one thread at any given point of time.
- Only method of the class can declared as synchronized and variable can not be declared as synchronized.
- If a thread tries to access a synchronized method then the thread acquires object lock on the given object. Once a thread acquires object lock other threads will not be able to access the same object at the same time.



### Program without synchronized

expected output for this question

t1 = 1    t2 = 1

t2 =0    t1 = 0

Program

**package** com.threads;

**class** Count

{

**static int** value = 0;

**public void** increment()

    {

        value++;                    }

**public void** decrement()

    {

        value--;                   }

**public void** showValue()

    {

        System.out.println("value = "+value);

    }}

**public class** Mainclass5 {

**public static void** main(String[] args) {

        System.out.println("program start");

        Count c1=new Count();

        Runnable r1 = () ->

        {

            c1.increment();

            c1.showValue();            };

        Runnable r2 = () ->

        {

            c1.decrement();

            c1.showValue();            };

        Thread t1 = new Thread(r1);

        Thread t2 = new Thread(r2);

        t1.start();

        t2.start();    } }



**op –**  
**program start**  
**value = 0**  
**value = 0**

**Output will be changed. But the above output is not expected**

**Program with synchronized keyword**

```
package com.threads;
class Count
{
    static int value = 0;
    public synchronized void increment()
    {
        value++;
    }
    public void decrement()
    {
        value--;
    }
    public synchronized void showValue()
    {
        System.out.println("value = "+value);
    }
}

public class Mainclass5 {
    public static void main(String[] args) {
        System.out.println("program start");
        Count c1=new Count();
        Runnable r1 = () ->
        {
            c1.increment();
            c1.showValue();
        };
        Runnable r2 = () ->
        {
            c1.decrement();
            c1.showValue();
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```

**op-**  
**program start**  
**value = 1**  
**value = 0**

## Synchronized Block

- Synchronized blocks are used to perform thread safe operations on the object which are not threadsafe.
- Synchronized blocks locks the given objects until all the codes written in the body of synchronized blocks are executed.
- Syntax :

```
synchronized (ref_var of object)
{
    statement ;
}
```

## Program

```
package com.threads;
public class Mainclass6 {
    static String s1= new String("hi");
    public static void main(String[] args) {
        System.out.println("program start");
        Runnable r1 = () ->
        {
            System.out.println("t1 waitingg to lock s1");
            synchronized(s1)
            {
                System.out.println("t1 locked to s1");
                s1=s1+"java";
                System.out.println("s1 = "+s1);
            }
            System.out.println("t1 released lock on s1");
        };
        Runnable r2 = () ->
        {
            System.out.println("t2 waiting to lock s1");
            synchronized(s1)
            {
                System.out.println("t2 locked to s1");
                s1=s1+"hello";
                System.out.println("s1 = "+s1);
            }
            System.out.println("t2 released lock on s1");
        };
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();    }}
```

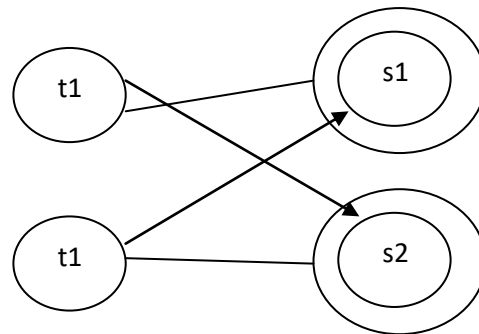
```

output
program start
t1 waiting to lock s1
t2 waiting to lock s1
t1 locked to s1
s1 = hijava
t1 released lock on s1
t2 locked to s1
s1 = hijavahello
t2 released lock on s1

```

## Deadlock

- Deadlock is a situation where thread one is waiting to acquire the object lock of thread two and thread two is waiting to acquire object lock on thread one and both threads wait for infinite period of time.



- Deadlock can be overcome with the help of **Interthread Communication (ITC)**.
- **Interthread Communication** can be done with the help of wait() and notify() of object class.

## Program for deadlock

```

package com.threads;
public class Mainclass7 {

    static String s1= new String("hi");
    static String s2= new String("android");

    public static void main(String[] args) {
        System.out.println("program start");

        Runnable r1 = () ->
        {
            System.out.println("t1 waiting to lock s1");
            synchronized(s1)

```

```

{
    System.out.println("t1 locked to s1");

    System.out.println("t1 waiting to lock s2");
    synchronized(s2)
    {
        System.out.println("t1 locked to s2");
        System.out.println("t1 released lock on s2");
    }
}
System.out.println("t1 released lock on s1");
};
Runnable r2 = () ->
{
    System.out.println("t2 waiting to lock s2");
    synchronized(s2)
    {
        System.out.println("t2 locked to s2");
        synchronized(s1)
        {
            System.out.println("t2 locked to s1");
            System.out.println("t2 released lock on s1");
        }
    }

    System.out.println("t2 released lock on s2");
};
Thread t1 = new Thread(r1);
Thread t2 = new Thread(r2);

t1.start();
t2.start();

}}

```

**op-**

**program start**

**t1 waiting to lock s1**

**t1 locked to s1**

**t1 waiting to lock s2**

**t2 waiting to lock s2**

**t2 locked to s2**

**output will be changed.**

### **wait() :**

- wait() will pause the execution of the thread and also releases all the object locks the given thread is holding.
- wait() can be called only from synchronized block or synchronized method.
- If you call wait() from a non-synchronized method or non-synchronized block then JVM throws **IllegalMonitorStateException**

### **notify()**

- notify() is used to communicate termination state of one thread to another thread.

### **notifyAll()**

- This methods is used to communicate the termination status of one thread to every other thread of the given programmes.

### **Program**

```
package com.threads;
public class Mainclass8 {
    static String s1= new String("hi");
    static String s2= new String("android");

    public static void main(String[] args) {
        System.out.println("program start");

        //s1.wait(); // we can not call it from non-synchronized method
        Runnable r1 = () ->
        {
            System.out.println("t1 waiting to lock s1");
            synchronized(s1)
            {
                System.out.println("t1 locked to s1");

                try {
                    System.out.println("t1 going to wait state ");
                    s1.wait();
                }
                catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
            System.out.println("t1 waiting to lock s2");
        }
    }
}
```

```

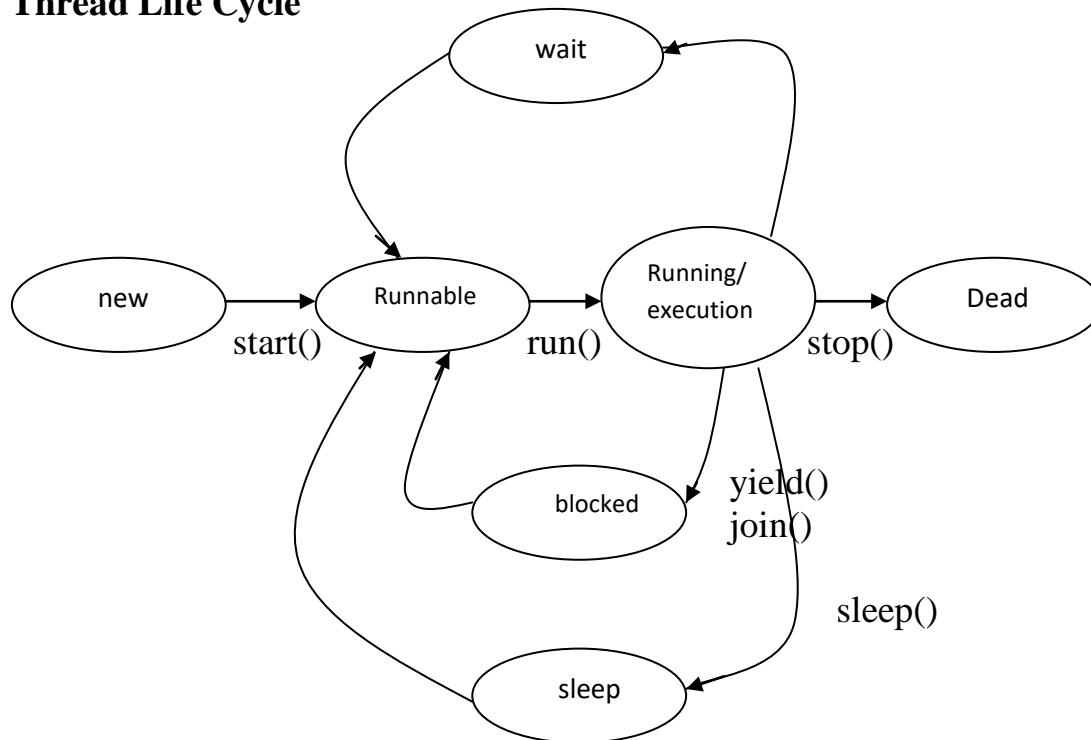
        synchronized(s2)
        {
            System.out.println("t1 locked to s2");
            System.out.println("t1 released lock on s2");
        }
        System.out.println("t1 released lock on s1");
    };
    Runnable r2 = () ->
    {
        System.out.println("t2 waiting to lock s2");
        synchronized(s2)
        {
            System.out.println("t2 locked to s2");
            synchronized(s1)
            {
                System.out.println("t2 locked to s1");
                System.out.println("t2 released lock on s1");
                s1.notify();
            }
        }
        System.out.println("t2 released lock on s2");
    };
    Thread t1 = new Thread(r1);
    Thread t2 = new Thread(r2);

    t1.start();
    t2.start();
}

```

**op**  
**program start**  
**t1 waiting to lock s1**  
**t1 locked to s1**  
**t1 going to wait state**  
**t2 waiting to lock s2**  
**t2 locked to s2**  
**t2 locked to s1**  
**t2 released lock on s1**  
**t1 waiting to lock s2**  
**t2 released lock on s2**  
**t1 locked to s2**  
**t1 released lock on s2**  
**t1 released lock on s1**

## Thread Life Cycle



## Differences

<b>Run()</b>	<b>Sleep</b>
1- Run() executes all the business logic written in main thread	1- It creates a new thread and executes all the business logics of run() in the new thread
2- Run method should be overridden to write the business logics.	2- It can be overridden but not recommended because all the business logic will be executed in main thread.

<b>Sleep()</b>	<b>wait()</b>
1- sleep() pause the execution and does not release the object lock	1- wait() pause the execution of the thread and release the object lock.
2- it can not be used for Inter Thread Communication	2- It should be used for Inter Thread Communication
3- sleep() is present in thread class	3- wait() is present in object class

## Garbage Collector

- Garbage collector is used to remove the unused objects which are de-referenced from the heap area.
- Garbage collector can be called explicitly by using `System.gc()`.
- `finalize()` of object class will be called implicitly by the garbage collector just before the object is removed from the heap area.
- If you want to execute any lines of codes just before the object is removed from heap area, then we should override `finalize` of object class.
- It is always a best practice to re-initialize referenced variables to **NULL**, once the uses of object is over.
- Garbage collector only clear the de-referenced object.

```
package com.jsp.garbagecollector;
class Student{
    String name = "Smith";
    int id=1234;
    @Override
    protected void finalize()
    {
        System.out.println("Object removed....");
    }
}
public class Mainclass {
    public static void main(String[] args) {
        System.out.println("Program starts....");

        Student s1= new Student();
        System.out.println(s1.id + " "+s1.name);
        s1=null; // dereference
        System.gc();
        for(int i=1; i<=5;i++)
        {
            System.out.println(i);
        }
        System.out.println("Program ends....");
    }
}
```

```
op -      Program starts....
          1234 Smith
          1
          2
          3
          Object removed....
          Program ends....
```



- If we use object after calling garbage collector then it show **java.lang.NullPointerException**

### Program

```
package com.jsp.garbagecollector;
class Student{
    String name = "Smith";
    int id=1234;
    @Override
    protected void finalize()
    {
        System.out.println("Object removed....");
    }
}
public class Mainclass {
    public static void main(String[] args) {
        System.out.println("Program starts....");

        Student s1= new Student();
        System.out.println(s1.id + " "+s1.name);
        s1=null; // dereference
        System.gc();
        System.out.println(s1.id + " "+s1.name); //returns a runtime error

        for(int i=1; i<=5;i++)
        {
            System.out.println(i);
        }
        System.out.println("Program ends....");
    }
}
```

op-

Program starts....

1234 Smith

Object removed....

Exception in thread "main"

java.lang.NullPointerException

at

com.jsp.garbagecollector.Mainclass.main(Mainclass.jav  
a:23)

## File Handling

- **File** : File is an entity which contains information.
- **Folder/Directory** : It is a container where file are stored.
- **File System** : It is a logical arrangement done by the operating system to manage files and folders.
- **File Handling** : Writing java programs to manage or create, delete, read, write etc. the files is called file handling.

### Creating a Folder

#### Program

```
import java.io.File;
public class Mainclass {
    public static void main(String[] args) {

        File f1 = new File("G:\\JavaFile"); // create a folder

        if(f1.exists()==false) // to check files is already present or not
        {
            f1.mkdir(); // for make directory
            System.out.println("folder created....");
        }
        else
        {
            System.out.println("folder is already exists");
        }
    }
}
```

op – folder created....

### Creating a new file

#### Program

```
import java.io.File;
public class Mainclass {
    public static void main(String[] args) {
        File f1 = new File("G:\\JavaFile/File1.txt"); // create a file
        if(f1.exists()==false) // to check files is already present or not
        {
            try {
                f1.createNewFile(); //create the file
                System.out.println("file created");
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

else
{
    System.out.println("folder is already exists");
}}

```

**op- file created**

**Write the data to file**

```

import java.io.FileWriter;
import java.io.IOException;
public class Mainclass3 {
    public static void main(String[] args) {
        String path = "G:\\JavaFile/File1.txt";
        FileWriter fw=null;
        try {
            fw = new FileWriter(path);
            fw.write("this is file handling classes");
            fw.flush(); // without this method file is not write inside a file

            System.out.println("writing data completed....");
        }
        catch (IOException e) {

            System.out.println("error in writing data to file ");
            e.printStackTrace();
        }
        finally //these blocks executed even exception is occurred or not
        { //used to execute the code irrespective of occurrence of exception
            try {
                fw.close();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}

```

**op- writing data completed....**

## Reading data from file

### Program

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class Mainclass4 {
    public static void main(String[] args) {

        String path = "G:\\JavaFile/File1.txt";
        File f1 = new File(path);
        FileReader fr=null;
        try {
            fr = new FileReader(path);
            int size = (int) f1.length(); // return long
            char [] c1 = new char [size];
            fr.read(c1);
            System.out.println(c1);
            System.out.println("reading data completed....");
        }
        catch (IOException e) {
            System.out.println("error in reading data to file ");
            e.printStackTrace();
        }
        finally //these blocks executed even exception is occurred or not
        { //used to execute the code irrespective of occurrence of exception
            try {
                fr.close();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

**op-**

**this is file handling classes  
reading data completed....**

## **File Class**

- The file class allows the user to create files and folders in the system.

### **Methods of File Class**

- (i) **mkdir()** – make directory - This method creates a new folder of directory in the specified location.
- (ii) **exists()** – This method checks if the given file or folder is present in the specified location and returns true, if the file or folder already exists, else it return false.
- (iii) **createNewFile()** – This method creates a new file at the specified location.
- (iv) **length()** – This method returns number of characters present in the given file. The return type of this method is long.

## **FileWriter class**

- FileWriter helps the programmer to write the data to the file specified in the location.

### **Method of FileWriter**

- (i) **write ()** – This methods writes all the data with the output stream.
- (ii) **flush()** – This method writes all the data from the stream to the given file.

**Note :** The stream should be closed by the programmer explicitly by using **close()** methods. It always a good practice to call close methods of the stream within the finally blocks.

## **FileReader class**

- FileReader reads the data from the file one character at a time.

### **Methods of FileReader**

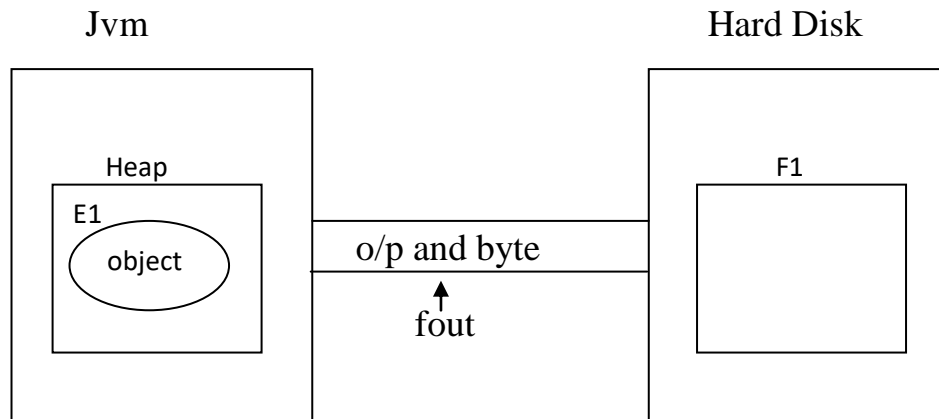
- (i) **read()** – The read() reads one character at a time and stores it within the given character array.  
To pass the character Array we have to create a character array matching the number of characters present in given file.

## **Finally block :**

- It is a block of code which will be executed irrespective of occurrence of exception, means exception occurred or not occurred.
- Finally block should be always written after catch block.

## Serialization

- The process of converting java object into stream of byte is known as serialization.



### There are two types of Stream :

- InputStream
- OutputStream

### There are two types of value present

- Char stream
- Byte stream

### Steps to Achieve Serialization

1. Establish the o/p and byte stream using FileOutputStream class.
2. Create an object for ObjectOutputStream for converting java object into byte stream.
3. Invoke writeObject method.
4. Close the connection which is written in finally block.

### Program

//serialization

**package** com.jsp.file\_handling;

**import** java.io.File;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**class** Employee **implements** Serializable

{

**int** id;

```

String name;
public Employee(int id, String name)
{
    super();
    this.id=id;
    this.name=name;
}
}
public class Mainclass5 {
    public static void main (String ar[])
    {
        Employee e1 = new Employee(123,"Abc");
        FileOutputStream fout=null;
File f1 = new File ("G:\\JavaFile/emp.ser"); // extension for serializable file =
ser

        try {
            f1.createNewFile();

            fout=new FileOutputStream(f1); // creating output byte stream
ObjectOutputStream out = new ObjectOutputStream(fout); // object stream
out.writeObject(e1); // used to convert java object into stream of bytes and
make it to transfer0
            System.out.println("Object written");
        }
        catch (IOException e) {

            e.printStackTrace();
        }
        finally
        {
            try {
                fout.close();
            } catch (IOException e) {

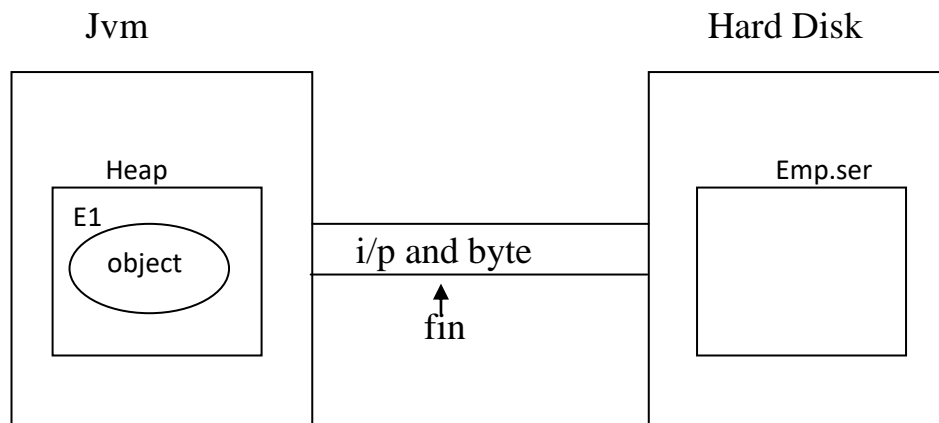
                e.printStackTrace();
            }
        }
    }
}

```

Op –

**Object written**

## Deserilization



## Constructor also throws an exception

### Steps to Achieve Serialization

1. Establish the input & byte stream.
2. Create an object for object inputStream.
3. Invoke read object method.
4. Close the connection.

### Use of Marker interface

- To make class eligible for any operation such as serializable, clonable is a marker interface which helps class to perform serialization.

### Program

//De-serialization

```
package com.jsp.file_handling;
```

```
import java.io.File;
```

```
import java.io.FileInputStream;
```

```
import java.io.IOException;
```

```
import java.io.ObjectInputStream;
```

```
import java.io.Serializable;
```

```
class Employee implements Serializable
```

```
{
```

```
    int id;
```

```
    String name;
```

```
    public Employee(int id, String name)
```

```
    {
```

```
        super();
```

```
        this.id=id;
```

```
        this.name=name;
```

```
    }
```

```
}
```



```

public class Mainclass6 {

    public static void main (String ar[])
    {

File f1 = new File ("G:\\JavaFile/emp.ser"); // extension for serializable file =
ser

        FileInputStream fin=null;
        try
        {
            fin=new FileInputStream(f1);
ObjectInputStream in = new ObjectInputStream(fin); // object stream
            Object ob=in.readObject();
            Employee e = (Employee)ob;
            System.out.println(e.id);
            System.out.println(e.name);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        finally
        {
            try {
                fin.close();
            } catch (IOException e) {

                e.printStackTrace();
            }
        }
    }
}

Op -
123
Abc

```

## Object Cloning

- Creating an object from an existing object with the same data is called as object cloning.
- If you want to clone an object then the class should implements cloneable interface(which is marker interface).
- You should also override clone method of object class to perform cloning.
- **Copying all the data from existing object to cloned object is called as deep-copying.**

## Program

```
package com.jsp.clone;
class Emp extends Object implements Cloneable
{
    String name;
    int id;
    Double salary;
    Emp(String name, int id, Double salary){
        this.name=name;
        this.id=id;
        this.salary=salary;
    }
    @Override
    public String toString()
    {
        String info = id+ " "+name+ " "+salary;
        return info; }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class Mainclass2 {
    public static void main (String ar[])
    {
        Emp e1 = new Emp("Blake", 123, 2314.14);
        System.out.println("HashCode =" +e1.hashCode());
        System.out.println(e1);
        try
        {
            Emp e2 = (Emp) e1.clone();
            System.out.println("HashCode =" +e2.hashCode());
            System.out.println(e2);
        }
    }
}
```

```

    catch (CloneNotSupportedException e)
    {
        e.printStackTrace();
    }}

```

**Op-**

**HashCode =792791759**

**123 Blake 2314.14**

**HashCode = 434176574**

**123 Blake 2314.14**

- Copying only required data from the existing object to cloned object is called as shallow copying.

**Program**

**package** com.jsp.clone;

**class** Emp **extends** Object **implements** Cloneable

```

{
    String name;
    int id;
    Double salary;
    Emp(String name, int id, Double salary){

        this.name=name;
        this.id=id;
        this.salary=salary;
    }

    @Override
    public String toString()
    {
        String info = id+ " "+name+ " "+salary;
        return info;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException
    {
        Emp emp = new Emp (name, 0, salary);
        return emp;
    }
}

```

```

public class Mainclass2 {

    public static void main (String ar[])
    {
        Emp e1 = new Emp("Blake", 123, 2314.14);
        System.out.println("Hashcode =" +e1.hashCode());
        System.out.println(e1);
        try
        {
            Emp e2 = (Emp) e1.clone();
            System.out.println("Hashcode = " +e2.hashCode());
            System.out.println(e2);
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}

```

**Op-**  
**Hashcode =792791759**  
**123 Blake 2314.14**  
**Hashcode = 434176574**  
**0 Blake 2314.14**

- If you try to clone an object using clone method without implementing clonable interface then JVM throws cloneNotSupportedException.

### Program

```

package com.jsp.clone;
class Emp extends Object // here we not implement clonable interface
{
    String name;
    int id;
    Double salary;
    Emp(String name, int id, Double salary){

        this.name=name;
        this.id=id;
        this.salary=salary;
    }
    @Override
    public String toString()
    {
        String info = id+ " "+name+ " "+salary;
        return info;
    }
}

```

```

@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
}
public class Mainclass2 {

    public static void main (String ar[])
    {
        Emp e1 = new Emp("Blake", 123, 2314.14);
        System.out.println("HashCode =" +e1.hashCode());
        System.out.println(e1);
        try
        {
            Emp e2 = (Emp) e1.clone();
            System.out.println("HashCode = " +e2.hashCode());
            System.out.println(e2);
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}

```

Op-

HashCode =792791759

123 Blake 2314.14

[java.lang.CloneNotSupportedException:](#)

[com.jsp.clone.Emp](#)

[at java.base/java.lang.Object.clone \(Native](#)

[Method\)](#)

[at com.jsp.clone.Emp.clone \(Mainclass2.java:23\)](#)

[at](#)

[com.jsp.clone.Mainclass2.main \(Mainclass2.java:35\)](#)

- Cloneable is a marker interface which is used to mark the permission as the object can be cloned by JVM.

### Normal Program

```
package com.jsp.clone;
```

```
class Emp extends Object implements Cloneable
```

```
{
```

```
    String name;
```

```
    int id;
```

```
    Double salary;
```

```

Emp(String name, int id, Double salary)
{
    this.name=name;
    this.id=id;
    this.salary=salary;
}
@Override
public String toString()
{
    String info = id+ " "+name+ " "+salary;
    return info;
}
@Override
protected Object clone() throws CloneNotSupportedException {
    return super.clone();
}
public class Mainclass2 {

    public static void main (String ar[])
    {
        Emp e1 = new Emp("Blake", 123, 2314.14);
        System.out.println("HashCode =" +e1.hashCode());
        System.out.println(e1);
        try
        {
            Emp e2 = (Emp) e1.clone();
            System.out.println("HashCode =" +e2.hashCode());
            System.out.println(e2);
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    }
}

Op –
HashCode =792791759
123 Blake 2314.14
HashCode = 434176574
123 Blake 2314.14

```