**DATE : 17-05-2022**

# VARIABLE

=> Variable is a memory, which is used for storing the data.

=> Every variable has 4 characteristics.

1.) Variable name

2.) Variable type

3.) Variable memory size

( Memory is measured by using terminology bit / byte)

4.) Variable data/value

**Syntax for Variable:**

```
data_type variable_name = data / value;
        where
        LHS = Variable Declaration / Create Memory
        RHS = Variable Initialization / Store Data
```

Example:

1.) int k=10;

k=99; // Variable Re-initialization

=>Variable Re-initialization means update memory by removing old data and by assigning new data.

# KEYWORDS

=> Keywords are special words in java.

=> All the keywords will have special meaning.

=> All the keywords must be used in lowercase.

## List of Keywords

| | | | | |
|---|---|---|---|---|
| int | false | break | class | extends |
| double | if | default | static | this |
| char | else | for | void | super |
| boolean | switch | while | return | final |
| true | case | do | new | |

## IDENTIFIERS

=> Identifiers are the names given by developers.

=> Developer is responsible for providing names for class_name , variable_name , method_name

## Rules of Identifiers

=> We can use **A-Z,a-z,0-9,$,'_'** in Identifier names.

=> Identifiers can start with alphabet/symbol but identifiers not start with numbers.

---

**NOTE:**

→Identifiers can have numbers but should not start with numbers

---

Example:

| **valid** | **invalid** |
|---|---|
| TestVariable | Test Variable**(Space is not allowed)** |
| testvariable | 123javapoint**(Not begin with number)** |
| Test_Variable | java+tpoint**( '+' symbol cannot be used)** |
| _testvariable | a-javapoint**( '-' symbol is not allowed)** |
| $testvariable | java_&_tpoint**( '&' symbol is not used)** |

## Naming formats of Identifiers

=>Naming formats are followed to improve readability of the program.

=> Naming formats are not rules.

=> Class name should follow PASCAL casing.

=> Variable name should follow CAMEL casing.

=> PASCAL casing means , every word starting letter should be Capital.

Example :

Demo DemoProgram StudentFinalResult

=> CAMEL casing means,

a.) If there is only single letter (or) single word,use lower casing

Example :

k i z age height

name count sum

b.) If there are multiple words , first word should be in lowercase , from second word starting letter should be capital.

Example :

employeeId studentPercentage

areaOfCircle employeeAnnualSalary

**DATE : 18-05-2022**

## METHODS

=> Methods are used for performing operation.

=> Other than main method we can also create our own methods.

=> Methods are divided into 2 categories

1.) Static Methods 2.) Non-static Methods

=> In order to execute our method, we should perform method calling.

**//Program for Method**

```
class Sample
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		display();
		System.out.println("Main Ends");
	}
	public static void display()
	{
		System.out.println("Display Method");
	}
}
```

**OUTPUT:**

```
Main Starts
Display Method
Main Ends
```

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE TWO STATIC METHODS**

**B.) CALL BOTH THE METHODS FROM MAIN METHOD**

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

```
        System.out.println("Main Starts");
        display();
        show();
        System.out.println("Main Ends");
    }
    public static void display()
    {
        System.out.println("Display Method");
    }
    public static void show()
    {
        System.out.println("Show Method");
    }
}
```

**OUTPUT:**

```
Main Starts
Display Method
Show Method
Main Ends
```

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE NON-STATIC METHOD**

**B.) CALL BOTH THE METHODS FROM MAIN METHOD**

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

```
    {
        System.out.println("Main Starts");
        Methods m1= new Methods();
        m1.run();
        System.out.println("Main Ends");
    }
    public void run()
    {
        System.out.println("Run Method");
    }
}
```

**OUTPUT:**

```
        Main Starts
        Run Method
        Main Ends
```

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE TWO NON-STATIC METHODS**

**B.) CALL BOTH THE METHODS FROM MAIN METHOD**

```
class NMethods
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        NMethods m1= new NMethods();
        m1.display()
```

```
        m1.run();
        System.out.println("Main Ends");
    }
    public void display()
    {
        System.out.println("Display Method");
    }
    public void run()
    {
        System.out.println("Run Method");
    }
}
```

**OUTPUT:**

```
Main Starts
Display Method
Run Method
Main Ends
```

**Q-->WRITE A PROGRAM TO CREATE ONE STATIC AND ONE NON STATIC METHODS AND CALL BOTH THE METHODS FROM MAIN CLASS**

```
class Methods
{
    public static void print()
    {
        System.out.println("Print Method");
```

```
    }
    public void scan()
    {
        System.out.println("Scan Method");
    }
    public static void main(String[] args)
    {
        System.out.println("Main Method Starts");
        print();
        Methods m1=new Methods();
        m1.scan();
        System.out.println("Main Method Ends");
    }
}
```

**Syntax of Method:**

```
Access_Specifier Acess_Modifier void method_name();
//Method Signature
{
    ----
    //Method Implementation
    ----
}
```

==>Static Methods can be accessed(called) directly

**Syntax for accesing Static method:**

```
Method_Name();
```

==>Non-Static method cannot be accessed directly,rather we have to create an object and use object name for accessing non static method.

**Syntax for creating an object:**

Class_Name Object_Name = new Class_Name();

**Syntax for accessing Non-Static Method:**

Object_name.Method_name();

==>Method_Name should follow CAMEL Casing.

==>Object_Name should follow CAMEL Casing.

## Common Q/A

**1.Where to create a method?**

A: Anywhere inside the scope({ }) of the class.

**2.How to call the method?**

A: By using Method_name.

**3.Where to call the method?**

A: From the main method.

**4.Can we create mutiple methods?**

A: Yes

**5.Can we perform compilation without main method?**

A: Yes, We can perform compilation with or without main method.

**6.Can we execute without main method?**

A: No, Main method is mandatory for execution purpose.

**DATE : 19-05-2022**

# VARIBLES

Variables are categorized into 2 types

1.) Local variable 2.) Global variable

## Local variables

1.) Local variables are created inside the method.

2.) Local variables cannot be used in another method,rather use it only within the same method.

3.) Local variable initialization is mandatory.

4.)Without initializing local variable,if local variables are used, it leads to error.

**Syntax of Local variable :**

Datatype variable_name= data;

**//Program1 for Local variable**

```
class Sample
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		int k=10;	//k is local to main method
		System.out.println(k);
		display();
		Sample s1=new Sample();
		s1.result();
		System.out.println("Main Ends");
	}
	public static void display()
	{
		System.out.println("Display Starts");
		double d=5.4;		//d is local to display method
		System.out.println(d);
```

```
        System.out.println("Display Ends");
    }
    public void result()
    {
        System.out.println("Result Starts");
        boolean b=true;      //b is local to result method
        System.out.println(b);
        System.out.println("Result Ends");
    }
}
```

**Output:**

```
Main Starts
10
Display Starts
5.4
Display Ends
Result Starts
true
Result ends
Main Ends
```

**//Program2 for Local variable**

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

```
    {
        System.out.println("Main Starts");
        int k;
        System.out.println(k);
    }
}
```

**Output:**

error because variable k is not initialized

## Global variables

1.) Global variables are created within the class, outside the method.

2.) There are 2 types of global variable.

1,)Static variable

2.) Non-Static variable

3.) Global variables can be accessed in any method.

4.) Static variables are accessed directly.

5.) Non-Static variables are accessed through object.

6.) Initialization of global variables is not mandatory.

7.) In case global variables are not initialized, we will **not get** error , rather compiler will initialize to default values.

| DATATYPE | DEFAULT VALUE |
|---|---|
| int | 0 |
| double | 0.0 |
| boolean | False |
| char | Empty character |
| String | NULL |

**Syntax for Global Variable:**

Access_Specifier data_type variable_name=data;

**//Program1 for static variable**

```
class Test
{
	public static int i=99;
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(i);
		System.out.println("Main Ends");
	}
}
```

**Output:**

```
Main Starts
99
MainEnds
```

**//Program2 for Non-static variable**

```
class Run
{
	public boolean b=true;
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Run r1= new Run();
```

```
        System.out.println(r1.b);
        System.out.println("Main Ends");
    }
}
```

**Output:**

```
Main Starts
99
MainEnds
```

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.)CREATE 2 STATIC VARIBLES OF DIFFERENT DATATYPES**

**B.)PRINT BOTH THE VARIABLES FROM MAIN METHOD**

```
class GlobalS
{
    public static char ch='V';
    public static boolean b=true;
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(ch);
        System.out.println(b);
        System.out.println("Main Ends");
    }
}
```

**Output:**

Main Starts

V

true

MainEnds

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.)CREATE TWO NON-STATIC VARIABLE OF SAME DATATYPE**

**B.)PRINT BOTH THE VARIABLES FROM MAIN METHOD**

```
class GlobalNS
{
    public char ch1='V';
    public char ch2='S';
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        GlobalNS g1= new GlobalNS();
        System.out.println(g1.ch1);
        System.out.println(g1.ch2);
        System.out.println("Main Ends");
    }
}
```

**Output:**

Main Starts

V

S

MainEnds

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.)CREATE STATIC VARIABLE,NON-STATIC VARIABLE,LOCAL VARIABLE OF SAME DATATYPE**

**B.)PRINT ALL THE VARIABLES FROM MAIN METHOD**

```
class Snslvs
{
    public static char ch1='V';
    public char ch2='S';
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        char ch='P';
        System.out.println(ch);
        System.out.println(ch1);
        Snslvs s1= new Snslvs();
        System.out.println(s1.ch2);
        System.out.println("Main Ends");
    }
}
```

**Output:**

```
Main Starts
P
V
S
```

MainEnds

**//Program For Global variable**

```
class Test1
{
    public static int i;
    public static void  main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(i);
        System.out.println("Main Ends");
    }
}
```

**Output:**

Main Starts

0

MainEnds

## Common Q/A

**1.) Can Local variable be declared as Static and Non-Static?**

A: No,Local variable cannot be declared as Static and Non-Static

**2.) Can Global variable be declared as Static and Non-Static?**

A: Yes, Local variable cannot be either declared as Static and Non-Static variable

**3.) Is Global variable initialization mandatory?**

A: No, If not initialize, compiler will initialize to default value

**4.) Can we access Non-Static variable directly?**

A: No, We have to access Non-static variable through object.

**5.) Is default value applicable for local variables?**

A: No, Default value applicable for only to global variables

**6.) Is default value applicable only for static variables?**

A: No, Default value applicable for both static and non-static variables

**7.) Can we access Local variable in all the methods?**

A: No, We can access only within the same method

**Q-->WRITE A PROGRAM TO CREATE 2 STATIC AND NON- STATIC VARIABLE , DO NOT INITIALIZE THE VARIABLE, DATATYPES OF 4 VARIABLE ARE DOUBLE,CHAR, BOOLEAN,STRING**

```
class Snsv2
{
	public static double d = 5.4;
	public static char ch = 'V';
	public boolean b = false;
	public String s = "COREJAVA";
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(d);
		System.out.println(ch);
		Snsv2 s2= new Snsv2();
		System.out.println(s2.b);
```

```
        System.out.println(s2.s);
        System.out.println("Main Ends");
    }
}
```

**Output:**

```
        Main Starts
        5.4
        V
        false
        COREJAVA
        MainEnds
```

**Q-->WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE STATIC VARIABLE,NON-STATIC VARIABLE.**

**B.) CREATE STATIC METHOD,NON-STATIC METHOD,EACH METHOD WILL HAVE LOCAL VARIABLE OF DIFFERENT DATATYPE.**

**C.) PRINT LOCAL VARIABLES IN THE SAME METHOD**

**D.)PRINT GLOBAL VARIABLES IN MAIN METHOD,CALL STATIC AND NON-STATIC METHOD FROM MAIN METHOD**

```
class AllMethsVars
{
    public static int i=30;
    public double d=3.5;
    public static void display()
    {
        System.out.println("Display Starts");
```

```
        boolean b=true;
        System.out.println(b);
        System.out.println("Display Ends");
    }
    public void show()
    {
        System.out.println("Show Starts");
        char ch='V';
        System.out.println(ch);
        System.out.println("Display Ends");
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(i);
        AllMethsVars a1= new AllMethsVars();
        System.out.println(a1.d);
        display();
        a1.show();
        System.out.println("Main Ends");
    }
}
```

**DATE : 20-05-2022**

## Method with Argument

=> Method with arguments are created when input is required to perform the operation.

=> Arguments are part of method signature.

=> Arguments are created inside the bracket('( )').

=> Argument is nothing but variable declaration.

=> Arguments are initialized during method calling.

=> Arguments are also called local variables.

**Syntax for Method with Argument :**

```
Access-Specifier Access-Modifier void Method_Name
(Arg_Type Parameter_Type,.....,Arg_TypeN Parameter_TypeN)
//Method Signature
{
        -----
                //Method Implementation
        -----
}
```

**Example:**

```
class Sample
{
        public static void text1(int k)
        {
                -----   //System.out.println(k);
        }
        public static void text2(boolean b)
        {
                -----   //System.out.println(b);
        }
        public void text3(char ch)
        {
                -----   //System.out.println(ch);
        }
```

```
	public void text4(double d,String str)
	{
		-----    //System.out.println(str);
	}
	public static void main(String[] args)
	{
		text1(65);
		text2(false);
		Sample s1= new Sample();
		s1.text3('S');
		s1.text4(5.5,"JAVA");
	}
}
```

## Method with Return-Type

=> Method with return type will return the output after performing operation.

=> Return statement is used for returning output produced by method

=> Return type specifies the output returned by the method.

**Syntax for Method with Return-type :**

```
Access-Specifier  Access-Modifier  Return_Type  Method_Name()
//Method Signature
{
	-----
		//Method Implementation
	-----
	return output
}
```

**Example :**

```
class MethWRt
{
	public static void main(String[] args)  //Go to down for "="
	{
		int res1 = text1();   //move to right and go to text1()"
		boolean res2 = text2();
		MethWRt m1= new MethWRt();
```

```
        double res3=m1.text3();
    }
    public static int text1()
    {
        -----
        return 65;     //go back to main method and store the value
            in the "res1" variable
    }
    public static boolean text2()
    {
        -----
        return false;
    }
    public double text3()
    {
        -----
        return 5.5;
    }
}
```

**DATE : 21-05-2022**

**Q-->WRITE A PROGRAM FOR PERFORMING ADDITION OF TWO INTEGERS**

```
class Addition

{

    public static void add(int num1,int num2)

    {

        int sum=num1+num2;

        System.out.println("Sum of"+num1+" and "+num2+" is
            "+sum);

    }

    public static void main()

    {

        add(9,5);
```

```
    }
}
```

**OUTPUT:**

```
    Main Starts
    Sum of 9 and 5 is 14
    Main Ends
```

**Q-->WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.)PROGRAM FOR CIRCLE.**

**B.)CREATE A METHOD TO FIND THE RADIUS OF CIRCLE BASED ON THE GIVEN DIAMETER.**

```
class Circle
{
    public void findRadius(double diameter)
    {
        double radius=diameter/2;
        System.out.println("The radius of circle for given diameter
            "+diameter+" is "+radius);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Circle circle=new Circle();
        circle.findRadius(12.6);
        System.out.println("Main Ends");
    }
```

```
}
```

**OUPTUT:**

```
Main Starts
The radius of circle for given diameter 12.6 is 6.3
Main Ends
```

**Q-->WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.)PROGRAM FOR RECTANGLE**

**B.) FIND THE AREA OF RECTANGLE BASED ON THE GIVEN DIMENSION**

```
class Rectangle
{
    public void findArea(double length,double breadth)
    {
        double area=length*breadth;
        System.out.println("The length of rectangle is "+length);
        System.out.println("The breadth of rectangle is "+breadth);
        System.out.println("The area of rectangle is "+area);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Rectangle rectangle=new Rectangle();
        rectangle.findArea(5.2,3.5);
        System.out.println("Main Ends");
    }
```

```
}
```

**OUPTUT:**

```
Main Starts
The Length of rectangle is 5.2
The Breadth of rectangle is 3.5
The area of rectangle is 18.2
Main Ends
```

**Q-->WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.)PROGRAM FOR CIRCLE**

**B.)FIND THE AREA OF CIRCLE BASED ON THE GIVEN DIAMETER.**

```
class AreaOfCircle
{
    public staic double findRadius(double diameter)
    {
        double radius=diameter/2;
        return radius;
    }
    public static void findArea(double radius)
    {
        double area=3.14*radius*radius;
        System.out.println("The area of circle is "+area);
    }
    public static void main(String[] args)
    {
```

```
        System.out.println("Main Starts");
        double result=findRadius(14.4);
        findArea(result);
        System.out.println("Main Ends");
    }
}
```

**OUPTUT:**

```
    Main Starts
    The area of rectangle is 162.7776
    Main Ends
```

**DATE : 23-05-2022**

**Q-->WRITE A PROGRAM FOR THE FOLLOWING SCENARIO**

**"A MAN WALKING AT THE SPEED OF 5M/S WANTS TO CROSS A BRIDGE OF 120M LENGTH. FIND THE TIME TAKEN TO CROSS THE BRIDGE IN MINUTES."**

**→OUPUT SHOULD BE PRINTED IN THE BELOW MANNER**

**//A MAN WALKING AT THE SPEED OF 5M/S WILL TAKE 0.4 MINUTES TO CROSS THE BRIDGE OF 120M LENGTH//**

```
class Bridge
{
    public static int timeInSec(int distance,int speed)
    {
        int time=distance/speed;
        return time;
    }
    public static void timeInMin(int distance,int speed,int timeSec)
```

```
    {
        double timeMin=timeSec/60.0;
        System.out.println("A man walking at the speed of "+speed+" m/s
        will take "+timeMin+" minutes to cross the bridge of "+distance+"
        meters");
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        int result=timeInSec(120,5);
        timeInMin(120,5,result);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
A man walking at the speed of 5 m/s will take 0.4 minutes to cross the
bridge of 120 meters
Main Ends
```

**Q→WRITE A PROGRAMFOR BELOW SCENARIO**

**"A WOMAN WILL PURCHASE THE BOOK FOR 200RS , AFTER SELLING THE BOOK SHE WANTS PROFIT OF 10%"**

```
class SellBook
{
    public static int priceProfit(int profitPercent,int costPrice)
```

```
{
        int profitPrice=(profitPercent*costPrice)/100;
        return profitPrice;
}
public static void priceSelling(int costPrice,int profitPrice,)
{
        int sellingPrice=costPrice+profitPrice;
        System.out.println("Selling Price of the book is "+sellingPrice+"
rupees);
}
public static void main (String[] args)
{
        System.out.println("Main Starts");
        int result=priceProfit(10,200);
        priceSelling(200,result);
        System.out.println("Main Ends");
}
}
```

**OUTPUT**

Main Starts

Selling price of the book is 220

Main Ends

## Common Q/A

**1.) Can we develop multiple arguements?**

A: Yes

**2.) Can we have multiple return types?**

A: No, Only single return type is allowed.

**3.) When return statement is compulsory?**

A: When return type is other than void.

**4.) When return statement is not necessary?**

A: When return type is void.

**5.) Should return type and return statement match?**

A: Yes, Datatype of return type and return statement should match.

**6.) What is the condition to create method with return type?**

A: Perform operation, Produce output ,Suppose output will be used as input to perform another operation .For this operation we should create method with return type.

**7.) Within the same method, local variables can have same name?**

A: No ,Within the same method local variable names must be different.

**8.) Between different methods can local variable have same name?**

A: Yes, Between different methods local variable can have same name.

**Q→WRITE A PROGRAM FOR THE FOLLOWING SCENARIO**

**"A TRAIN TRAVELLING AT THE SPEED OF 60 KM/HR WILL CROSS AN ELECTRICITY POLE IN 9 SEC. FIND THE LENGTH OF THE TRAIN IN METERS"**

**→OUTPUT SHOULD BE IN THE BELOW MANNER**

**"A TRAIN OF LENGTH 149.9 M WILL TAKE 9 SEC TO CROSS ELECTRCITY POLE IF THE TRAIN MOVES AT THE SPEED OF 16.666 M/S"**

**DATE : 24-05-2022**

# METHOD OVERLOADING

==>Process of creating two or more methods with the same method name but with different argument types .This process is called Method Overloading.

==> Method overloading does not depends on

1.)Access Specifier 2.)Access Modifier

3.)Return type 4.)Argument Name

==> Overloaded methods can have same or different access specifier, access modifier, return type, argument name.

==> Method overloading depends only on 2 factors

1.) All the method names must be same.

2.) Method should have different argument types.

==> We go for Method overloading

"**To perform same operation in multiple ways"**.

==> Method overloading provides flexibility for user to perform operation at user's choice.

//**PROGRAM**

```
class Sample
{
    public static void move(int i,int j)
    {
        System.out.println("Inside move method with int,int arguments");
    }
    public static void move(double i,double j)
    {
```

```
        System.out.println("Inside move method with double,double
        arguments");
    }
    public static void move(int i,int j,int k)
    {
        System.out.println("Inside move method with int, int, int arguments");
    }
    public static void move(double i,double j,double k)
    {
        System.out.println("Inside move method with double, double, double
        arguments");
    }
    public static void main(String [] args)
    {
        System.out.println("Main Starts");
        move(3,5);
        move(30.0,50.0);
        move(5,7,9);
        move(50.0,70.0,90.0);
        System.out.println("Main Ends");
    }
}
```

**Q→ WRITE A PROGRAM TO PERFORM ADDITION IN THREE WAYS**

**A.) ADDITION OF TWO INTEGERS**
**B.) ADDITION OF TWO DECIMALS**
**C.) ADDITION OF TWO INTEGERS AND ONE DOUBLE**

```
class Addition
{
	public static void add(int a,int b)
	{
		System.out.println("Addition of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("a+b = "+(a+b));
	}
	public static void add(double a,double b)
	{
		System.out.println("Addition of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("a+b = "+(a+b));
	}
	public static void add(int a,int b,double c)
	{
		System.out.println("Addition of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("c = "+c);
		System.out.println("a+b+c = "+(a+b+c));
	}
	public static void main(String[] args)
```

```
    {
        System.out.println("Main Starts");
        add(5,9);
        add(12.0,13.0);
        add(8,4,18.0);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Addition of
a=5
b=9
a+b=14
Addition of
a=12.0
b=13.0
a+b=25.0
Addition of
a=8
b=4
c=18.0
a+b+c=30.0
Main Ends
```

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

- **A.) PERFORM MULTIPLICATION**
- **B.) PERFORM MULTIPLICATION IN 4 WAYS**
  - **a.) INTEGER AND DOUBLE**
  - **b.) DOUBLE AND INTEGER**
  - **c.) 3 INTEGERS**
  - **d.) 2 DOUBLES AND ONE INTEGER**

**FIRST TWO NON-STATIC AND LAST TWO STATIC METHODS**

```
class Multiplication
{
	public void mul(int a,double b)
	{
		System.out.println("Multiplication of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("a*b = "+(a*b));
	}
	public static void mul(double a,int b)
	{
		System.out.println("Multiplication of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("a*b = "+(a*b));
	}
	public static void mul(int a,int b,int c)
	{
		System.out.println("Addition of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("c = "+c);
		System.out.println("a*b*c = "+(a*b*c));
	}
	public static void mul(double a,double b,int c)
	{
		System.out.println("Addition of");
		System.out.println("a = "+a);
		System.out.println("b = "+b);
		System.out.println("c = "+c);
		System.out.println("a*b*c = "+(a*b*c));
```

```
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Multiplication m1=new Multiplication();
        m1.mul(4,5.0);
        m1.mul(6.0,9);
        mul(3,5,7);
        mul(4.0,6.0,8);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Multiplication of
a=4
b=5.0
a*b=20.0
Multiplication of
a=6.0
b=9
a*b=54.0
Multiplication of
a=3
b=5
c=7
a*b*c=105
Multiplication of
a=4.0
```

b=6.0

c=8

a*b*c=192.0

Main Ends

**Q→ WRITE A PROGRAM TO OVERLOAD MAIN METHOD WITH STATIC AND NON-STATIC METHOD.**

```
class MainOverload
{
	public static void main(int i)
	{
		System.out.println("Static Main"+i);
	}
	public void main(double i)
	{
		System.out.println("Non-Static Main"+i);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Method");
		main(10);
		MainOverload m1=new MainOverload();
		m1.main(20.0);
	}
}
```

**OUTPUT**

Main Method
Static Method : 10
Non-Static Method : 20

## Common Q/A

**1.) Can we overload static and non-static methods together?**

A: Yes, We can overload static and non-static methods together.

**2.) Can we overload methods with different return types?**

A: Yes, We can have overloaded methods with different return types.

**3.) Can we overload main method?**

A: Yes, We can overload main method but these main methods should be called for execution.

**4.) Can we create non-static main method?**

A: Yes, Main method can be non-static with the help of overloading.

**5.) Can we overload methods with different number of arguments?**

A: Yes, Overloaded methods can differ by number of arguments.

**6.) Can we overload methods with same number of arguments but different argument type?**

A: Yes, Overloaded methods can have same number of arguments but change in argument type.

**DATE : 25-05-2022**

# Execution Process

==> When java program is executed two memory areas are created.

1.)Stack area 2.)Heap area

==> Stack area is used for execution purpose.

==> Heap area is used for storage purpose.

==> JVM will make use of 3 different resources for execution purpose

==> 3 resources are

1.) Class loader 2.) Main method 3.)Garbage collector

==> JVM will call each and every resource one-by-one to enter into stack area for execution.

==> Each resource, after completion of execution should exit from stack area.

==> Class loader resource purpose is to load(store) all the static members into static pool area.

==> Main method resource purpose is to execute each and every statement one-by-one.

---

**NOTE:**

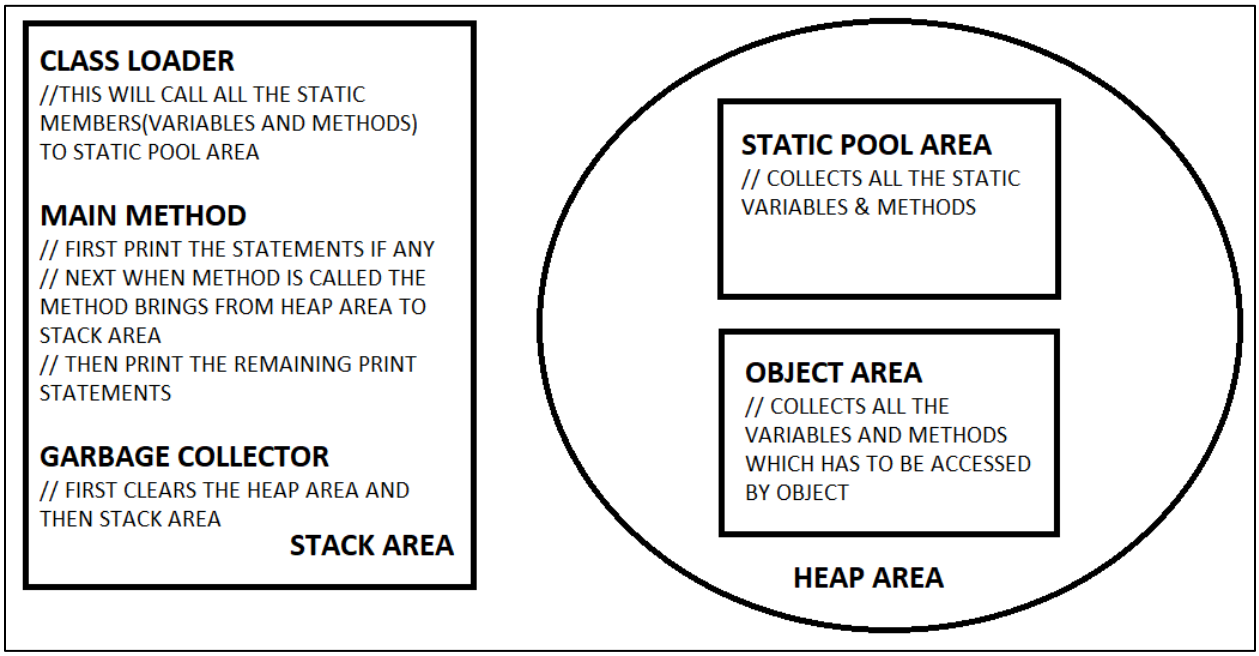
When object creation statement is executed,below steps will occur

1.) Create an object in Heap area.
2.) Load all non-static members into object.
3.) Give the object address.

} These 3 steps are executed by new operator

---

==> Garbage collector resource purpose is to clear Heap area.




**//Program**

```
class Demo
{
	public static int a=12;
	public int b=65;
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(a);
		Demo demo=new Demo();
		System.out.println(demo.b);
		System.out.println("Main Ends");
	}
}
```

**OUTPUT :**

Main Starts

---

**NOTE:**

==>Object name is also called as reference variable.

==> Reference variable contains object address.

==> By using Object address, JVM can refer the object for accessing non-static members

---

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.)CREATE 2 STATIC AND NON-STATIC VARIABLE**

**B.)PRINT ALL THE VARIABLES IN MAIN METHOD**

**C.)DRAW EXECUTION DIAGRAM**

```
class Variables
{
	public static int a=15;
	public static int b=35;
	public int c=55;
	public int d=75;
	public staic void main(String[] args)
	{
		System.out.println(“Main Starts”);
		System.out.println(a);
		System.out.println(b);
		Variables variables=new Variables();
		System.out.println(variables.c);
		System.out.println(variables.d);
		System.out.println(“Main Ends”);
	}
}
```

**OUTPUT**

Main Starts

15

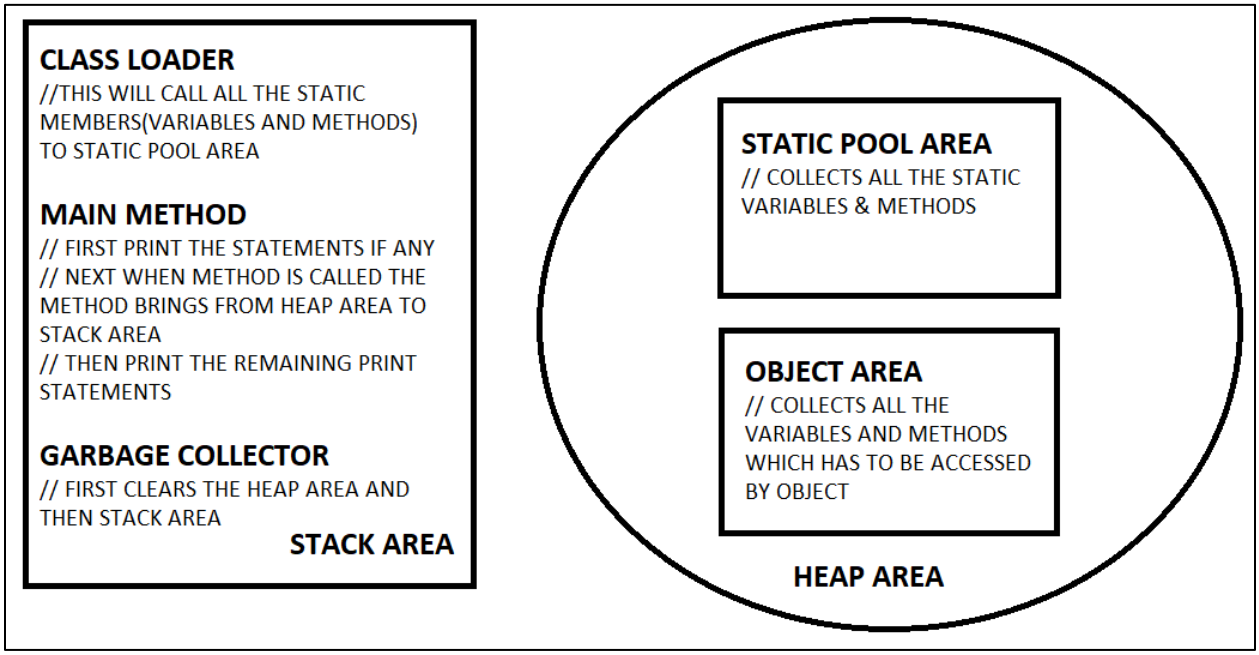
25

55

75

Main Ends

## EXECUTION DIAGRAM




**DATE : 26-05-2022**

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE STATIC METHOD**

**B.) CALL THE METHOD FROM MAIN METHOD**

**C.) DRAW EXECUTION DIAGRAM**

```
class Test
{
	public static void move()
	{
		System.out.println("Move Method");
	}
	public static void main(String args[])
	{
		System.out.println("Main Starts");
		move();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
	Main Starts
	Move Method
```

Main Ends

---

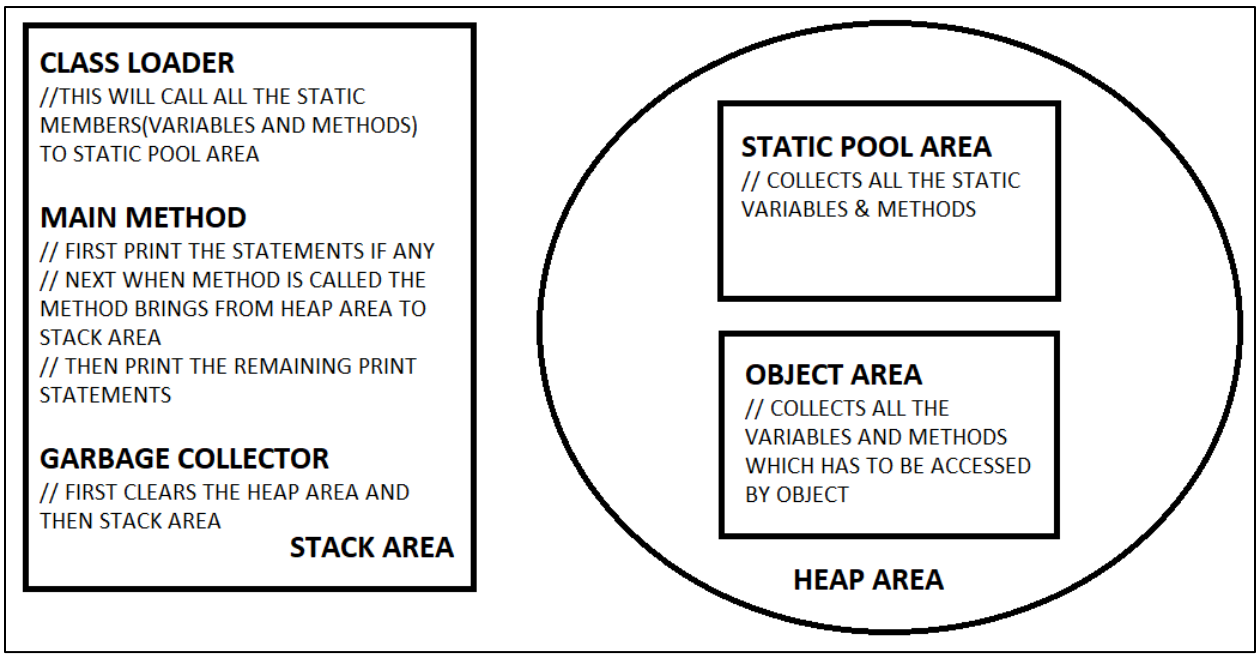
**NOTE :**

-> Whenever method is called method must enter into stack area for execution.

-> After execution of the method, method will exit from the stack area.

---

**EXECUTION DIAGRAM**




**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE NON-STATIC METHOD**

**B.) CALL THE NON-STATIC METHOD FROM MAIN METHOD**

**C.) DRAW EXECUTION DIAGRAM**

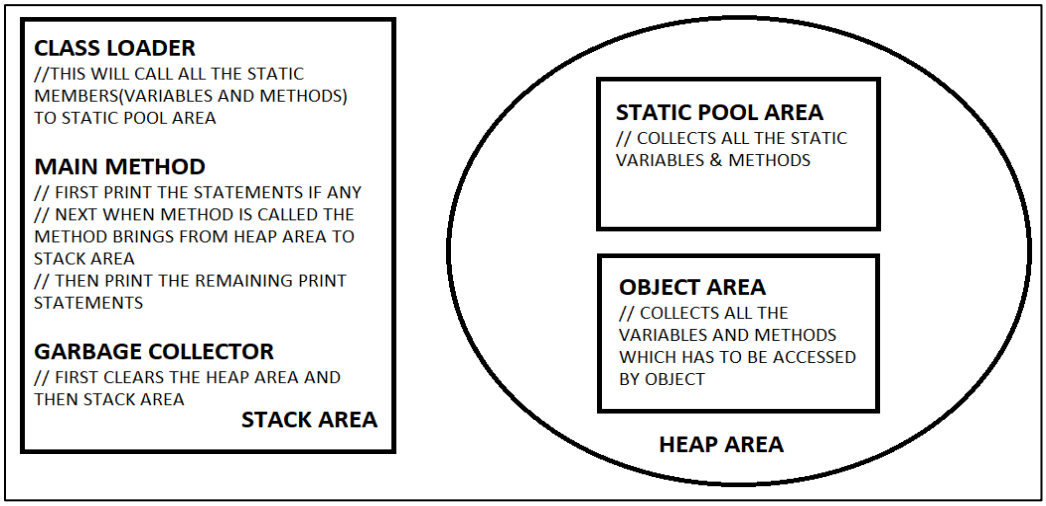
```
class Ntest
{
    public void create()
    {
        System.out.println("Move Method");
    }
    public static void main(String args[])
```

```
    {
        System.out.println("Main Starts");
        Ntest ntest=new Ntest();
        ntest.create();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

Main Starts
Move Method
Main Ends

**EXECUTION DIAGRAM**




**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE STATIC VARIABLE,NON-STATIC VARIABLE,LOCAL VARIABLE**
**B.) PRINT ALL THE VARIABLES FROM MAIN METHOD**
**C.) DRAW EXECUTION DIAGRAM**

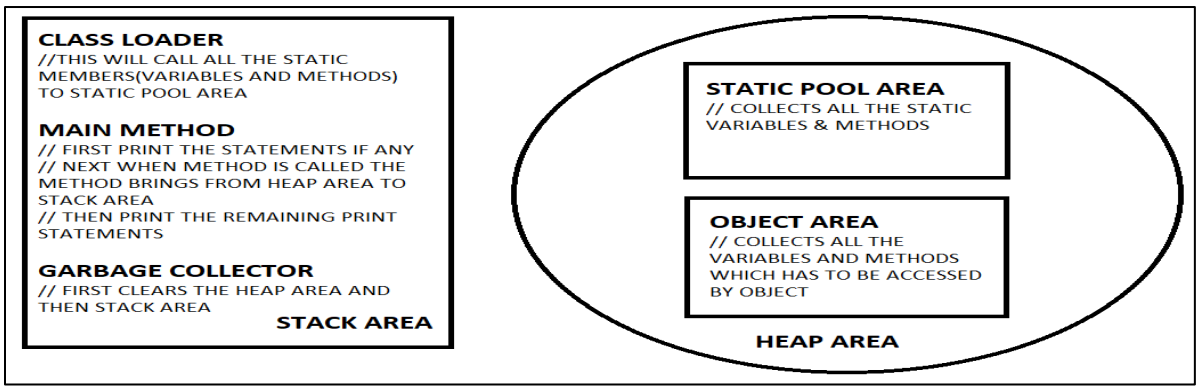
```
class Snslv
{
    public static int a=25;
    public int b=30;
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        int c=55;
        System.out.println(a);
```

```
        Snslv snslv=new Snslv();
        System.out.println(snslv.b);
        System.out.println(c);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

Main Starts
25
30
55
Main Ends

**EXECUTION DIAGRAM**




==> All the local variables are created in Stack Area.

==>All the global variables are created in Heap Area.

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE STATIC METHOD AND NON-STATIC METHOD**
**B.) CREATE LOCAL VARIABLE IN BOTH THE METHODS**
**C.) PRINT THE VARIABLE WITHIN THE SAME METHOD**
**D.) CALL BOTH THE METHODS FROM MAIN METHOD**
**E.) DRAW EXECUTION DIAGRAM**

## COPIES OF STATIC AND NON-STATIC MEMBERS

**//PROGRAM**

```
class Copies
{
	public static int i=76;
	public int j=33;
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(i);
		Copies cs1=new Copies():
		Copies cs2=new Copies():
		Copies cs3=new Copies():
		System.out.println(cs1.j);
		System.out.println(cs2.j);
		System.out.println(cs3.j);
		System.out.println("Main Ends");
	}
}
```

==> We can have only single copy of static members,because class loader will be executed only once for each class.

==> We can have only multiple copies of non-static members,because developer can create multiple objects.

**DATE : 27-05-2022**

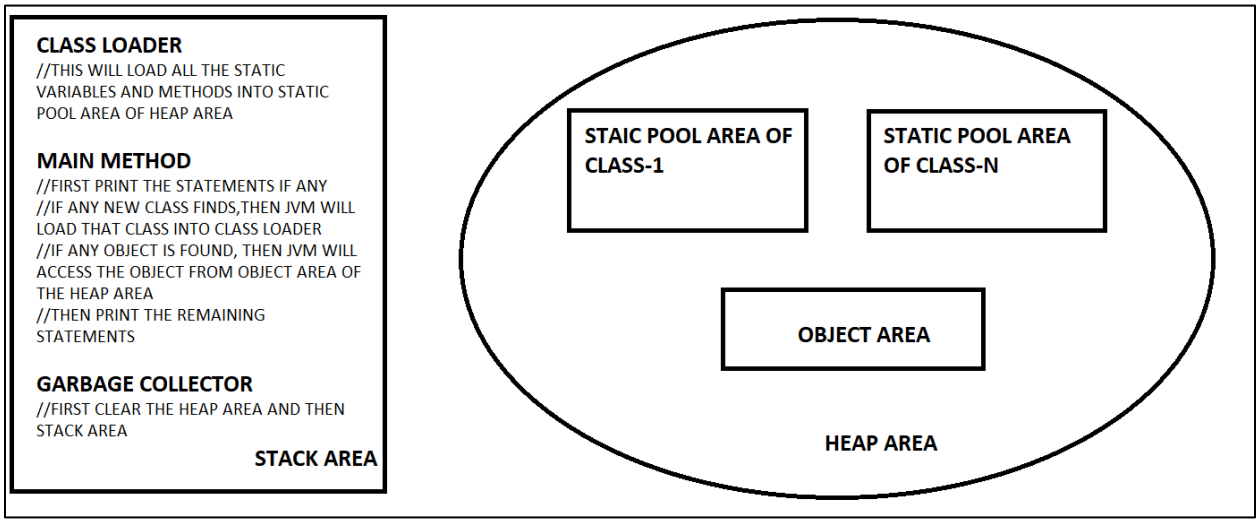
## ACCESSING MEMBERS OF CLASS INN ANOTHER CLASS

==> Static members can be accessed in another class by using class name.

==> Non-Static members can be accessed in another class by creating an object and by using object reference variable.

==> Class Loader will be called by JVM, whenever JVM encounters the class name for the first time.

==> JVM will call the class loader only once for each class.




**//PROGRAM**

```
class Demo
{
        public static int a=30;
        public int b=20;
}
class Sample
{
        public static void main(String[] args)
```

```
	{
		System.out.println("Main Starts");
		System.out.println(Demo.a);
		Demo demo=new Demo();
		System.out.println(demo.b);
		System.out.println("Main Ends");
	}
}
```

**//PROGRAM**

```
class Demo
{
	public static int a=30;
	public int b=20;
}
class Sample
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(Demo.a);
		Demo demo=new Demo();
		//Eventhough,this is object creation statement,JVM will not create an object.
		Because JVM encounters the class name for the first time.
		So, JVM will class loader for loading all the static members.
```

```
            Later object will be created , Non-static members are loaded into
            object.

            System.out.println(demo.b);

            System.out.println("Main Ends");

        }

}
```

==> From the above program Static members are always loaded first before Non-Static members

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE TWO CLASSES**
**B.) FIRST CLASS CONTAINS STATIC AND NON-STAIC METHODS,SECOND CLASS CONTAINS MAIN METHOD.**
**C.) CALL BOTH THE METHODS FROM MAIN METHOD.**

```
class Sns

{

        public static void compile()

        {

            System.out.println(“ Compile Method”);

        }

        public void execute()

        {

            System.out.println(“ Execute Method”);

        }

}

class SnsMain

{

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

```
    {
        System.out.println("Main Starts");
        Sns.compile();
        Sns sns=new Sns();
        sns.execute();
        System.out.println("Main Ends");
    }
}
```

---

**NOTE:**

New method for calling the object is as follows

new ClassName().MethodName();

---

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE TWO CLASSES**
**B.) FIRST CLASS CONTAINS STATIC, NON-STATIC VARIABLES AND METHODS.**
**C.) SECOND CLASS CONTAINS MAIN METHOD**
**D.) PRINT ALL THE VARIABLES,CALL ALL THE METHODS**

```
class Assign
{
    public static int i=45;
    public int j=60;
    public static void print()
    {
        System.out.println("Print Method");
```

```
    }
    public void scan()
    {
        System.out.println("Scan Method");
    }
}
class  MainAssign
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(Assign.i);
        Assign.print();
        Assign assign=new Assign();
        System.out.println(assign.j);
        assign.scan();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
    Main Starts
    45
    Print Method
    60
    Scan Method
```

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE THREE CLASSES**
**B.) FIRST CLASS CONTAINS STATIC ,NON-STATIC METHODS.**
**C.) SECOND CLASS CONTAINS STATIC ,NON-STATIC VARIABLES**
**D.) THIRD CLASS CONTAINS MAIN METHOD**
**E.) PRINT ALL THE VARIABLES,CALL ALL THE METHODS**

```
class Triple
{
	public static void run()
	{
		System.out.println("Run Method");
	}
	public void walk()
	{
		System.out.println("Walk Method");
	}
}
class NTriple
{
	public static int m=35;;
	public int n=70;
}

class  MainTriple
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(NTriple.m);
        Triple.run();
        NTriple ntriple=new NTriple();
        System.out.println(ntriple.n);
        new Triple().walk();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
35
Run Method
70
Walk Method
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE THREE CLASSES**
**B.) FIRST CLASS CONTAINS STATIC VARIABLE AND METHOD**
**C.) SECOND CLASS CONTAINS ,NON-STATIC VARIABLE AND METHOD**
**D.) THIRD CLASS CONTAINS MAIN METHOD**
**E.) PRINT ALL THE VARIABLES,CALL ALL THE METHODS**

```
class Alter
{
	public static int x=20;
	public static void compile()
	{
		System.out.println("Compile Method");
	}
}
class NAlter
{
	public int y=40;
	public void execute()
	{
		System.out.println("Execute Method");
	}
}
class  MainAlter
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		System.out.println(Alter.x);
		Alter.compile();
		NAlter nalter=new NAlter();
		System.out.println(nalter.y);
```

```
        nalter.execute();
        System.out.println("Main Ends");
    }
}
```

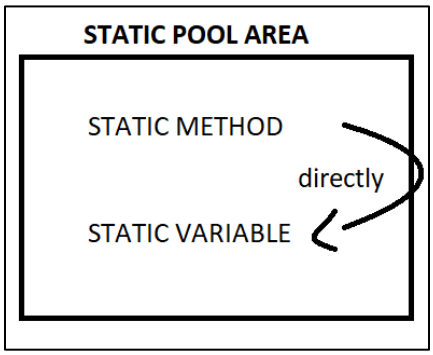
**OUTPUT**

```
Main Starts
20
Compile Method
40
Execute Method
Main Ends
```

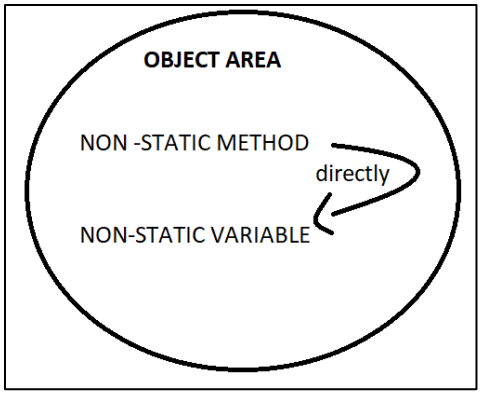
**DATE : 28-05-2022**

## METHOD ACCESS VARIABLE

==> Static method can access static variable directly because both the members are located in same static pool area.

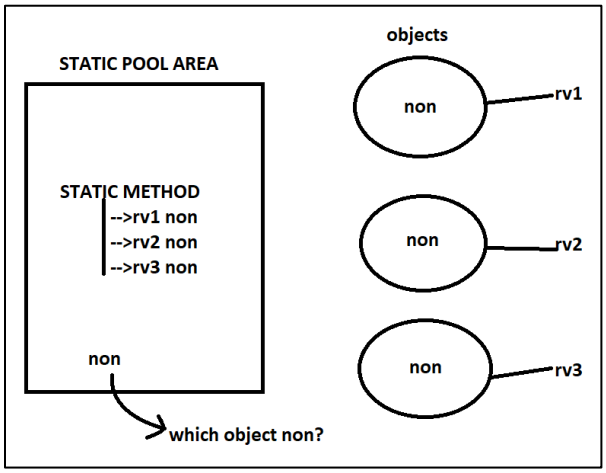



==> Non-Static method can access non-static variable directly because both the members are located in same object.

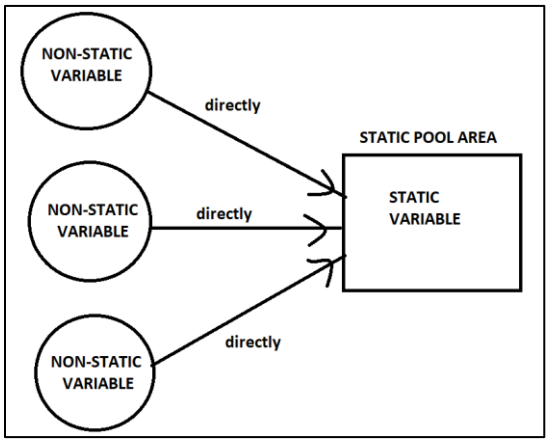



==> Static method cannot access non-static variable directly because there can be multiple copies of Non-static variable; each copy is present in the object. Hence Static methods gets confused **"Which object Non-static variable?" .**

Static method can access Non-static variable by using Reference variable.




==> Non-static method can access static variable directly. Because there is only single copy of Static variable.



**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE STATIC VARIABLE**
**B.) CREATE STATIC METHOD**
**C.) STATIC METHOD SHOULD PRINT STATIC VARIABLE**
**D.) CALL THE METHOD FROM MAIN METHOD**

```
class Demo
{
	public static double d=12.5;
	public static void print()
	{
		System.out.println(d);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		print();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
12.5
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE NON-STATIC VARIABLE**
**B.) CREATE NON-STATIC METHOD**
**C.) PRINT NON-STATIC VARIABLE IN NON-STATIC METHOD**

**D.) CALL THE METHOD FROM MAIN METHOD**

```
class  NDemo
{
	public int n=50;
	public void scan()
	{
		System.out.println(n);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		new NDemo().scan();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
50
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE NON-STATIC VARIABLE**
**B.) CREATE STATIC METHOD**
**C.) PRINT NON-STATIC VARIABLE IN STATIC METHOD**
**D.) CALL THE METHOD FROM MAIN METHOD**

```
class  SmNsv
{
```

```
	public char ch='V';
	public static void show()
	{
		System.out.println(new SmNsv().ch);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		show();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
V
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE STATIC VARIABLE**
**B.) CREATE NON-STATIC METHOD**
**C.) PRINT STATIC VARIABLE IN NON-STATIC METHOD**
**D.) CALL THE METHOD FROM MAIN METHOD**

```
class SvNsm
{
	public static boolean b=true;
	public void display()
	{
```

```
        System.out.println(b);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        new SvNsm().display();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
true
Main Ends
```

**DATE : 30-05-2022**

**METHOD CALL METHOD**

==> Static Method call Static Method →directly

==> Non-Static Method call Non-Static Method →directly

==> Static Method call Non-Static Method →using reference variable

==> Non-Static Method call Static Method →directly

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE TWO STATIC METHODS**
**B.) CREATE MAIN METHOD**
**C.) PERFORM STATIC METHOD CALLING STATIC METHOD**
**D.) CALL STATIC METHOD FROM THE MAIN METHOD**

```
class Books
{
```

```
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        read();
        System.out.println("Main Ends");
    }
    public static void read()
    {
        System.out.println("Read Starts");
        write();
        System.out.println("Read Ends");
    }
    public static void write()
    {
        System.out.println("Write Starts");
        System.out.println("Write Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Read Starts
Write Starts
Write Ends
Read Ends
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE MAIN METHOD**
**B.) CREATE TWO NON-STATIC METHODS**
**C.) MAIN METHOD WILL CALL FIRST NON-STATIC METHOD**
**D.) FIRST NON-STATIC METHOD WILL CALL FIRST NON-STATIC METHOD**

```
class Employee
{
	public void main(String[] args)
	{
		System.out.println("Main Starts");
		new Employee().salary();
		System.out.println("Main Ends");
	}
	public void salary()
	{
		System.out.println("Salary Starts");
		work();
		System.out.println("Salary Ends");
	}
	public static void work()
	{
		System.out.println("Work Starts");
		System.out.println("Work Ends");
	}
}
```

**OUTPUT**

Main Starts

Salary Starts

Work Starts

Work Ends

Salary Ends

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE MAIN METHOD**
**B.) CREATE STATIC METHODS**
**C.) CREATE NON-STATIC METHODS**
**D.) MAIN METHOD WILL CALL STATIC METHOD**
**E.) STATIC METHOD WILL CALL NON-STATIC METHOD**

```
class DebitCard
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        changePin();
        System.out.println("Main Ends");
    }
    public static void changePin()
    {
        System.out.println("Change Pin Starts");
        new DebitCard().withdraw();
        System.out.println("Change Pin Ends");
```

```
    }
    public void withdraw()
    {
        System.out.println("Withdraw Starts");
        System.out.println("Withdraw Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Change Pin Starts
Withdraw Starts
Withdraw Ends
Change Pin Ends
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE MAIN METHOD**
**B.) CREATE NON-STATIC METHOD**
**C.) CREATE STATIC METHOD**
**D.) MAIN METHOD WILL CALL NON-STATIC METHOD**
**E.) NON-STATIC METHOD WILL CALL STATIC METHOD**

```
class TV
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        new TV().changeChannel();
```

```
        System.out.println("Main Ends");
    }
    public void changeChannel()
    {
        System.out.println("Change Channel Starts");
        changeVolume();
        System.out.println("Change Channel Ends");
    }
    public static void changeVolume()
    {
        System.out.println("Change Volume Starts");
        System.out.println("Change Volume Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Change Channel Starts
Channel Volume Starts
Channel Volume Ends
Change Channel Ends
Main Ends
```

**SUMMARY OF ACCESSING**

**WITHIN SAME CLASS**

==> Static Method access Static Variable →directly

==> Static Method access Static Method →directly

==> Non-Static Method access Non-Static Variable →directly

==> Non-Static Method access Non-Static Method →directly

==> Static Method access Non-Static Variable →using reference variable

==> Static Method access Non-Static Method →using reference variable

==> Non-Static Method access Static Variable →directly

==> Non-Static Method access Static Method →directly

**WITH ANOTHER CLASS**

==> Static Variable access Static Method→ using class name

==> Static Variable access Non-Static Method→ using class name

==> Static Method access Static Method→ using class name

==> Static Method access Non-Static Method→ using class name

==> Non-Static Variable access Static Method→ using reference variable

==> Non-Static Variable access Non-Static Method→ using reference variable

==> Non-Static Method access Static Method→ using reference variable

==> Non-Static Method access Non-Static Method→ using reference variable

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE THREE CLASSES**

**B.) FIRST CLASS CONTAIN STATIC VARIABLE,NON-STATIC VARIABLE,STATIC METHOD,NON- STATIC METHOD**

**C.) SECOND CLASS CONTAIN STATIC AND NON-STATIC METHODS**

**D.) THIRD CLASS CONTAINS MAIN METHOD**

**E.) PRINT STATIC VARIABLE IN STATIC METHOD, PRINT NON-STATIC VARIABLE IN STATIC METHOD**

**F.) CALL FIRST CLASS STATIC METHOD IN SECOND CLASS STATIC METHOD, CALL FIRST CLASS NON-STATIC METHOD IN SECOND CLASS NON-STATIC METHOD,**

## G.) CALL SECOND CLASS STATIC METHOD AND NON-STATIC METHOD IN MAIN METHOD

```
class First
{
	public static int i=25;
	public double j=40.0;
	public static void display()
	{
		System.out.println("Display Method");
		System.out.println(i);
	}
	public void show()
	{
		System.out.println("Show Method");
		System.out.println(j);
	}
}
class Second
{
	public static void print()
	{
		System.out.println("Print Method");
		First.display();
	}
	public void scan()
```

```
	{
		System.out.println("Scan Method");
		new First().show();
	}
}
class  MainClassFS
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Second.print();
		new Second().scan();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Print Method
Display Method
25
Scan Method
Show Method
40.0
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A.) CREATE FOUR CLASSES**

**B.) FIRST CLASS CONTAINS STATIC VARIABLE AND NON-STATIC VARIABLE OF SAME DATATYPE.**

**C.) SECOND CLASS CONTAINS STATIC METHOD WHICH WILL PRINT STATIC VARIABLE OF FIRST CLASS, NON-STATIC METHOD WHICH WILL PRINT NON-STATIC VARIABLE OF FIRST CLASS**

**D.) THIRD CLASS CONTAINS NON-STATIC METHOD WHICH WILL CALL STATIC AND NON--STATIC METHOD OF SECOND CLASS**

**E.) FOURTH CLASS CONTAINS STATIC METHOD AND MAIN METHOD,STATIC METHOD WILL CALL THIRD CLASS NON-STATIC METHOD,MAIN METHOD WILL CALL SAME CLASS STATIC METHOD**

```
class A
{
	public static int i=35;
	public int j=45;
}
class B
{
	public static void print()
	{
		System.out.println("Print Method");
		System.out.println(A.i);
	}
	public void show()
	{
		System.out.println("Show Method");
		System.out.println(new A().j);
```

```
        }
}
class C
{
        public void call()
        {
                System.out.println("Call Method");
                B.print();
                new B().show();
        }
}
class  MainABC
{
        public static void develop()
        {
                System.out.println("Develop Method");
                new C().call();
        }
        public static void main(String[] args)
        {
                System.out.println("Main Starts");
                develop();
                System.out.println("Main Ends");
        }
}
```

**OUTPUT**

Main Starts

Develop Method

Call Method

Print Method

35

Show Method

45

Main Ends

**DATE : 31-05-2022**

## MULTIPLE WAYS TO ACCESS STATIC MEMBERS WITHIN THE SAME CLASS

==> There are 3 ways to access static members within the static method of same class.

1.) Direct access

2.) By using Class name

3.) By creating object and using reference variable

**//Program**

```
class Demo
{
    public static int a=100;
    public static void display()
    {
        System.out.println(a);
        System.out.println(Demo.a);
```

```
        Demo demo=new Demo();
        System.out.println(demo.a);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        display();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
100
100
100
Main Ends
```

**MULTIPLE WAYS TO ACCESS NON-STATIC MEMBERS WITHIN THE SAME CLASS**

==> There are 2 ways to access Non-static members within Non-static method of same class.

1.) Direct access

2.) By using **this** keyword

---

**NOTE:**

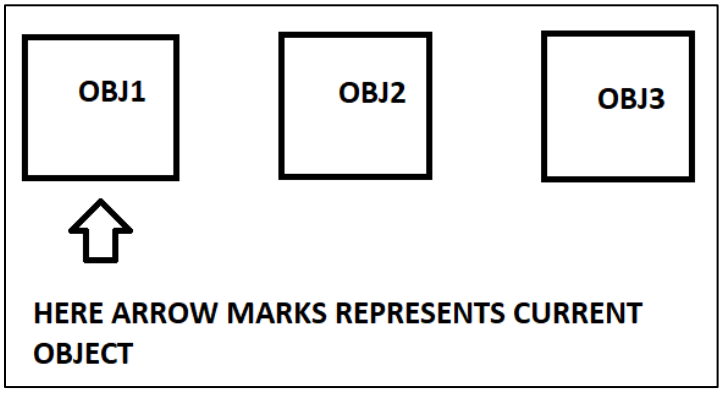
→ **this** keyword represents current object in execution.

---

==========

## Q→ HOW TO RECOGNIZE CURRENT OBJECT?

**A:** Whichever statement is executed by JVM , that statement object is the current object.




==========

**//Program**

```
class Sample
{
	public int x=20;
	public void result()
	{
		System.out.println(x);
		System.out.println(this.x);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Sample sample1 = new Sample();
```

```
        Sample sample2 = new Sample();
        Sample sample3 = new Sample();
        sample1.result();
        sample2.result();
        sample3.result();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
20
20
20
20
20
20
Main Ends
```

## LOCAL VARIABLE AND GLOBAL VARIABLE NAMES ARE SIMILAR

### LOCAL VARIABLE AND STATIC VARIABLE

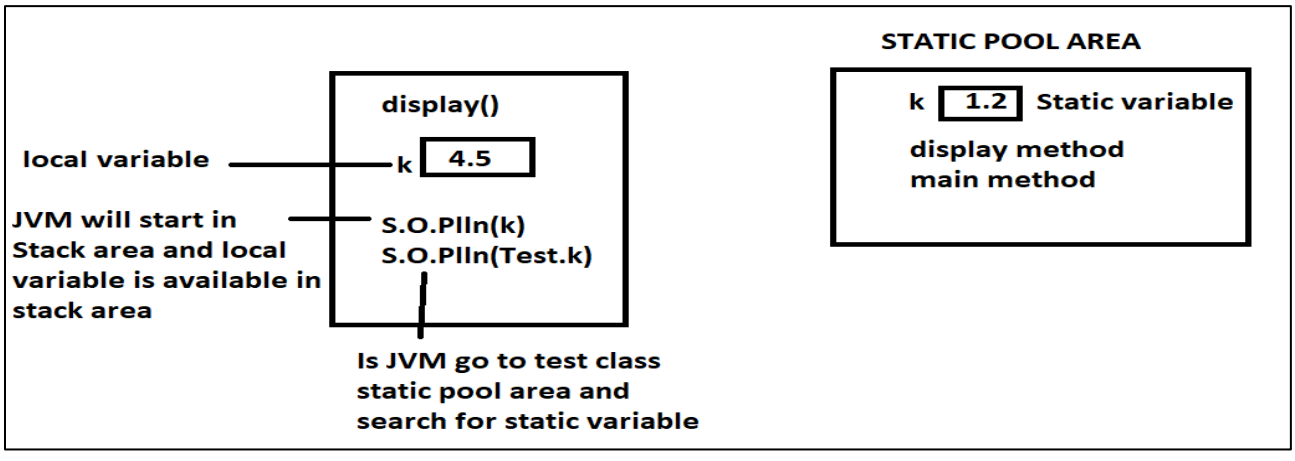
==> Static variable and local variable can have same name.

==> In this case static variable cannot be access directly because local variables are created in stack area and JVM also will search for the variable in Stack Area. Hence, local variables are given direct access.

==> In order to access static variable, we have to use Class name, which will indicate JVM to go to Static pool area for searching static variable.

**//PROGRAM**

```
class Test
{
	public static double k=1.2;
	public static void display()
	{
		double k=4.5;
		System.out.println(k);
		System.out.println(Test.k);
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		display();
		System.out.println("Main Ends");
	}
}
```



**OUTPUT**

Main Starts

4.5

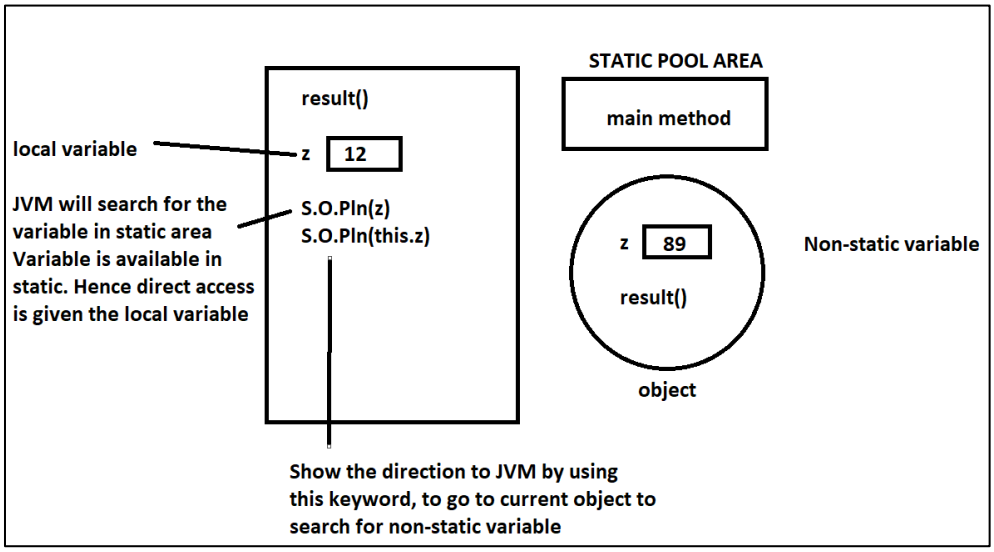
1.2

Main Ends

==> Non-static variable and local variables names can be same.

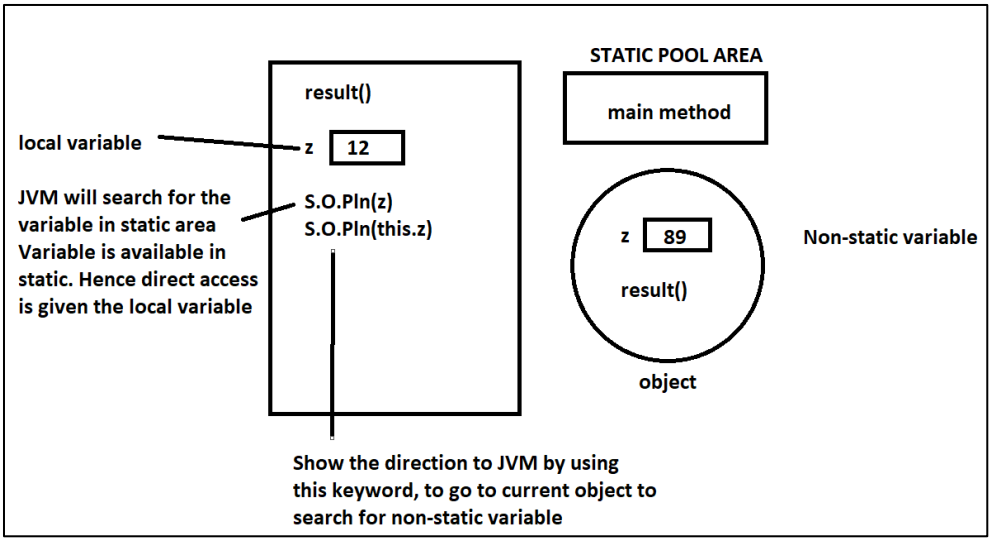
==> In this case Non-static variable cannot be access directly because local variables are created in stack area and JVM also will search for the variable in Stack Area. Hence, local variables are given direct access.

==> In order to access Non-static variable we have to use **this** keyword, which will indicate current object for searching non-static variable.




```
class Run
{
    public int z=89;
    public void result()
    {
        int z=12;
        System.out.println(z);
```

```
        System.out.println(this.z);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Run run=new Run();
        run.result();
        System.out.println("Main Ends");
    }
}
```




**OUTPUT**

```
Main Starts
12
89
Main Ends
```

**Q→ WRITE A PRGRAM FOR THE BELOW REQUIREMENTS**

**A.) CREATE STATIC METHOD WITH ARGUMENT**
**B.) CREATE STATIC VARIABLE**
**C.) STATIC VARIABLE NAME AND ARGUMENT NAME MUST BE SAME**
**D.) PRINT BOTH THE VARIABLES IN THE SAME METHOD**
**E.) CALL STATIC METHOD FROM MAIN METHOD**

```
class Assign1
{
	public static void cricket(int runs)
	{
		System.out.println("Target for team : " + runs);
		System.out.println("Target for team is  " + Assign1.runs);
	}
	public static int runs=164;
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		cricket(160);
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Target for team : 160
Target for team is 164
Main Ends
```

**Q→ WRITE A PRGRAM FOR THE BELOW REQUIREMENTS**

**A.) CREATE TWO NON-STATIC VARIABLES**

**B.) CREATE NON-STATIC METHOD WITH TWO ARGUMENTS, ARGUMENT NAME MUST BE AS NON-STATIC VARIABLE NAME**

**C.) PRINT NON-STATIC VARIABLE AND ARGUMENT WITHIN THE SAME METHOD**

**D.) CALL THE NON-STATIC METHOD FROM MAIN METHOD**

```
class Assign2
{
	public int scores=120;
	public int wickets=5;
	public void display(int scores,int wickets)
	{
		System.out.println("Team scored "+ scores +" runs by "+ wickets +" wickets");
		System.out.println("Team scored "+ this.scores +" runs with loss of "+ this.wickets +" wickets");
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		new Assign2().display(125,3);
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
```

Team scored 125 runs by 3 wickets

Team scored 120 runs with loss of 5 wickets

Main Ends

**DATE : 01-06-2022**

## MULTIPLE CHOICE QUESTIONS TRACING

**1.) Determine Output**

```
class Increment
{
	public static void main(String[] args)
	{
		Simple s= new Simple();
		Simple r=new Simple();
		s.incr();
		r.incr();
		s.display();
		r.display();
	}
}
class Simple
{
	public static int a=5;
	public void incr()
	{
		a++;
	}
```

```
    public void display()
    {
        System.out.println("a="+a);
    }
}
```

**OUTPUT** a = 7 a = 7

**2.) Determine Output**

```
public class Test
{
    public void method(int x,int y)
    {
        System.out.println("Hello");
    }
    public static void method(int x,int y)
    {
        System.out.println("Haii");
    }
    public static void main(String[] args)
    {
        method(1,3);
    }
}
```

**OUTPUT** Error, because both the methods have same argument type

**3.) Determine Output**

```
class SomeClass
```

```
{
        char batch = 'A';
}
public class TestingMethods11
{
        public static void main(String[] args)
        {
                SomeClass a1 = new SomeClass();
                System.out.println("Before : "+a1.batch);
                SomeClass a2 = new SomeClass();
                a2.batch = 'B';
                System.out.println("After : "+a1.batch);
        }
}
```

**OUTPUT**

Before : A

After : A

---

**NOTE:**

→Any changes made to non-static variable the changes will affect only the current object, not all the objects.

---

**4.) Determine Output**

```
class overload
{
```

```
        int x;
        int y;
        public void add(int a)
        {
                x=a+1;
        }
        public void add(int a,int b)
        {
                x=a+1;
        }
}
class Overload_methods
{
        public static void main(String[] args)
        {
                overload obj = new overload();
                int a=0;
                obj.add(6);
                System.out.println(obj.x);
        }
}
```

**OUTPUT** 7

**5.) Determine Output**

```
class Cricket
{
```

```
    int runs;
}
class Testing19
{
    public static void main(String[] args)
    {
        Cricket c1= new Cricket();
        c1.runs=250;
        Cricket c2;          // only created reference variable
        c2=c1;               // c1 and c2 reference variable is cricket type
        c2.runs=300;
        System.out.println("Runs = "+c1.runs);
    }
}
```

**OUTPUT** Runs=300

---

**NOTE:**

→If reference variable are of same type, we can copy object address from one reference variable to another reference variable.In this case single object is referred by two reference variable. Hence if anyone of the reference variable makes changes to object, it will effect the other reference variable

```
int a=10;
int b=a;
```

Since a and b variable are of same type, we can copy data from one variable to another variable.

---

**DATE : 02-06-2022**

## Class and Object

==> Class is used for designing blueprint for object.

==> Object is a mirror image of the blueprint created by class.

==> Using single blueprint we can create multiple objects.

==> To design the blueprint we need to think. Hence, class is called logical entity.

==> Objects exists in our real world. Hence, object is real entity.

**Eg:**

```
class CM          class Marker        class Mobile
{                 {                   {
      ------            ------              ------
}                 }                   }
```

//By using class we are displaying blueprint of class. Description of class is mention within the class//

==> Every object consists of **state** and **behaviour**.

==> State represents “What object knows? “.

==> Behaviour represents “What object does? “.

==> State is nothing but data which is stored in variable.

==> Behaviour is nothing but operation, which will be defined within the method.

Eg:

| WhatsApp | Car | Dog |
|---|---|---|
| Profile name<br>Display picture //State<br>Mobile number | Brand name<br>Colour<br>Mileage | Name<br>Breed<br>Colour |
| Send message<br>Upload status //Behaviour<br>View status | Accelerate<br>Brake<br>Change gear | Play<br>Bark<br>Bite |

==> If the value of the same variable changes from one object to another object, then we need multiple copies of the same variable. In order to get multiple copies of the same variable, declare variable as Non-static.

==> If the value of the variable **remains** same for all the objects, we need single copies of the variable. In order to get single copy of the variable, declare variable as Static.

Eg:

| Circle | Rectangle | Fan |
|---|---|---|
| Radius //Non-static<br>Pi //Static | Length //Non-static<br>Breadth //Non-static<br>Sides //Static | Company //Non-static<br>Speed //Non-static<br>Wings //Static |

==> If the method performs operation by using Non-static variable then declare method as a Non-static method. Because only Non-static method can access variable directly.

==> For Static method there are no such conditions.

Eg:

| Account | Mobile | Triangle |
|---|---|---|
| Balance //Non-static variable | InternetData //Non-static<br>RechargeAmt //Non-static | Base //Non-static<br>Height //Non-static |
| withdraw(B) //Non-static method | watchYoutube(ID) //Non-static<br>performCalling(RA) //Non-static | area(B,H) //Non-static |

==> Static variable is also called as Class Variable because there is only single copy of static variable, same single copy belongs to whole class. Since, there is only single copy, Static method, Non-static method, object can access the Static variable. In other words entire class can access Static variable directly.

==> Any changes made to Static variable, the changes will reflect to the whole class.

==> Non-static variable is also called as Instance variable because Non-static variable nature is multiple copy, each copy belongs to each and every object.

---

**NOTE:**

→Instance means object.

---

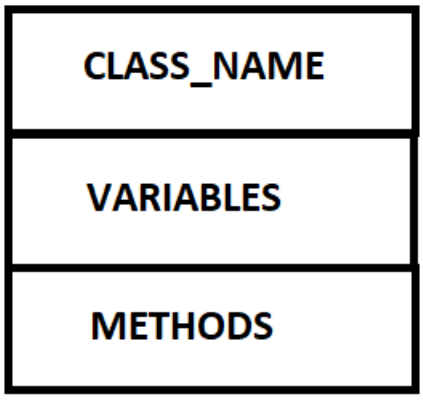
==> Any changes made to Non-static variable, the changes will reflect only to the current object.

**DATE : 03-06-2022**

## Class Diagram

==> Class diagram represents pictorial representation of java program.

==> Before we write any program we should always write class diagram.




where

==>'**+**' indicates **public** ==> '**-**' indicates **private**

==> '**#**' indicates **protected** ==> '**~**' indicates **default**

==> '**c**' indicates **class** ==> '**i**' indicates **interface**

==> '**s**' indicate **static** ==> **'a'** indicates abstract

==> Syntax of variable is **variable_name : data_type**

==> Syntax of method is **method_name(argument_type) : return_type**

## Constructor

==> Constructor is a special type of method which gets executed during object creation.

==> Constructor name must be same as class name.

==> Constructors are called by new operator.

==> There are 2 types of constructor

1.) Constructor with argument

2.) Constructor without argument

**Syntax of Constructor:**

```
Access_Specifier Constructor_name (args/no args)
{
        --------
}
```

==> Within the constructor we can perform any operation, but the main purpose of constructor is to initialize instance variables.

==> If developer does not create any constructor, compiler will create a default constructor. (Default constructor means constructor without arguments and with empty body.)

**//Program**

```
class Demo
{
        public Demo()
        {
                System.out.println("Execute Constructor without argument");
        }
        public static void main(String[] args)
        {
```

```
        System.out.println("Main Starts");
        Demo d1=new Demo();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
    Main Starts
    Execute Constructor without argument
    Main Ends
```

**Q→ WRITE A PROGRAM TO CREATE A CONSTRUCTOR WITH INTEGER ARGUMENT EXECUTE THE CONSTRUCTOR WITH 3 TIMES.**

```
class Sample
{
    public Sample( int n)
    {
        System.out.println("Execute Constructor with argument value : "+n);
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Sample s1=new Sample(5);
        Sample s2=new Sample(7);
        Sample s3=new Sample(9);
        System.out.println("Main Ends");
```

```
    }
}
```

**OUTPUT**

```
    Main Starts
    Execute Constructor with argument value: 5
    Execute Constructor with argument value: 7
    Execute Constructor with argument value: 9
    Main Ends
```

---

**NOTE:**

→By using single constructor we can construct multiple objects.

---

**//PROGRAM**

```
class Flower
{
    public String fName;
    public Flower(String name)
    {
        fName=name;
    }
    public void display()
    {
        System.out.println("Flower name is "+fName);
```

```
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Flower f1= new Flower("Lilly");
        Flower f2= new Flower("Rose");
        Flower f3= new Flower("Jasmine");
        f1.display();
        f2.display();
        f3.display();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Flower name is Lilly
Flower name is Rose
Flower name is Jasmine
Main Ends
```

**Q→ WRITE A PROGRAM FOR BELOW SCENARIO**

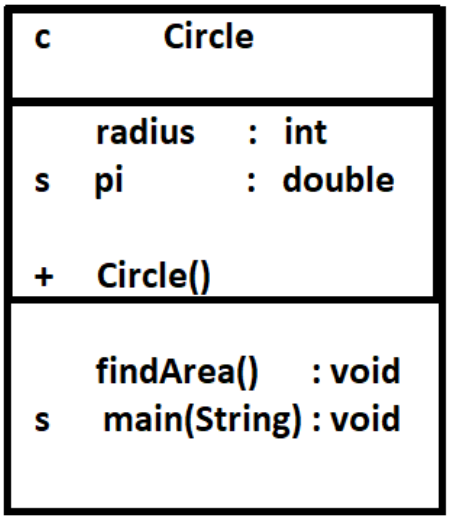
**A.) TWO GUNS OF DIFFERENT NAME WITH DIFFERENT NUMBER OF BULLETS IS USED FOR SHOOTING**

```
class Guns
{
    public String gName;
```

```
	public int bullets;
	public Guns(String a, int b)
	{
		gName=a;
		bullets=b;
	}
	public void shooting()
	{
		System.out.println("Gun name is "+gName);
		for(int i=0;i<bullets;i++)
		{
			System.out.println("Fire");
		}
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Guns g1= new Guns("M416",5);
		Guns g2= new Guns("AKM",5);
		g1.shooting();
		g2.shooting();
		System.out.println("Main Ends");
	}
}
```

**DATE : 06-06-2022**

**Q→ WRITE A PROGRAM FOR BELOW CLASS DIAGRAM**




```
class Circle
{
	public int radius;
	public static double pi=3.14;
	public Circle(int a)
	{
		radius=a;
	}
	public void findArea()
	{
		double area=pi*radius*radius;
		System.out.println("Area of circle = "+area);
	}
}
```

```
class MainCircle
{
    public static void main(String[] args)
    {
        System.out.println(" Main Starts ");
        Circle c1=new Circle(4);
        Circle c2=new Circle(5);
        c1.findArea();
        c2.findArea();
        System.out.println(" Main Ends ");
    }
}
```

**OUTPUT**

```
Main Starts
Area of circle = 50.24
Area of circle = 78.5
Main Ends
```

**Q→ WRITE A PROGRAM FOR BELOW SCENARIO**

**"TWO CARS OF SAME BRAND MOVES AT DIFFERENT SPEED AND WILL TAKE DIFFERENT AMOUNT OF TIME TO REACH THE SAME DESTINATION."**

**FIND THE DISTANCE TRAVELLED BY THE CAR.**

```
class Car
{
    public static String brand="KIGER";
```

```
    public int speed;
    public double time;
    public static String destination="JNTU";
    public Car(int speed,double time)
    {
        this.speed=speed;
        this.time=time;
    }
    public void calculateDistance()
    {
        double distance=speed*time;
        System.out.println("Distance travelled by "+brand+" car  with the
        speed of "+speed+" km/hr and it has taken time of "+time+"hrs to
        cover the distance of "+distance+" km");
    }
}
class MainCar
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Car c1=new Car(50,2.5);
        Car c2= new Car(70,1.5);
        c1.calculateDistance();
        c2.calculateDistance();
```

```
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

Main Starts

Distance travelled by KIGER car with the speed of 50 km/hr and it has taken time of 2.5hrs to cover the distance of 125 km

Distance travelled by KIGER car with the speed of 70 km/hr and it has taken time of 1.5hrs to cover the distance of 105 km

Main Ends

**DATE : 07-06-2022**

## CONSTRUCTOR OVERLOADING

==> Developing multiple constructors with different argument types is called Constructor Overloading.

==> We go for Constructor Overloading, to create same class object in multiple ways.

==> In Constructor overloading, chances are we may repeat the code.

==> In order to avoid repetition code in constructors. We go for Constructor Chaining.

**//PROGRAM**

```
class Demo
{
    public Demo(int a)
    {
```

```
            System.out.println("Execute Demo constructor with int argument");
            System.out.println("a = "+a);
        }
        public Demo(double a)
        {
            System.out.println("Execute Demo constructor with double
            argument");
            System.out.println("a = "+a);
        }
        public Demo(String i,boolean j)
        {
            System.out.println("Execute Demo constructor with String boolean
            argument");
            System.out.println("i = "+i);
            System.out.println("j = "+j);
        }
}
class DemoMain
{
        public static void main(String[] args)
        {
            System.out.println("Main Starts");
            Demo demo1=new Demo(5);
            Demo demo2=new Demo(3.0);
            Demo demo3=new Demo("JAVA",true);
```

```
        System.out.println("Main Ends");        }
}
```

**OUTPUT**

```
Main Starts
Execute Demo constructor with int argument
a = 5
Execute Demo constructor with double argument
a = 3.0
Execute Demo constructor with String boolean argument
i = JAVA
j = true
Main Ends
```

**//PROGRAM**

```
class Student
{
    public String name;
    public double tenth;
    public double inter;
    public double degree;
    public double masters;
    public Student(String name, double tenth, double inter, double degree,
    double masters)
    {
        this.name=name;
        this.tenth=tenth;
```

```
        this.inter=inter;
        this.degree=degree;
        this.masters=masters;
}

public Student(String name, double tenth, double inter, double degree)
{
        this.name=name;
        this.tenth=tenth;
        this.inter=inter;
        this.degree=degree;
}

public void display()
{
        System.out.println("====================");
        System.out.println("Name : "+name);
        System.out.println("Tenth percentage : "+tenth+"%");
        System.out.println("Inter percentage : "+inter+"%");
        System.out.println("Degree Percentage : "+degree+"%");
        if(masters!=0.0)
        {
                System.out.println("Masters Percentage :"+masters+"%");
        }
        System.out.println("====================");
```

```
    }
}
class StudentMain
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Student s1=new Student("JAY",75.0,79.0,84.0,90.0);
        s1.display();
        Student s2=new Student("VIJAY",80.0,75.0,65.0);
        s2.display();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
====================
Name: Jay
Tenth Percentage: 75.0%
Inter Percentage: 79.0%
Degree Percentage: 84.0%
Masters Percentage: 90.0%
====================
====================
Name: Vijay
```

Tenth Percentage: 80.0%

Inter Percentage: 75.0%

Degree Percentage: 65.0%

==========

Main Ends

---

**NOTE:**

→ int is converted to double automatically

Eg: 75→75.0

---

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**=> CREATE A BLUEPRINT FOR BOX**
**=> BOX CAN BE CREATED IN 2D OR 3D MANNER, YOUR PROGRAM SHOULD PROVIDE BOTH THE OPTION**
**=> PRINT THE DIMENSIONS OF THE BOX ACCORDINGLY**

```
class Box
{
    public double length;
    public double width;
    public double height;
    public Box(double length,double width)
    {
        this.length=length;
        this.width=width;
    }
    public Box(double length,double width,double height)
```

```
{
        this.length=length;
        this.width=width;
        this.height=height;
}

public void dimensions2D()
{
        System.out.println("====================");
        System.out.println("2D BOX Dimensions");
        System.out.println("Length : "+length+"cm");
        System.out.println("Width : "+width+"cm");
        System.out.println("====================");
}
public void dimensions3D()
{
        System.out.println("====================");
        System.out.println("3D BOX Dimensions");
        System.out.println("Length : "+length+"cm");
        System.out.println("Width : "+width+"cm");
        System.out.println("Height : "+height+"cm");
        System.out.println("====================");
}
public void dimension()
{
```

```
            if(height==0.0)
            {
                dimensions2D();
            }
            else
            {
                dimensions3D();
            }
        }
}

class BoxMain
{
        public static void main(String[] args)
        {
            System.out.println("Main Starts");
            Box b1=new Box(10.0,5.0);
            b1.dimension();
            Box b2=new Box(7.0,5.0,6.0);
            b2.dimension();
            System.out.println("Main Ends");
        }
}
```

**OUTPUT**

```
Main Starts
```

==========

2D BOX Dimensions

Length: 10.0cm

Width: 5.0cm

==========

==========

3D BOX Dimensions

Length: 7.0cm

Width: 5.0cm

Height: 6.0cm

==========

Main Ends

**DATE : 09-06-2022**

## CONSTRUCTOR CHAINING

==> Process of calling the constructor from another constructor is called Constructor Chaining.

==> We can perform Constructor Chaining within the same class by using call to this statement.

**Syntax of Call to this statement:**

**this(args/no-args);**

==> Call to this statement must be the first statement in Constructor.

==> Within the constructor, we can use only one call to this statement.

**i.e** Constructor can perform constructor chaining with only one constructor.

==> We go for Constructor Chaining to avoid Code repetition, rather to implement code reusability.

**//PROGRAM**

```
class Student
{
	public String name;
	public double tenth;
	public double inter;
	public double degree;
	public double masters;
	public Student(String name, double tenth, double inter, double degree,
	double masters)
	{
		this(name,tenth,inter,degree);
		this.masters=masters;
	}

	public Student(String name, double tenth, double inter, double degree)
	{
		this.name=name;
		this.tenth=tenth;
		this.inter=inter;
		this.degree=degree;
	}
	public void display()
	{
		System.out.println("====================");
```

```
            System.out.println("Name : "+name);
            System.out.println("Tenth percentage : "+tenth+"%");
            System.out.println("Inter percentage : "+inter+"%");
            System.out.println("Degree Percentage : "+degree+"%");
            if(masters!=0.0)
            {
                  System.out.println("Masters Percentage :"+masters+"%");
            }
            System.out.println("====================");
      }
}
class StudentMain
{
      public static void main(String[] args)
      {
            System.out.println("Main Starts");
            Student s1=new Student("JAY",75.0,79.0,84.0,90.0);
            s1.display();
            Student s2=new Student("VIJAY",80.0,75.0,65.0);
            s2.display();
            System.out.println("Main Ends");
      }
}
```

**OUTPUT**

```
      Main Starts
```

==========

Name: Jay

Tenth Percentage: 75.0%

Inter Percentage: 79.0%

Degree Percentage: 84.0%

Masters Percentage: 90.0%

==========

==========

Name: Vijay

Tenth Percentage: 80.0%

Inter Percentage: 75.0%

Degree Percentage: 65.0%

==========

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**==> PROGRAM FOR EMPLOYEE**

**==> THERE ARE 2 TYPES OF EMPLOYEES "FRESHER AND EXPERIENCED"**

**==> EMPLOYEE PROPERTIES ARE EID,ENAME,ESALARY,E YEARS OF EXPERIENCE**

**==> DISPLAY EACH TYPE OF EMPLOYEES INFORMATION ACCORDINGLY**

```
class Employee
{
    public int id;
    public String name;
```

```
public int salary;
public int year_of_exp;
public Employee(int id,String name, int salary,int year_of_exp)
{
        this(id,name,salary);
        this.year_of_exp=year_of_exp;
}

public Employee(int id,String name, int salary)
{
        this.id=id;
        this.name=name;
        this.salary=salary;
}
public void display()
{
        System.out.println("====================");
        System.out.println("Employee Id : "+id);
        System.out.println("Employee Name : "+name);
        System.out.println("Employee salary : "+salary);
        if(year_of_exp!=0)
        {
                System.out.println("Employee Year of
                experience:"+year_of_exp);
        }
```

```
            System.out.println("====================");
        }
}
class EmployeeMain
{
        public static void main(String[] args)
        {
            System.out.println("Main Starts");
            Employee e1=new Employee(1,"AJAY",25000,5);
            e1.display();
            Employee e2=new Employee(2,"JAY",10000);
            e2.display();
            System.out.println("Main Ends");
        }
}
```

**OUTPUT**

```
Main Starts
====================
Employee Id : 1
Employee Name : AJAY
Employee salary : 25000
Employee Year of experience: 5
====================
====================
Employee Id : 2
```

Employee Name : JAY

Employee salary : 10000

==========

Main Ends

---

**NOTE:**

==> Reusing the code present in the method for following code

```
public void displayFresher()
{
        System.out.println("Employee Id : "+id);
        System.out.println("Employee Name : "+name);
        System.out.println("Employee salary : "+salary);
}
public void display()
{
        displayFresher();
        System.out.println("Employee Year of experience:"+year_of_exp);
}
```

---

## Common Q/A

1. **Can we access non-static members by using class name?**

**A:** No, class name is used only on the static members

2. **Can we access static members by using object?**

**A:** Yes, objects can be used for accessing static members because every object is connected to static pool area.

3. **Can we create an object without creating the reference variable?**

**A:** Yes

4. **Can a single object have multiple reference variable?**

**A:** Yes

5. **Can we give Constructor name Method name same?**

**A:** Yes, Because Constructor name will follow PASCAL casing, Method name will follow CAMEL casing.

6. **Is Access modifier allowed for Constructor?**

**A:** Constructor is a special type of method which does not allow access modifier but by default it is static.

7. **Can we initialize static variable in Constructor?**

**A:** Yes

8. **Can we call a method from Constructor?**

**A:** Yes

9. **What is Default Constructor?**

**A:** Constructor without any argument with empty body

10. **If developer creates constructor without argument, will compiler create default constructor?**

**A:** No, Because developer has already created a constructor.

11. **If developer creates constructor with argument, will compiler create default constructor?**

**A:** No, Because developer has already created a constructor.

12. **When constructors are loaded, where constructors are loaded?**

**A:** Constructors are loaded during class loader and they are loaded into static pool area.

**13.Write a program to count object creation automatically?**

**A:**
```
class TV
{
    public static int count;
    public TV()
    {
        count++;
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        TV t1=new TV();
        TV t2=new TV();
        TV t3=new TV();
        System.out.println("Number of objects: "+count);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Stats
Number of objects : 3
Main Ends
```

---

**DATE : 11-06-2022**

# INHERITANCE

==> Super class will provide separate copy of properties to each and every object of sub class. This process is called Inheritance.

==> Inheritance can be achieved by using a keyword "**extends**".

==> Only non-static members can be inherited, static members cannot be inherited rather shared.

==> There are 4 types of Inheritance.

1.) Single-level Inheritance

2.) Multi-level Inheritance

3.) Hierarchical Inheritance

4.) Multiple Inheritance: which cannot be achieved using classes.

==> Inheritance is also known as "**IS-A relationship**"

==> According to the syntax we can create super class object but according to the real world, super class objects should not be created because super class objects does not exist in the real world.

==> Objects are called real entities, Hence anything that is not real. Object should not be created.

**Eg:**

Draw circle : We can draw because it is existing

Draw Shape : Which shape? because we don't know which to draw.

==> The main purpose of inheritance is to achieve Code reusability. **i.e**, define the common code (or) repetitive code (or) duplicated code, reuse the code in sub class.

## OTHER ADVANTAGES OF INHERITANCE

1.) Method over-riding

2.) Abstract

3.) Interface

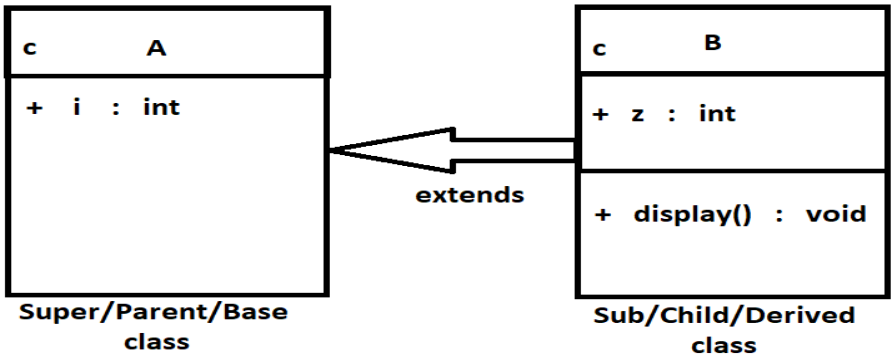
4.) Up-casting and Down-casting

5.) Generalization

6.) Polymorphism

7.) Abstraction

## SINGLE-LEVEL INHERITANCE




**// PROGRAM**

```
class A
{
        public int i=10;
}
class B extends A
{
        public int z=100;
        public void display()
        {
                System.out.println(i);
                System.out.println(z);
        }
}
```

```
class ABMain
{
	public static void main(String[]args)
	{
		System.out.println("Main Starts");
		B b1=new B();
		b1.display();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
10
100
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

- **A.) PROGRAM FOR SINGLE LEVEL INHERTANCE**
- **B.) CREATE TWO NON-STATIC VARIABLES IN SUPER CLASS**
- **C.) CREATE NON-STATIC METHOD IN SUPER CLASS**
- **D.) CREATE NON-STATIC VARIABLES IN SUB CLASS**
- **E.) PRINT SUPER CLASS VARIABLES IN SUB CLASS METHOD,CALL SUPER CLASS METHOD IN SUB CLASS METHOD**
- **F.) CREATE A SUB CLASS OBJECT IN MAIN METHOD AND CALL SUB CLASS METHOD**

```
class SingleA
{
	public int a=40;
```

```
	public int b=60;
	public void print()
	{
		System.out.println("Print Method");
	}
}
class SingleB extends SingleA
{
	public void scan()
	{
		print();
		System.out.println("Scan Method");
		System.out.println("a : "+a);
		System.out.println("b : "+b);
	}
}
class Single_AB_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		SingleB sb=new SingleB();
		sb.scan();
		System.out.println("Main Ends");
	}
```

}

**OUTPUT**

Main Starts

Print Method

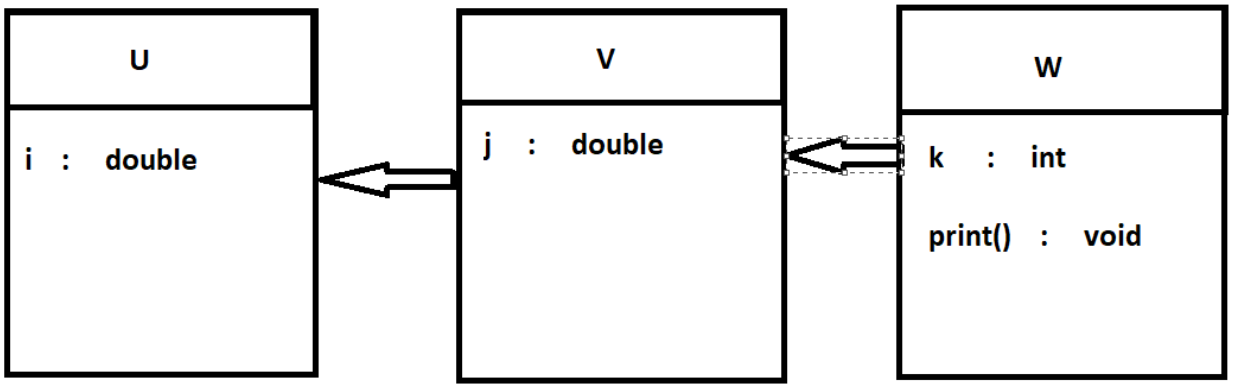
Scan Method

a : 40

b : 60

Main Ends

**MULTI-LEVEL INHERITANCE**




**// PROGRAM**

```
class U
{
        public double a=4.5;
}
class V extends U
{
        public double b=7.5;
}
```

```
class W extends V
{
	int c=20;
	public void print()
	{
		System.out.println(a);
		System.out.println(b);
		System.out.println(c);
	}
}
class UVW_Main
{
	public static void main(String[]args)
	{
		System.out.println(“Main Starts”);
		W w1=new W();
		w1.print();
		System.out.println(“Main Ends”);
	}
}
```

**OUTPUT**

```
Main Starts
4.5
7.5
20
```

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.) PROGRAM FOR SINGLE LEVEL INHERTANCE**
**B.) CREATE SINGLE NON-STATIC VARIABLES IN SUPER MOST CLASS**
**C.) CREATE NON-STATIC METHOD IN SUPER CLASS WHICH WILL PRINT NON-STATIC VARIABLE OF ITS SUPER CLASS**
**D.) CREATE SUB CLASS WITH NON-STATIC METHOD WHICH WILL CALL THE METHOD OF ITS SUPER CLASS**
**E.) CREATE SUB CLASS OBJECT AND CALL SUB CLASS METHOD**

```
class MultiA
{
	public int a=50;
}
class MultiB extends MultiA
{
	public void show()
	{
		System.out.println("Show Method");
		System.out.println("a : "+a);
	}
}
class MultiC extends MultiB
{
	public void result()
	{
		show();
```

```
        System.out.println("Result Method");
    }
}
class MultiABC_MAIN
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        MultiC c= new MultiC();
        c.result();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Show Method
a : 50
Result Method
Main Ends
```

**HIERARCHICHAL INHERITANCE**

**// PROGRAM**

```
class H_P
{
    public int a=10;
}
```

```
class H_Q extends H_P
{
	public double b=25.5;
	public void display()
	{
		System.out.println(a);
		System.out.println(b);
	}
}
class H_R extends H_P
{
	public double c=30.8;
	public void print()
	{
		System.out.println(a);
		System.out.println(c);
	}
}
class H_PQR_Main
{
	public static void main(String[]args)
	{
		System.out.println(“Main Starts”);
		new H_Q().display();
		new H_R().print();
```

```
            System.out.println(“Main Ends”);
        }
}
```

**OUTPUT**

Main Starts

10

20.5

10

30.8

Main Ends

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A.) CREATE A METHOD IN SUPER CLASS**
**B.) CREATE 3 SUB CLASSES FOR SAME SUPER CLASS**
**C.) EACH SUB CLASS WILL HAVE NON-STATIC VARIABLE AND METHOD**
**D.) IN SUB CLASS METHOD CALL SUPER CLASS METHOD AND ALSO PRINT OWN CLASS VARIABLE**

```
class HIER_A
{
    public void run()
    {
            System.out.println("Run Method");
    }
}
class HIER_B extends HIER_A
{
    public String b="HI";
```

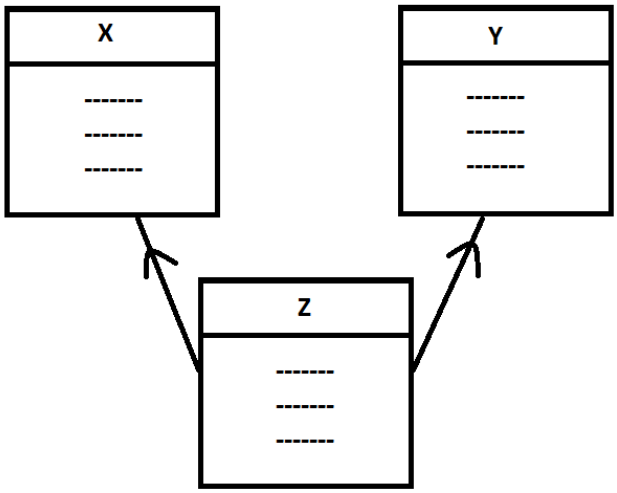
```
    public void runB()
    {
            run();
            System.out.println("Run B Method");
            System.out.println(b);
    }
}
class HIER_C extends HIER_A
{
    public String c="HELLO";
    public  void runC()
    {
            run();
            System.out.println("Run C Method");
            System.out.println(c);
    }
}
class HIER_D extends HIER_A
{
    public String d="WELCOME";
    public void runD()
    {
            run();
            System.out.println("Run D Method");
            System.out.println(d);
```

```
    }
}
class HIER_ABCD_MAIN
{
    public static void main(String[] args)
    {
            System.out.println("Main Starts");
            new HIER_B().runB();
            new HIER_C().runC();
            new HIER_D().runD();
            System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Run Method
Run B Method
HI
Run Method
Run C Method
HELLO
Run Method
Run D Method
WELCOME
Main Ends
```

## MULTIPLE INHERITANCE




==> Multiple inheritance means single sub class having multiple super class.

==> This is not possible in java by using classes.

==> At any point of time sub class should have only one immediate super class.

**DATE : 13-06-2022**

## NON-STATIC METHODS ARE INHERITED, STATIC METHODS ARE SHARED

```
class E
{
	public static int a=10;
	public int b=98;
}
class F extends E
{
	public void print()
	{
		System.out.println("Shared Variable a= "+a);
```

```
            System.out.println("Inherited Variable b= "+b);
        }
}
class EF_MAIN
{
        public static void main(String[] args)
        {
            F f1=new F();
            F f2=new F();
            F f3=new F();
            F f4=new F();
            f1.print();
            f2.print();
            f3.print();
            f4.print();
        }
}
```

**OUTPUT**

```
Shared Variable a = 10
Inherited Variable b = 98
Shared Variable a = 10
Inherited Variable b = 98
Shared Variable a = 10
Inherited Variable b = 98
Shared Variable a = 10
```

Inherited Variable b = 98

**PURPOSE OF INHERITANCE**

```
class Circle
{
	public void rotate()
	{
		System.out.println("Clock-wise");
	}
}
class Rectangle
{
	public void rotate()
	{
		System.out.println("Clock-wise");
	}
}
class Triangle
{
	public void rotate()
	{
		System.out.println("Clock-wise");
	}
}
class  CRT_MAIN
{
```

```
    public static void main(String[] args)
    {
        Circle c=new Circle();
        Rectangle r=new Rectangle();
        Triangle t= new Triangle();
    }
}
```

==> In the above program we have repeated the code. Due to which we will face maintenance problem in future.

==> To avoid repetitive code among the class and to implement code reusability among the classes, we go for inheritance.

**//PROGRAM**

```
class Shape
{
    public void rotate()
    {
        System.out.println("Clock-wise");
    }
}
class Rectangle extends Shape
{
}
class Circle extends Shape
{
```

```
}
class Triangle extends Shape
{
}
class Shape_CRT_MAIN
{
	public static void main(String[] args)
	{
		Circle c=new Circle();
		c.rotate();
		Rectangle r=new Rectangle();
		r.rotate();
		Triangle t= new Triangle();
		t.rotate();
	}
}
```

**OUTPUT**

```
Clock-wise
Clock-wise
Clock-wise
```

==> In the above program we have defined repetitive code in super class. We have reused the code in sub classes due to reusability of code, maintenance of program is easy in future. **i.e**, even if code change, we need not change multiple times. Rather we have to modify only single time in super class, which will by default effect all the sub classes.

**//PROGRAM**

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A) PROGRAM FOR EMPLOYEES AND TYPES OF EMPLOYEES**
**B) DEVELOPER IS A TYPE OF EMPLOYEE**
**C) TEST ENGINEER IS A TYPE OF EMPLOYEE**
**D) DEVELOPER CHARACTERSTICS ARE ID,NAME,SALARY,LANGUAGE**
**E) TEST ENGINEER CHARACTERSTICS ARE ID,NAME,SALARY,TYPE OF TESTING**
**F) DISPLAY EACH AND EVERY EMPLOYEE DETAILS**

```
class CR_Employee
{
	public String id;
	public String name;
	public double salary;
	public CR_Employee(String id, String name,double salary)
	{
		this.id=id;
		this.name=name;
		this.salary=salary;
	}
	public void employeeDetails()
	{
		System.out.println("Id : "+id);
		System.out.println("Name : "+name);
		System.out.println("Salary : "+salary);
```

```
        }
}
class CR_Developer extends CR_Employee
{
        public String lang;
        public CR_Developer(String id, String name,double salary,String lang)
        {
                super(id,name,salary);
                this.lang=lang;
        }
        public void developerDetails()
        {
                employeeDetails();
                System.out.println("Language : "+lang);
        }
}
class CR_Testing extends CR_Employee
{
        public String typeTesting;
        public CR_Testing(String id, String name,double salary,String typeTesting)
        {
                super(id,name,salary);
                this.typeTesting=typeTesting;
        }
        public void testingDetails()
```

```
	{
		employeeDetails();
		System.out.println("Type Testing : "+typeTesting);
	}
}
class CR_EDT_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		CR_Developer d1=new
CR_Developer("121","RAM",50000.0,"JAVA");
		CR_Testing t1=new
CR_Testing("131","JAY",80000.0,"MANUAL");
		d1.developerDetails();
		t1.testingDetails();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Id: 121
Name: RAM
Salary: 50000.0
Language: JAVA
```

Id: 131

Name: JAY

Salary: 80000.0

Type Testing: MANUAL

Main Ends

**DATE : 15-06-2022**

# METHOD OVER-RIDING

==> During Inheritance process, Sub class can change the method implementation of inherited method. This process is called Method Over-riding.

==> Method Over-riding depends on 2 factors.

1.) Inheritance is must.

2.) Method signature should be same.

==> We go for Method Over-riding, to make changes to method implementation of inherited method for particular sub class, without affecting other sub classes. This is the main purpose of Method Over-riding.

## OTHER ADVANTAGES METHOD OVER-RIDING

1.) Abstract

2.) Interface

3.) Polymorphism

4.) Abstraction

**//PROGRAM**

```
class Demo
{
	public void move()
	{
		System.out.println("Moving North");
```

```
	}
}
class Sample extends Demo
{
	public void move()
	{
		System.out.println("Moving South");
	}
}
class DS_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Sample s1=new Sample();
		s1.move();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Moving South
Main Ends
```

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A) PROGRAM FOR MULTILEVEL INHERITANCE**

**B) CREATE A NON STATIC METHOD IN SUPER CLASS**
**C) OVERRIDE THE METHOD IN LAST SUBCLASS**

```
class MULTI_OR_1
{
	public void run()
	{
		System.out.println("Run Method");
	}
}
class MULTI_OR_2 extends MULTI_OR_1
{
}
class MULTI_OR_3 extends MULTI_OR_2
{
	public void run()
	{
		System.out.println("New Run Method");
	}
}
class MULTI_OR_123_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		MULTI_OR_3 mr=new MULTI_OR_3();
```

```
        mr.run();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

Main Starts

New Run Method

Main Ends

==> Method overriding can happen at any level of sub class

**Q→ WRITE A PROGRAM FOR BELOW REQUIREMENTS**

**A) PROGRAM FOR HIERARCHICAL INHERITANCE(Min 3 sub)**
**B) CREATE A NON STATIC METHOD IN SUPER CLASS**
**C) OVERRIDE THE METHOD IN FIRST AND THIRD SUB CLASS**

```
class HIER_OR_1
{
    public void display()
    {
        System.out.println("Display Method");
    }
}
class HIER_OR_2 extends HIER_OR_1
{
    public void display()
    {
        System.out.println("Display Method HI");
```

```
	}
}
class HIER_OR_3 extends HIER_OR_1
{
}
class HIER_OR_4 extends HIER_OR_1
{
	public void display()
	{
		System.out.println("Display Method HELLO");
	}
}
class HIER_OR_1234_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		HIER_OR_2 hr2=new HIER_OR_2();
		hr2.display();
		HIER_OR_4 hr4=new HIER_OR_4();
		hr4.display();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

Main Starts

Display HI

Display HELLO

Main Ends

==> When a sub class over rides the method, the over-riding will not affect all the sub classes rather it affects only the current sub class.

==> Over-riding is not mandatory, it depends on requirement of application.

**//PROGRAM**

```
class Shape
{
	public void rotate()
	{
		System.out.println("Clock-wise");
	}
}
class Rectangle extends Shape
{
	public void rotate()
	{
		System.out.println("Anti-Clock-wise");
	}
}
class Circle extends Shape
{
```

```
}
class Triangle extends Shape
{


}
class Shape_OR_MAIN
{
        public static void main(String[] args)
        {
                Circle c=new Circle();
                c.rotate();
                Rectangle r=new Rectangle();
                r. rotate();
                Triangle t= new Triangle();
                t.rotate();
        }
}
```

**OUTPUT**

```
Clock-wise
Anti-Clock-wise
Clock-wise
```

## FINAL KEYWORD

==> final is a keyword which represents no more changes.

==> final keyword can be used with respect to variable, method, class.

## FINAL VARIABLE

==> If variables are declared as final, the variable cannot be re-initialized because the given value is the final value (last value).

==> final variables are called constant variables

**//PROGRAM**

```
class Demo
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		final int a=10;
		a=20;		//Error, Remove this statement for successful compliation and execution
		System.out.println(a);
		System.out.println("Main Ends");
	}
}
```

## FINAL METHOD

==> If method is defined as final, we cannot perform method over-riding because the given implementation by super class is final implementation (last implementation).

==> Sub class can inherit and use final method but sub class does not have rights to over-ride.

**//PROGRAM**

```
class Coach
{
        final public void run()
        {
                System.out.println("Run 5kms")
        }
}
class Runner extends Coach
{
        public void run()
        {
                System.out.println("Run 18kms")
        }
//Error, Remove this method for successful compilation and execution
}
class   CR_MAIN
{
        public static void main(String[] args)
        {
                Runner r=new Runner();
                r.run
        }
}
```

## FINAL CLASS

==> If class is defined as final, further inheritance is not possible because final class is the last sub class.

==> final class can have super class but final class cannot have sub class.

**//PROGRAM**

```
class A
{
        //statements
}
final class B extends A
{
        //statements
}
class C extends B
{
        //statements
}
```

//Error, Remove this class for successful compilation and execution

## MULTIPLE INHERITACE PROBLEM

==> Single sub class having multiple super classes is called Multiple Inheritance. But, according to java single sub class cannot have multiple super classes because of "**Diamond problem**".

==> Diamond problem says

1.) Ambiguity in constructor chaining

```
class A
{
```

```
        public A(int a)
        {
        }
}
class B
{
        public B(double b)
        {
        }
}
class C extends A,B
{
        C()
        {
                super()?// Confusion for sub class constructor "Which super class constructor should call?"
        }
}
```

2.) Ambiguity in execution of method implementation

```
class A
{
        public void walk()
        {
                System.out.println("Walk 5 kms")
        }
```

```
}
class B
{
        public void walk()
        {
                System.out.println("Walk 10 kms")
        }
}
class C extends A,B
{
        walk()?// Confusion for sub class, " Which super class method implementation to execute.
}
```

---

## Common Q/A

1. **Can we inherit static members?**

   **A:**

2. **Can we inherit main method?**

   **A:**

3. **Constructors can be inherited?**

   **A:**

4. **Is multiple inheritance possible using classes?**

   **A:**

5. **Can we overload and over-ride non-static method?**

   **A:**

6. **Can we overload and over-ride static method?**

**A:**

7. **Can we over-ride main method?**

**A:**

8. **Can we create an object of final class?**

**A:**

9. **Can we declare final variable and initialize final variable separately in case of local variable? * * *(IMP-1)**

**A:** Yes, we can declare and initialize final local variable in separate statement.

```
class FLocal
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		final int x;
		x=10;
		System.out.println(x);
		System.out.println("Main Ends");
	}
}
```

10. **Can we declare final variable and initialize final variable separately in case of static variable? * * *(IMP-2)**

**A:** No, we cannot declare and initialize final static variable in separate statements. Because compiler will provide default value, that default value becomes final value.

Hence we conclude final static variable should be initialized during declaration.

```
class FStatic
{
    public final static int x;
    public static void main(String[] args)
    {
            System.out.println("Main Starts");
            x=10;
            System.out.println(x);
            System.out.println("Main Ends");
    }
}
```

**// FOR SUCCESSFUL COMPILATION**

```
class CQ2
{
    public final static int x=10;
    public static void main(String[] args)
    {
            System.out.println("Main Starts");
            System.out.println(x);
            System.out.println("Main Ends");
    }
}
```

==> Conclusion is final static variables must be initialized during declaration

**// USING CONSTRUCTOR**

```
class FStatic
```

```
{
    public final static int x;
    public FStatic()
    {
        x=10;          //error
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(x);
        System.out.println("Main Ends");
    }
}
```

==> Conclusion is final static variables cannot be initialized by using constructor. (Normal static variables can be initialized by using constructor.)

**11. Can we declare final variable and initialize final variable separately in case of non-static variable? * * ***

**A:** In case of final Non-static variable we have 2 choice

1.) Declare and initialize in the same statement
2.) Declare separately but compulsory initialize by using constructor.

```
class FNonStatic
{
    public final int x;
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
```

```
                FNonStatic fns=new FNonStatic();
                x=10;
                System.out.println(fns.x);
                System.out.println("Main Ends");
        }
}
```

**// FOR SUCCESSFUL COMPILATION**

```
class FNonStatic
{
        public final int x=10;
        public static void main(String[] args)
        {
                System.out.println("Main Starts");
                FNonStatic fns=new FNonStatic();
                System.out.println(fns.x);
                System.out.println("Main Ends");
        }
}
```

==> Conclusion is final Non-static variables must be initialized during declaration

**// USING CONSTRUCTOR**

```
class FNonStatic
{
        public final int x;
        public FNonStatic()
```

```
    {
        x=10;
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        FNonStatic fns=new FNonStatic();
        System.out.println(fns.x);
        System.out.println("Main Ends");
    }
}
```

==> Conclusion is final Non-static variables can be initialized by using constructor.

**12. Can we inherit final method?**

**A:**

**13. Can we over-ride final method?**

**A:**

---

**DATE : 16-06-2022**

## ABSTRACT

==> abstract is a keyword which represents incompleteness.

==> There are 2 types of method.

1.) Complete Method: Method with signature and Implementation.

2.) Incomplete Method: Method with signature and without Implementation.

==> Complete method is also called as Concrete method.

==> Incomplete method is also called as Abstract method.

**STEPS TO DESIGN ABSTRACT METHOD**

==> Method will have only method signature.

==> Method signature should end with semicolon.

==> Method signature will have abstract keyword.

==> If class contains abstract method, then the whole class becomes abstract class.

**STEPS TO USE ABSTRACT METHOD**

==> Inherit the abstract method.

==> Over-ride the method (or) Complete the method (or) Implement the method

**//PROGRAM**

```
abstract class Demo
{
	abstract public void test();
}
class Sample extends Demo
{
	public void test()
	{
		System.out.println("Test Completed");
	}
}
class DS_MAIN
{
	public static void main(String[] args)
	{
```

```
        new Sample().test();
    }
}
```

**OUTPUT**

Test Completed

**DATE : 17-06-2022**

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.) PROGRAM FOR SINGLE LEVEL INHERITANCE**
**B.) CREATE TWO ABSTRACT METHOD AND COMPLETE THE METHODS**

```
abstract class Super
{
    abstract public void run();
    abstract public void walk();
}
class Sub
{
    public void run()
    {
        System.out.println(" Run Method");
    }
    public void walk()
    {
        System.out.println(" Walk Method");
    }
```

```
}
class SupSub_Main
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Sub sb=new Sub();
        sb.run();
        sb.walk();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Run Method
Walk Method
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A.) PROGRAM FOR SINGLE LEVEL INHERITANCE**

**B.) CREATE A COMPLETE AND INCOMPLETE METHOD IN SUPER CLASS,COMPLETE THE INCOMPLETE METHOD**

```
abstract class SL_Super
{
    abstract public void print();
    public void scan()
```

```
	{
		System.out.println("Scan Method");
	}
}
class SL_Sub extends SL_Super
{
	public void print()
	{
		System.out.println("Print Method");
	}
}
class SL_SupSub_Main
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		SL_Sub sb=new SL_Sub();
		sb.print();
		sb.scan();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
	Main Starts
	Print Method
```

Scan Method

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A. PROGRAM FOR HIERARCHICAL INHERITANCE**
**B. CREATE AN ABSTRACT METHOD IN SUPER CLASS**
**C. COMPLETE THE METHOD**

```
abstract class HIER_Super
{
    abstract public void write();
}
class HIER_Sub extends HIER_Super
{
    public void write()
    {
        System.out.println("Write Java");
    }
}
class HIER_Sub1 extends HIER_Super
{
    public void write()
    {
        System.out.println("Write JavaScript ");
    }
}
class HIER_SupSub_Main
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        new HIER_Sub().write();
        new HIER_Sub1().write();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Write Java
Write JavaScript
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

**A. PROGRAM FOR MULTI-LEVEL INHERITANCE**
**B. CREATE TWO ABSTRACT IN SUPER MOST CLASS**
**C. COMPLETE FIRST METHOD IN NEXT SUB CLASS, COMPLETE THE SECOND METHOD IN NEXT LEVEL SUB CLASS (* * * * *)**

```
abstract class ML_Super
{
    abstract public void listen();
    abstract public void speak();
}
abstract class ML_Sub extends ML_Super
```

NOTE : ML_Sub class is abstract because, still there is an incomplete method gave

```
{
	public void listen()
	{
		System.out.println("Listen Method");
	}
}
class ML_Sub1 extends ML_Sub
{
	public void speak()
	{
		System.out.println("Speak Method");
	}
}
class ML_SupSub_Main
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		ML_Sub1 sb= new ML_Sub1();
		sb.listen();
		sb.speak();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

Main Starts

Listen Method

Speak Method

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **PROGRAM FOR MULTI-LEVEL INHERITANCE**
B. **CREATE COMPLETE AND INCOMPLETE METHOD IN SUPER MOST CLASS**
C. **OVERRIDE THE FIRST METHOD IN FIRST SUB CLASS, OVERRIDE THE SECOND METHOD IN NEXT LEVEL SUB CLASS**

```
abstract class MLOR_Super
{
    abstract public void message();
    public void call()
    {
        System.out.println("Call Method");
    }

}
abstract class MLOR_Sub extends MLOR_Super
{
    public void call()
    {
        System.out.println("New Call Method");
    }
```

```
}
class MLOR_Sub1 extends MLOR_Sub
{
	public void message()
	{
		System.out.println("Message Method");
	}
}
class MLOR_SupSub_Main
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		MLOR_Sub1 sb= new MLOR_Sub1();
		sb.message();
		sb.call();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Message Method
New Call Method
Main Ends
```

**PURPOSE OF ABSTRACT**

==> Abstract methods are created to provide standard method signature to all the sub classes.

==> Abstract methods are created to achieve loose coupling.

**OTHER ADVANTAGES OF ABSTRACT**

1.) Polymorphism
2.) Abstraction

**DATE : 18-06-2022**

# INTERFACE

==> Interface is one of the type definition block used for creating user defined data types.

==> Interface allows only static variables.

==> All the variables in interface are by default final and static.

==> Interface allows only incomplete methods.

==> All the methods in interface are by default public and abstract.

# ABSTRACT

==> Multiple Inheritance is possible by using interface because there is no diamond problem. **i.e**, there is no Ambiguity (confusion) in Constructor chaining. Because interface does not allows constructor.

==> There is no Ambiguity in Execution of method implementation because does not allow method implementation even if all the interfaces are having same signature, we have tom implement one time.

**PURPOSE OF INTERFACE**

==> By defining interface, we can provide standard method signature for all the implementation class.

==> By defining interface, we can achieve loose coupling.

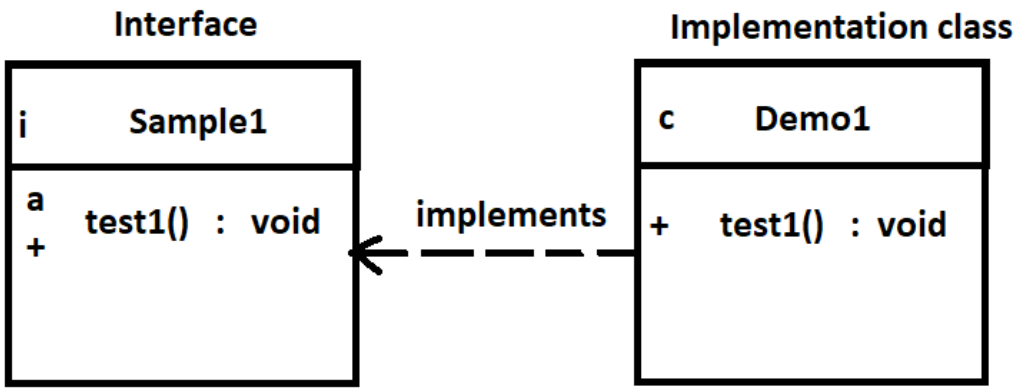
==> By using Interface we can achieve Multiple Inheritance.

**OTHER ADVANTAGES**

1.) Polymorphism

2.) Abstraction

**//PROGRAM**




```
interface Sample
{
	public void test();
}
class Demo implements Sample
{
	public void test()
	{
		System.out.println("Test completed");
	}
}
class INTER_MAIN
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        new Demo().test();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Test Completed
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE 2 INTERFACES AND PERFORM SINGLE LEVEL INHERITANCE**
B. **CREATE IMPLEMENTATION CLASS**

```
interface Super
{
    public void run();
}
interface Sub extends Super
{
    public void walk();
}
```

```
class Inter implements Sub
{
	public void run()
	{
		System.out.println("Run Method");
	}
	public void walk()
	{
		System.out.println("Walk Method");
	}
}
class SuperSub_Main
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Inter i= new Inter();
		i.run();
		i.walk();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

Main Starts

Run Method

Walk Method

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE 3 INTERFACES AND PERFORM MULTI-LEVEL INHERITANCE**
B. **CREATE IMPLEMENTATION CLASS**

```
interface Mobile
{
	public void call();
}
interface Car extends Mobile
{
	public void drive();
}
interface Cycle extends Car
{
	public void ride();
}
class ML_INTER implements Cycle
{
	public void call()
	{
		System.out.println("Call Method!");
```

```
	}
	public void drive()
	{
		System.out.println("Drive Method!");
	}
	public void ride()
	{
		System.out.println("Ride Method!");
	}
}
class  ML_INTER_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		ML_INTER ml=new ML_INTER();
		ml.call();
		ml.drive();
		ml.ride();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
	Main Starts
	Call Method
```

Drive Method

Ride Method

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE 3 INTERFACES AND PERFORM HIERARCHICAL INHERITANCE**
B. **CREATE IMPLEMENTATION CLASS**

```
interface HTML
{
	public void design();
}
interface CSS extends HTML
{
	public void decorate();
}
interface JAVASCRIPT extends HTML
{
	public void link();
}
class HIER_INTER implements CSS
{
	public void design()
	{
		System.out.println("Design Method");
	}
```

```
	public void decorate()
	{
		System.out.println("Decorate Method");
	}
}
class HIER_NEW_INTER implements JAVASCRIPT
{
	public void design()
	{
		System.out.println("Design Method");
	}
	public void link()
	{
		System.out.println("Link Method");
	}
}
class  HIER_NEW_INTER_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		HIER_INTER h1= new HIER_INTER();
		h1.design();
		h1.decorate();
		HIER_NEW_INTER h2= new HIER_NEW_INTER();
```

```
        h2.design();
        h2.link();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Design Method
Decorate Method
Design Method
Link Method
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE 3 INTERFACES AND PERFORM HIERARCHICAL INHERITANCE**
B. **CREATE IMPLEMENTATION CLASS (* * * * *)**

```
interface Sample1
{
    public void test1();
}
interface Sample2
{
    public void test2();
}
interface Sample3 extends Sample1,Sample2
```

```
{
        public void test3();
}
class Demo implements Sample3
{
        public void test1()
        {
                System.out.println("Test1 completed");
        }
        public void test2()
        {
                System.out.println("Test2 completed");
        }
        public void test3()
        {
                System.out.println("Test3 completed");
        }
}
class MUL_MAIN
{
        public static void main(String[] args)
        {
                System.out.println("Main Starts");
                Demo d=new Demo();
                d.test1();
```

```
        d.test2();
        d.test3();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

Main Starts

Test1 completed

Test2 completed

Test3 completed

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE 2 INTERFACE**
B. **CREATE SINGLE IMPLEMENTATION CLASS WHICH IMPLEMENTS BOTH THE INTERFACE(* * * * *)**

```
interface Demo1
{
    public void demo1();
}
interface Demo2
{
    public void demo2();
}
class Sample_INTER implements Demo1,Demo2
```

```
{
	public void demo1()
	{
		System.out.println("First Demo");
	}
	public void demo2()
	{
		System.out.println("Second Demo");
	}
}
class Sample_INTER_Main
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Sample_INTER si=new Sample_INTER();
		si.demo1();
		si.demo2();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
First Demo
Second Demo
```

Main Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE AN INTERFACE**
B. **CREATE AN ABSTRACT CLASS**
C. **CREATE A CLASS WHICH BEHAVES LIKE SUB CLASS AS WELL AS IMPLEMENTATION CLASS**

```
interface Inter
{
	public void scan();
}
abstract class Abstract implements Inter
{
	abstract public void print();
}
class INTER_ABS extends Abstract
{
	public void scan()
	{
		System.out.println("Scan Method");
	}
	public void print()
	{
		System.out.println("Print Method");
	}
}
class INTER_ABS_MAIN
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        INTER_ABS ias=new INTER_ABS();
        ias.scan();
        ias.print();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Scan Method
Print Method
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE THREE INTERFACES AND PERFORM MULTIPLE INHERITANCE**
B. **ALL THE INTERFACE CONTAINS SAME METHOD**
C. **PROVIDE SAME IMPLEMENTATION CLASS**

```
interface Room1
{
    public void guest();
}
interface Room2
{
```

```
        public void guest();
}
interface Room3 extends Room1,Room2
{
        public void guest();
}
class Room implements Room3
{
        public void guest()
        {
                System.out.println("Only Room2 is available");
        }
}
class Room_MAIN
{
        public static void main(String[] args)
        {
                System.out.println("Main Starts");
                Room r=new Room();
                r.guest();
                System.out.println("Main Ends");
        }
}
```

**OUTPUT**

```
Main Starts
```

Only Room2 is available

Mani Ends

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE A COMPLETE CLASS**
B. **CREATE AN INTERFACE**
C. **CREATE A CLASS BEHAVES LIKE SUB CLASS / IMPLEMENTS CLASS**

```
class Run
{
	public void cycle()
	{
		System.out.println("Cycle 5 kms");
	}
}
interface Sample
{
	public void test();
}
class Demo extends Run implements Sample
{
	public void test()
	{
		System.out.println("Test completed");
	}
}
class RunDemoSample_MAIN
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        Demo d=new Demo();
        d.cycle();
        d.test();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Cycle 5 kms
Test Completed
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENTS**

A. **CREATE A COMPLETE CLASS**
B. **CREATE AN INTERFACE**
C. **CLASS AND INTERFACE HAS SAME METHOD**
D. **CREATE A CLASS BEHAVES LIKE SUB CLASS / IMPLEMENTS CLASS**

```
class NewRun
{
    public void move()
    {
        System.out.println("Move North");
```

```
        }
}
interface NewSample
{
        public void move();
}
class NewDemo extends NewRun implements NewSample
{


}
class NewRunDemoSample_MAIN
{
        public static void main(String[] args)
        {
                System.out.println("Main Starts");
                NewDemo nd=new NewDemo();
                nd.move();
                System.out.println("Main Ends");
        }
}
```

**OUTPUT**

```
        Main Starts
        Move North
        Main Ends
```

==> NewDemo need not compulsory complete the method because same method is already completed in NewRun. Hence NewDemo is indirectly complete method.

**(OR)**

**//PROGRAM**

```
class A_NewRun
{
	public void move()
	{
		System.out.println("Move North");
	}
}
interface A_NewSample
{
	public void move();
}
class A_NewDemo extends A_NewRun implements A_NewSample
{
	public void move()
	{
		System.out.println("Move South");
	}
}
class A_NewRunDemoSample_MAIN
{
	public static void main(String[] args)
```

```
    {
        System.out.println("Main Starts");
        A_NewDemo and=new A_NewDemo();
        and.move();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
    Main Starts
    Move South
    Main Ends
```

==> NewDemo can also implement the method once again. In that case Super class implementation override with Sub class implementation.

**//PROGRAM**

```
interface Junior
{
    public int i=10; /* Variable i is by default static and final*/
}
class Junior_MAIN
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        System.out.println(Junior.i);
        Junior i=99;
```

```
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

error

---

**NOTE:**

→Interface allows

==> Static variable (Possible)

==> Non-static variable (Not possible)

==> Complete Static method (Not Possible)

==> Complete Non-static method (Not Possible)

==> Abstract Static method (Possible)

==> Abstract Non-static method (Possible)

==> Constructor (Not Possible)

---

**DATE : 22-06-2022**

---

**NOTE-1:**

abstract public void test();

| | **Inheritance** | **Over-riding** |
|---|---|---|
| **Absract** | Possible | Possible |
| **Non-static** | - | - |

==> Incomplete static methods cannot be created. Because Incomplete method requires Inheritance and over-riding. But Static method does not support

Inheritance and Over-riding. Hence, we can conclude Static methods cannot be Incomplete methods rather Static methods are always Complete methods

**NOTE-2:**
abstract public void test();

| | Inheritance | Over-riding |
|---|---|---|
| **Abstract** | Possible | Possible |
| **Non-static** | Possible | Possible |

==> Non-static methods can be Incomplete. Because Incomplete method requires Inheritance and over-riding whereas Non-static supports Inheritance and Over-riding. Hence, Non-static methods can be Incomplete as well as Complete method.

**NOTE-3:**
final public void test();

| | Inheritance | Over-riding |
|---|---|---|
| **Abstract** | Possible | Possible |
| **Final** | Possible | - |

==> final Incomplete method cannot be created because Incomplete method requires Inheritance and Over-riding but final method supports Inheritance, does not supports Over-riding. Hence, final method cannot be developed as Incomplete method rather final method must be Complete Method.

---

**DATE : 23-06-2022**

---

## Common Q/A

1. **Can we create constructor in abstract class?**

   **A:** Yes, because class allows Non-static variable, in order to initialize Non-static variable we need constructor.

2. **Does Interface allow constructor?**

**A:** No, because interface does not allow Non-static variable. Hence interface does not allow constructors.

**3. Can we create Static method in Interface?**

**A:** Yes, complete static method is allowed in Interface.

**4. Is Multiple Inheritance possible using abstract class?**

**A:** No, because of Diamond Problem.

**5. Can we create an abstract class without abstract method?**

**A:** Yes, there is no rule abstract method is mandatory in abstract class.

**6. Can we create an object of abstract class?**

**A:** No, because class is abstract.

**7. Can we create reference variable of abstract class?**

**A:** Yes, because abstract reference variable can be used for up-casting.

**8. Can we create an object of interface?**

**A:** No, because interface is by default abstract.

**9. Can we create reference variable of interface?**

**A:** Yes, because interface reference variable can be used for up-casting.

**10. Can we declare interface as abstract?**

**A:** Yes, but it is unnecessary because interface is by default abstract.

**11. Can we use class and interface (or) interface and class keywords?**

**A:** No

```
class interface Run( Not Possible )
{
}
interface class Run( Not Possible )
{
```

}

Conclusion is Class should be used separately, Interface should be used seperately

---

# OBJECT TYPE CASTING

==> Casting means converting

A(Super class properties)

B(Super class properties+ Sub class properties)

where A→B is Down-casting (Sub class properties are visible)

B→A is Up-casting (Sub class properties are hidden)

==> Process of converting one object type into another object type is called Object Type Casting.

==> Object Type Casting is classified into 2 types.

1. Up-casting
2. Down-casting

==> Process of converting sub class object to look like super class object is called Up-casting.

==> Up-casting is possible because sub class object will have properties of super class.

==> During Up-casting process, Sub class properties are hidden, Super class properties are shown.

==> Up-casting can be achieved in Implicit manner as well as Explicit manner.

==> Up-casting can be achieved in Implicit manner because every sub class will have only one immediate super class. Hence, Sub class object can be automatically converted to Super class object.

==> Converting super class object back to sub class object is called Down-casting.

==> Direct down-casting is not possible because super class object will not have properties of sub class.

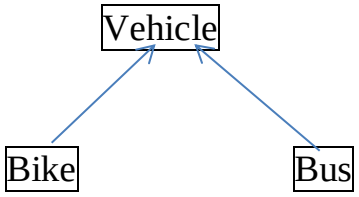
==> Down-casting is possible only after Up-casting.

==> When object is down-casted hidden sub class properties becomes visible.

==> Down-casting can be achieved only in Explicit way.

==> Down-casting is explicit because single super class can have multiple sub classes. Hence, Developer should involve for converting super class object to sub class object.

**Example for Implicit and Explicit**




Explicit Up-casting : Vehicle v1=(Vehicle) new Bike();

Implicit Up-casting : Vehicle v1= new Bike();

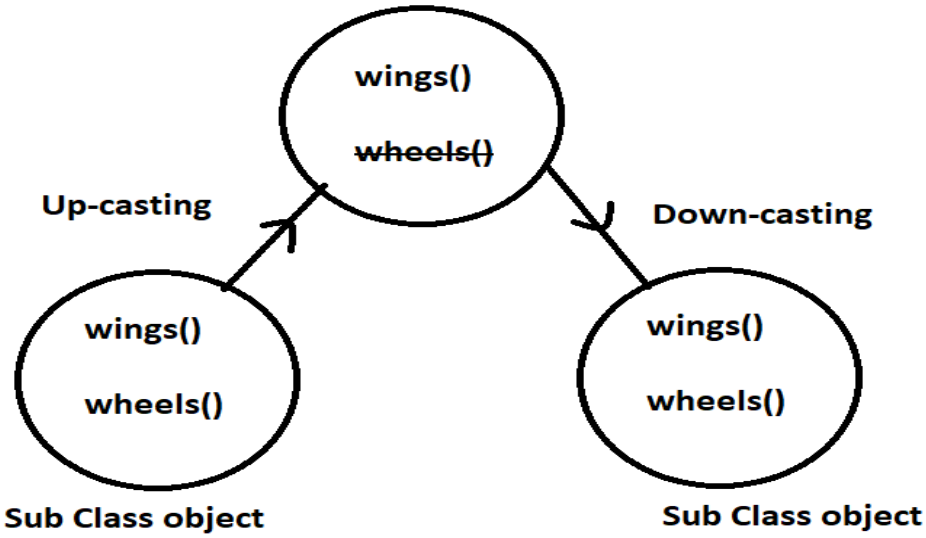
Explicit Down-casting : Bike b1=(Bike) v1;

Explicit Up-casting : Vehicle v2=(Vehicle) new Bus();

Implicit Up-casting : Vehicle v2= new Bus();

Explicit Down-casting : Bus b2=(Bus) v2;

**//PROGRAM**




```
class A
{
	public void wings()
	{
		System.out.println("Use Wings");
	}
}
class B extends A
{
	public void wheels()
	{
		System.out.println("Use Wheels");
	}
}

class AB_MainClass
```

```
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        //Up-casting
        A a= (A)new B();
        a.wings();
        //Down-casting
        B b=(B)a;
        b.wings();
        b.wheels();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Use Wings
Use Wings
Use Wheels
Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A. PROGRAM HIERARCHICAL INHERITANCE**
**B. PERFORM UP-CASTING AND DOWN-CASTING**

```
class HIER_1
{
```

```
	public void hello()
	{
		System.out.println("Hello Method");
	}
}
class HIER_2 extends HIER_1
{
	public void hai()
	{
		System.out.println("Hai Method");
	}
}
class HIER_3 extends HIER_1
{
	public void welcome()
	{
		System.out.println("Welcome Method");
	}
}
class HIER_MAINCLASS
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		HIER_1 h1=(HIER_1) new HIER_2();
```

```
        h1.hello();
        HIER_2 h2=(HIER_2)h1;
        h2.hello();
        h2.hai();
        HIER_1 h3=(HIER_1) new HIER_3();
        h3.hello();
        HIER_3 h4=(HIER_3)h3;
        h4.hello();
        h4.welcome();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Hello Method
Hello Method
Hai Method
Hello Method
Hello Method
Welcome Method
Main Ends
```

**DATE : 25-06-2022**

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A. PROGRAM FOR SINGLE LEVEL INHERTIANCE**
**B. PERFORM METHOD OVER-RIDING**

## C. PERFORM UPCASTING AND CALL THE METHOD

```
class SL_SUPER
{
	public void move()
	{
		System.out.println("Move Method");
	}
}
class SL_SUB extends SL_SUPER
{
	public void move()
	{
		System.out.println("New Move Method");
	}
}
class SL_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		SL_SUPER sl=(SL_SUPER)new SL_SUB();
		sl.move();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

Main Starts

New Move Method

Main Ends

==> After method over-riding if we perform Up-casting and call the method, we will get sub class implementation output because method is already over-rided.

==> Down-casting is unnecessary if sub class does not have sub class property.

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

A. **PROGRAM FOR CREATING ABSTRACT CLASS WITH ABSTRACT METHOD.**
B. **PERFORM UPCASTING AND CALL THE METHOD**

```
abstract class ABST_SUPER
{
	abstract public void scan();
}
class ABST_SUB extends ABST_SUPER
{
	public void scan()
	{
		System.out.println("Scan Method");
	}
}
class ABST_MAIN
{
	public static void main(String[] args)
	{
```

```
        System.out.println("Main Starts");
        ABST_SUPER abs= (ABST_SUPER) new ABST_SUB();
        abs.scan();
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
    Main Starts
    Scan Method
    Main Ends
```

**Q→ WRITE A PROGRAM FOR THE BELOW REQUIREMENT**

**A. CREATE AN INTERFACE WITH ABSTRACT METHOD.**
**B. CREATE AN IMPLEMENTATION CLASS.**
**C. PERFORM UPCASTING AND CALL THE METHOD.**

```
interface SUPER
{
    void print();
}
class SUB implements SUPER
{
    public void print()
    {
        System.out.println("Print Method");
    }
}
```

```
class INTER_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		SUPER s= (SUPER) new SUB();
		s.print();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
Main Starts
Print Method
Main Ends
```

# METHOD BINDING

==> Process of connecting method signature with method implementation is called Method Binding.

==> There are 3 stages in Java process.

1. Coding
2. Compilation
3. Execution

==> Method Binding will never occur during coding stage.

==> Method Binding will occur either during compilation (or) execution stage.

==> Static methods are binded by Compiler.

==> Non-Static methods are binded by JVM.

==> If methods are binded at the time of compilation, it is called Compile Time Binding.

==> If methods are binded at the time of execution, it is called Run Time Binding.

==> Compile Time Binding is also known as Static Binding.

==> It is called Static Binding because binding between method signature and method implementation cannot be changed.

==> Run Time Binding is also known as Dynamic Binding.

==> It is called Dynamic Binding because binding between method signature and method implementation can change based on object creation.

==> Compile Time Binding is also known as Early Binding.

==> It is called Early Binding because methods are already binded before execution and methods are ready to execute.

==> Run Time Binding is also known as Late Binding.

==> It is called Late Binding because JVM will not perform method binding as soon as execution starts. There is a delay for binding Non-Static method. During the delay time, JVM will perform pre-condition activities. Hence, since there is a delay for binding it is called Late Binding.

==> Pre-condition activities are

1. Creation of Stack area and Heap area.
2. Execution of Class loader.
3. Execution of Main method.

---

**NOTE:**

→ During execution of method, if object is created, then Non-Static methods are binded.

---

**DATE : 27-06-2022**

## SPECIALIZATION

==> Process of creating a method which can handle multiple objects of same type. These methods are called Specialized method and the process is called Specialization.

## STEPS TO CREATE SPECIALIZED METHODS

==> Create a method (Static or Non-Static)

==> Method should have arguments.

==> Argument should be reference variable type of same class.

==> Call the method and pass same class object.

---

**NOTE:**

→ Repetitive code should go inside specialized method. This Specialized method will be reused for each and every object.

---

**//Example**

```
class Demo
{
}
class Sample
{
}
public static void test1(int a)
{
}
public static void test2(Demo d1)        // Specialized for Demo type objects
{
```

```
}
public static void test3(Sample s1)      // Specialized for Sample type objects
{
}
class Main
{
        test1(79);
        test2( new Demo1);
        test3(new Sample());
}
```

**EXAMPLE FOR SPECIALIZATION**

```
class Book
{
        public String bName;
        public Book(String bName)
        {
                this.bName=bName;
        }
        public void write()
        {
                System.out.println("Write " +bName+ " book");
        }
        public void read()
        {
                System.out.println("Read " +bName+ " book");
```

```
	}
}
class MainClass1
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Book b1=new Book("JAVA");
		Book b2=new Book("HTML");
		Book b3=new Book("CSS");
		b1.write();    // Repeated code for each and every object
		b1.read();     // Repeated code for each and every object
		b2.write();
		b2.read();
		b3.write();
		b3.read();
		System.out.println("Main Ends");
	}
}
```

**// USING CODE REUSABILITY**

```
class Book
{
	public String bName;
	public Book(String bName)
	{
```

```
        this.bName=bName;
    }
    public void write()
    {
        System.out.println("Write " +bName+ " book");
    }
    public void read()
    {
        System.out.println("Read " +bName+ " book");
    }
}
class Book_MAIN
{
    public static void exam(Book book)
    {   //Specialized method for handling Book objects
        book.write();
        book.read();
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        exam(new Book("JAVA"));
        exam(new Book("HTML"));
        exam(new Book("CSS"));
        System.out.println("Main Ends");
```

```
    }
}
```

**OUTPUT**

```
Main Starts
Read JAVA book
Write JAVA book
Read HTML book
Write HTML book
Read CSS book
Write CSS book
Main Ends
```

**GENERALIZATION**

==> Process of creating a method which can handle multiple objects of different types. These methods are called Generalized method and the process is called Generalization.

**STEPS TO CREATE GENERALIZED METHODS**

==> Create a method (Static or Non-Static).

==> Method should have argument.

==> Argument should be reference variable type of Super class.

==> Call the method and pass sub class object.

---

**NOTE-1:**

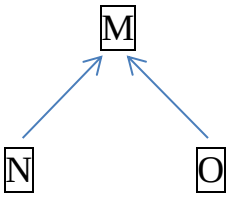
→ In Generalization there will be upcasting.

**NOTE-2:**

→ Repetitive code should be defined inside Generalized method and reuse the method for each and every sub class object.

---

**//Example**




```
class M
{
}
class N extends M
{
}
class O extends M
{
}
public static void test(M m1)     //Generalized method which can handle all the
                                   Sub class objects [ Different type of objects ]
{
}
class Main
{
        test(new N());
        test(new O());
}
```

---

**NOTE:**

→ We can also create a method with reference variable as an argument.

→ If reference variable is argument, we have to pass object when we call the method.

---

**EXAMPLE FOR GENERALIZATION**

```
class Employee
{
	public void work()
	{
		System.out.println("Work from 9AM to 6PM");
	}
	public void getSalary()
	{
		System.out.println("Salary will be credited by end of the month");
	}
}
class Developer extends Employee
{
}
class Testing extends Employee
{
}
class Admin extends Employee
{
}
class EDT_MAIN
{
	public static void main(String[] args)
	{
```

```
        System.out.println("Main Starts");
        Developer d = new Developer();
        Testing t = new Testing();
        Admin a = new Admin();
        d.work();      //Repetitive code for Developer type
        d.getSalary();
        t.work();      //Repetitive code for Testing type
        t.getSalary();
        a.work();      //Repetitive code for Admin type
        a.getSalary();
        System.out.println("Main Ends");
    }
}
```

**// USING CODE REUSABILITY**

```
class Employee
{
    public void work()
    {
        System.out.println("Work from 9AM to 6PM");
    }
    public void getSalary()
    {
        System.out.println("Salary will be credited by end of the month");
    }
}
```

```
class Developer extends Employee
{
}
class Testing extends Employee
{
}
class Admin extends Employee
{
}
class EDTA_MAIN
{
	public static void company(Employee emp)
	{	//Generalized method for handling Sub class objects
		emp.work();
		emp.getSalary();
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		company(new Developer());
		company(new Testing());
		company(new Admin());
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

Main Starts

Work from 9AM to 6PM

Salary will be credited by end of the month

Work from 9AM to 6PM

Salary will be credited by end of the month

Work from 9AM to 6PM

Salary will be credited by end of the month

Main Ends

**Conclusion:** "Specialization and Generalization are used for avoiding code repetition and to implement code reusability."

**DATE : 28-06-2022**

**STANDARDIZATION AND LOOSE COUPLING**

==>

**//EXAMPLE FOR STANDARDIZATION**

Associations

==> They have fixed charges.

==> They have to pay some amount for joining the association.

Mobiles

==> They have fixed chargers in olden days.

**//EXAMPLE FOR LOOSE COUPLING**

Pens

==> Some pens have refills to change into new refills.

Ice-cream cones

==> For Empty cones, we can fill with any flavours of ice-cream as we want.

# POLYMORPHISM

==> Single entity having multiple forms is called Polymorphism.

==> Polymorphism is classified into 2 types

1. Run Time Polymorphism
2. Compile Time Polymorphism

## RUNTIME POLYMORPHISM

==> Call to over-rided method is decided during runtime based on the object creation. This is known as Runtime Polymorphism.

==> In order to achieve Run Time Polymorphism we should make use of Inheritance, Method Over-riding, Generalization and Up-casting

**//PROGRAM**

```
interface Animal
{
	void noise();
}
class Lion implements Animal
{
	public void noise()
	{
		System.out.println("Lion Roars");
	}
}
class Dog implements Animal
{
	public void noise()
```

```
	{
		System.out.println("Dog Barks");
	}
}
class Cat implements Animal
{
	public void noise()
	{
		System.out.println("Cat Meows");
	}
}
class ALDC_MAIN
{
	public static void makeSound(Animal a)
	{
		a.noise();
	}
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		makeSound(new Lion());
		makeSound(new Dog());
		makeSound(new Cat());
		System.out.println("Main Ends");
	}
```

```
}
```

**OUTPUT**

```
Main Starts
Lion Roars
Dog Barks
Cat Meows
Main Ends
```

**//PROGRAM**

```
abstract class Shape
{
	abstract public void getArea();
}
class Circle extends Shape
{
	public final static double pi = 3.14;
	public int r;
	public Circle(int r)
	{
		this.r=r;
	}
	public void getArea()
	{
		System.out.println("Area of Circle :"+pi*r*r);
	}
```

```
}
class Rectangle extends Shape
{
	public double length;
	public double width;
	public Rectangle(double length,double width)
	{
		this.length=length;
		this.width=width;
	}
	public void getArea()
	{
		System.out.println("Area of Rectangle :"+length*width);
	}
}
class Triangle extends Shape
{
	public int base;
	public int height;
	public Triangle(int base,int height)
	{
		this.base=base;
		this.height=height;
	}
	public void getArea()
```

```
    {
        System.out.println("Area of Triangle :"+0.5*base*height);
    }
}
class SCRT_MAIN
{
    public static void printArea(Shape shape)
    {
        shape.getArea();
    }
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        printArea(new Circle(5));
        printArea(new Rectangle(4.0,6.0));
        printArea(new Triangle(2,8));
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Area of Circle: 78.5
Area of Rectangle: 24.0
Area of Triangle: 8.0
Main Ends
```

## COMPILE TIME POLYMORPHISM

==> Call to over-loaded method is decided during compile time based on the compile time. This is known as Compile Time Polymorphism.

==> In order to achieve Compile Time Polymorphism we should make use of Static method and Method Over-loading.

**//PROGRAM**

```
class Addition
{
	public static void add(int i, int j)
	{
		System.out.println(i+j);
	}
	public static void add(double i, int j)
	{
		System.out.println(i+j);
	}
	public static void add(int i, int j, int k)
	{
		System.out.println(i+j+k);
	}
}
class ADD_MAIN
{
	public static void main(String[] args)
	{
```

```
        System.out.println("Main Starts");
        Addition.add(3,5);
        Addition.add(9.5,7);
        Addition.add(4,5,6);
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
8
16.5
15
Main Ends
```

**DATE : 29-06-2022**

# ABSTRACTION

==> In general terms, abstraction means hiding unnecessary details and providing access only to necessary details.

==> In Java, Abstraction means hiding method implementation and providing access only to method signature.

## STEPS TO DESIGN ABSTRACTION PROGRAM

1. Create an interface.
2. Create a method inside interface.
3. Create an implementation class.
4. Implement the method of interface.
5. Create a Helper class.
6. Create a Helper method.

7. Helper method will create an object of implementation class, Up-cast the object to interface, return the object.

**STEPS TO USE ABSTRACTION PROGRAM**

1. Know the interface.
2. Know the methods of interface.
3. It is unnecessary to know the implementation class.
4. It is unnecessary to know "How method is implemented**?**".
5. Know the Helper class.
6. Know the Helper method is it static or Non-static.

---

**NOTE:**

→ If helper method is Static, call the method by using Helper class name

→ If helper method is Non-Static, call the method by using Helper class object.

---

7. After calling the Helper method, it will return the object, receive the object in interface reference variable.
8. Call the interface method by using interface reference variable.

**//PROGRAM**

**/* Completes STEP-1 and STEP-2*/**

```
interface A_Sample
{
        void test();
}
```

**/* Completes STEP-3 and STEP-4*/**

```
class A_Demo implements A_Sample
{
        public void test()
```

```
	{
		System.out.println("Logic for Test method");
	}
}
```

**/* Completes STEP-5,STEP-6 and STEP-7*/**

```
class A_Run
{
	public static A_Sample giveObject()
	{
		A_Sample s = new A_Demo();
		return s;
	}
}
class A_SDR_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		A_Sample rv = A_Run.giveObject();
		rv.test();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
	Main Starts
```

Logic for Test method

Main Ends

**Q→ WRITE A PROGRAM TO DESIGN ABSTRACTION USING ABSTRACT CLASS AND USE ABSTRACTION PROGRAM**

**/* Completes STEP-1 and STEP-2*/**

```
abstract class Vehicle
{
    abstract public void scan();
}
```

**/* Completes STEP-3 and STEP-4*/**

```
class Test extends Vehicle
{
    public void scan()
    {
        System.out.println("Logic for Scan method");
    }
}
```

**/* Completes STEP-5,STEP-6 and STEP-7*/**

```
class Run
{
    public static Vehicle getObject()
    {
        Vehicle v = new Test();
        return v;
    }
```

```
}
class VTR_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		Vehicle rv = Run.getObject();
		rv.scan();
		System.out.println("Main Ends");
	}
}
```

**// Non-Static Way**

**/* Completes STEP-1 and STEP-2*/**

```
abstract class N_Vehicle
{
	abstract public void scan();
}
```

**/* Completes STEP-3 and STEP-4*/**

```
class N_Test extends N_Vehicle
{
	public void scan()
	{
		System.out.println("Scan method");
	}
}
```

**/* Completes STEP-5,STEP-6 and STEP-7*/**

```
class N_Run
{
	public N_Vehicle getObject()
	{
		N_Vehicle v = new N_Test();
		return v;
	}
}
class N_VTR_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Starts");
		N_Run nr=new N_Run();
		N_Vehicle nv=nr.getObject();
		nv.scan();
		System.out.println("Main Ends");
	}
}
```

**OUTPUT**

```
	Main Starts
	Scan Method
	Main Ends
```

==> In Abstraction user faces 2 challenges

1. Since user does not know implementation class, user is unable to create an object of implementation class. So, Helper method will create an object of implementation class.

2. Since user does not know implementation class, user is unable to create implementation class reference variable. So, Helper method is up-casting the object of implementation class to interface.

**PURPOSE OF ABSTRACTION**

==> We go for abstraction to avoid complexity of implementation from user.

==> We go for abstraction to avoid confusion in implementation from the user.

==> We go for abstraction to make our program easier to use.

---

**NOTE:**

→ Abstraction can be achieved in two ways

1. By using Interface
2. By using Abstract class

---

**DATE : 30-06-2022**

# ENCAPSULATION

==> Binding data members(variables) with member function(methods) into single unit class. This is known as Encapsulation.

**STEPS TO DESIGN ENCAPSULATED PROGRAMS**

1. Declare data members as private.
2. Provide access to data members through public member function.

**PROGRAM WITHOUT ENCAPSULATION**

```
class Calender

{

    private int monthNumber;
```

```
}
class Cal_MAIN
{
	public static void main(String[] args)
	{
		System.out.println("Main Strats");
		Calender c = new Calender();
		c.monthNumber=100;
		// Even though invalid data is assigned, calendar program is accepting the invalid data. To overcome this problem we go for Encapsulation
		System.out.println("Main Ends");
	}
}
```

**PROGRAM WITH ENCAPSULATION**

```
class Calender
{
	private int monthNumber;

	public void giveMonthNumber(int monthNumber)
	{
		if(monthNumber>=1 && monthNumber<=12)
		{
			this.monthNumber=monthNumber;
		}
```

```
        else
        {
                System.err.println("INVALID MONTH NUMBER.....!");
        }
}
public void printMonthName()
{
        switch(monthNumber)
        {
                case 1: System.out.println("January");
                break;
                case 2: System.out.println("February");
                break;
                case 3: System.out.println("March");
                break;
                case 4: System.out.println("April");
                break;
                case 5: System.out.println("May");
                break;
                case 6: System.out.println("June");
                break;
                case 7: System.out.println("July");
                break;
                case 8: System.out.println("August");
                break;
```

```
                case 9: System.out.println("September");
                break;
                case 10: System.out.println("October");
                break;
                case 11: System.out.println("November");
                break;
                case 12: System.out.println("December");
                break;
            }
        }
}
class Cal_MAIN
{
        public static void main(String[] args)
        {
            System.out.println("Main Strats");
            Calender c = new Calender();
            c.giveMonthNumber(7);
            c.printMonthName();
            System.out.println("Main Ends");
        }
}
```

**OUTPUT**

```
Main Starts
July
```

Main Ends

==> We go for encapsulation to protect our variable from mis-usage. **i.e,** from invalid data

**Example**

Conditions

Email-id: varun@gmail.com

Password:

1. At least
    a. alphabet
    b. number
    c. symbol
2. Minimum 8 characters

Confirm Password: should be same as password

```
class GmailAccount
{
    private String pwd;
    public void setPassword(String pwd)
    {
        if(pwd.length()>=8)
        {
            this.pwd=pwd;
            System.out.println("Password Set Successfully");
        }
        else
        {
            System.err.println("INVALID PASSWORD.....!");
        }
```

```
    }
}
class Gmail_MAIN
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        GmailAccount g= new GmailAccount();
        g.setPassword("varun1234");
        System.out.println("Main Ends");
    }
}
```

**OUTPUT**

```
Main Starts
Password Set Successfully
Main Ends
```

**CHALLENGES FACED IN ENCAPSULATION**

==> Figuring out the condition for data member.

==> Providing logic for the condition inside the public method.

---

# OTHERS

---

## WAYS TO IMPROVE READABILITY

1.) Readability means, when we write a program , not only we must understand , even other developers should understand.
2.) There are multiple ways to make program readable.
    => Provide proper class name according to the program.
    => Provide proper variable name according to the data.
    => Provide proper method name according to the operation.
    => Follow PASCAL casing for class name, follow CAMEL casing for variable name, method name and object name.
    =>When result is printed, concat with meaningful message.
3.) Multiple classes should not be present in the same file because as the code length increases, maintenance of the program becomes challenging. Hence, we have to maintain separate file for each and every class.

4.) Blueprint code and main method code should be present in separate classes.

5.) If we are using argument for initializing non-static variable, argument name should be same as non-static variable name. In this case, follow the below syntax for initialization

```
this.Non-static_variable_name=argument_name;
```

---

---

# Common Q/A

**1. ?**

**A:**

**2. ?**

**A:**

**3. ?**

**A:**

**4. ?**

**A:**

**5. ?**

**A:**

**6. ?**

**A:**

**7. ?**

**A:**

**8. ?**

**A:**

**9. ?**

**A:**

**10.?**

**A:**

---

# PROBLEM OF CODE REPITITION

1.) Group of programs put together application.
2.) Every application will undergo updation.
3.) Updating application means
    a. Modifying existing feature
    b. Adding new feature
    c. Removing out-dated feature
4.) During updation of application, we should feel very easy to update. **i.e,** Maintenance of program should be easy. This is achievable only when we follow **DRY(Do-not Repeat Yourself)**.
5.) Dry principle says if the code is repeated multiple times, in future if there are any changes to be made to the code we have to modify the code multiple times. Hence, code should not be repeated, rather code should be reuse to the maximum.

---

**NOTE:**

→ 100% we cannot guarantee elimination of code repetition but we have to avoid code repetition to the maximum.

---

**FOLDER_LIST**

MAIN FOLDER: **java_program**

SUB FOLDERS:

1.) method_program

2.) variable_program

3.) method_with_arguement_return_type_program

4.) method_overloading_programs

5.) execution_process_programs

6.) access_in_another_class_programs

7.) method_access_variable_programs

8.) method_call_method_programs

9.) summary_of_accessing_programs

10.) multiple_ways_to_access_members_within_same_class

11.) constructor_programs

12.) constructor_overloading_programs

13.) constructor_chaining_programs

14.) inheritance_programs

15.) method_overriding_programs

16.) final_programs

17.) abstract_programs

18.) interface_programs

19.) object_typecasting_programs

20.) specialization_generaliztaion_programs

21.) abstraction_programs