Java provides 6 ways to read the data from keyboard.

1. Command Line Arguments
2. java.util.Scanner
3. java.io.BufferedReader
4. java.io.Console.
5. java.io.DataInputStream
6. GUI (awt,swing,applet).
7. System Properties

Command Line Arguments:

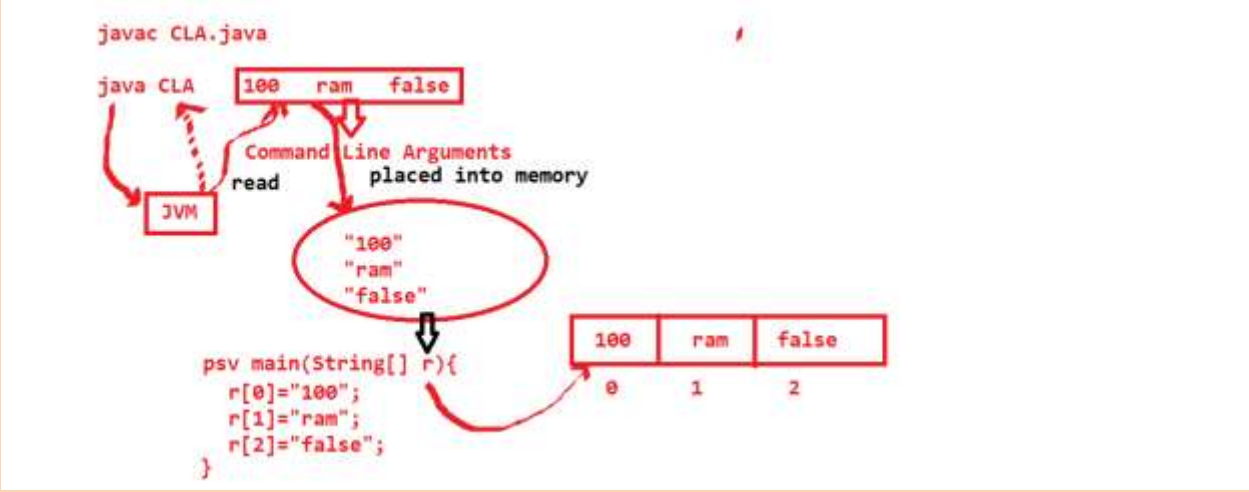
Passing values at mean while executing the program is called command line argument.

Syntax: java ArrayDemo 111 ram (click on enter button)

Here 111, ram are called command line arguments.

Whatever the data, which is sending from keyboard to java application, that data is always in the form of string.

The data may numerical ("123"), character (".net"), special character ("%^&*"), alphanumerical (“ram124”);



```
class CommandLine{
	static public void main(String[] ram){
		String s1 = ram[0];
		String s2 = ram[1];
		System.out.println(s1);
		System.out.println(s2);
		 System.out.println("s1+s2: "+(s1+s2));
	}
}
javac CommandLine.java
java CommandLine  php android
php
android
s1+s2: phpandroid

java CommandLine 123 234
123
234
s1+s2: 123234

java CommandLine "#$%^"   "&*("
#$%^
&*(
s1+s2: #$%^&*(
```

Invalid Execution:

java CommandLine (enter)

java.lang.ArrayIndexOutOfBoundsException: 0

java CommandLine 123 (enter)

java.lang.ArrayIndexOutOfBoundsException: 1

```
class CommandLine{
	static public void main(String[] ram){
		String s1 = ram[0];
		System.out.println(s1);
		String s2 = ram[1];
		System.out.println(s2);
		System.out.println(s1+s2);
		int i = Integer.parseInt(s1);
		System.out.println(i);
		int j = Integer.parseInt(s2);
		System.out.println(j);
		System.out.println(i+j);
	}
}
```

The drawback of above program is , we read the data in the form of string, later we will converting into respective data type(int, boolean, float .....).

So here we are writing some extra convertion logic to read the data in different format.

```
class CLA{
	public static void main(String[] s){
```

```
            String s1= s[0];
                boolean b1 = Boolean.parseBoolean(s1);
            System.out.println(b1);


        }
}
```

If we are sending otherthan "true" value from command line argument, we are always getting like false.But jvm never giving any proper Exception.

We need to remember the number(count) of data.

We need to remember the order(type) of data.

Based on order we need enter the data in keyboard.

To avoiding this problem, we should go for java.util.Scanner class.

```
class CommandLineArgument{
        public static void main(String[] s){
            System.out.println("main method");
            String s1 = s[0];
            boolean b = Boolean.parseBoolean(s1);
            System.out.println(b);
        }
}
```

If we are sending other than true value, we are always getting false only, but JVM never giving exception message like java.lang.NumberFormatException

---

```
public class Account {
       long accno;
       String accHolderName;
       short pin;
       double amount;
```

```
	public static void main(String[] args)
	throws IOException{
		BufferedReader br =
				new BufferedReader(new InputStreamReader(System.in));
		Account[] acc = new Account[2];
	for(int i=0;i<2;i++){
		Account cust1 = new Account();
		acc[i] = cust1;
		System.out.println("enter  accno of long type");
		String s = br.readLine();
		cust1.accno = Long.parseLong(s);
		System.out.println("enter accHolderName of STring type ");
		cust1.accHolderName = br.readLine();
		System.out.println("enter pin number of short type");
		String s1 = br.readLine();
		cust1.pin = Short.parseShort(s1);
		System.out.println("enter amount of double type");
		cust1.amount = Double.parseDouble(br.readLine());

		System.out.println(cust1.accno);
		System.out.println(cust1.accHolderName);
		System.out.println(cust1.pin);
		System.out.println(cust1.amount);
		System.out.println("========================");
	}
		System.out.println(acc[0].accHolderName);
		System.out.println(acc[1].accHolderName);
	}
}
```

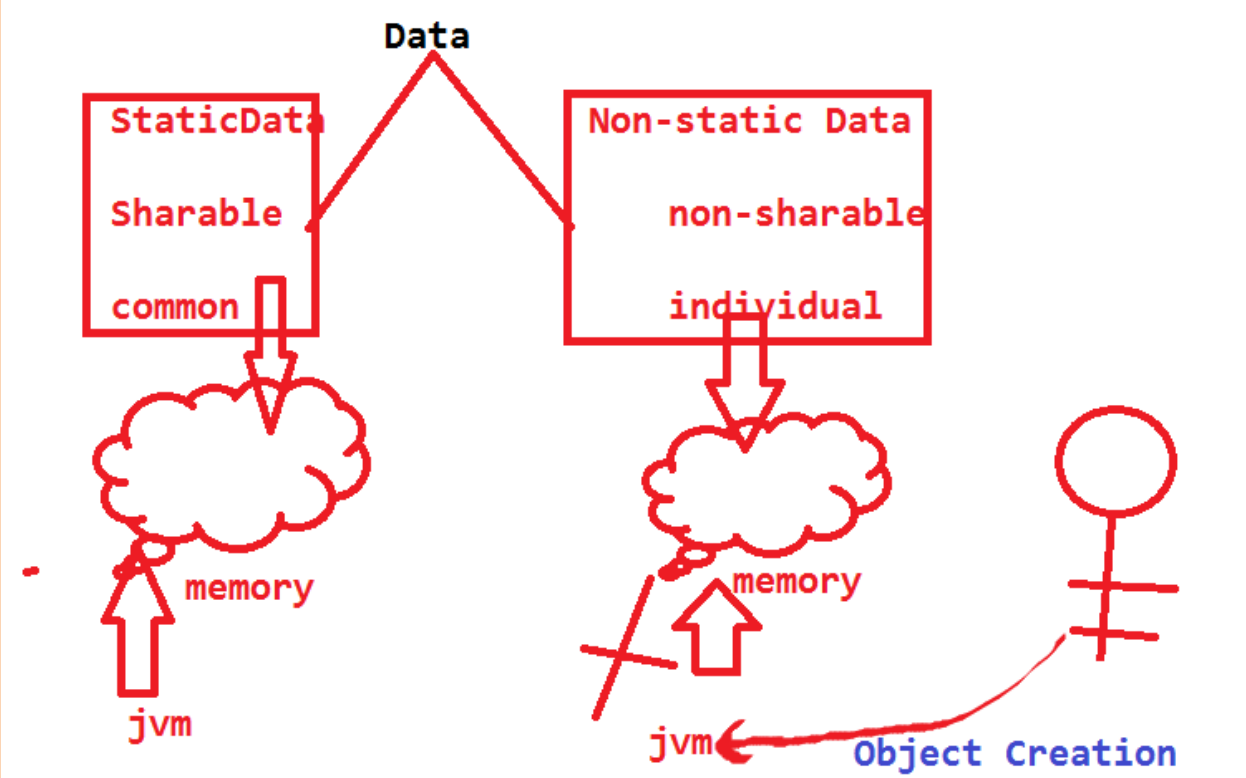
==========

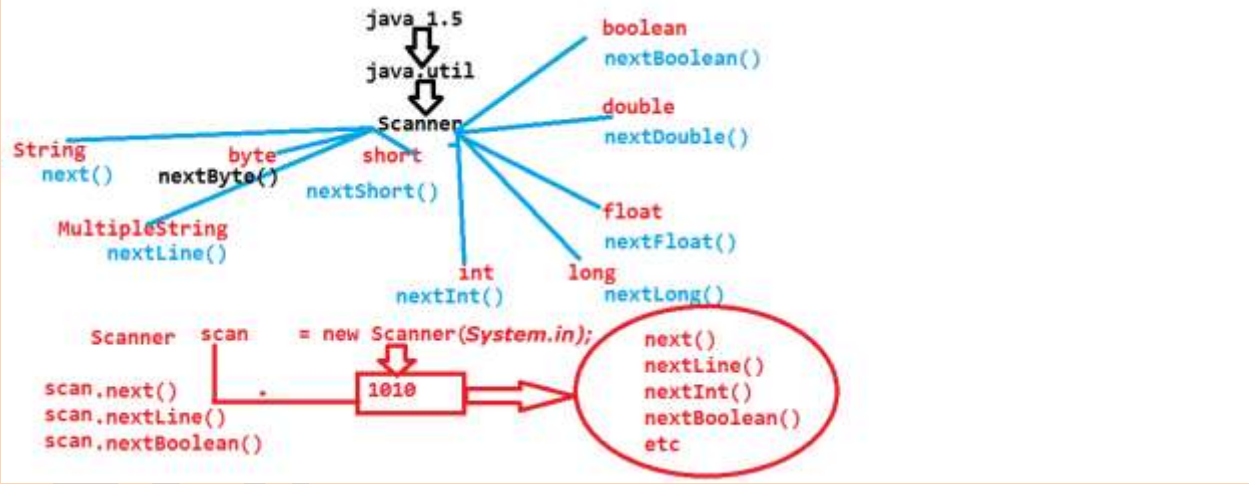
java.util.Scanner:

Scanner is an one predefine class, which is used to read the data in the different data format.

Different data format means either String, int, byte, short, long, float, double, boolean.

With the help of Scanner class we cannot read character data.






```
import java.util.Scanner;

public class ScannerDemo {

	public static void main(String[] args) {

		Scanner scan = new Scanner(System.in);

		System.out.println("enter some string value");

		String s = scan.nextLine();
```

```
            System.out.println(s);
            System.out.println("enter some string value");
            String s1 = scan.next();
            System.out.println(s1);
            System.out.println("enter some integer data");
            int i = scan.nextInt();
            System.out.println(i);
            System.out.println("enter some boolean data");
            boolean b = scan.nextBoolean();
            System.out.println(b);
            System.out.println("enter some float data");
            float f = scan.nextFloat();
            System.out.println(f);
        }
}

import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
            Scanner scan = new Scanner(System.in);
            System.out.println("enter some string value");
            String s1 = scan.next();
            System.out.println(s1);
            scan.nextLine();
```

```
            System.out.println("enter some string value");
            String s = scan.nextLine();
            System.out.println(s);
        }
}
```

Note: In the above program, if we are not

Using obj.nextLine () syntax, then we are unable to read more words by using second nextLine() method. The reason is next () will read only collection of characters after that if we click enter button, that enter button value will be read by nextLine ().

So we are unable type some words.

To overcome the above problem we need to use one extra nextLine() syntax in our program.

```
import java.util.Scanner;
public class ArrayDemo1 {
        public static void main(String[] args) {
            int a[] = new int[5];
            Scanner scan = new Scanner (System.in);
            System.out.println("Reading the valued from keyboard");
            for(int i = 0; i < a.length; i = i+1){
                a[i] = scan.nextInt();
            }
            System.out.println("printing the values of a array");

            for(int i = 0; i< a.length; i++){
                System.out.println(a[i]);
```

```
            }


        }
}
// Copy the elements from one array to another
public class ArrayDemo1 {
    public static void main(String[] args) {
        int a [] = {11,22,33,44,55};
        int b [] = new int[5];
        for(int i =0; i < a.length ; i++){
                //reading the elements from a array
                //b[i] = a[i];
                for(int j =i; j<i+1; j++ ){
                //placing elements into b array
                        b[j] = a[i];
                }
        }
        System.out.println("printing b array elements");
                for(int i =0;i<b.length;i++){
                        System.out.println(b[i]);
                }
        }
}
```

Whenever we read float data and double data by Scanner class don't send float and double followed by either 'f' or 'F' and 'd' or 'D'.

```java
import java.util.Scanner;
public class ScannerDemo {
	public static void main(String[] args) {

		/*long l1 = 100;
		long l2 = 100l;
		long l3 = 100L;

		//float f1 = 123.456;
		float f2 = 100;
		float f3 = 345.45f;
		float f4 = 345.45F;

		double d1 = 400;
		double d2 = 12.23;
		double d3 = 12.23d;
		double d4 = 12.23D;*/

		Scanner scan = new Scanner(System.in);
		System.out.println("enter some float data");
		float f1 = scan.nextFloat();
		System.out.println("f1: "+f1);

		System.out.println("enter some double data");
		double d1 = scan.nextDouble();
		System.out.println("d1: "+d1);
	}
}
```

Java provides one predefine method for copy the elements from one array to another array.

That is arrayCopy(). It is static method, which is available at java.lang.System.

System.arrayCopy(a,0,b,0,5);

a--> source array.

0-->from which index position we are reading the elements from source array.

b--> destination array.

0--> In which index position, we are going to be place the elements in destination array

5-->Number of elements.

Double Dimensional Array:

```
import java.util.Scanner;
public class ArrayDemo1 {
	public static void main(String[] args) {
		int a[][] = {{11,22,33},{44,55,66}};
		for(int i =0;i<2 ; i++){//rows
			for(int j=0;j<3;j++){//columns
				System.out.print(a[i][j]+" ");
			}
			System.out.println();
		}
	}
import java.util.Scanner;
public class ArrayDemo1 {
	public static void main(String[] args) {
		int a[][] = new int[3][3];
		System.out.println("enter some data for 3*3 matrix");
		Scanner scan = new Scanner(System.in);
		for(int i=0;i<3;i++){
			for(int j=0;j<3;j++){
			a[i][j] = scan.nextInt();
			}
		}
```

```
            System.out.println("data from a array");
            for(int i=0;i<3;i++){
                for(int j=0;j<3;j++){
                System.out.print(a[i][j]+" ");
                }
                System.out.println();
            }
        }
    }
}
```

Difference between char array to remaining data type arrays:

If we are printing char array reference variable we will get data. If we are print remaining data type array variables we will reference. (classname@memory)

```
public class ArrayDemo {
    public static void main(String[] ram) {
        char c[]={'1','a','o'};
        System.out.println(c);
        int a[] = {11,22,33,444};
        System.out.println(a);
        boolean b[] = {false,true,true,false};
        System.out.println(b);
    }
}
```

```
public class ArrayDemo {
public static void main(String[] ram) {
        byte b[] = new byte[2];
        System.out.println(b.getClass().getName());
        short s[] = new short[3];
        System.out.println(s.getClass().getName());
        int i[] = new int[2];
        System.out.println(i.getClass().getName());
        float f[] = new float[3];
        System.out.println(f.getClass().getName());
        double d[] = new double[7];
        System.out.println(d.getClass().getName());
        char c[] = new char[1];
        System.out.println(c.getClass().getName());
        System.out.println("****************");
        long l[] = new long[2];
        System.out.println(l.getClass().getName());
        boolean b1[] = new boolean[2];
        System.out.println(b1.getClass().getName());
    }
}
```

Anonymous arrays:

An array, which does not having any external reference, that array is called anonymous array.

Ex: new int []{11,12,13,14,15};

With the help of anonymous array we cannot reuse the memory.

With the help of anonymous array we can interact with the data only one time.

In different times we can send different number of values and different values. After using memories GC will clean them.

```
public class ArrayDemo {
	static void m1(int x[]){
		for(int i=0;i<x.length;i++){
			System.out.println(x[i]);
		}
		System.out.println("**********");
	}
		public static void main(String[] ram) {
			//reference array
			int a[] = new int []{11,22,33,44};
			m1 (a);
			//anonymous array
			m1 (new int []{111,222,333,444});
	}
}
```

Multidimensional Array:

```
import java.util.Scanner;
public class ArrayDemo {
	public static void main(String[] ram) {
		int a[][][] = new int[2][2][3];
```

```
        Scanner scan = new Scanner(System.in);
        System.out.println("please enter integer" +
                " values for 2*2*3 matrix");
        for(int i=0;i<2;i++){
            for(int j=0;j<2;j++){
                for(int k=0;k<3;k++){
                    a[i][j][k]= scan.nextInt();
                }
            }
        }
        System.out.println("elements of a array");
        for(int i=0;i<2;i++){
            for(int j=0;j<2;j++){
                for(int k=0;k<3;k++){
                    System.out.print(a[i][j][k]+" ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Jagged array:

If we want to save the memory while inserting the data into the array, we can go for jagged array.

```
        Scanner scan = new Scanner (System.in);
        int a[][] = new int[4][];
        a[0] = new int[1];
        a[1] = new int[2];
        a[2] = new int[3];
        a[3] = new int[4];
        System.out.println("enter elements for jagged array");
        for(int i=0;i<4;i++){//rows
            for(int j=0;j<i+1;j++){
                a[i][j] = scan.nextInt();
            }
        }
        System.out.println("a array elements");
        for (int i=0;i<4;i++){//rows
            for(int j=0;j<i+1;j++){
                System.out.print(a[i][j]+" ");
            }System.out.println();

        }
import java.util.Scanner;
public class ScannerDemo {
    public static void main(String[] args) {
        int[][][] a= new int[2][4][];
        for(int i=0;i<2;i++){
```

```
        a[i][0] = new int[1];
        a[i][1] = new int[2];
        a[i][2] = new int[3];
        a[i][3] = new int[4];
    }
    Scanner scan = new Scanner(System.in);
    System.out.println("enter some data for jagged array");
    for(int k=0;k<2;k++){
        for(int i=0;i<4;i++){
            for(int j=0;j<=i;j++){
                a[k][i][j]= scan.nextInt();
            }
        }
    }
    System.out.println(" jagged array data:");
    for(int k=0;k<2;k++){
        for(int i=0;i<4;i++){
            for(int j=0;j<=i;j++){
                System.out.print(a[k][i][j]+" ");
            }
            System.out.println();
        }
        System.out.println();
    }
```

```
	}

}
```

```
import java.util.Scanner;

public class Demo {
	public static void main(String[] args){
		Scanner scan = new Scanner(System.in);
		System.out.println("enter row size");
		int row = scan.nextInt();
		int[][] a = new int[row][];
		int []cc = new int[row];
		for(int i=0;i<row;i++){
	System.out.println("how many elements do you need in row-"+(i+1));
			int column = scan.nextInt();
			cc[i] = column;
			a[i] =new int[column];
			System.out.println("enter values");
			for(int j=0;j<column;j++){
				a[i][j] = scan.nextInt();
			}

		}
		for(int i=0;i<row;i++){
			for(int j=0;j<cc[i];j++){
				System.out.print(a[i][j]+" ");
			}
			System.out.println();
		}
		}
	}
```

Array with casting values:

Array can hold different type of data, but the data must be in the range of array type.

```
public class ArrayDemo {

	public static void main(String[] args) {

		int a[] = new int[5];

		a[0]=10;

		a[1]=(byte)120;

		a[2]=(short)250;

		a[3]='a';
```

```
        //a[4]=(float)120.35;//loss of precision
        //a[4]=true; //incompatible
        //a[4]=23.34;
        double b[] = new double[7];
        b[0]=10;
        b[1]=(short)10;
        b[2]=(long)10;
        b[3]=(byte)10;
        b[4]=(float)20;
        b[5]=(int)30;
        b[6]='a';
        //b[7]=true;
    }
}
```

Variable:

Variable is a named memory location, which can be holds the data temporarily.

The data can be change from one statement to another statement.

```
int a = 10;
a = a*20;
a=a-100;
```

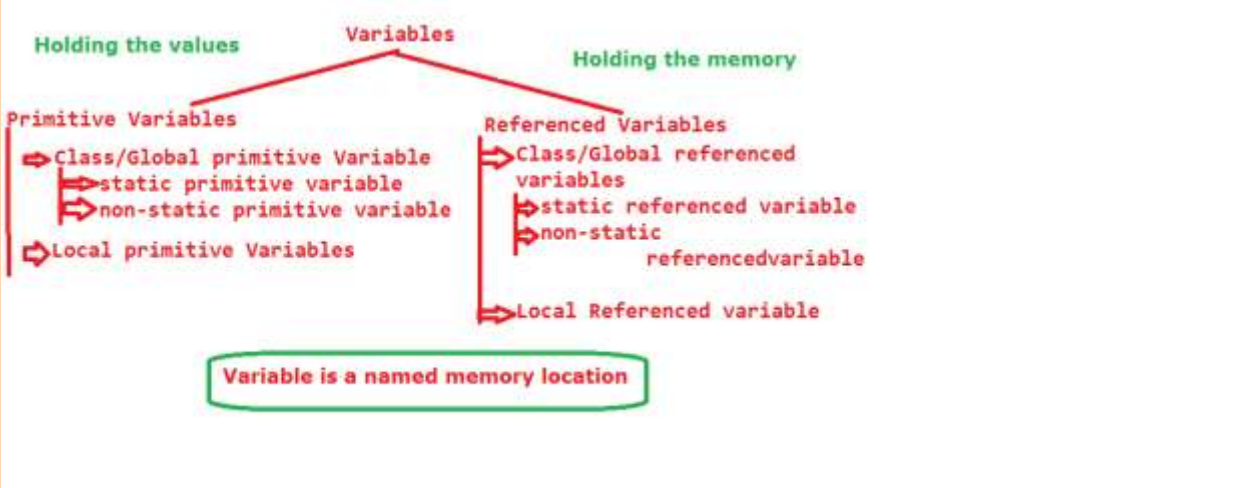
Syntax:

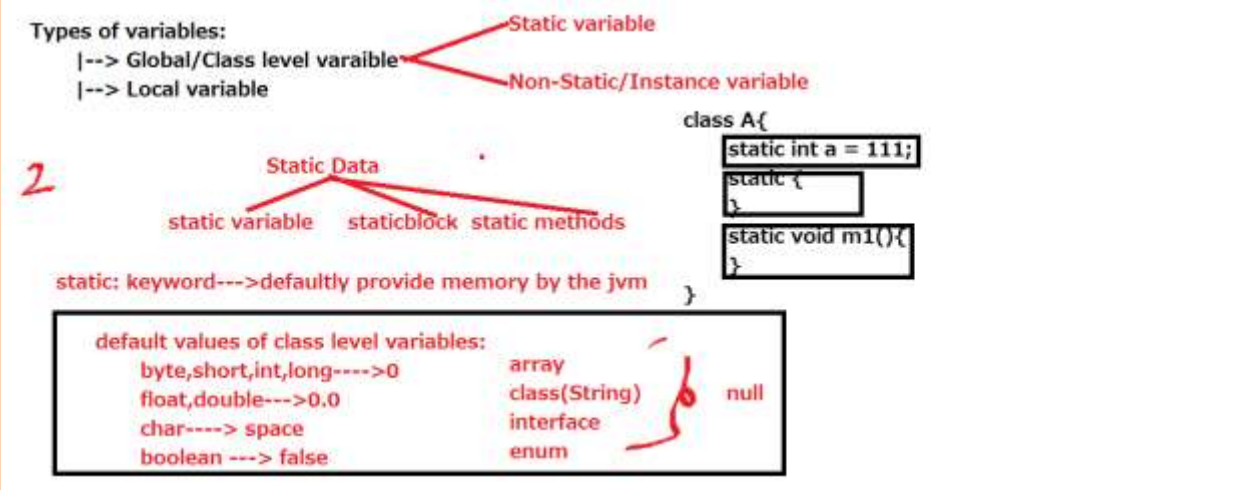
Access-Modifier modifier modifier data-type variable_name = value;

Example: public static final int a = 10;

Variable can be classified as two types

1. Primitive Variables.
2. Referenced variables.





Primitive Variable: A variable, which can be declared with the help of primitive data type is called primitive variable.

int x;

float y;

Referenced variables:

A variable which can be declared with the help of referenced datatype is called referenced variable.

int x[];

Student s;

Runnable r;

ArrayList al;

Primitive Variable:

Variable can be categorized in two types.

1. Class level variable
2. Local variable/method level variable.

Class Level Variable:

A variable, which is located at class scope those variable are called class level variable.

These are two types.

a. static variable

b. non-static/instance variable

a. static variables:

The variable, which have static keyword in its declaration is called static variable.

static int a = 111;

In java data is classified into two types.

Sharable data.

Non-sharable data.

Sharable data can be represents with "static" keyword.

In java we can allocate the memory in two ways

1. static
2. new

Non-sharable data comes under "new" keyword (object).

Before executing main method some static data is going to be executed.

static data means:

1. static variables
2. static blocks
3. static methods.

Before executing main () all static variables and static blocks will be executed, but methods are not executed automatically, if we want to execute we should call explicitly.

All the static variables having same priority, the execution priority is depend upon the way we mention in the program from top to bottom.

Variables execution means placing the data into memory.

All the static variables are memorized into two phases.

1. Loading phase.
2. Execution/initialization phase.

Loading Phase:

In this phase all the variables are memorized with default values. These default values will be placed by one special component in JVM that is "preparer".

Once Loading Phase is completed of all variables, then JVM control will goes to initialization phase.

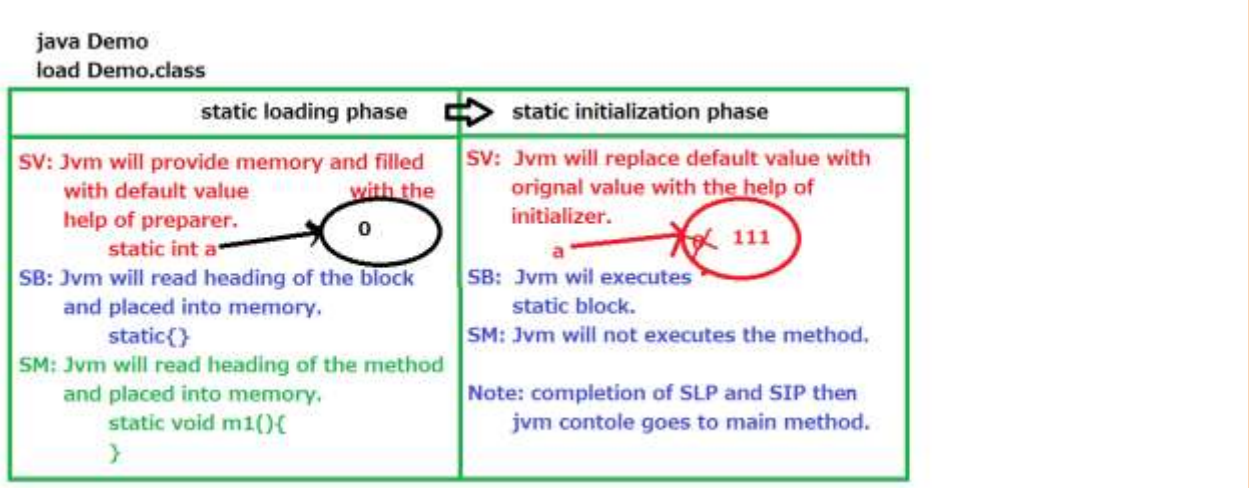
Initialization phase:

In this phase all the default values will be replaces with original values/actual values. These original values will be given by one of the special component of JVM that is "initializer".

static int a = 111;

Initially a value is 0 in the loading phase

Later 0 will be replaced with 111 in the initialization phase.




```
public class StaticVariableDemo {

    static int m2(){

        System.out.println(b);

        return 222;

    }

    static  int a = m1();

    public static void main(String[] args) {

        System.out.println("main method");

        System.out.println(a);

        System.out.println(b);

    }

    static int m1(){
```

```
        System.out.println(a);
        return 111;
    }
    static int b = m2();
}
```

Accessing the static variable:

We have three ways to access the static variables.

1. Class name
2. Directly (by variable name)
3. By Object or reference.

Note: If we want to call any other class static variable we have only two ways

1. Classname
2. Object or reference.

We cannot call other class static variable directly.

```
class A{
    static int b = 222;
}
public class StaticVariableDemo {
    static int a = 111;
    public static void main(String[] args) {
        System.out.println("main method");
        int i = StaticVariableDemo.a;
        System.out.println(i);
        System.out.println(StaticVariableDemo.a);
```

```
        int j = A.b;
        System.out.println(j);
        System.out.println(A.b);
        System.out.println("*****************");
        int k = a;
        System.out.println(k);
        System.out.println(a);
        //System.out.println(b);
        System.out.println("***************");
        StaticVariableDemo sd = new StaticVariableDemo();
        System.out.println(sd.a);
        System.out.println(new StaticVariableDemo().a);
    }
}
```

Note:

static data is sharable data between Objects.

static data having only one memory location.

That memory location can be sharable by all the objects in the application.

If anyone object is updating the static data that updated data will be effected to other objects also.

Static block:

Block means group of statements.

A block which contains only static keyword, that block is called static block.

Syntax: static{

}

It is used for executing the common logic for all objects before main method. That type of code, we should be writes in the static block.

To initialize the static variables.

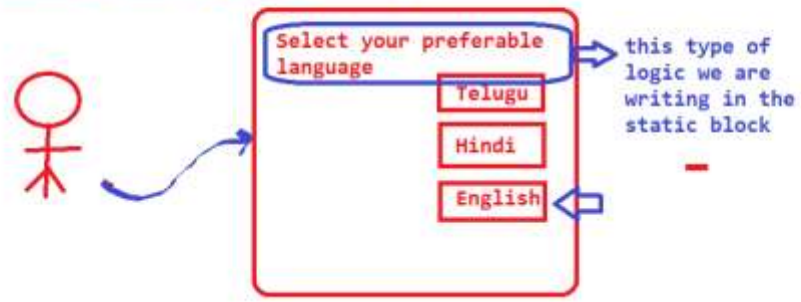
All the static blocks having same priority.

The execution of static blocks will be depends upon the way we mention in the program from top to bottom.

We cannot write one static block within the another static block and also another method, constructor, interface, annotation.

Static blocks holds logic which is used for doing or excuting actual logic.

static:
---> use to hold logic
---> logic is common for all
---> executes exactly one time.



static blocks:
snytax: static{
//logic
}

one static block we can not write in another static block and method.
static block we can write in class, abstract class, enum.

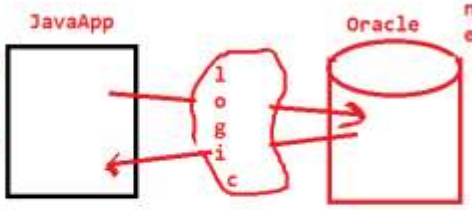
---> hold the logic.
---> common for all.

---> executes the logic exactly one time.

---> It is used to initialize the static variables

---> Registering the drivers.

when ever we loading the driver class (OracleDriver) we need to create object, so that object creation syntax we can write with int he static block is called registering driver.



we can not communicate non-static data in static block directly we need reference or object.

ojdbc6.jar

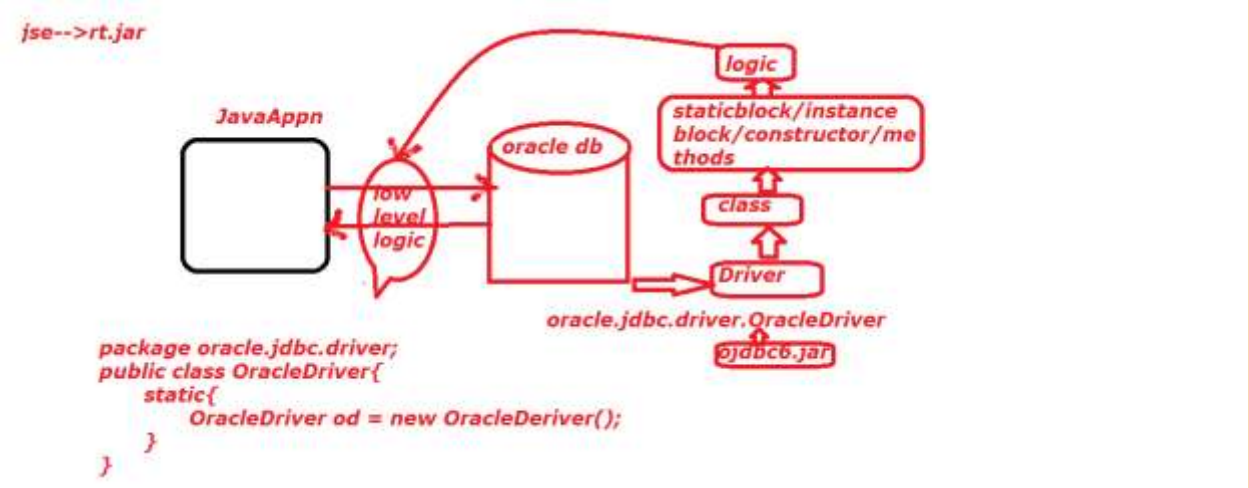
Driver

```
class  oracle.jdbc.driver.OracleDriver {
       static{
          OracleDriver od = new OracleDriver();
       }
}
```

Downloading ojdbc6/ojdbc14.jar file
oracle
jdbc
driver

OracleDriver

--> we need to load the OracleDriver class
--> we need to create Object for OracleDriver class
--> Oracle Driver class register with DriverManager
--> by using drivermanager we can get Connection between
javaapp to oracle

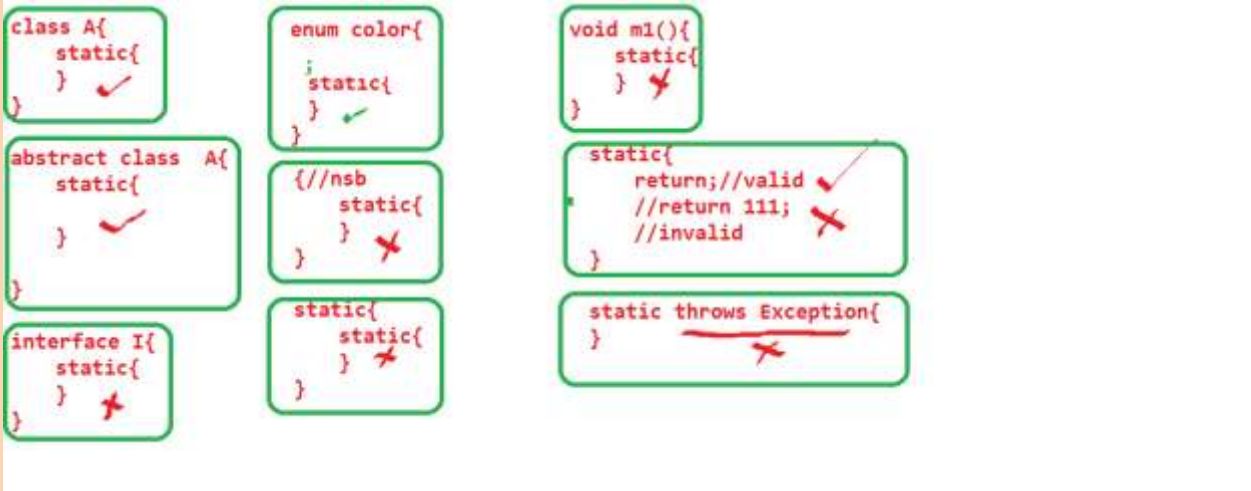



```
import java.util.Scanner;
public class Atm {
	static Scanner scan = new Scanner(System.in);
	static String language;
	static byte twoDigitNumber;
	static{
		System.out.println("wel come to sbi services");
	}
	public static void main(String[] args) {
			System.out.println("main method");
	}
	static{
		System.out.println("enter your language");
		language = scan.nextLine();
		System.out.println("you are choosing language like: "
					+language.toUpperCase());
	}
	static{
		System.out.println("enter two digit number for security");
		twoDigitNumber = scan.nextByte();
```

```
            System.out.println("you are enter number like: "+
                                    twoDigitNumber);
        }
}
        class A{
            static{}//valid
            static{
                static{}//invalid
            }
            void m1(){
                static{}//invalid
            }
        }
        interface I{
            static{}//invalid
        }

        @interface AAA{
            static{}//invalid
        }
```

Each and every static variables and static blocks having same priority. The priority is depends upon the way we mention in the program from top to bottom.

If both static variables and blocks are available in single class, then all static variables loading phase will be first completed. After that both static variables and static blocks will be executed (initialization phase) in the order wise. (From top to bottom).



Loading phase:

In this phase the static data is variable, then JVM provides memory, in that JVM will place default values.

If static data is block and method, then those heading will be read by the JVM and place into some memory.

Initialization phase:

In the phase, the static data is variable, then default values will be replaced with original values. If static data is block then automatically block will be executed. If static data is method then not executed, the reason is no method will be executed automatically except main.

After successfully completion of static variable, static blocks execution, then JVM will give chance to main method to execute.

static data can be sharable between all the objects. If anyone object is doing modification, that updated value effected to remaining objects also.

Note: All the static data are stored in Method Area.

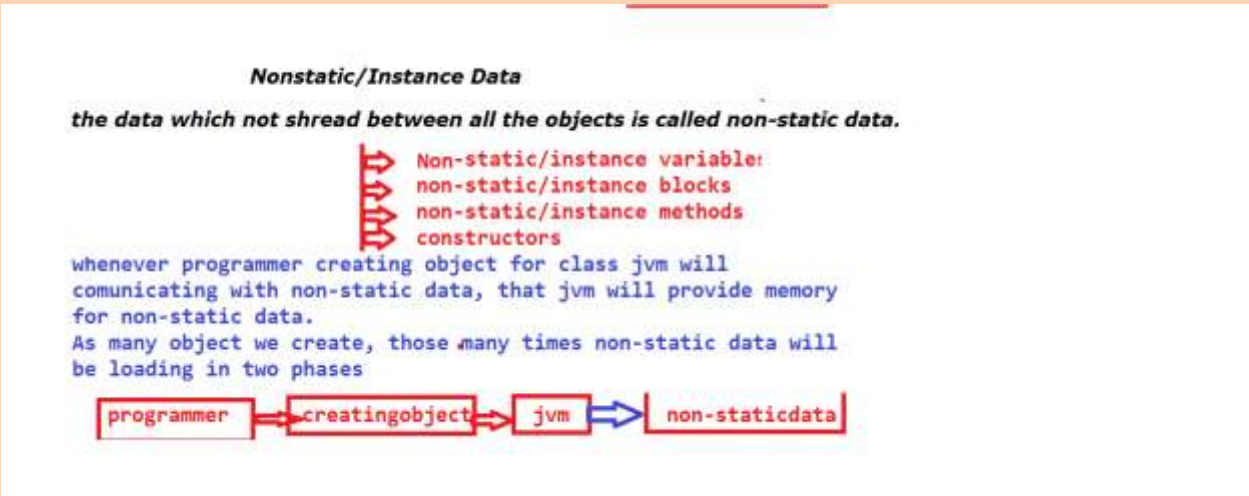
Non-static data:

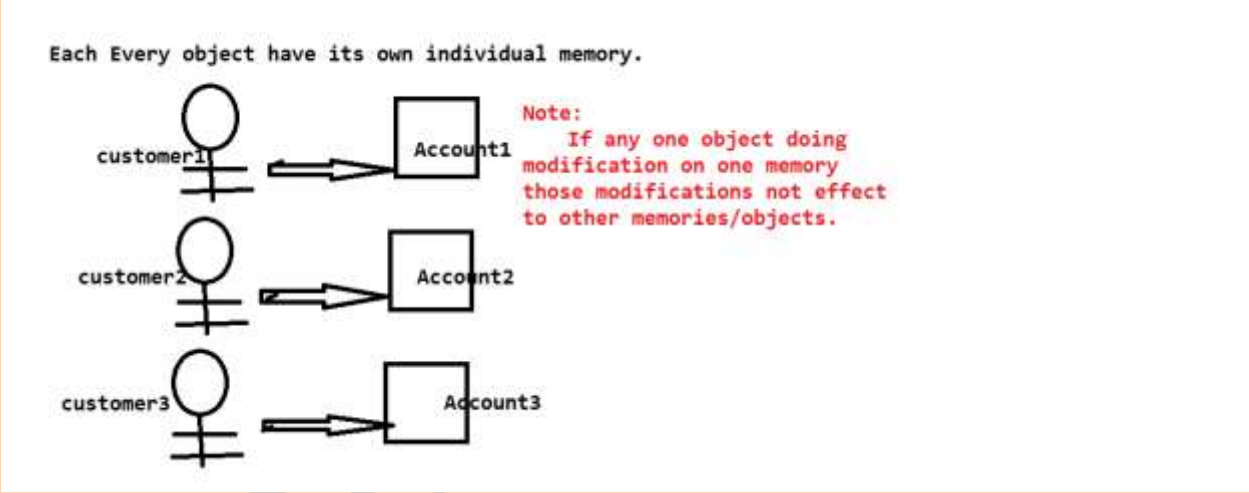
Non-static data comes under

non-static variable

non-static blocks

non-static methods.





Non-static variable:

A variable which doesn't have any static keyword in its declaration is called non-static variable.

Ex: int a = 111;

Non-static is not sharable between any two objects.

If one object is doing modification, that updated value not effected to remaining objects.

All the non-static data and objects are stored in the heap area.

Whenever we create an object JVM will do the following things.

static data is loading or memorized only one time for entire application.

As many objects we create in our program, those many times non-static data memorized /loaded.

Loading phase:

If data is non-static variable, then JVM blindly provides memory and place the default values.

If data is non-static block, JVM will read only block heading.

If data is non-static method, JVM will read only method heading.

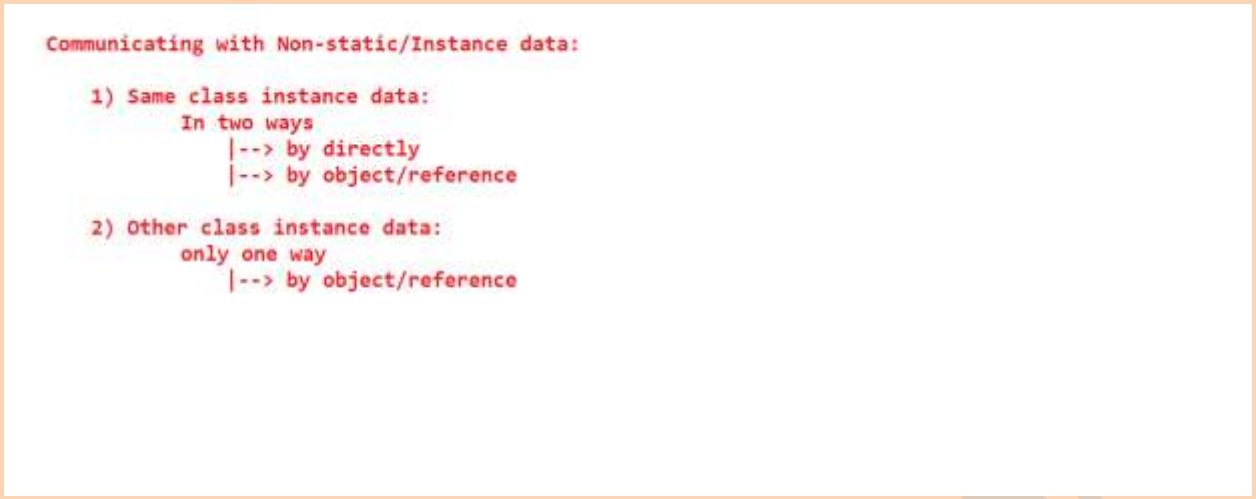
Initialization phase: If data is non-static variable, then jvm replace the default values with original/actual values.

If data is non-static block then block will be executed by jvm.

If data is non-static method then not executed.

All the non-static variables are having same priority, the execution of variables is depends upon, the way we mention in the program from top to bottom.

nonstatic/instance loading phase	nonstatic/instance initialization phase
NSV: Jvm will give memory and filled with default value with the help of constructor. NSB: Jvm will read the heading and placed into memory. NSM: Jvm will read the heading and placed into memory. Constructor: Jvm will read the heading and placed into memory	NSV: Jvm will be replaced default values with original/actual vlaues *with the help of constructor.* NSB: execute NSM: Not Execute Constructor: Not Execute Note: after succesfull completion of nonstatic loading phase and nonstatic initilization phase controle goes to constructor. *Once constructor is executed we can say object is sucessfully initialized.*




```
public class Nonstatic {
	int a = m1();
	int m2(){
		System.out.println(b);
		System.out.println("m2 method");
		return 222;
	}
	public static void main(String[] args) {
		System.out.println("main method");
		Nonstatic  ns = new Nonstatic();
	}
	int m1(){
		System.out.println(a);
		System.out.println("m1 method");
		return 111;
	}
	int b = m2();
```

}

Note: static int a = m1();

"static int a" is comes under declaration part.

The space between equal operator and semicolon is called initialization part.

If any method calling syntax is available in initialization part, that must be carry the information from called area to calling area.

A method which is carries the information is called non-void method.

Information must be same or compatible with variable data type.

If method having return-type, that method must be have return statement with value.

Again value must be compatible with return type.

Accessing the non-static variable:

We have two ways to access the non-static data

1. By directly.
2. By object or reference.

//non-static variable we cannot call from static area directly. We should always by using reference or object.

//if we want to call any other class non-static data, we have only one way. That is object or reference.

```
class B{
    int b = 222;
}
public class Nonstatic {
    int a = 111;
    void m1(){
        System.out.println("m1 method");
```

```
            System.out.println(a);
        }
        public static void main(String[] args) {
            System.out.println("main method");
            Nonstatic  ns = new Nonstatic();
            System.out.println(ns.a);
            System.out.println(new Nonstatic().a);
            ns.m1();
            //System.out.println(a);
            //System.out.println(b);
            B obj = new B();
            System.out.println(obj.b);
            System.out.println(new B().b);
        }
}
```

Non-static data is non sharable between objects. If anyone object is doing modification those modifications are not affected to other objects the reason every object have its own memory. If we are doing modification on one memory those are not affected to another memory.

```
public class C {
        int a = 111;
        int b = 222;
        public static void main(String[] args) {
            C obj1 = new C();
            System.out.println(obj1.a);
```

```
            System.out.println(obj1.b);
            System.out.println("---------");
            C obj2 = new C();
            System.out.println(obj2.a);
            System.out.println(obj2.b);
            System.out.println("---------");
            obj1.a = 888;
            System.out.println(obj1.a);
            System.out.println(obj1.b);
            System.out.println("---------");
            obj2.a = 999;
            System.out.println(obj2.a);
            System.out.println(obj2.b);
        }
}
import java.util.Scanner;
class Account{
        long accNo;
        String accHolName;
        double amount;
}
class AccountDemo{
        public static void main(String[] s){
            Scanner scan = new Scanner(System.in);
```

```
        System.out.println("enter number of customers");
        int noCustomers = scan.nextInt();
        Account customer[] = new Account[noCustomers];
        for(int i=0;i<noCustomers;i++){
            customer[i] = new Account();
         System.out.println("enter "+(i+1)+" customer details");
            System.out.println("enter accNo");
            customer[i].accNo = scan.nextLong();
                System.out.println("enter accholdername");
            customer[i].accHolName=scan.next();
            System.out.println("enter amount");
            customer[i].amount= scan.nextDouble();
        }
        for(int i=0;i<noCustomers;i++){
System.out.println((i+1)+" customer details");
        System.out.println(customer[i].accNo);
        System.out.println(customer[i].accHolName);
        System.out.println(customer[i].amount);

        }

        customer[0].amount=15000;
System.out.println(customer[0].amount);
System.out.println(customer[1].amount);
```

```
    }
}
```

Non-static Blocks:

A block which doesn't contains static keyword in its declaration is called non-static block.

All the non-static blocks having same priority.

The execution priority is depends upon, the way we mention in the program from top to bottom.

```
public class C {
    {
        System.out.println("non-static block 1");
    }
    {
        System.out.println("non-static block 3");
    }
    public static void main(String[] args){
        System.out.println("main method");
            C obj = new C();
    }
    {
        System.out.println("non-static block 2");
    }
}
```

```
public class C {
```

```
static {
	System.out.println("static block 1");
}
int a = m1();
{
	System.out.println("non-static block 1");
}
static int c = m3();
int m1(){
	System.out.println(a);
	System.out.println("m1 method");
	return 2222;
}
static int m3(){
	System.out.println(c);
	System.out.println("m3 method");
	return 4444;
}
{
	System.out.println("non-static block 3");
}
int b = m2();
public static void main(String[] args) {
	System.out.println("main method");
```

```
        C obj = new C();
        System.out.println(obj.a);
        System.out.println(obj.b);
    }
    {
        System.out.println("non-static block 2");
    }
    int m2(){
        System.out.println(b);
        System.out.println("m2 method");
        return 3333;
    }
}
```

Valid statement:

```
    {
        {
        }
    }
    static{
        {
        }
    }
    C (){
        {
```

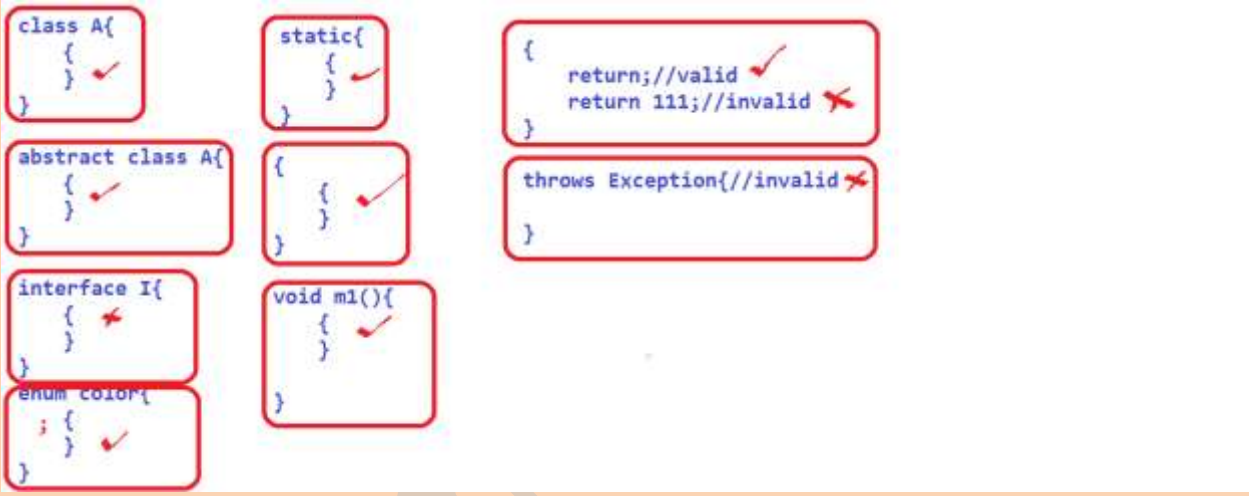
```
            }
        }

interface D{
        //{}
}
enum E{
        ;
        {

        }
}
abstract class F{
        {

        }
}
@interface I{
        //{}
}
public class C {
        {
                System.out.println("non-static block 1");
                {
```

```
            System.out.println("non-static inner block 1 ");
        }
}
static{
        System.out.println("static block 1");
        {
            System.out.println("non-static inner block2");
        }
}
C(){
        System.out.println("constructor");
        {
            System.out.println("non-static inner block 3");
        }
}
static void m2(){
        {
            System.out.println("m2 method non-static block");
        }
}
public static void main(String[] args) {
        {
            System.out.println("main method non-static block");
        }
```

```
        System.out.println("mainmethod");
        C obj = new C();
    }
}
```

Non-static block we can write with in the class, enum, abstract class, static block, static method, non-static method, constructor but we cannot write in interface and annotation.

static block we can write only within the class, enum, abstract class.



Q) Difference between static and new keyword?

static:

1) It is a keyword, is used allocate the memory for static data.

2) static data means, static variable, blocks, methods.

3) static data is sharable data

4) static data having only one memory

5) Only one time static data is loaded.

6) If any one object is doing updating on static data, that updated value will be effected to remaining objects.

7) Meanwhile of class loading static-data is loaded by the JVM.

8) Static data can be access with in 3 ways.

Directly, Class name, Object or reference.

9) static data we can call anywhere in the program.

10) other class static can be access only in two ways.

a.by classname

b.by object or reference

new:

1) It is keyword is used to allocate the memory for non-static data.

2) Non-static data means, non-static variables, non-static blocks, non-static methods.

We have one more name to non-static data that is instance data.

3) Non-static data is non-sharable data.

4) Non-static data having multiple memories.

5) As many object we created those many times non-static data is going to be loaded.

6) If any one object is doing updating on non-static data that updated data/value will not be effects to remaining objects.

7) Meanwhile of object creation non-static data executed.

8) Non-static can be access in two ways.

By using directly, object or reference.

9) Non-static we cannot call directly in static area.

10) Other class non-static can be access only in one way.

a. By object or reference

Nsb sb constructor

throw	no	no	yes
return	no	no	yes
return value	no	no	no

Local variable:

A variable, which is resides or locate within the parameter of a method, constructor and body of method, constructor, block (static, non-static) is called local variable.

Ex:

```
 void m1(int x){
        int y;
 }
{
        int z;
}
static{
        int l;
}
class  A{
        A(int n){
                int m;
        }
}
```

x, y, z, l, m, n all are local variables.

Local variables always either default or final.

```
void m1(){
        int a;          //valid
        final int b;    //valid
        private int x;      //invalid
        public int y;//invalid
        protected int z;    //invalid
        static int l;  //invalid
        transient int m;    //invalid
        volatile int n;     //invalid
}
```

Local variables may be default or final.

There are no default values for local variables.

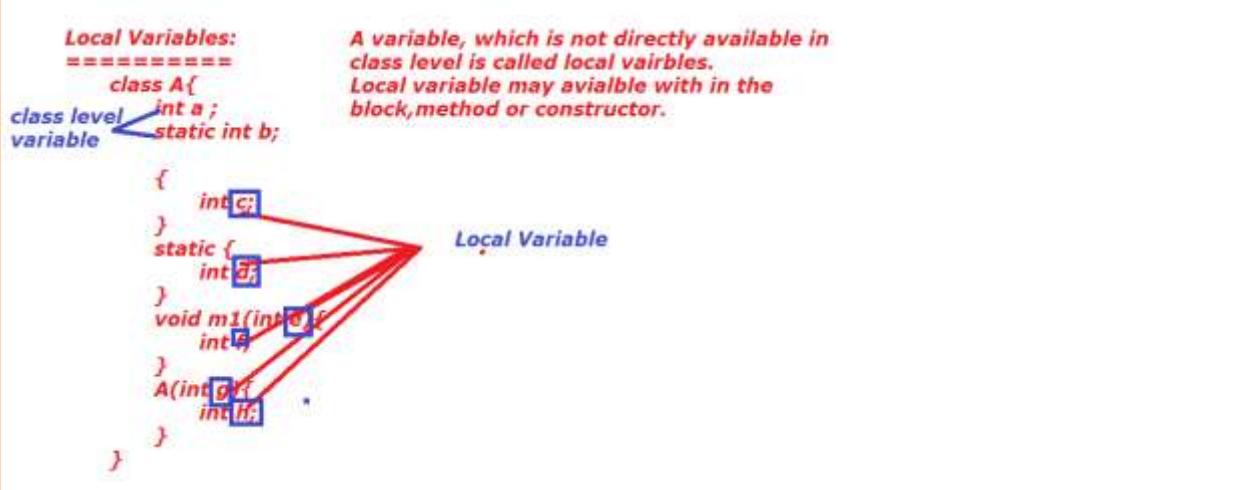
Before using local variables we should be initialized.

if we are using local variables without initialization, we will get one compile time error.

Same variable name we can declare in different scopes.

final keyword we can be applied on variables, method, classes.

Meanwhile of block execution local variables are going to be memorized and after block execution completed all the local variables are destroyed.



final variables:

A variable which have final keyword in its declaration is called final variable.

final keyword we can be applied on top of class level and local variables also.

class level final variable must be initialized meanwhile of declaration.

local level final variable must be initialized before using that variable.

final variable can be use more than one time but we cannot update/change the data.

If we are trying to change we will get CE.

```
final class Student{
    String sname="suji";
}
public class FinalVariable {
    final int a = m1();
    //final int b; //class level final variable must be
    //intialize mean while of declaration.
    int m1(){
```

```
        System.out.println(a);
        System.out.println("m1 method");
        return 111;
}
final void m2(){
        System.out.println("m2 method");
}
static void m3(final int x){
        System.out.println(x);
        //x=x+1;
}
public static void main(String[] args) {
        FinalVariable fv = new FinalVariable();
        System.out.println(fv.a);
        //fv.a = fv.a*2;
        int c = fv.a*2;
        final int d;
        d=888;
        System.out.println(d);
        Student s = new Student();
        System.out.println(s.sname);
        s.sname="ram";
        System.out.println(s.sname);
        fv.m2();
```

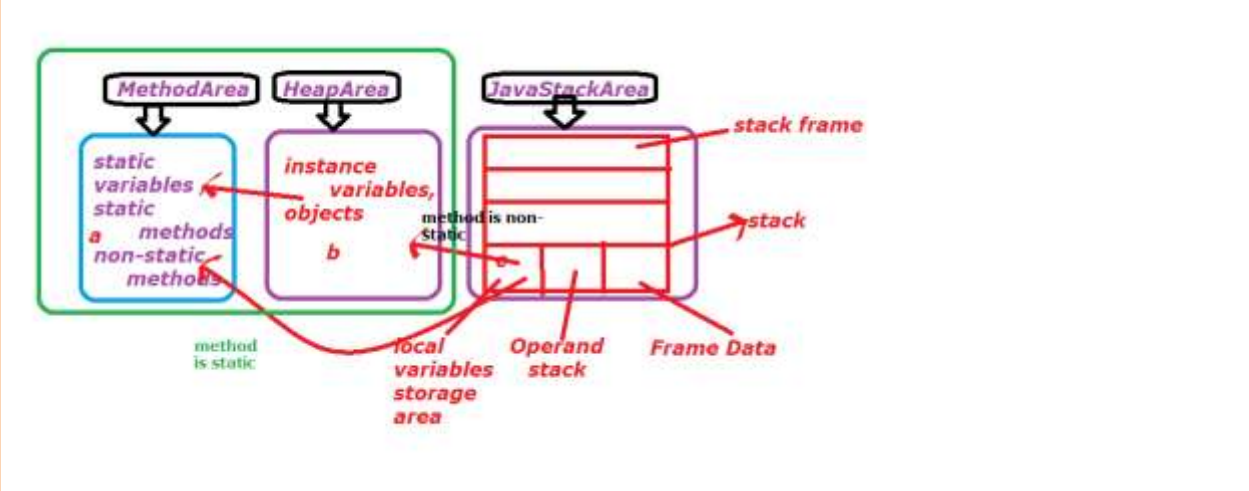
```
    }
}
```

Scope of variables:

Local variables: only within the block(method, constructor, static and non-static block).

Non-static variable: Based on object. once object created automatically non-static variables loaded, once object is destroy non-static variables are destroyed.

static variable: Based on byte-code (.class).



When ever we call any variable directly first prefarence gives to local variable(Javastackarea).

--> If local variables is not existed then controle goes to either HeapArea or MethodArea.

--> if method is static then controle goes to methodarea

--> if method is non-static then controle goes to heaparea.

(if not available in heap area then controle to method area)

==========

If we call any variable by reference or object first prefarence gives to HeapArea if data is not available then controle goes to MethodArea.

==========

If we call any variable by using class name then controle goes to MethodArea.

private, protected,public,static variables are unable to write in block level (static,non-static,constructor,method), the reason is, the main intension of

private variable intension is accessible with in the class, if we write in the block level those are not accessible with in the entire class.

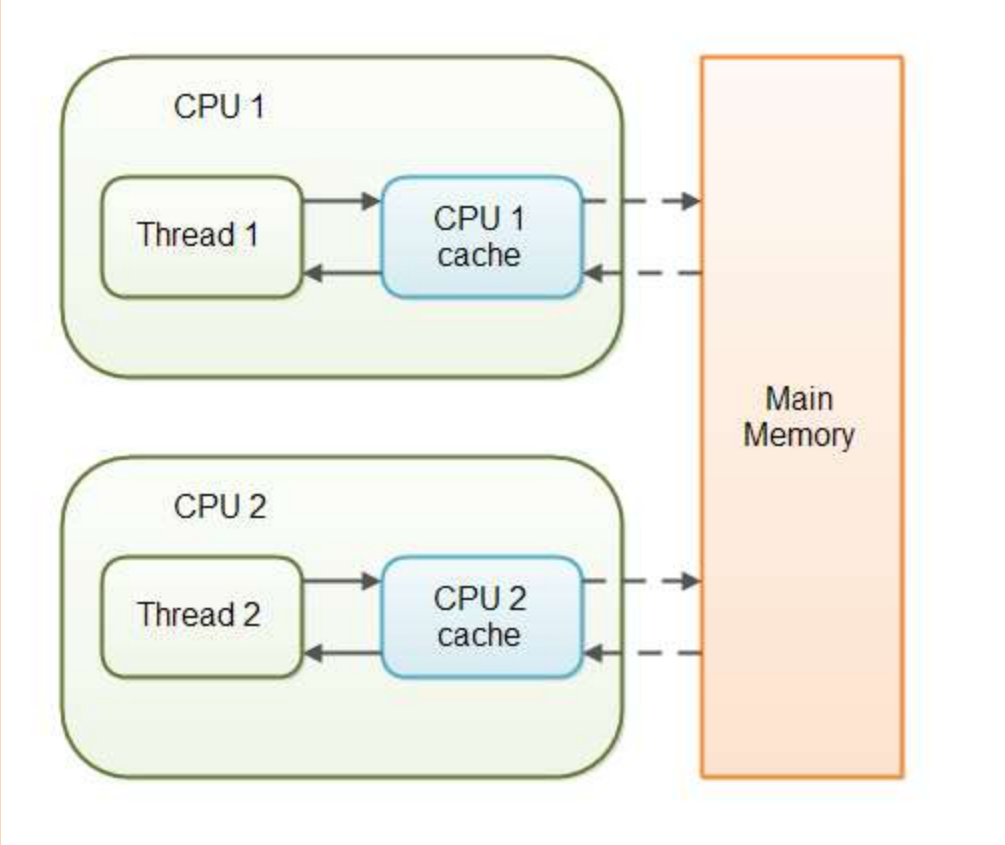
protected variable intension is accessible with in the class, and its inherited class, if we write at block level, we are unable to accessible both with in the class and its inherited classes.

public variable intension is accessible with in the any where in the project, if we declare public variable at block level, those are not available with in the project.

static variables are common for all objects(threads), if we declared at block level all threads are not accesible.

The Java volatile keyword guarantees visibility of changes to variables across threads. This may sound a bit abstract, so let me elaborate.

In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs. This is illustrated here:



http://tutorials.jenkov.com/java-concurrency/volatile.html

transient variables intension is not partispated at serialization, only class level data is partispated in serialization, so writing transient at block level there is no meaning.

Where shouldn't use volatile keyword?

If the variables is not shared between two threads.

When should use volatile keyword?

If the variable is shared between two threads frequently, then we should use volatile keyword.

6/2/19

* Same class, non-static / instance data:

→ by directly (from non-static area / context)
(non static of non-static method of constructor).

→ by object / reference.

Other class non-static / instance data:

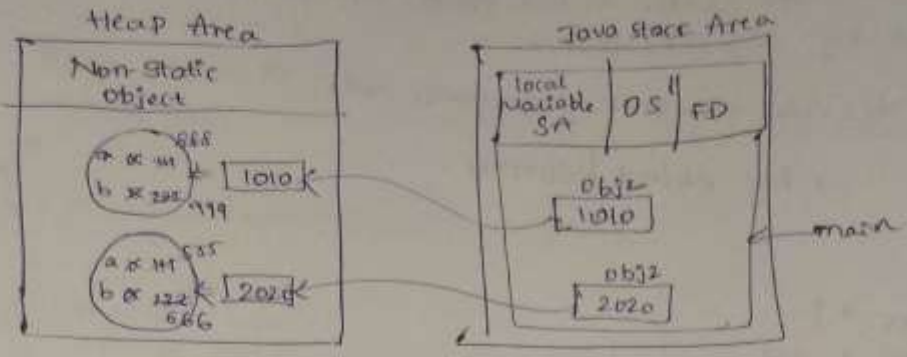
→ by object reference.

Prgm:

```
class A {
  int b=222;
}
public class Variable {
    int a=111;
    {
    S.O.P ("from nsb:"+a);
    }
    void m1() {
      S.O.P ("from non-static method:"+a);
    }
    Variable() {
      S.O.P ("from constructor:" +a);
    }
    static {
      // S.O.P ("from sb:"+a);
      S.O.P ("from sb:" +new Variable().a);
    }
    public static void main (String[] args) {
      //S.O.P ("by directly:"+a);
      Variable obj=new Variable();
      S.O.P ("by reference in main:"+obj.a);
      obj.m1();
      A obj1=new A();
      S.O.P ("A class b variable:" + obj1.b);
    }
}
```

```
from nsb: 111
from constructor:111
from sb: 111
from nsb:111
from constructor:111
by reference in main:111
from non-static method:111
A class b variable:222
```

* If any one object is doing modification on non-static data these modifications are not effected to other objects.



Program:

```
public class Variable {
  int a=111;
  int b=222;
public static void main (String[] args) {
    Variable obj1 = new Variable();
    Variable obj2 = new Variable();
    S.O.P("obj1:" +obj1.a+"..."+obj1.b);
    S.O.P ("obj2:" +obj2.a+"..."+obj2.b);
    obj1.a=888;
    obj1.b=999;
    S.O.P ("=======");
    S.O.P ("obj1:" +obj1.a+"..."+obj1.b);
    S.O.P ("obj2:" + obj2.a+"..."+obj2.b);
    obj2.a=555;
    obj2.b=666;
    S.O.P ("=======");
    S.O.P ("obj1:"+obj1.a+"..."+obj1.b);
    S.O.P ("obj2:"+obj2.a+"..."+obj2.b);
  }
}
```

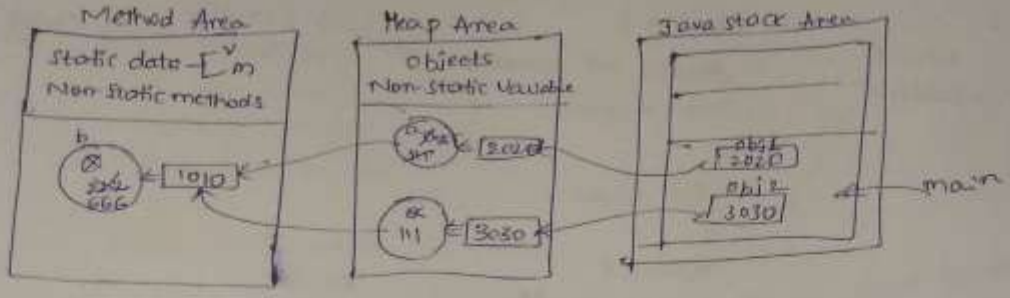
O/P

```
111....222
111....222
888....999
111....222
888....999
555....666
```

O/P

111 ... 222
111 ... 222
555 ... 666
111 ... 666

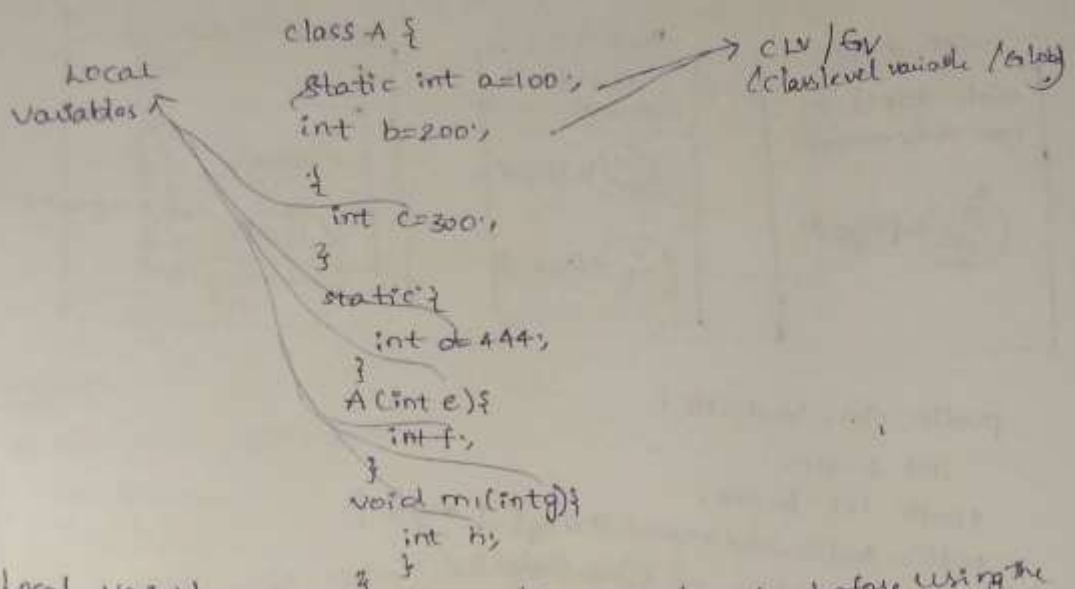
Pgm




```
public class Variable I
  int a=111;
 static int b=222;
 public static void main(String[] args) {
 Variable obj1 = new Variable();
 Variable obj2 = new Variable();
  S.O.P("obj1 :"+obj1.a+"..."+obj1.b);
  S.O.P("obj2:"+obj2.a+"..."+obj2b);

  Obj1.a=555;
  obj1.b=666;
   S.O.P("= = = = =");
 S.O.P("obj1 :"+obj1.a+"..."+obj1.b);
 S.O.P("obj2:"+obj2.a+"..."+obj2.b);
 }
}
```

Local variable (Local primitive variable):

A variable which is not a direct part of the class is called local variable.



* Local variables doesn't have default values so, before using the (CLV. have) local variables we need to initialize.

pgm:

```
public class variable {
    int a;
  static int b;
   public int e;
   private int f;
   protected int g;
  transient int h;
   final int i=555;
 void m1(){
     S.o.p("non-static m1 method");
     int d=333;
     S.o.p ("d: "+d);
 }
public static void main (String [ ] args) {
    S.o.p("static main method");
    variable obj1= new variable();
```

```
S.O.P ("nsv :"+objl.a);
S.O.P ("sv, "+variable.b);
int c;
//S.O.P ("local variable:"+c);
objl.ml();
/* public int el;
   private int ft;
   protected int gl;
   transient int hl;
   */
final int il=666;
}
}.
```

*Note The scope of the local variable is within that particula block, after initialized (declaration) one block local variable we can't be used in other blocks directly.

Prgm:

```
public class Variable {
    int m1(){
     S.O.P ("m1 method");
    /* S.O.P("c:"+c);
    S.O.P ("d:"+d);*/
     int c=333;
     int d=444;
     S.O.P ("c:"+c);
     S.O.P ("d:"+d);
     return c;
    }
    public static void main (String[] args) {
        S.O.P ("static main method);
        int a=111;
         int b=222;
        Variable V= new Variable();
        V.m1();
```

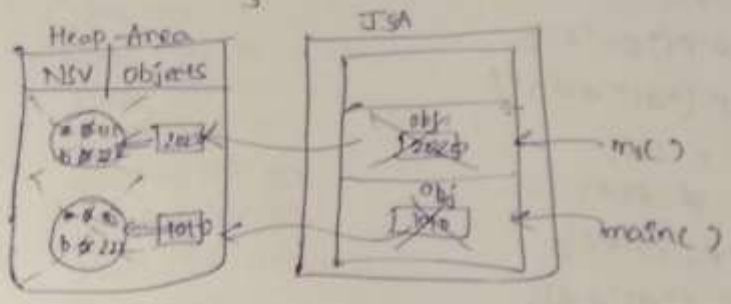
```
/* S.o.p ("c:"+c);
   S.o.p ("d:"+d); */
}
```

7/2/12

Scope of non-static variable:

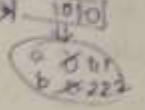
Whenever object is created, all the non-static variables getting memory of created. once. object is destroyed all the non-static variables memory destroyed that means the scope of the non-static variable is lifetime of object.

```
class A {
 int a=111; int b=222;
 void m1(){
  A obj= new A();
 }
 PSVm(--) {
  A obj1 = new A();
   obj1.m1();
 }
}
```

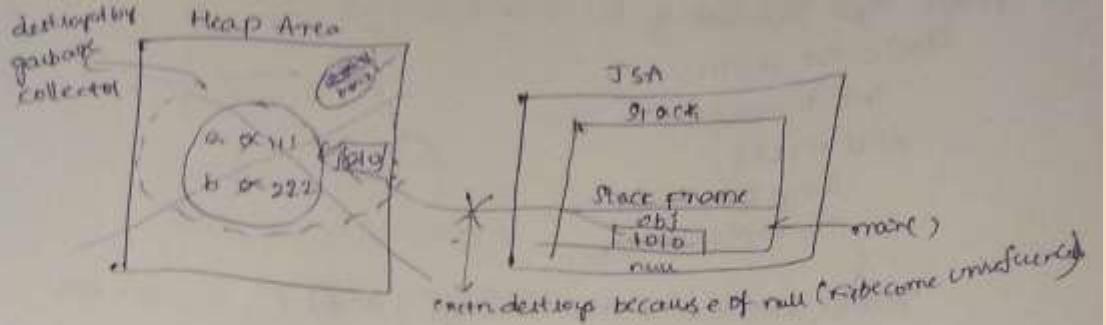


*

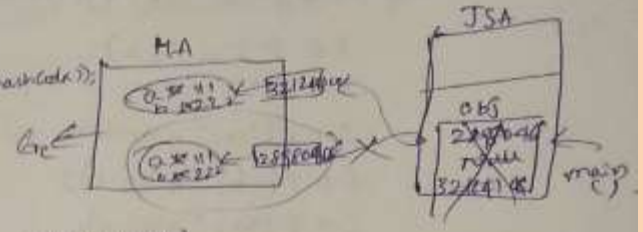
```
class A {
 int a=111;
 int b=222;
  PSVm(--) {
   1. A obj = new A();
111 <- 2. Sop(obj.a);
222 <- 3. Sop(obj.b);
   4. Obj =null; // programmer destroying in understanding
                    only mem destroy not object.
```



1000 sop(obj.a); } If we execute again by writing these statements then
sop(obj.b) } we will get an error like
Null pointer Exception.



* class A{
int a=111;
int b=222;
PSVM(-){
1. A obj=new A();
2. sop(obj.a); S.o.p(obj.hashcode));
3. sop(obj.b);
4. obj=null;
:
500. obj=new A(); //sop(obj.hashcode())
1000. sop(obj.a)
sop(obj.b)



obj.hashcode() → To know the address.

Note

* Whenever we call any variable directly first preference gives to local variable (JavaStackArea).
→ If local variable is not existed then control goes to either Heap Area of method Area.
→ if method is static then control goes to method area.
→ if method is non-static then control goes to heap area
(if not available in heap area then control to method area).

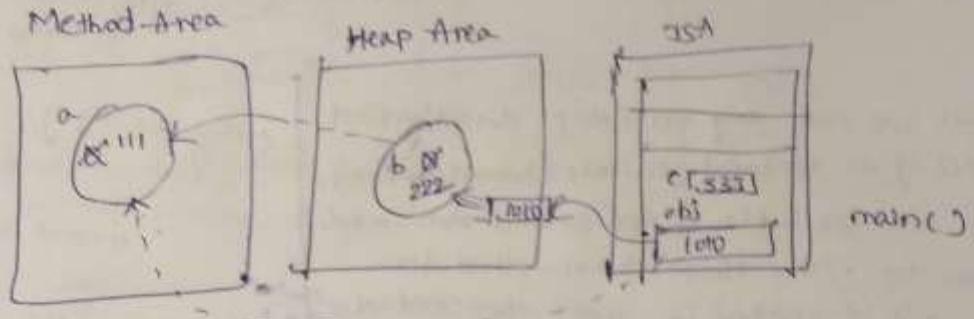
calling directly → preference local variable
calling class → method Area
obj → HA.

* If we call any variable by reference of object first preference gives to heap area, if data is not available then control goes to method area.

* If we call any variable by using class name then control goes to method area.

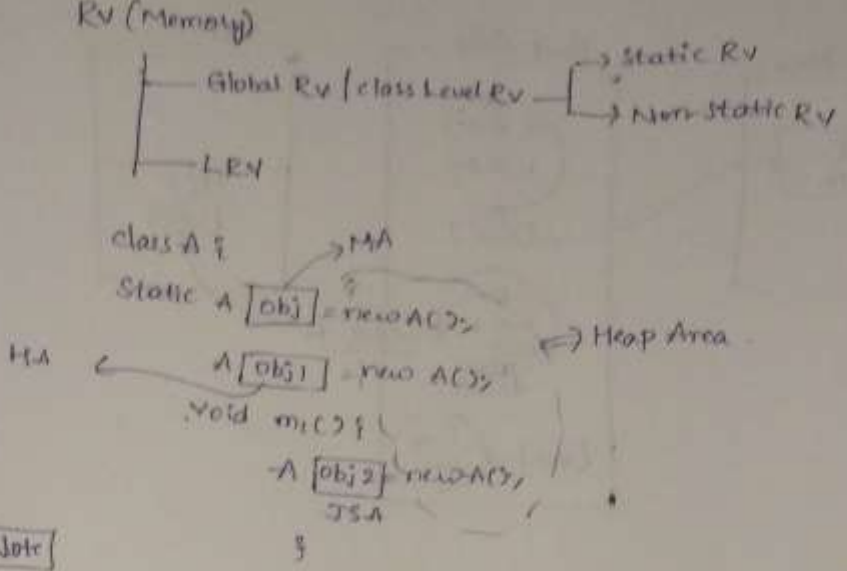
Prgm:

```
Public class variable {
    Static int a=111;
      int b=222;
    void m1(){
       S.o.p ("a:"+a);
       S.o.p ("b:"+b);
    }
    Public static void main(String[] args) {
        int c=333;
      variable obj= new variable();
       S.o.p ("c:"+c);
      // S.o.p ("b:"+b); // invalid // b is method is static
        S.o.p ("a:"+a);
        S.o.p("= = = = = =");
        S.o.p("a:"+variable.a);
        S.o.p("b:"+obj.b);
        S.o.p("a:"+obj.a);
    }
}
```



for a variable

Referenced Variables:



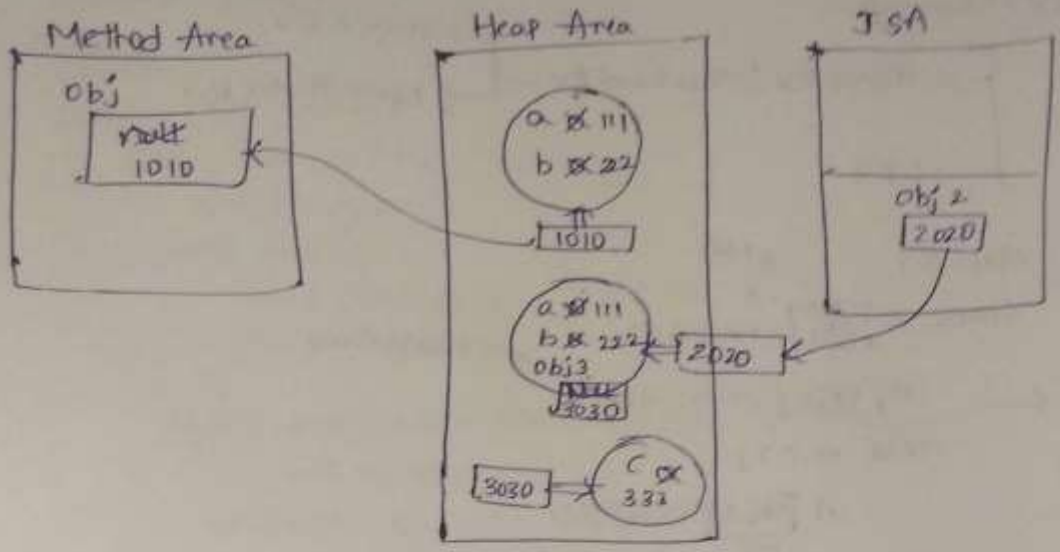
*Note

Reference variable will be created in different areas but memories are always created in Heap Area.

Prgm:

```
class A{
    int c=333;
}
public class Variable {
    int a=111;
    int b=222;
    A obj3 = new A();
    //static Variable obj=new Variable();
    Variable obj1= new Variable();
    public static void main(String[] arg) {
        S.o.p ("*****");
        Variable obj2 = new Variable();
    }
}
```

Internal flow:



Referenced Variables:

Referenced variables are categorized into two types.

1. Class level referenced variable
2. Local referenced variable.

Class level referenced variable:

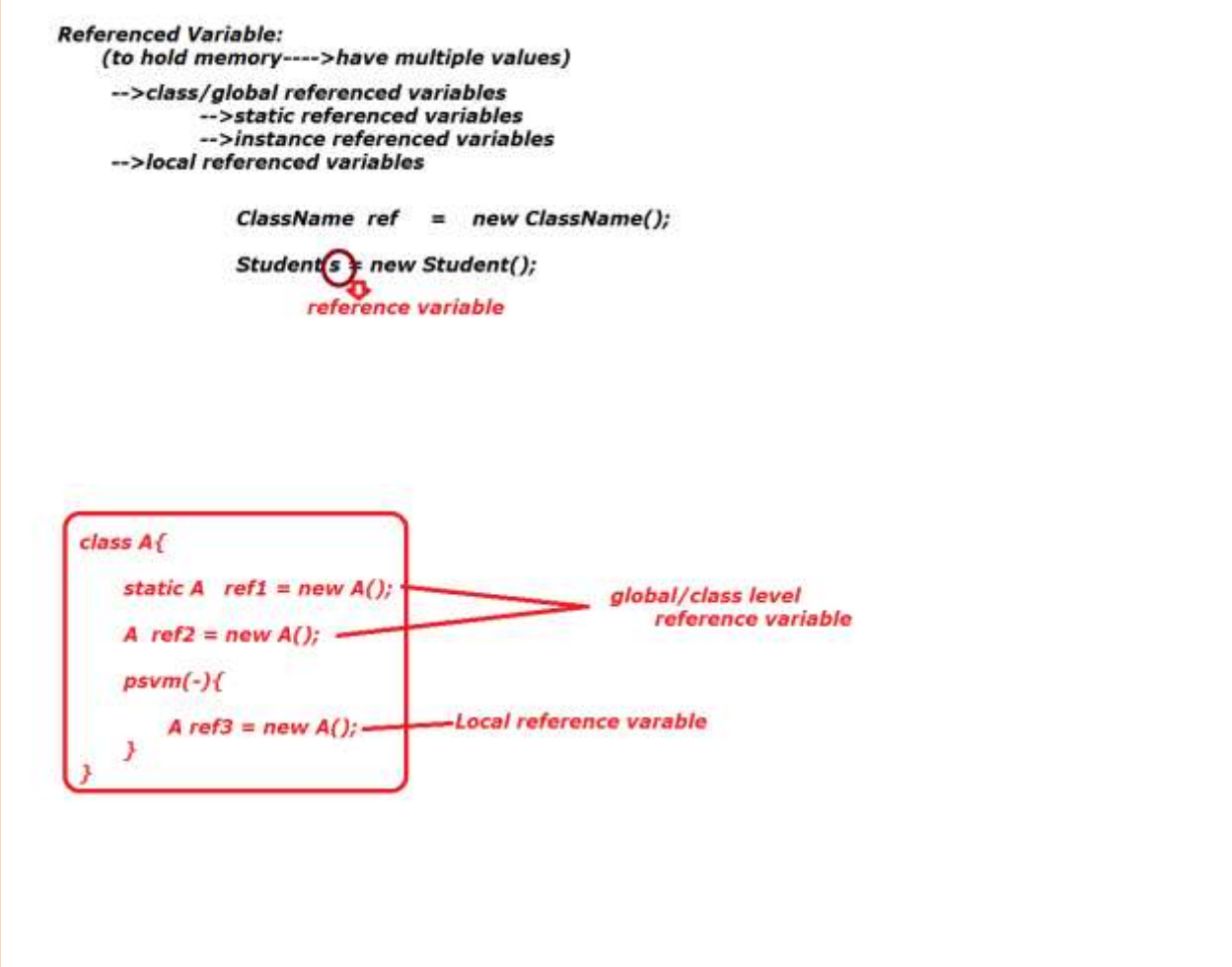
A referenced variable available at class level is called CLRV.

It has been categorized into two types

a. static referenced variable

b. non-static/instance referenced variable.

```
class Student{
    Student s = new Student();
    static Student s1 = new Student();
}
```



Local referenced variable:

A variable which is available at block or method level is called LRV.

Ex:

```
class Student(){
    p s v main(-){
    Student s = new Student ();
    }
}
```

```
package oops;

public class Variable {
```

```
	int a = 111;
	static Variable obj2 = new Variable();
	//Variable obj = new Variable();

	public static void main(String[] args) {
		System.out.println("************");
		Variable obj1 = new Variable();
	}
}

/*Note: the combination of bellow statements at class level
	illegal
	static Variable obj2 = new Variable();
	Variable obj = new Variable();
*/

/* we can not write following statements, two times at class level
and method level illegal
	Variable obj = new new Variable();

*/
```

Arrays can hold more than one value with same type, but it is unable hold different type of data , to overcome this problem then we can go for another referenced datatype is called "class".

Class is an imaginary thing, which is not existed in the real world.

Class is a model.

Class is a model for creating objects. Means the properties and actions of the objects are written in the class.

Properties are represented by variables.

Actions of the objects are represented by methods.

So a class contains variables and methods.

The same variable and methods are also available in the objects because they are created from the class.

These variables are also called as instances.

Q) How can we provide functionality to a class?

A) By creating an object.

Object:

Whatever the functionalities the class having, we should get those functionalities by creating object only.

Creating an object is nothing but, allocating memory, necessary to store the actual data of the variable.

By creating the object only, we can give the memory to name, age variables

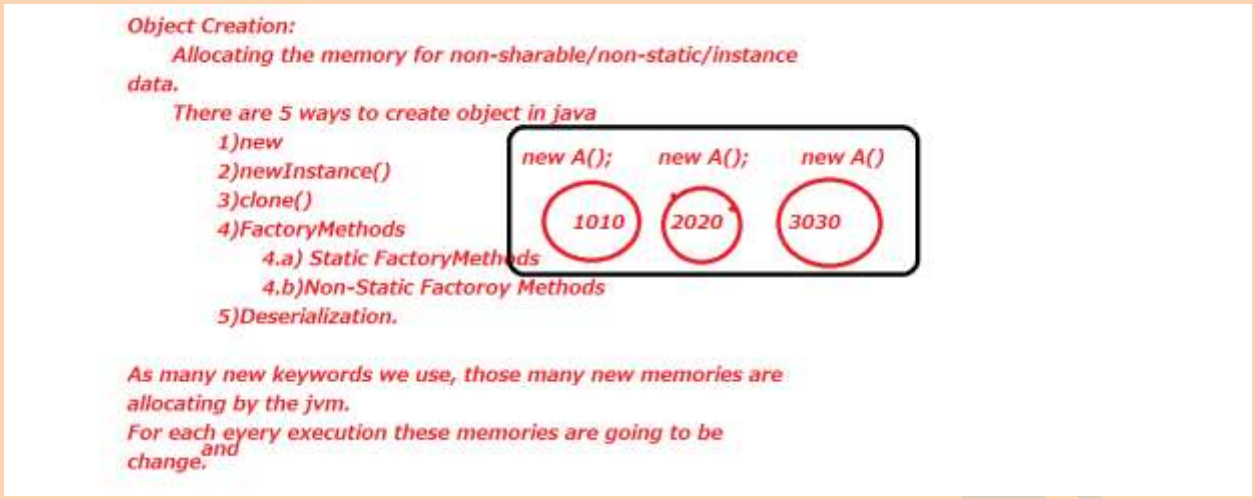
Class Def:

It is a java programming element. It can be hold both static and non-static variables, static and non-static method, static and non-static block, both static and non-static inner class, interface, enum.

Object Creation:

In java we have 6 ways to create an object.

1. new keyword
2. clone()
3. newInstance()
4. Factory methods.
    a. static factory methods
    b. non-static factory methods
5. Deserialization.
6. Literals.



1. new Keyword:

In java we can create two types of objects.

a. referenced object.

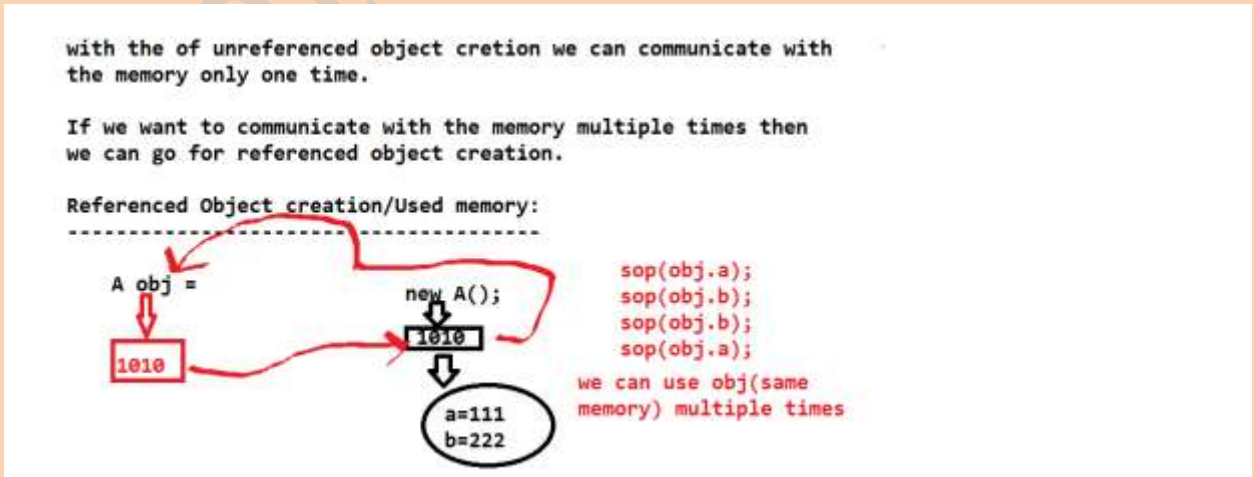
b. un-referenced object.

Referenced Object Creation or Used Memory:

Syntax: Class_name reference_name = new class_name();

Whenever JVM see the new keyword blindly provides new memory.

With the help of referenced object creation we can reuse the memory more than one time.



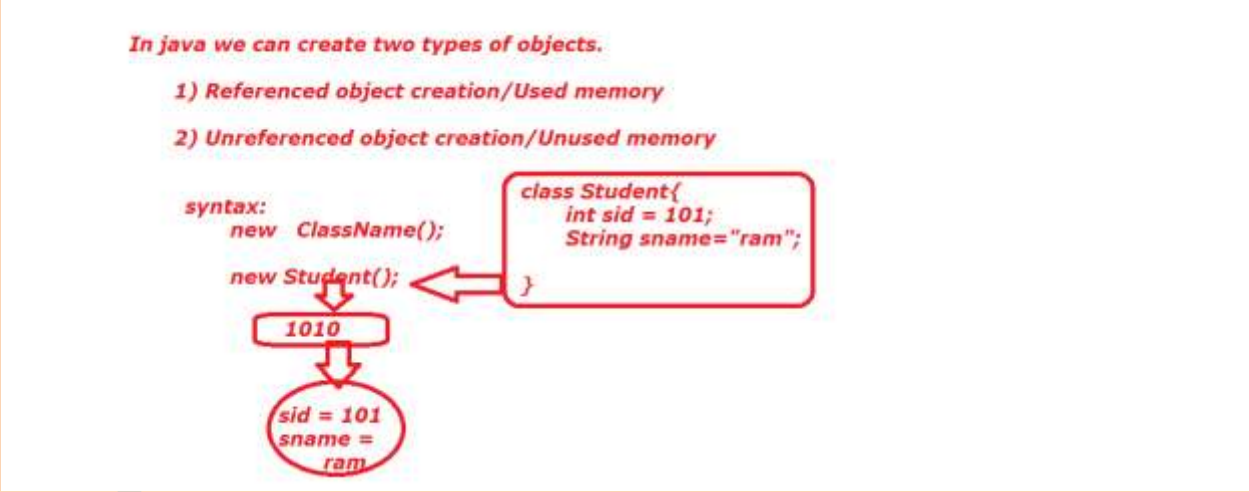
Unreferenced object creation:

new class_name();

With help of unreferenced object we can use the memory only one time.

With the help of object we can communicate with both static and non-static data also.

```
class Student {
    int a = 111;
    int b = 222;
    psv main(){
    Student s1 = new Student();
    new Student();
    }
}
```



Difference between object and reference:

Object means allocating memory.

Reference means holding the memory, which is generated by object.

But both object and reference will points to same memory.

With the help of reference, we can use the same memory more than one time.

With the help of object, we can use the memory only one time.

```
public class SRV {
        int x = 100;
        int y = 200;
    public static void main(String[] args) {
        SRV obj = new SRV();
        System.out.println(obj.x);

        System.out.println(obj.y);
        new SRV();
        int a = new SRV().x;
        int b = new SRV().y;
        System.out.println(new SRV().x);
        System.out.println(new SRV().y);
    }
}
public class SRV {
    SRV obj = new SRV();
    public static void main(String[] args) {
        SRV obj1 = new SRV();
    }
    /*static SRV obj = new SRV();
    public static void main(String[] args) {
```

```
            SRV obj1 = new SRV();
            System.out.println(obj1);
            System.out.println(obj);
      }*/
}


public class SRV {
      int a = 111;
      static int b = 222;
      public static void main(String[] args) {
            int a = 333;
            int b = 444;
            SRV obj = new SRV();
            System.out.println(a);
            System.out.println(obj.a);
            System.out.println(obj.b);
            System.out.println(b);
      }
}
```

JVM always gives priority to local data.

If we are calling variable directly first compiler will within the local area (block), if not then it will go to class level. (Heap or Method area).

s.o.p(obj.a) then compiler directly check in object are(heap area),f available that a value bind to obj, then JVM will print a value in runtime.

If not compiler will check in the method area or static are

If available compiler will bind the data to obj, then JVM will print a value in runtime.

If not available then compile time error.

```
public class NSD {
        static int b = 333;
        int a = 111;
//static int a = 222;
//same we cannot be define more than one time
//in the same scope
public static void main(String[] args) {
                int a = 222;
                System.out.println(a);(1)
                new NSD();//no use
                System.out.println(new NSD().a);(2)
                NSD ns = new NSD();
                System.out.println(ns.a);(3)
                System.out.println(ns.b);(4)
                System.out.println(NSD.b);(5)
                System.out.println(b);(6)
        }
}
```

Note:

In step 1) JVM will give priority to local data first

If data is not available in block level (Java Stack Area) then control goes to method area.

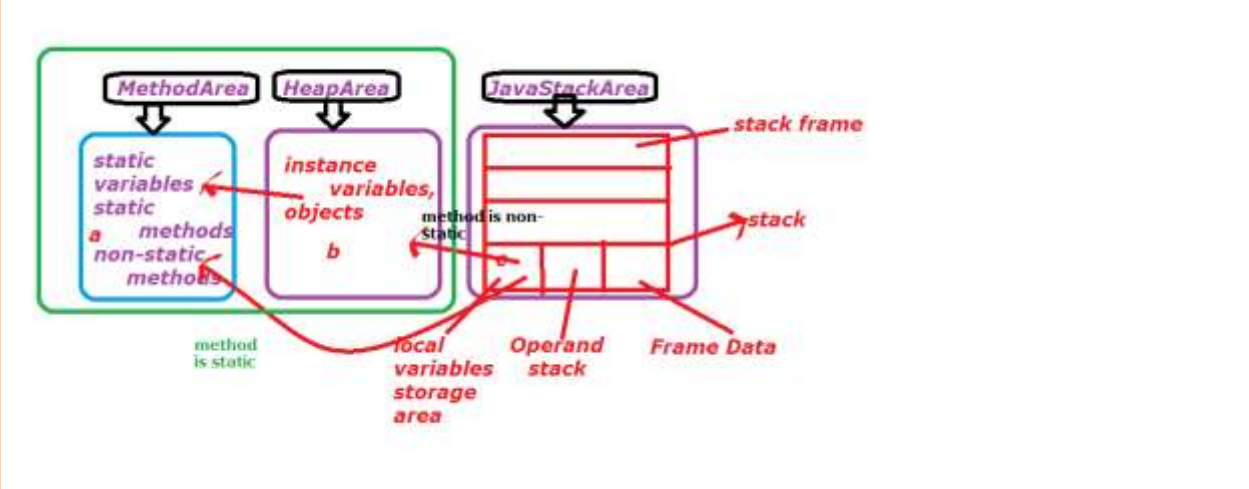
2) JVM control directly goes to heap area if not available then goes to method area

3) Same as.

4) Same as 2.

5) Same control directly goes to method area

6) JVM control goes to Java Stack Area, if not available then control to goes to method area



Scopes of variables:

We have two scopes.

1. Class scope
2. Block scope

Class scope variables we can access within the entire class level

Block scope variables we can access within the same scope.

```
class Student1{
    int sid = 111;
    String sname = "suji";
}
```

```
public class SRV {

public static void main(String[] args) {

        Student1 s1 = new Student1();

        Student1 s2 = new Student1();

        Student1 s3 = new Student1();

        System.out.println("s1: "+s1.sid+"...."+s1.sname);

        System.out.println("s2: "+s2.sid+"...."+s2.sname);

        System.out.println("s3: "+s3.sid+"...."+s3.sname);

        }

}
```

Output:

```
s1: 111....suji

s2: 111....suji

s3: 111....suji
```

In above program we are getting same id, name for all the students, it is not possible in real time, to avoid this drawback we should provides different values for different students. If we want to achieve this requirement then we can go for constructors.

Constructors:

Constructors are special blocks in class level, to initialize the non-static variables meanwhile of object creation.

Constructors are used to creating object.

Constructors are class level blocks.

Constructor name is same as class name.

After successfully executing of all non-static variables, blocks then constructors will be executed.

Once constructor is successfully executed, then we can say this, our object is successfully initialized.

Constructors are plays major role in object creation.

In constructor block we can write throws keyword and return statement without value. But we cannot write return statement with value.

Without constructor we cannot create object for a class "with new keyword".

object creation means calling or invoking the constructor to provide memory and initialize the non-static data.

Constructors are three types.

a. Default constructor.

b. Zero argument/non-parameterize constructor.

c. Parameterized/ augmented constructor.

Default constructor:

If we are not specify any constructor in our class by default compiler will provide one constructor with zero parameters is called default constructor.

In that default constructor compiler will provides one non-static variable that is "super ()".

Ex: class A{

}

javac A.java

javap A

```
class A extends java.lang.Object{
    A(){
        super();
    }
}
```

javap command is used to check the .class files information.

javap command always needs fully qualified name(package name + class name /interface name/ enum /abstract class/ annotation).

If our class is public compiler generated constructor also public.

If our class is default compiler generated constructor also default.

We cannot write protected, private, static, native, volatile, transient, synchronized modifiers in front of outer class name.

We can write only default, public, final, abstract, strictfp in front of the outer class names.

Zero-argument constructor:

The constructor, which is created by the programmer without any argument is called zero argument constructor.

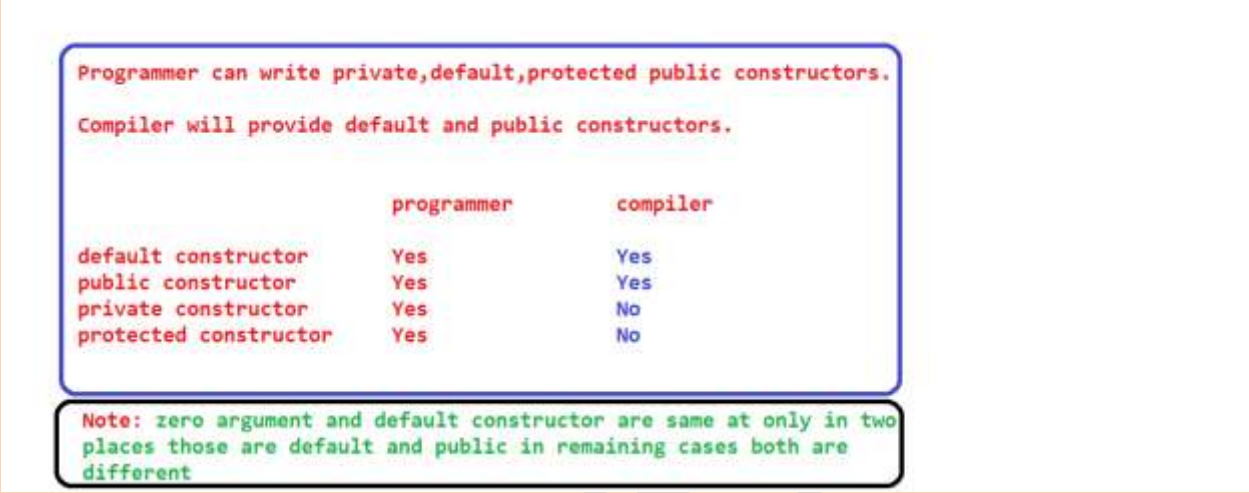
```
public class A{
    A(){//zero argument
    }
}
```

Zero argument constructor can allows, all the access modifier (private, default, protected, public)

Zero argument constructor is same as default constructor in two areas (default, public), in remaining two areas both are different.

Note:

If the class is default, then compiler provide constructor (default constructor) and zero argument constructor with default access modifier then only both default and zero argument constructor are same, otherwise different.



Parameterized constructor:

The constructor, which is created by the programmer with argument/parameters is called parameterized constructor.

The parameter count must be at least one.

Ex:

```
class Student{
        Student(){//zero argument
        }
                Student(int x){//parameterized
        }
}
```

Within the one class we can write multiple constructors.

Q) Difference between method and constructor?

Method name and constructor name can be same as class name, but method always having return type, whereas constructor doesn't.

Method having identity where as constructor doesn't have.

Constructors are automatically executed meanwhile of object creation.

Whereas methods never executed automatically meanwhile of class loading as well as object creation. If we want to execute method we need to call.

```
class Student1{
	int sid = m1();
	int m1(){
		System.out.println(sid);
		System.out.println("m1 method");
		return 111;
	}
	Student1(){
	   System.out.println("zero argument constructor");
	}
	{
		System.out.println("non-static block");
	}
	Student1(int x){
	     System.out.println("parameterized constructor");
		System.out.println(x);
	}
}
public class SRV {
```

```
	public static void main(String[] args) {
		Student1 s1 = new Student1();
		Student1 s2 = new Student1(200);
	}
}
```

For single object creation only one respectable constructor is executed.

If we want create an object we need respective constructor otherwise we will not create object with the help of new keyword.

As many object we created those many times non-static variables, blocks, constructors (respective) will be executed.

Object creation means communicating with the constructor only for initializing the non-static data/variables and forwarding the controle from sub class to super class with the help of super keyword.

At time we can call only one constructor.

```
class A{
	A(int x){
	}
	public static void main(String[] s){
		A obj = new A();
	}
}
```

above program gives compile time error. the reason is

new A() this statement meaning is calling the either zero or default constructor.

but in the program we didn't write zero argument constructor aswellas compiler not provides default constructor the reason programmer already written parameterized constructor.

We can avoids above errors in two ways.

a. by writing zero argument constructor by the programmer

```
A(){
}
```

b. changing the object creation statement.

```
A obj = new A(111);
```

Constructor overloading:

Wring the same constructor more than one time with in the same class with different parameters is called constructor overloading.

```
class A{
    A(){}
    A(int x){}
    A(float y){}
    A(int x, float y){}
    A(float x, int y){}
}
```

Note: Don’t consider variable names.

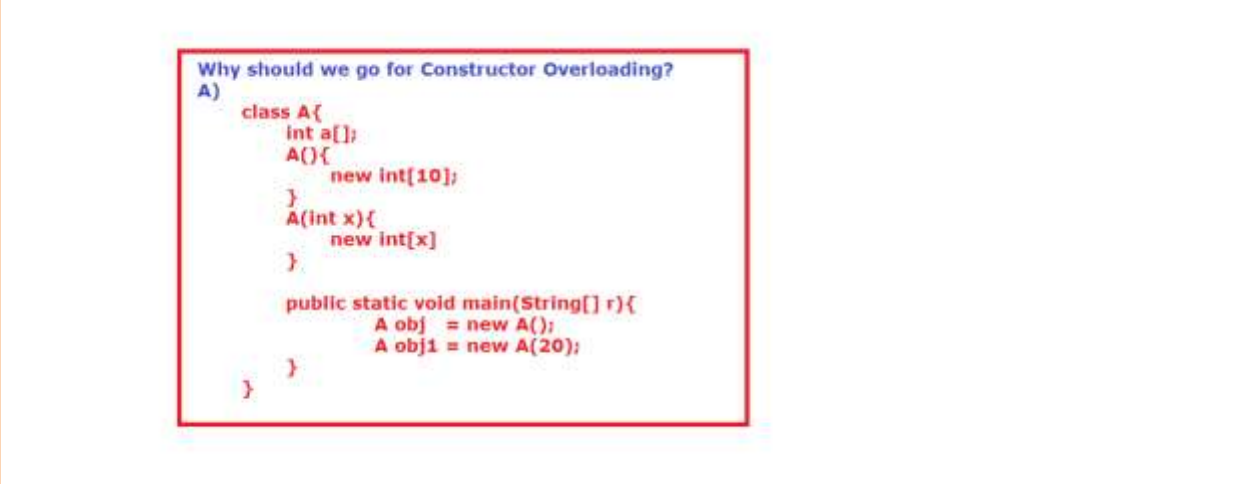
Note: one class constructor can‘t be placed in another class.

```
class A{
    B(){//invalid
    }
}
```

```
class B{
}
```

Executing the non-static variable and blocks is called object creation.

Executing constructor is called object initialization




```
import java.util.Scanner;
class Account{
    double totalBalance;
    Account(){
    }
    Account(double minimumBalance){
        totalBalance = minimumBalance;
    }
    public static void main(String[] s){
        System.out.println("enter type of person");
        Scanner scan = new Scanner(System.in);
        String personType = scan.next();
        if(personType.equalsIgnoreCase("Student")){
```

```
            Account obj1 = new Account();
        System.out.println("balance: "+obj1.totalBalance);
        }
        else
          if(personType.equalsIgnoreCase("Employee")){
            Account obj2 = new Account(5000);
        System.out.println("balance: "+obj2.totalBalance);
         }
        else {
            Account obj3 = new Account(500);
        System.out.println("balance: "+obj3.totalBalance);
        }
    }

}
```

Constructor chaining:

Communicating with one constructor to another constructor is called constructor chaining.

If we want develop constructor chaining in java, we need any one of these keywords

1. this.
2. super.

constructor chaining:

Making a communication between one constructor to another.

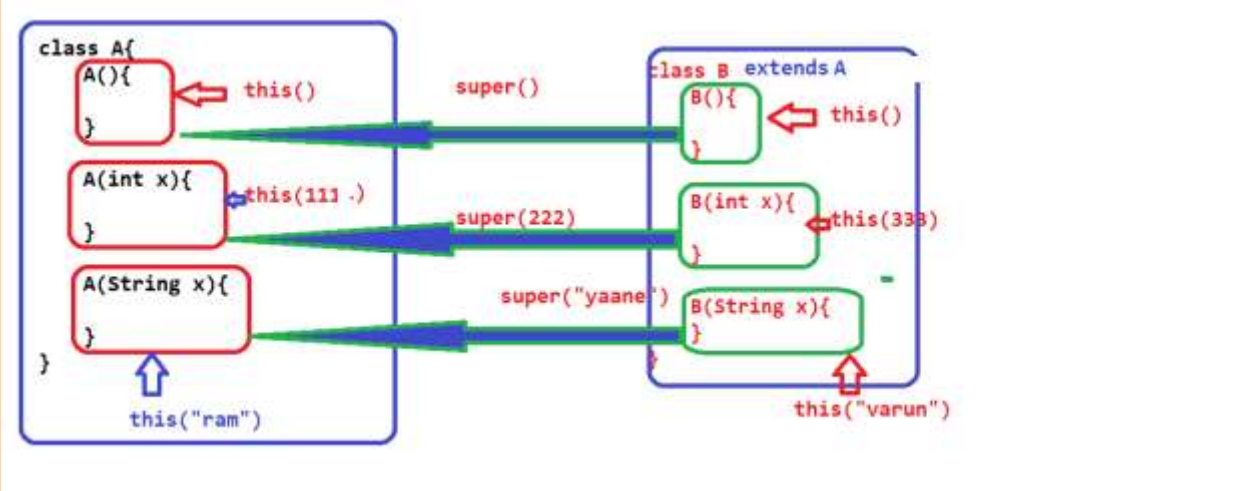
communication between

subclass constructor--> super class constructor(super())

one constructor to another--->this()

we can represents constructors

A()	**this()**	**B() ---> super()**
A(int x)	**this(111)**	**B(int x) --->super(111);**
A(String x)	**this("ram");**	**B(String x) --->super("anjali")**
A(int x, float y()	**this(100,34.00f)**	**B(int x, float y)--->super(100,20f)**
A(float y, int x)	**this(34f,2000)**	**B(float y, int x)---> super(1f,2)**




```
class Student{
        int sid;
        String sname;
        Student(){
                this(111);
                System.out.println("zero argument constructor");
        }
        Student(int x){
                this("ram");
                System.out.println("single int parmeterized constructor");
```

```
            System.out.println(x);
        }
        Student(String s){
            System.out.println("single String parmeterized constructor");
            System.out.println(s);
        }
}
public class ConstructorChaining {
        public static void main(String[] args) {
            Student s = new Student();
        }
}
class B{
        B(int x){
            System.out.println("super class int argument");
        }
}
class C extends B{
}
```

In the above code we will get compile time error.

The reason is in the C class compiler will provide default constructor and super () (zero argument).

Like as bellow

```
        class C extends java.lang.Object{
            C(){
```

```
                super();
        }
    }
```

But in the super class we don't have any zero argument constructors.

Note: If super class not having any zero argument constructor, in the sub class programmer must be write constructor and with appropriate super () with argument.

```
package reference;

class B{
    B(){
        System.out.println("super class int argument");
    }
}
class C extends B{
    private C(){
    }
    public static  C getC(){
        return new C();
    }
    int a = 111;
}
public class ConstructorChaining {
    public static void main(String[] args) {
        //C obj = new C();
        C obj1 = C.getC();
```

```
        System.out.println(obj1.a);
    }
}
```

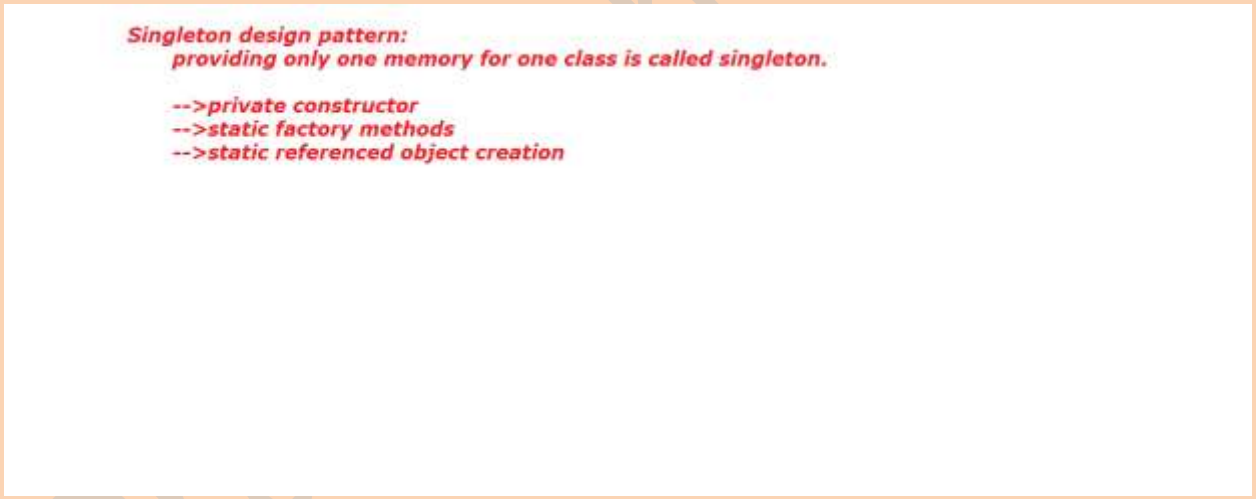
If any class having private constructor, that class must be have static factory method to communicate its(class) non-static data.

Singleton Design pattern:

A design patterns are well-proved solution for solving the specific problem/task.

SDP represents allocating memory for a particular class only one time. The same memory can be use multiple times. With the SDP we can save the memory.

If we want to develop SDP class we need one private constructor and public static reference variable and one static factory method.




```
class Servlet{
public static Servlet obj = new Servlet();
    private Servlet(){
    }
    public static Servlet getServlet(){
        return obj;
```

```
	}
}
public class SingletonDesignPattern {
public static void main(String[] args) {
	//System.out.println(new Servlet());

	Servlet obj1 = Servlet.getServlet();
	Servlet obj2 = Servlet.getServlet();
	System.out.println(obj1.hashCode());
		System.out.println(obj2.hashCode());
	}
}
```

Additional Program on constructor chaining:

```
package variable;

class A{
	A(){
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		System.out.println("super class int argument constructor");
	}
	A(String x){
		System.out.println("super class String argument constructor");
	}
}
class B extends A{
	B(){
		//super();
		System.out.println("sub class zero argument constructor");
	}
	B(int x){
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		System.out.println("sub class String argument constructor");
	}
}
```

```
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}
o/p:
super class zero argument constructor
sub class zero argument constructor

package variable;

class A{
	A(){
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		System.out.println("super class int argument constructor");
	}
	A(String x){
		System.out.println("super class String argument constructor");
	}
}
class B extends A{
	B(){
		this(123);
		System.out.println("sub class zero argument constructor");
	}
	B(int x){
		//super();
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		System.out.println("sub class String argument constructor");
	}
}
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}

package variable;

class A{
	A(){
		//super();
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		System.out.println("super class int argument constructor");
	}
	A(String x){
		System.out.println("super class String argument constructor");
	}
}
```

```
class B extends A{
	B(){
		this(123);
		System.out.println("sub class zero argument constructor");
	}
	B(int x){
		this("ram");
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		//super();
		System.out.println("sub class String argument constructor");
	}
}
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}
```

```
package variable;

class A{
	A(){
		this(234);
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		this("varun");
		System.out.println("super class int argument constructor");
	}
	A(String x){
		System.out.println("super class String argument constructor");
	}
}
class B extends A{
	B(){
		this(123);
		System.out.println("sub class zero argument constructor");
	}
	B(int x){
		this("ram");
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		//super();
		System.out.println("sub class String argument constructor");
	}
}
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}
package variable;
```

```java
class A{
	A(){
		this(234);
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		this("varun");
		System.out.println("super class int argument constructor");
	}
	A(String x){
		super();
		System.out.println("super class String argument constructor");
	}
}
class B extends A{
	B(){
		this(123);
		System.out.println("sub class zero argument constructor");
	}
	B(int x){
		this("ram");
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		//super();
		System.out.println("sub class String argument constructor");
	}
}
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}
```

```java
package variable;

class A{
	A(){
		this(234);
		System.out.println("super class zero argument constructor");
	}
	A(int x){
		this("varun");
		System.out.println("super class int argument constructor");
	}
	A(String x){
		super();
		System.out.println("super class String argument constructor");
	}
}
class B extends A{
	B(){

		this(123);
```

```
		//super();

		System.out.println("sub class zero argument constructor");
		//this(123);
		//super();
	}
	B(int x){
		this("ram");
		System.out.println("sub class int argument constructor");
	}
	B(String x){
		System.out.println("sub class String argument constructor");
	}
}
public class ConstructorChaining {
	public static void main(String[] args) {
		new B();
	}
}
```

Copy Constructor:

To Placing one object data into another object we required one constructor that is copy constructor.

The constructor which have same class reference variable as parameter is called copy constructor.

```
package inheritance;

class Student{

	int sid ;

	String sname;

	int sage;

	Student(int sid,String sname,int sage){

		this.sid = sid;

		this.sname = sname;
```

```
        this.sage = sage;
    }
    Student(Student obj){//copy constructor
        this.sid = obj.sid;
        this.sname = obj.sname;
        this.sage = obj.sage;
    }
}
public class CopyConstructor {
    public static void main(String[] args) {
        Student s = new Student(101,"mounika",25);
        Student s1 = new Student(s);
        System.out.println("s: "+s.sid+"..."+s.sname+".."+s.sage);
        System.out.println("s1:"+s1.sid+"..."+s1.sname+".."+s1.sage);
    }
}
```

"this" keyword:

"this" is a non-static final reference variable, which can be created by compiler and memory filled by the JVM, meanwhile of object creation, to hold current object.

Current Object means, the object which is used by the programmer currently/at present.

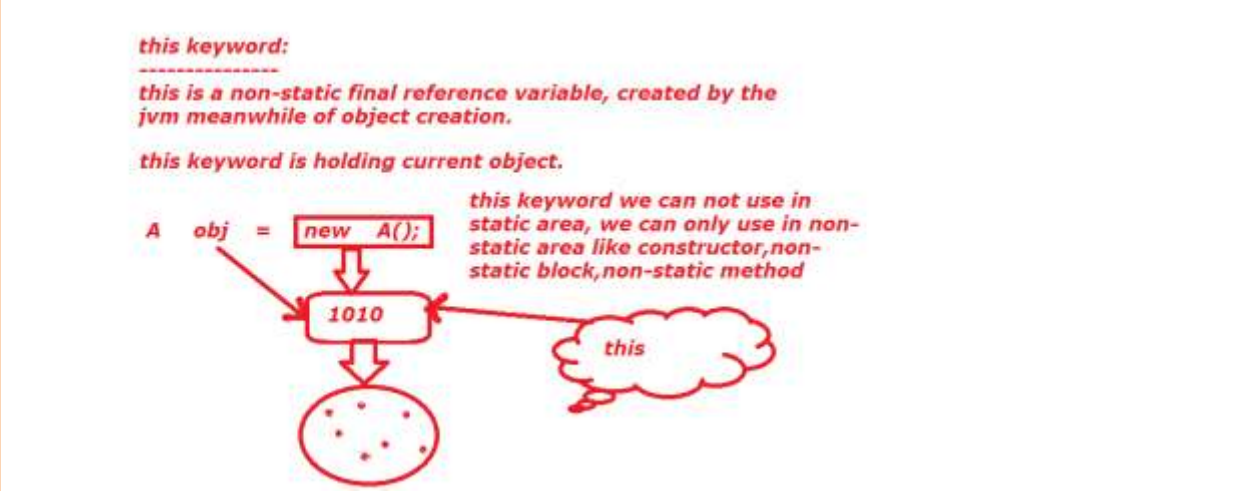
Every memory pointed by one non-static final referenced variable that is "this" reference variable.

Whenever we can any variable directly, compiler gives first preference to current location, in the current location if variable not existed, then compiler will that syntax as bellow

Syntax: this.variablename.

1) this keyword is a non-static final reference variable.

2) this keyword can't be used from static area(static method, static block). It can be accessed only within the non-static area(non-static method, non-static block, constructor). We can't access this keyword even by using object also.



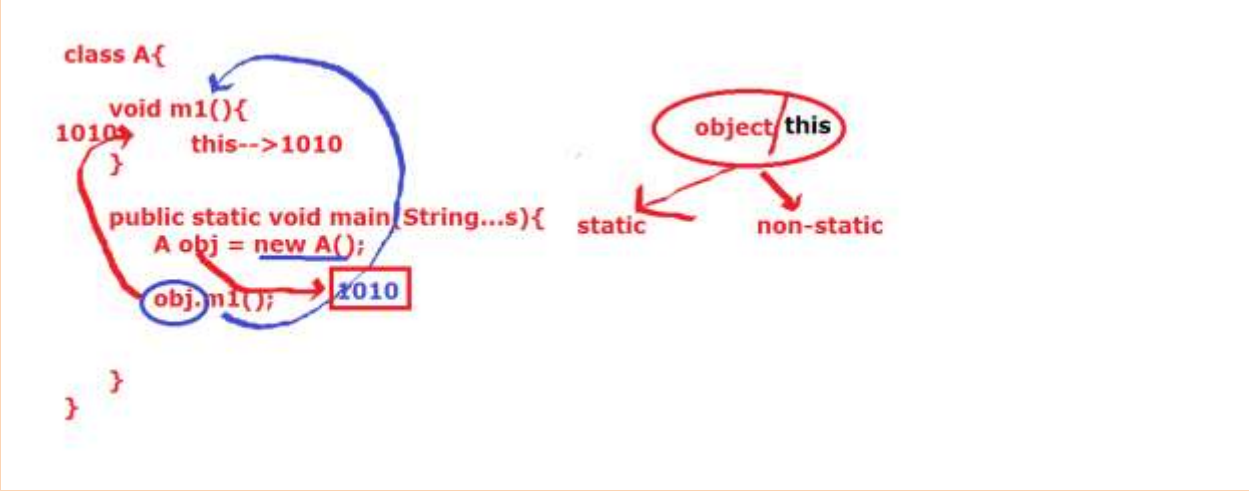

```
public class ThisDemo {
        int a = 111;
        public static void main(String[] args)          {
        ThisDemo td = new ThisDemo();
        System.out.println(td.a);
        //System.out.println(this.a);
        }
}
```

3) We cannot change the value of this keyword.

```
public class ThisDemo {
	int a = 111;
	void m1(){
	System.out.println(this);
	//System.out.println(this+100);
	}
	public static void main(String[] args)	{
	ThisDemo td = new ThisDemo();
	System.out.println(td.a);
	//System.out.println(this.a);
	td.m1();
	}
}
```

4) Whenever we calling m1 () on top of "td" reference (current object), the m1 () is going to be executed, in the meanwhile td reference value is also going to m1 ().

If we want to hold td reference value in m1() we need "this" keyword.



5) this keyword is also communicating with both static and non-static data.

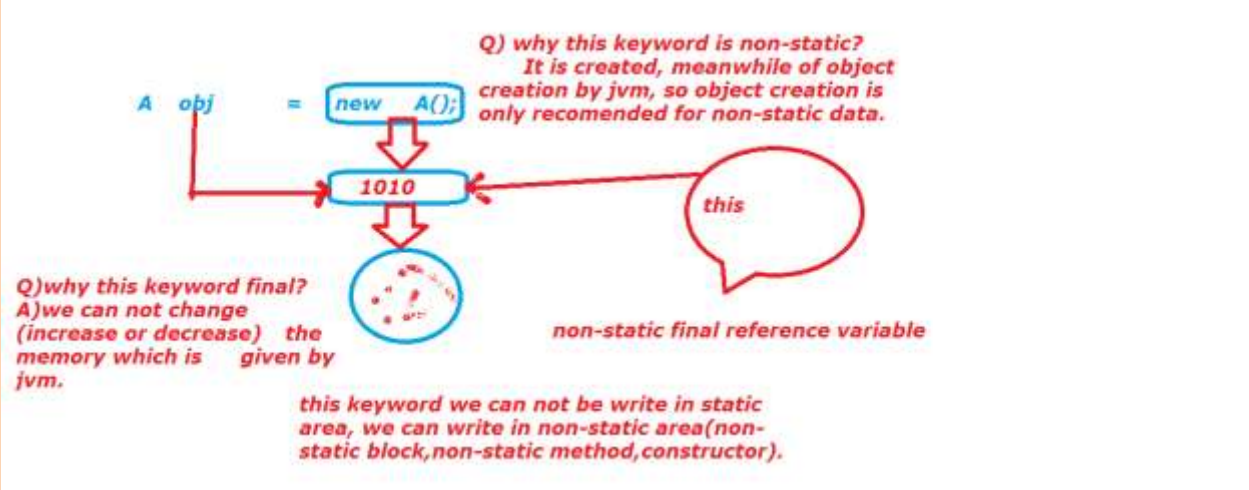
6) this keyword is used to make a communication between constructor, with in the same class.

7) this keyword must be first statement in the constructor.

8) We should leave one constructor for "super" keyword.

That means we can't use this keyword in all constructor.

9) We can't call same constructor, within its body.



If any object is suppose to call any block or method to execute either directly or indirectly, that object memory will avialble in those blocks, that memory we can hold with the support of "this" keyword.

```
class Student{
    Student(){
        this(111);
        System.out.println("zero arguement constructor");
        //this();
    }
```

```
	Student(int x){
		//this();
		System.out.println("parmeterized constructor");
	}
}
public class ThisDemo {
	int a = 111;
	static int b = 222;
	void m1(){
		System.out.println(this.a);
		System.out.println(this.b);
	}
	public static void main(String[] args) {
	ThisDemo td = new ThisDemo();
		td.m1();
		Student s = new Student();
	}
}
public class ThisDemo {
	int a = 111;
	static int b = 222;
	static {
		System.out.println("static block");
		//System.out.println(this);
```

```
}
ThisDemo(){
System.out.println("constructor: "+this);
}
{
System.out.println("non-static block");
System.out.println("nsb:****** "+this);
}
void m1(){
        System.out.println("m1: "+this);
        //System.out.println(this++);
        System.out.println("m1 method");
        System.out.println(this.a);
        System.out.println(this. b);
}
public static void main(String[] args) {
        System.out.println("main method");
        //System.out.println(this);
        ThisDemo td = new ThisDemo();
        System.out.println("td: "+td);
        td.m1();
        System.out.println("********");
        ThisDemo td1 = new ThisDemo();
        System.out.println("td1:"+td1);
```

```
        td.m1();
    }
}
```

10) this keyword is used forwarding the current object to another method as an argument.

```
class Student{
void m1(){
    System.out.println("m1 method");
    System.out.println("this: "+this);
    m2(this);//compiler conver into this.m2(this);
    System.out.println("--------------");
    Student s1 = new Student();
    System.out.println("s1: "+s1);
    m2(s1);//compiler conver into this.m2(s1);
    System.out.println("--------------");
    m2(new Student());//compiler convert into//  this.m2(new Student());
    System.out.println("--------------");
    m2(this);
    }
    void m2(Student s){
        System.out.println("m2 method s: "+s);
    }
}
public class ThisDemo {
    public static void main(String[] args) {
```

```
		Student s = new Student();
		System.out.println("s: "+s);
		s.m1();
	}
}
```

11) this is used to differentiate class level variable to local variable.

```
class Student{
	int sid = 111;
	String sname = "ram";
	/*Student(int id, String name){
	sid = id;		//this.sid = id;
	sname = name;//this.sname = name;
	}*/
	/*Student(int sid, String sname){
		sid = sid;
		sname = sname;
	}*/
	Student(int sid, String sname){
		this.sid = sid;
		this.sname = sname;
	}
}
public class ThisDemo {
   public static void main(String[] args) {
```

```
        Student s = new Student(222,"suji");

        System.out.println(s.sid);

        System.out.println(s.sname);

    }

}
```

```
package variable;

public class L {

        L(){
                System.out.println("constructor:this     : "+this.hashCode());
        }
        {
                System.out.println("non-static block:this: "+this.hashCode());

        }
        public static void main(String[] args) {
                System.out.println("main method");
                L obj = new L();
                System.out.println("main:obj:          : "+obj.hashCode());
                System.out.println("==============");
                L obj1 = new L();
                System.out.println("main:obj1:         : "+obj1.hashCode());
        }
}
```

```
If any non-static context(non-static block, constructor,
non-static method) excuting with the help of any object,
that object only the current object in that particular
scenario, so "this" keyword always pointing to that
object only.
```

super keyword:

super keywords is an non-static final reference variable, to make communication between subclass constructor to super class constructor.

Whenever using extends keyword whatever the data which is available in super class that data will comes to subclass. If we want extract the data first we need to forward our control from subclass to super class.

For this purpose we need one predefine keyword that is "super".

If we are not writing any super (), this () in constructor then compiler will place super (), in our constructor.

With in the single constructor we can write either this () or super (), we cannot write both. The reason is both super () and this () always needs first statement in the constructor.

All static and non-static variable, static and non-static block and constructor are loaded from super class to sub class.

```
class A extends java.lang.Object{
    int a = m1();
    static int b = m2();
    int m1(){
        System.out.println(a);
        System.out.println("A class non-static m1 method");
        return 111;
    }
    A(){
        super();
        System.out.println("A class constructor");
    }
    static{
```

```
		System.out.println("A class static block");
	}
	{
		System.out.println("A class non-static block");
	}
	static int m2(){
		System.out.println(b);
		System.out.println("A class static m2 method");
		return 222;
	}
}
class B extends A{
	int c = m3();
	static int d = m4();
	int m3(){
		System.out.println(c);
		System.out.println("B class non-static m3 method");
		return 333;
	}
	B(){
		super();
		System.out.println("B class constructor");
	}
	B(int x){
```

```
        super();
        //this();

        System.out.println("B class argument constructor");
    }
    static{
        System.out.println("B class static block");
    }
    {
        System.out.println("B class non-static block");
    }
    static int m4(){
        System.out.println(d);
        System.out.println("B class static m4 method");
        return 444;
    }
}
public class SuperDemo {
    public static void main(String[] args) {
        B obj = new B();
    }
}
```

"super" is used to differentiate the subclass variable to super class variable.

"super" is used to differentiate the subclass method to super class method.

"super" keyword can’t used in static area.

With the help of super keyword we can call both static and non-static data.

super keyword always represent super class memory.

```
package constructor;

class B{

	int a = 111;

	static int b = 222;

	void m1(){

		System.out.println("B class non-static m1()");

	}

	static void m2(){

		System.out.println("B class static m2()");

	}

}

public class SuperDemo extends B{

	int a = 333;

	static int b = 444;

	void m1(){

		int a = 555;

		System.out.println("SuperDemo class non-static m1()");

		System.out.println(a);

		System.out.println(this.a);
```

```
            System.out.println(super.a);
            System.out.println(b);
            System.out.println(this.b);
            System.out.println(super.b);
            m2();//this.m2();
            this.m2();
            super.m2();
            super.m1();
        }
        static void m2(){
            System.out.println("SuperDemo class static m2()");
        }
        public static void main(String[] args) {
            //System.out.println(super.a);
            SuperDemo td = new SuperDemo();
            td.m1();
        }
}
class L extends java.lang.Object{
        int a = 1000;
}
class A extends L{
        //int a = 111;
        static int b = 222;
```

```
	static void m1(){
		System.out.println("super class- static m1()");
	}
	void m2(){
		System.out.println("super class- non-static m2()");
	}
}
class B extends A{
	int a = 333;
	static int b = 444;
	static void m1(){
		System.out.println("sub class- static m1()");
	}
	void m2(){
		System.out.println("sub class- non-static m2()");
		int a = 555;
		System.out.println(a);
		System.out.println(this.a);
		System.out.println(super.a);
		m1();//this.m1();
		this.m1();
		super.m1();
		System.out.println(this.b);
		System.out.println(super.b);
```

```
            System.out.println(this.hashCode()+"....."+super.hashCode());
        }
}
class Test{
        public static void main(String[] s){
            B obj = new B();
            obj.m2();

        }
}
```

note: Both this and super keywords are pointing to same memory.

but this is starts searching process from sub class to its all super classes.

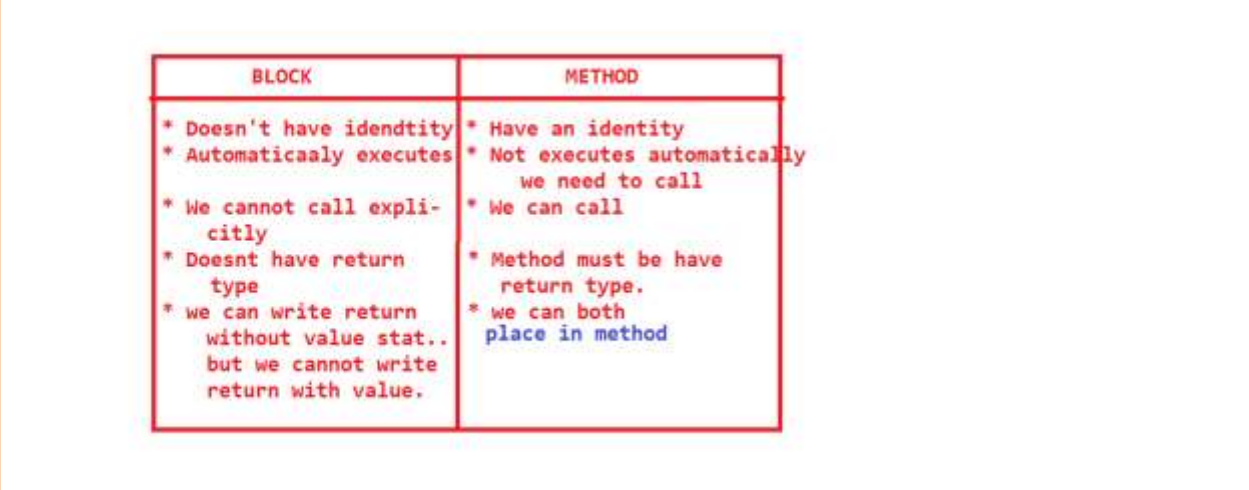
where as super keyword starts searching process from super class to its all

super classes.

Q) what is difference between blocks and methods?

A) **Blocks doesn't have identity whereas method have.**
Blocks automatically executes whereas method not.
We can't call blocks where methods we can.
Blocks doesn't have returntype whereas method have.
Blocks doesn't holds/have throws keyword whereas method have.
Blocks doesn't return statement with value whereas method have.

Blocks are executes from top to bottom whereas methods execution depends way of calling.



BLOCK	METHOD
* Doesn't have idendtity * Automaticaaly executes * We cannot call expli-citly * Doesnt have return type * we can write return without value stat.. but we cannot write return with value.	* Have an identity * Not executes automatically we need to call * We can call * Method must be have return type. * we can both place in method

Methods:

In java every value is hold by the variables.

If we want to update the value of an variables, we need some logic or operations.

In java, we can't do the operations in class level or we can't write logic in the class level.

So if we want to do the operations in java we have the following areas.

1. static Block.
2. Non-static block.
3. Constructors.

4. Methods.

If we are writing logic or operation in static block, that static block logic can be executed only one time, if we want to execute more than one we can't call explicitly.

If we are writing logic or operation in non-static block and constructor, those blocks execution need object creation, if we are executing those blocks multiple times, programmer should create multiple objects.

If we are keep on creating objects then we are unnecessarily wasting our memory.

To avoiding above drawbacks we have one special block in class level that is method.

We can execute the method multiple times based on our requirement.

The difference between block and method is blocks automatically executed whereas methods not executed, we need to call explicitly. Blocks don't have any specific name to call, whereas method have specific name to call.

Methods are mainly design for carrying the information from one place to another place and also used to holding the logic.

Ex:

```
class A{
	System.out.pritnln("not valid");//invalid
	static {
		s.o.p("only one time executed");
	}
	{
		s.o.p("object creation mandatory");
	}
	A(){
		s.o.p("object creation mandatory");
```

```
    }
    void m1(){
    s.o.p("writing the logic in method very flexible");
    }
}
```

Note: No method will be executed automatically, except main method.

Each and every class having it own functionalities, we can achieve these functionalities through methods.

```
 class Account{
    double accountNo=123456789;
    String accHoldName="bhuvana";
    double amout=10000;
}
class Deposit{
    public void deposit(int amount){
        Account acc = new Account();
        acc.amout = acc.amout+amount;
    }
}
class Withdrawl{
    public void withdrawl(int amount){
        Account acc = new Account();
        acc.amount=acc.amount-amount;
    }
}
```

```
class Transaction{
        psvm(--){
                Deposit d = new Deposit();
                d.deposit(10000);
                Withdrawl w = new  Withdrawl();
                w.withdrawl(5000);
                }
}
```

Note:

In the above program every class has its own functionalities.

We can differentiate with the help of different methods.

According to above information we can specify "method" is a block in the class, which contains set of statements, mostly we can say about these statements are called as actions or behavior or operation of a class.

All these statement have its own logic according to class specification.

That means each and every class has its own action and behavior.

For example sports class does not contain logic of Faculty class.

So the conclusion is, the logic of statements, which are located in the sports class method is different from logic of statements, which are located in the Faculty class methods.

The logic of a statement must be written in a method, not outside of a method.

If we attempt to write logic outside of method, the compiler will display the error message called as "identifier expected" System.out.println("hi");

Getting Modularity (differentiate from one logic to another logic).

We are always writing one logic within the one method.

Reduce Time Consuming.

Note: Java doesn't allow nested methods.

blocks are automatically executed whereas methods are not executed (we need to call).

blocks doesnot have identity whereas methods have identity.

by using identity we can able to call methods explicitly whereas block doesnot have identity, so we are unable to call explicitly.

blocks are executed the way we mention in the program from top to bottom, where as method are executed the way(order) we calling the methods.

Methods/Operations/Behaviour/Member methods:

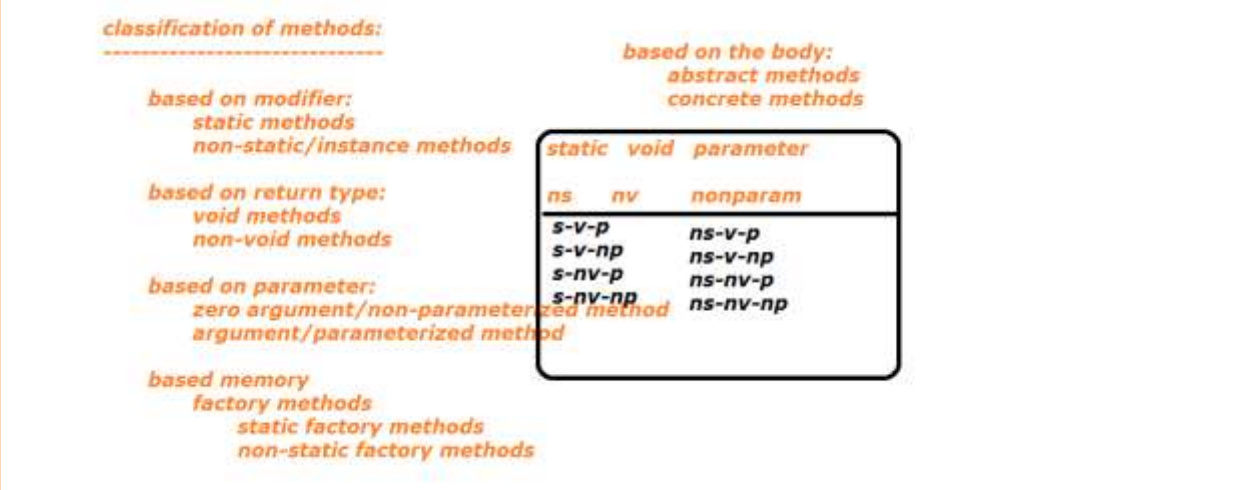
Method is sub block of class.
Method is holding the logic, which doing some operations on properties

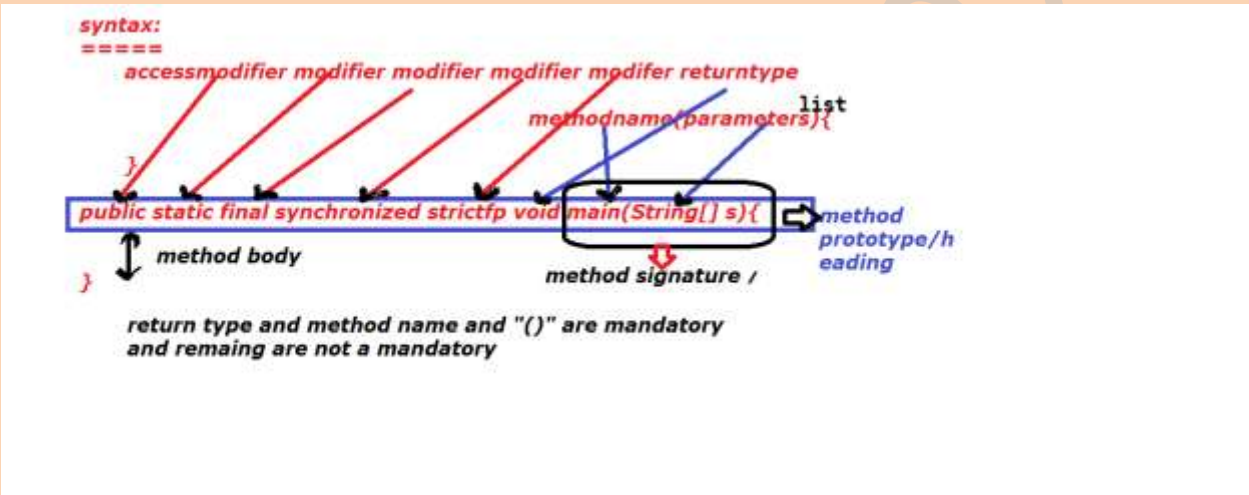
one object is communicating with another object with the hlep of methods.

Methods are having identity, which help of identity we can differentiate from one method to another method.

If we writing logic in static block we can execute only one time but we cannot multiple times, with the help of instatnce block and constructor, we can execute the logic mutltiple times, but we required more object (memory).

To avoiding above drawback, we are always prefer to writing the logic within methods.





Types of methods:

Based on modifier:

a. static methods

b. non-static methods

Based on return type:

a. void methods

b. non-void methods

Based on parameters

a. parameterized method

b. non-parameterized method

Factory methods:

a. static factory methods

b. non-static factory methods

Based on body:

a. abstract methods.

b. concrete methods.

Based On modifier:

a. static methods: A method which contains static keyword in its declaration or definition is called static method.

Ex:

```
static void m1(){

}

static int m2(){

        return 100;

}

static void m3(int x){

}
```

static methods can be called by using the following ways.

1. By directly (method name)
2. By class name
3. by object or reference.

Note: Other class static method can be call by using two ways only

1. by class name
2. by object or reference

b. Non-static methods: A method which doesn't have static keyword in its declaration or definition is called non-static method.

Ex:

```
void m1(){
}
int m2(){
        return 111;
}
void m3(int x){
}
```

We can call non-static method by using two ways.

1. by directly
2. by object or reference

Note:

Other class non-static can be calls by using object or reference only.

```
class AA{
        static void m3(){
                System.out.println("AA class static m3 method");
        }
        void m4(){
                System.out.println("AA class non-staticm4 method");
        }
}
public class MethodDemo {
        static void m1(){
```

```
        System.out.println("this is static m1 method");
    }
    void m2(){
        System.out.println("this is non-static m2 method");
        m5();
    }
    void m5(){
        System.out.println("this is non-static m5 method");
    }
    public static void main(String[] args) {
        m1();
        MethodDemo.m1();
        MethodDemo md = new MethodDemo();
        md.m1();
        new MethodDemo().m1();
        AA.m3();
        AA obj = new AA();
        obj.m3();
        new AA().m3();
        System.out.println("---------------");
        //m2();
        md.m2();
        obj.m4();
    }
```

}

Note:

We cannot call non-static data directly from static area.(method/block)

Based on method return type:

a. void method:

A method, which doesn't carrying any information from one place to another place, is called void method.

Void methods are not calling from initialization phase.

void means nothing to define.(void means not zero, not false, not null simply nothing to define).

Within the void methods we can write return statement.

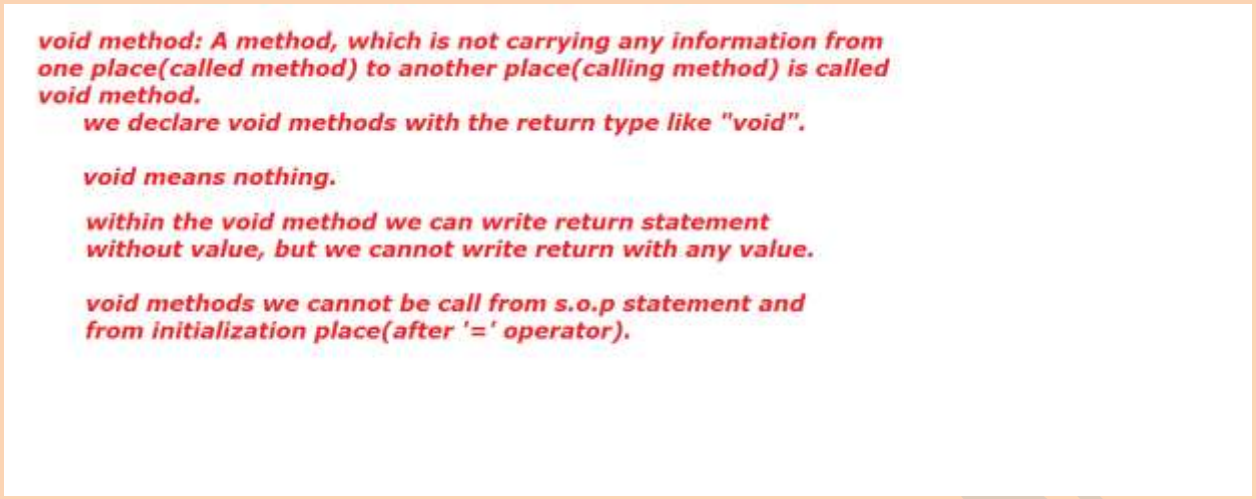
The return statement must be without value.

Note: void method can't be call from System.out.println().

```
void m1(){
}
s.o.p(m1());//invalid
```

Ex:

```
void m1(){
        return ;
        //return 100;//invalid
}
```



b. Non-void method:

A method, which carrying the information is called non-void method.

Here information means

a. primitive type

b. primitive arraytype

c. referenced type

d. referenced array type

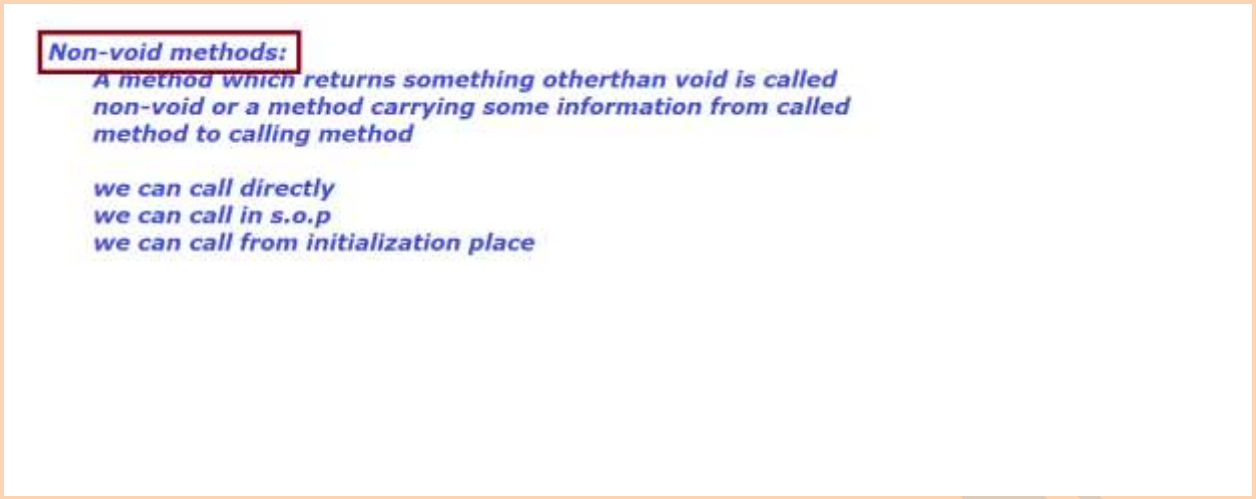
Every non-void method must be ended with return statement with respective value.

We can call non-void methods from s.o.p() statement.

Calling Method:

A method which is calling to another method, that method is called "Calling Method".

Called method: A method, which is called by other method, is called "Called Method".




```
class NStudent{
	String name="ramcharan";
}
public class MethodDemo {
	void m1(){
		System.out.println("non -static void m1()");
		m2();
		return;
	}
	static void m2(){
		System.out.println("static void m2()");
		//return 100;
	}
	int m3(){
		System.out.println("m3 method");
		return 100;
	}
```

```
boolean m4(){
        System.out.println("m4 method");
        return false;
}
String m5(){
        System.out.println("m5 method");
        return "ram";
}
int[] m6(){
        System.out.println("m6 method");
        //int a[] ={10,20,30};
        //return a;
        int a =11;
        int b= 22;
        int c=33;
        //return new int[]{a,b,c};
        return new int[]{11,22,33};
}
NStudent m7(){
        System.out.println("m6 method");
        NStudent n = new NStudent();
        return n;
}
NStudent[] m8(){
```

```
        System.out.println("m6 method");
        NStudent n1 = new NStudent();
        NStudent n2 = new NStudent();
        NStudent n3 = new NStudent();
        NStudent n4[]={n1,n2,n3};
        //return n4;
        //return new NStudent[]{n1,n2,n3};
return new NStudent[]{new NStudent(),new NStudent(),new NStudent()};
    }
    public static void main(String[] args) {
        m2();
        //System.out.println(m2());
        MethodDemo md = new MethodDemo();
        md.m1();
        //System.out.println(md.m1());
        System.out.println(md.m3());
        System.out.println(md.m4());
        System.out.println(md.m5());
        System.out.println(md.m6());
        int a[] = md.m6();
        for(int i=0;i<a.length;i++){
            System.out.println(a[i]);
        }
        System.out.println(md.m7());
```

```
        NStudent ns = md.m7();
        System.out.println(ns.name);
        NStudent[] ns1 = md.m8();
        for(int i=0;i<ns1.length;i++){
            System.out.println(ns1[i].name);
        }
    }
}
```

Based on Parameter:

a. zero argument method/non-parameterized method.

A method, which doesn't have any parameters in its parameter place is called non-parameterized method.

Ex:

```
    void m1(    ){
    }
    int m2(   ){
        return 111;
    }
```

b. Parameterized method:

A method, which contains parameters is called parameterized method.

The parameters may be

a. primitive type

b. primitive array type/reference

```
		c. referenced  type
		d. referenced array type.
	void m1(int x){
	}
	void m2(int x[]){
	}
	void m3(Student s){
	}
	void m4(Student s[]){
	}
import java.util.Scanner;
class MMM{
	int x=999;
}
class PANP{
	void m1(    ){
		System.out.println("non-parameterized m1() method");
	}
	void m2(int x){
		System.out.println("param m2 method");
		System.out.println(x);
	}
	void m3(String x){
		System.out.println("param m3 method");
```

```
            System.out.println(x);
        }
        void m4(String x[]){
            System.out.println("param m4 method");
            for(int i=0;i<x.length;i++){
                System.out.println(x[i]);
            }
        }
        void m5(int x,String y){
            System.out.println("param m5 method");
            System.out.println(x);
            System.out.println(y);
        }
        void m6(MMM obj){
            System.out.println("m6 method");
            System.out.println(obj);
            System.out.println(obj.x);
        }
}
public class Demo {
        public static void main(String[] args) {
            PANP obj = new PANP();
            MMM obj1 = new MMM ();
            obj.m6(obj1);
```

```
        obj.m6(new MMM());
        obj.m1();
        obj.m2(111);
        obj.m3("abhinava");
        String s[] = {"ram","java","php","oracle"};
        obj.m4(s);
        String s1[] = new String[]{"cobol","naresh i","techonologies"};
        obj.m4(s1);
        obj.m4(new String[]{"maths","english","telugu"});
        String s2[] = new String[3];
        Scanner scan = new Scanner(System.in);
        System.out.println("enter some string values");
        for(int i=0;i<s2.length;i++){
            s2[i]= scan.next();
        }
        obj.m4(s2);
        obj.m5(111,"basha");
    }
}
```

To call the method we need to follows the following things.

- **AccessModifier**
- **Static or non-static**
- **method name**
- **parameters(number of and type of)**
- **throws keyword**

```
class Test{
    private void m5(boolean b){
        System.out.println("boolean-argument method");
        System.out.println("b: "+b);
    }
    public void m6(){
        System.out.println("zero-argument method");
        m5(true);
    }
}
class MethodDemo{
    public static void m1(){
        System.out.println("zero argument method");
    }
    public void m2(int x){
```

```
        System.out.println("int-argument method");
        System.out.println("x: "+x);
    }
    public static void m3(int x,String y){
    System.out.println("int-String-argument method");
        System.out.println("x: "+x);
        System.out.println("y: "+y);
    }
    public static void main(String[] s){
        m1();
        MethodDemo md = new MethodDemo();
        //md.m2();
        md.m2(123);
        m3(456,"Annie");
        Test t = new Test();
        //t.m5(true);
        t.m6();
    }
}
```

In the above program Test class m5() is private, so we can't access from the outside of the

class(MethodDemo). If we want to access that m5() method we need to take the support of non-private method.

Abstract Method:

A method, which does not having body is called abstract method.

Abstract method is always ended with semicolon (;).

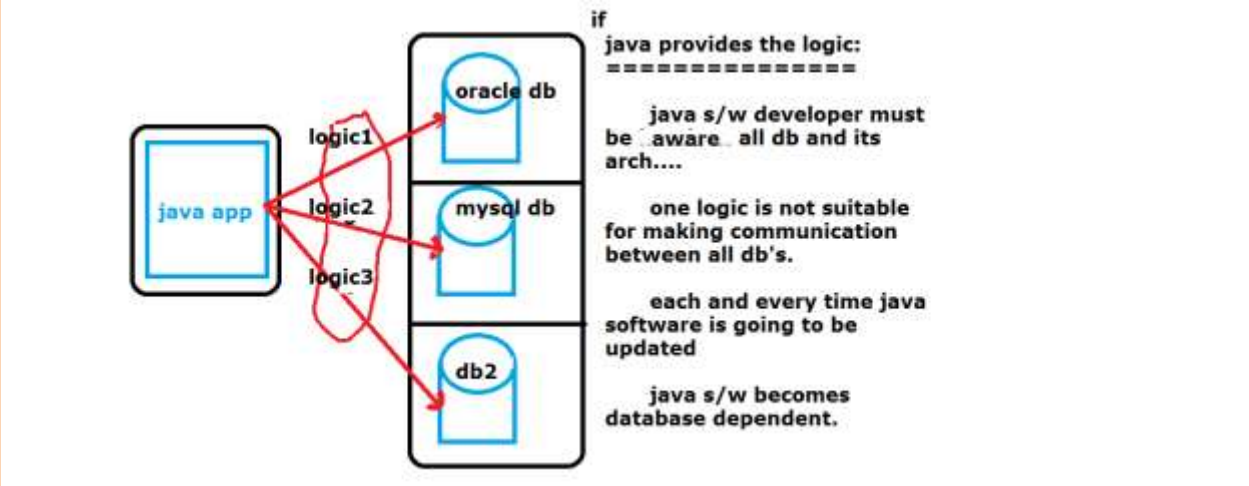
Abstract method must be needs "abstract" keyword in its declaration.

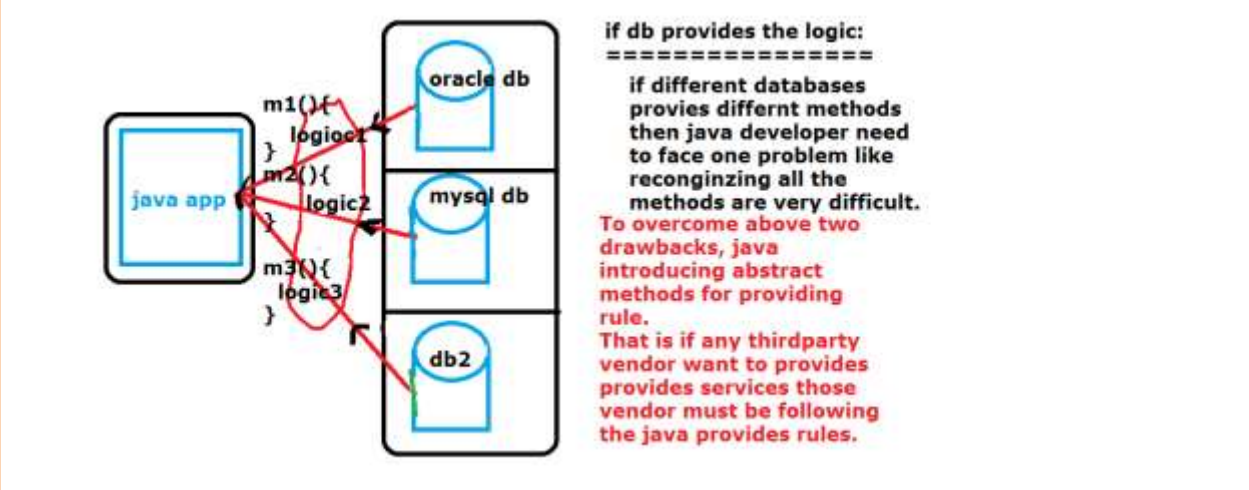
Abstract method always provides specification/rule.

If we want to use those methods we need to provide implementation for specification.

Java provides rules/specifications in the form of interfaces with the support of abstract methods(a method which doesnt have body or ended with semicolon), for these rules databases provides logics/implementation in the form of classes with the support concrete methods(the method which have body or logic)

Ex: abstract void m1 ();





In our java application if we are using db provided terminology our java application strictly interact with only one db server at a time, if we want to communicating with another db we must and should change the logic.

Concrete method:

A method which has body is called concrete method.

Ex:

```
void m1(){


}
```

Data Hiding:

The data which is not accessible from outside of the class is called data hiding.

(or)

Doesn't provide any permissions to unauthorized user to interact with our secure data is called Data Hiding.

We can achieve data hiding concept with the help of private access modifier.

```
class A{
        private int i=111;
        private int j = 222;
}
```

Data abstraction:

Hiding unnecessary information from enduser and provides only necessary information is called data abstraction.

Hiding the implementation logic from user to another user is called data abstraction.

We can achieve this data abstraction with the help of abstract keyword.

```
class A{
        private int i;
        public setI(int i){
                this.i = i;
        }
}
```

```
class X{
 psvm(){
   A obj = new A();
    obj.setI(111);
 }
}
```

```
class Y{
    psvm(){
            A obj = new A();
        obj.setI(222);
    }
}
```

In above syntax setI() will hide the both X,Y class logics.

JavaBean: It is a pojo, maintaing some standards.

those are:

1. class must be public.
2. class must implements java.io.Serializable or java.io.Externalizable.
3. public zero argument constructor
4. Properties must be private
5. public setters and getters

POJO: is a normal java class.

note: Ever javabean is pojo but pojo is not a java bean.

Example on Java Bean:

```
public class Account implements java.io.Serializable/Externalizable{
    public Account(){
    }

    private long accountNo;
    private short pin;


    public void setAccountNo(long accountNo){
        this.accountNo = accountNo;
    }

    public long getAccountNo(){
        return accountNo;
    }
        // setters and getters for pin also
}
```

Encapsulation:

Combination of providing services and security is called encapsulation.

Combination of private variable and public methods is called encapsulation.

Wrapping of data and its related methods is called encapsulation.

Placing the object information into a class is called encapsulation.

Hiding the information with the help of private modifier and accessing that hiding information with the help of public methods from outside of the program/class through permissions is called encapsulation.

To develop encapsulation we required CLASS.

Private data we can access outside of the class by using public method and reflection api.

```
class Bank{
	private long accountNo;
	private String accountHoldName;
	private double amount;
	public void setAccountNo(long accountNo) {
		this.accountNo = accountNo;
	}
	public long getAccountNo() {
		return accountNo;
	}
	public void setAccountHoldName(String accountHoldName) {
		this.accountHoldName = accountHoldName;
	}
	public String getAccountHoldName() {
		return accountHoldName;
	}
	public void setAmount(double amount) {
		this.amount = amount;
	}
	public double getAmount() {
		return amount;
	}
```

```
}
public class EncapsulationDemo {
	public static void main(String[] args) {
		Bank obj = new Bank();
		obj.setAccountNo(35568987);
		obj.setAccountHoldName("Srinivas");
		obj.setAmount(5000);
		long accountNo = obj.getAccountNo();
		String accountHoldName = obj.getAccountHoldName();
		double amount = obj.getAmount();
	System.out.println(accountNo+"   "+accountHoldName+"   "+amount);
	}
}
package oops;
import java.util.Scanner;
class Account{
	private long accountNo;
	private String accountHolderName;
	private short pin;
	private double amount;
	public long getAccountNo() {
		return accountNo;
	}
	public void setAccountNo(long accountNo) {
		this.accountNo = accountNo;
```

```
	}
	public String getAccountHolderName() {
		return accountHolderName;
	}
	public void setAccountHolderName(String accountHolderName) {
		this.accountHolderName = accountHolderName;
	}
	public short getPin() {
		return pin;
	}
	public void setPin(short pin) {
		this.pin = pin;
	}
	public double getAmount() {
		return amount;
	}
	public void setAmount(double amount) {
		this.amount = amount;
	}
}
public class Encapsulation {
	static Scanner scan = new Scanner(System.in );
	public static void main(String[] args) {
		System.out.println("WELCOME TO SBI BANK");
```

```
System.out.println("enter your  pin to interact with your data");
short pin = scan.nextShort();
Account acc = new Account();
if(pin == 5555){
    System.out.println("you are valid user ");
    acc.setAccountHolderName("RAMCHANDRA");
    System.out.println("what type of service do you want?");
    System.out.println("1.withdrawl\t 2.deposit");
    System.out.println("3.balance\t 4.accholdername");
System.out.println("do want to continue if yes click on Y if No click N");
    String service = scan.next();
    boolean process = true;
    while(process){
    if(service.contains("y")){
        System.out.println("choose ur service number");
        byte serviceType = scan.nextByte();
        switch(serviceType){
case 1: System.out.println("you are choosing withdrawl enter your
amount:");
            double amount = scan.nextDouble();
            if(acc.getAmount() >= amount){
             double amount1 = acc.getAmount()-amount;
           System.out.println("your balance is:"+amount1);
             acc.setAmount(amount1);
            }
```

```
                    else
                System.out.println("you dont have sufficient amount");
                    break;
case 2: System.out.println("you are choosing deposit enter your amount:");
                    double amount1 = scan.nextDouble();
                    double amount2 = acc.getAmount()+amount1;
                    acc.setAmount(amount2);
                System.out.println("your balance is: "+acc.getAmount());
                    break;
                case 3: System.out.println("you are choosing balance");
                System.out.println("your balance is: "+acc.getAmount());
                     break;
        case 4: System.out.println("you are choosing account holdername: ");
                    System.out.println("Account Holder name is:
                "+acc.getAccountHolderName());
                break;
                    }
                }
                else {
                    System.out.println("THANK YOU FOR VISITING SBI
SITE");
                }
                    System.out.println("do you want to continue if yes
click 'y' or if no click 'n'");
                    String ss = scan.next();
```

```
                        if(ss.contains("y"))
                                process=true;
                        else
                                process=false;
                    }
                }
                System.out.println("THANK YOU FOR VISITING SBI SITE");
        }
}
class A{
public static void main(String...ram){
        B obj = new B();
        System.out.println(obj.a);
}
}
```

FileName: A.java

Steps to load the B's class information:

--> Compiler will check B .class information in A.java file. If not available

--> Compiler will check B. class in current working directory. If not available

-->compiler will check B.java file in current working directory. If not available compiler will one error, that is cannot find symbol class B.

If B.java file is available in current working directly then B.java file converted into B. class file then that byte code will be used in A.java file..

If B. class file is available, then compiler will check B.java file available or not. If B.java file is not available only B. class available then compiler use B. class file information in A.java file.

If both B. class and B.java file are available then compiler will check both file creation time. If B. class file creation time is more than B.java file then compiler blindly use B. class file.

Otherwise means B.java file creation time is more than the B. class file creation time then compiler first compile B.java file and updated the existed B. class with new B. class file information, later compiler use updated B. class file information.

Singleton Design pattern:

To stop to create multiple object for a particular class is called singleton design pattern.

With the help of singleton design patter we can save the memory.

Ex: Servlet.

```
class Bank{
        private static Bank b = new Bank();
        private Bank(){
        System.out.println("Bank constructor");
        }
        public static Bank getObject(){
            return b;
        }
}
public class SingletonDemo {

    public static void main(String[] args) {
        //Bank b = new Bank();
        Bank obj = Bank.getObject();
```

```
            System.out.println(obj);
            Bank obj1 = Bank.getObject();
            System.out.println(obj1);
            Bank obj2 = Bank.getObject();
            System.out.println(obj2);
        }

}
class Account{
        Account(int x){
            System.out.println("account const");
            System.out.println(x);
        }
}
class Deposit extends Account{
        Deposit(){
            super(100);
            System.out.println("Deposit const");
        }
}
public class EncapsulationDemo {
        public static void main(String[] args) {
            Deposit d = new Deposit();
        }
```

```
}
```

Note:

If super class having parameterized constructor, programmer must be write one constructor in subclass, and programmer must be write super (---) with parameter in that constructor.

Each every class is always holds its own constructor not other class constructor.

```
class A{
   A(){
   }
    /*
        B (){//invalid
       }
   */
}
class B{
}
```

Factory Method:

A method which returns same class object or some other class object is called factory method.

Factory methods are two types

1. static factory method
2. non-static factory method

```
package constructor;
```

```
class Parent{
	String parentName = "KrishnaRao";
}
class Child{
	String childName ="RamaChandraRao";
	static Child getChild(){
		return new Child();
	}
	Parent getParent(){
		return new Parent();
	}
}
public class FactoryMethodDemo {
	public static void main(String[] args) {
		Child child = Child.getChild();
		Parent parent =  child.getParent();
		System.out.println(child);
		System.out.println(child.childName);
		System.out.println(parent);
	System.out.println(parent.parentName);
	}
}
```

Interface:

A java element which provides 100% abstraction is called interface.

or

If we don't know any implementation then we can go for interface.

Ex:

```
interface I{
        public abstract void m1();
        public abstract void m2();
        public abstract void m3();
}
```

Abstract Class:

A java element which provides some part of abstraction is called abstract class.

If we know some part of implementation then we can go for abstract class.

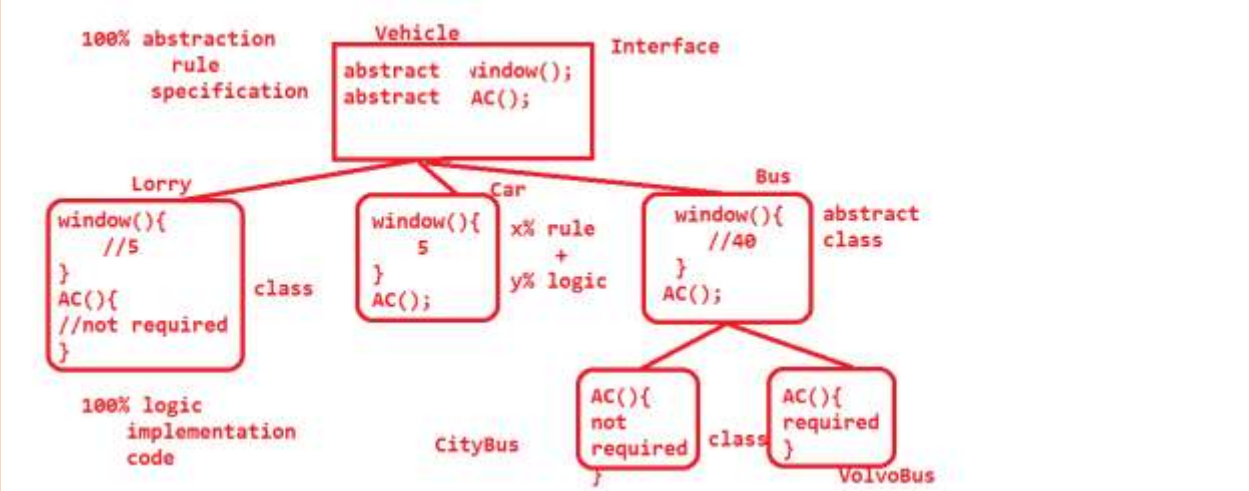
Ex:

```
abstract class AC{
        public abstract void m1();
        public abstract void m2();
        public  void m3(){
            ----------//logic
        }
}
```

Class:

A java element which provides 100% implementation is called class.

```
class Student{
        public  void m1(){
                ---------//logic
        }
        public  void m2(){
                ---------//logic
        }
        public  void m3(){
                ---------//logic
        }
}
```



default, private, public , protected, abstract, synchronized, final, static, strictfp, native.

The above are applied on methods

Bellow keywords are not applied on methods.

volatile

transient

Parameter:

A variable which is available at method parenthesis to hold value is called parameter.

Argument:

The value which is send to parameter is called argument.

```
class A{
    static void m1(int x){
            //x is a parameter
    }
    psvm(-){
        A obj = new A();
        obj.m1(111);//111 is a argument
    }
```

Varargs:

Var args stands for variable argument.

It is used to hold zero or one or more values.

It is introduced in java 1.5 version

It can be represents with the help of "..." (3dot operators)

Don't use more than 3 dots.

Don't use less than 3 dots.

These 3 dots, we can be called as elipse.

Don't give any space between dots.

Valid syntax:

String ... x;

String… x;

String …x;

Invalid syntax:

String . . . x;

String . . . . x;

String .. x;

String . ..x;

String.. .x;

String x...;

...String x;

If we are using the arrays, we can't be hold values more than it size;

int a[] = new int[5];

In the above syntax we can assign only 5 values, not more than 5 values.

If we are assigns less than 5 values, we have memory usage problem.

To overcome above two problems we can go for varargs.

Varargs must be use last parameter of a method, otherwise we will get compile time exception.

```
public class VarArgsDemo {
    static void m1(int []a){
    System.out.println("int array parameter m1 method");
        for(int i=0;i<a.length;i++){
            System.out.println(a[i]);
        }
    }
```

```
static void m2(int ... a){
	System.out.println("----------------");
	System.out.println("m2 method");
	for(int i=0;i<a.length;i++){
		System.out.println(a[i]);
	}
}
static void m3(int x, int ...y){
	System.out.println("-------------");
	System.out.println(x);
	System.out.println("-------------");
	for(int i=0;i<y.length;i++){
		System.out.println(y[i]);
	}
}
/*static void m1(int...a){
}*/
static void m1(int a){
	System.out.println("int prameter m1 method");
	System.out.println(a);
}
/*static void m4(int...a,int x){


}*/
```

```
	/*static void m5(int a,int...b,int c){


	}*/
	static void m1(){
		System.out.println(" zero argument m1 method");
	}
	public static void main(String[] args) {
		m1();
		m1(100);
		//m1(200)
		m1(new int[]{11,22,33});
		m2();
		m2(111);
		m2(222,333);
		m2(444,555,666,777,888,999);
		m3(2323);
		m3(444,555);
		m3(666,777,888);
		//m4();
		//m4(111);
		m4(111,222);
	}
}
class Demo{
```

```
    void m1(int... x){
        System.out.println("m1-var args");
        for(int i=0;i<x.length;i++)
        System.out.println(x[i]);
    }
    void m1(int x, int...y){
        System.out.println("m1-int-varargs");
    }

    /*void m1(int x, int...y,int z){
        //var args variable must be last parameter of method
    }*/

    public static void main(String...r){
        System.out.println("main");
        Demo d = new Demo();
        //d.m1(10); //ambiguity
        //d.m1(10,20);
        d.m1(10,new int[]{11,22,33});
    }

}
```

Interface:

If we don't know any implementation or if we want to provide 100% abstraction then we can go for interface.

Interface is the communication channel between service provider and service consumer.

Interface is the combination of public static final variables and public abstract methods.

If we want to develop interface concept in java, we need to use one predefine keyword that is "interface". This is always followed by Interface_Name.

The naming conventions of interface is same as Class_Name.

All the variables are of interface must be initialized; otherwise we will get compile time error.

By default all the variables are public static final.

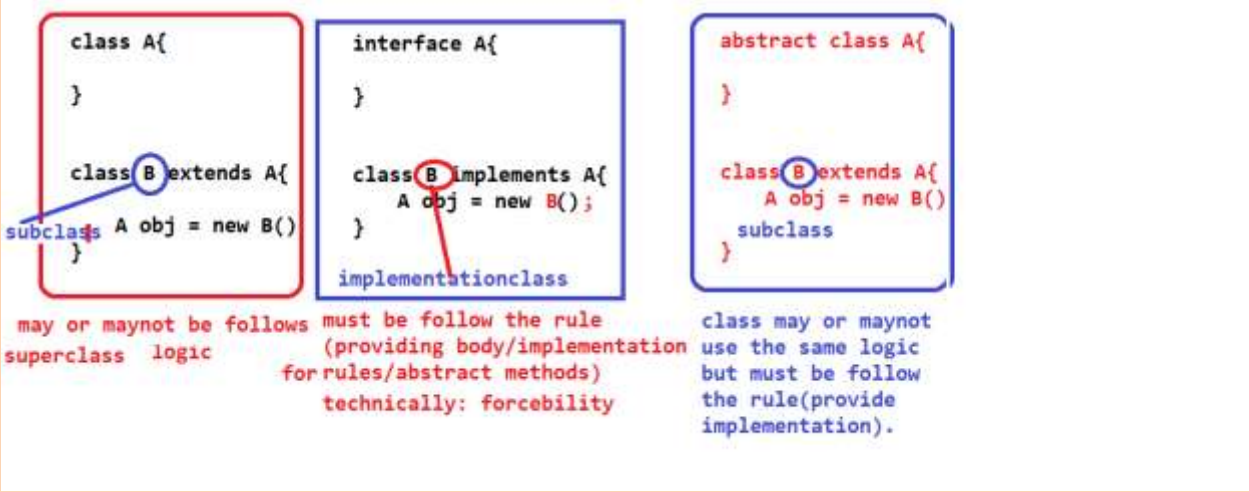
By default all the methods are public abstract.

static and abstract are invalid combination.

final and abstract are invalid combination. Interface provides commonality, forcibility.

If any class, abstract class want to communicate with interface we need one special keyword that is "implements" keyword.

We can write "abstract" keyword in front of interface keyword.



```
interface I{

}
```

```
Javac I.java

Javap  I
```

```
interface I{

}
```

interface is by default single programming element, that means by default interface never communicating with other interface or classes.

```
interface I{
        int i=111;
        void m1();
}
```

```
javac I.java

javap I
```

```
interface I{
        public static final int a =111;
        public abstract void m1();
}
```

```
interface I{
        int i=111;
        final abstract void m1();//invalid
        static abstract void m2();//invalid
}
```

Based on, the methods of interface can be classified as three types from java 1.8 onwards.

a. Marker interface

b. Functional interface

c. Normal interface.

Marker interface:

An interface which doesn't have any method is called marker interface.

Ex: java.io.Serializable, java.lang.Cloneable, java.util.RandomAccess.

If any class, which is implements interface that class must be provide implementation for all the methods of interface. It is comes under forcibility.

If any class, which implements marker interface that class is treated as a special class by the JVM.

Functional Interface:

An interface, which contains only one method, is called Functional interface.

If we want to mention our interface as a functional interface, we should use one java 1.7 features, that @FunctionalInterface annotation.

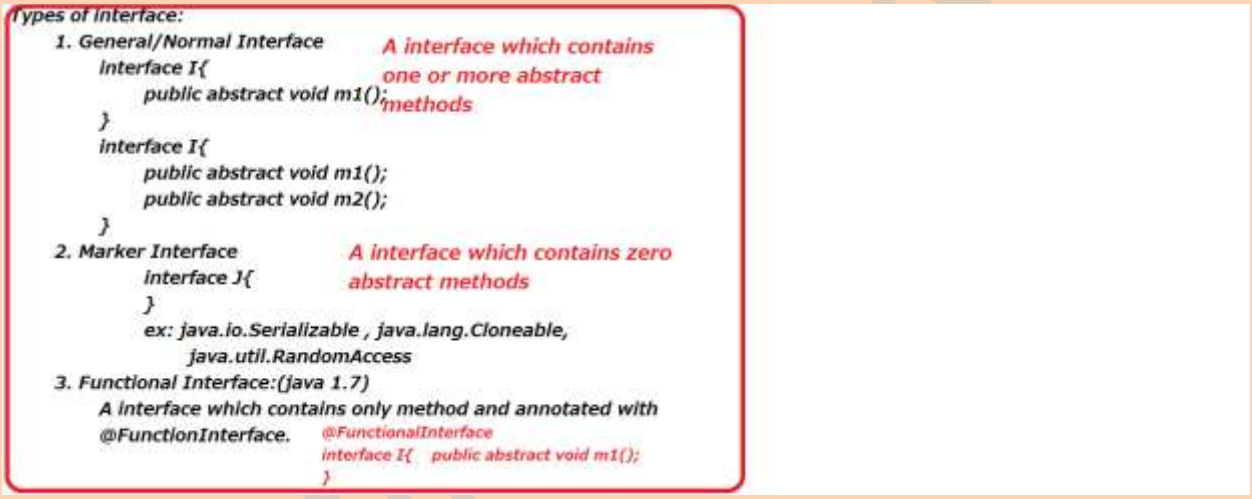
Ex:

```
@FunctionalInterface
interface I{
        public abstract void m1();
}
```

Normal interface:

An interface which contains more than one method or only one method without @FunctionalInterface annotation is called normal interface.

```
interface I{

        public abstract void m1();

        public abstract void m2();

}

interface J{

        public abstract void m1();

}
```



Note:

We cannot create object for interface, but we can create reference. With the help of reference, we can hold its implementation class.

```
interface J{

        public abstract void m1();

}

class M implements J {

}
```

Here M is called implementation class.

We can write abstract keyword in front of interface keyword.

We can't write static and non-static blocks and constructors.

```
interface J{
        static{//invalid
        }
        {       // invalid
        }
}
interface K{
        K(){//invalid
        }
}
```

There is no super class for interface.

If we are trying to write concrete methods in java we are getting compile time error.

```
package interfaces;
interface I {     //extends java.lang.Object{
        int a=111;
        public abstract void m1();
        void m2();
        /*void m3(){
                System.out.println("concrete methods");
        }
        {
```

```
            System.out.println("non-static block");
        }
        static{
            System.out.println("static block");
        }
        I(){
            System.out.println("constructor");
        }*/
}
class J implements I{
        public void m1(){
            System.out.println("J class m1 method");
        }
        public void m2(){
            System.out.println("J class m2 method");
        }
}
public class InterfaceDemo {
        public static void main(String[] args) {
            //I obj = new I();
            /*I obj ;
            obj = new J();*/
            I obj = new J();
            obj.m1();//synatax (1)
```

```
            obj.m2();//synatax (2)
        }
}
```

In above program syntax--1 compiler first check the type of obj, here type of obj is interface I. so first compiler will goes to I and check whether m1 method is available or not, if not compile time error, if yes m1() will bind to obj.

Then JVM will having the capability to check value.

JVM checks value, here value is J class memory(new J()).

So JVM executes the m1 method from J class.

Syntax--2 executions is also same as syntax---1.

interface: 100% abstraction specifications (rules) all methods of interface must be abstract.(upto 1.7)

interface is communication channel between service provider and utilizer.
*)No super class for interface
*)we cannot write constructor,static and non static block and concrete methods
*)by default variables are public static final and intialized.
*)by default methods are public abstract
*)we cannot create object
*)we can create reference to hold its implementation class memory.
*) we can write abstract keyword infront of the class.
*) we can not abstract static,abstract private, abstract final methods.
*)we can extract the one interface functionalities to another

Abstract Class:

If we want to provide some part of implementation or some part of abstraction then we can go for abstract class.

In the abstract class we can write both abstract and concrete method.

Class should be mention with "abstract" keyword.

Abstract class provides both forcibility and reusability.

Ex:

```
abstract class AC{
    abstract void m1();
    void m2(){
    }
}
```

We can't create object for abstract class

```
AC obj = new AC();//invalid syntax:
```

But we can create reference.

```
AC obj;
```

With the help of abstract class reference, we can hold sub class implementation logic/memory.

If we want use functionalities of interface we need implementation class, in same manner if we want use the abstract class functionalities we need sub class.

The class which is extends abstract class is called sub class.

Ex:

```
class SAC extends AC{
    void m1(){
    }
    void m2(){
    }
}
```

Note:

If any class which implements interface or extends abstract class, we should provide the body for all the method of interface and abstract class in implementation or sub class.

```
Interface I{
        public abstract void m1();
        public abstract void m2();
}
abstract class AC implements I{
        public  void m1(){}
        public abstract void m2();
        abstract void m3();
}
class SAC extends AC implements I{
        public void m1(){
        }
        public void m2(){
        }
        void m3(){
        }
}
```

With in the abstract class we can write constructors.

With in the abstract class we can write both static and non-static blocks.

```
abstract class AC{
        AC(){
                S.o.p("constructor");
```

```
        }
        {
                S.o.p("non-static block");
        }
        static{
                S.o.p("static block");
        }
}
```

Abstract class is also subclass of java.lang.Object class.

```
abstract class AC{
}
```

javac AC.java

After compilation compiler will convert in the following manner.

javap AC

```
abstract class AC extends java.lang.Object{
        AC(){
                super();
        }
}
```

Q) Why abstract class have constructor, why not interface?

interface doesn't need any predefine functionalities from java.lang.Object class, so need of forward the control from interface to Object class.

If we are not forwarding the control no need of constructor.

But whereas abstract class need some predefine functionalities of Object class, so we need to forward our control from abstract class to Object class.

If we want to forward control from abstract class to Object class we need "super ()".

This "super ()" must be write in the constructor.

So for forwarding the control purpose we need constructor in abstract class.

Q) Why abstract class have static block why not interface?

A) All the variables of interface by default static final, so if we want to declare static final variables in interface level we need to initialize the static variables first. So already static variables are initializes by the programmer, so no need of using the static block again in interface.

But whereas static variables in abstract class may or may not be initialized. If not initializes, we can use static block for static variables initialization purpose.

Q) Why abstract class have non-static block why not interface?

A) Non-static blocks are used to initialize the non-static variables. But in interface every variable is static so no need of non-static block.

Whereas in abstract class, either we can write static and non-static variable, so if we want initializes the non-static variable, so we need non-static block.

Q) Difference between interface and abstract class?

Interface:

1. If we don't know any implementation then we can go for interface.
2. Interface provides 100% abstraction.
3. We cannot write protected, final, static, private, strictfp, native methods.
4. Every variable must public static final variables.
5. Interface variables don't have private, protected, transient, volatile keywords.

6. Interface variables must be initializes.

7. In the interface we can't write static, non-static blocks and constructors.

8. No super class for interfaces.

Abstract Class:

1. If we know some part of implementation then we go for abstract class.

2. Abstract class provides some part (percentage) of abstraction based on abstract methods.

3. We can write any type of method.

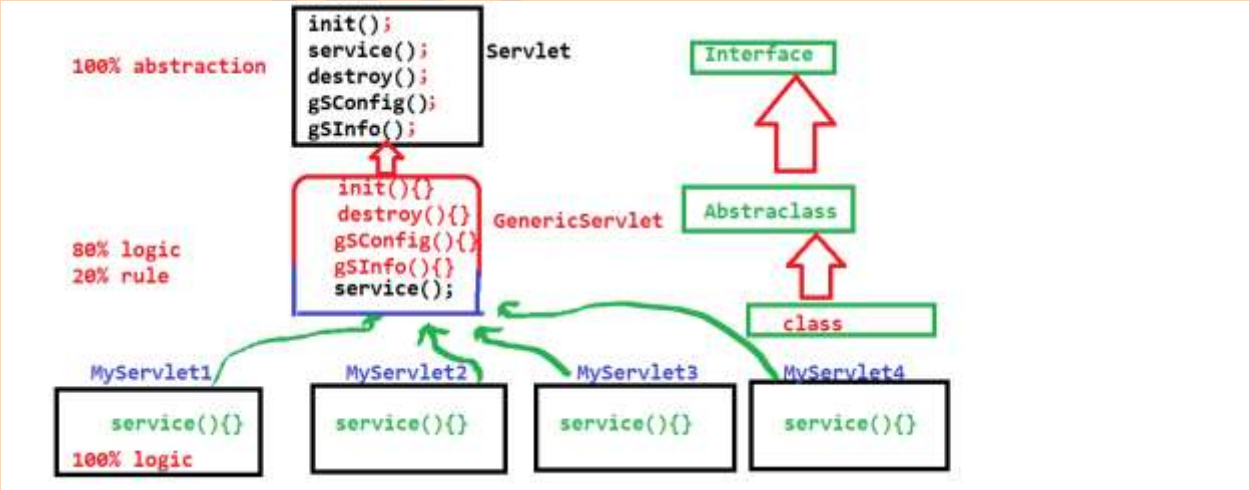
4. No need of public static final variable, if we want we can write.

5. We can write private, protected, transient, volatile keyword before variable names.

6. No need of initialization

7. We can write static, non-static blocks and constructor.

8. Abstract class is the sub class of java.lang.Object.



static block usage:

static block are used to initialize the static variables.

static block are used to registering the Driver class with DriverManager.

static block are used to provide common information.

Note: Abstract classes no need of having abstract methods.

abstract keyword is used to represent the non-body methods.

abstract keyword is used to provides some restrictions to end user to create an object for class.

```
abstract class AC{
        void m1(){
        }
        void m2(){
        }
}
```

If we want to use functionalities of abstract class we need subclass.

If any class which is not implements all the abstract methods of interface, we should be mentioning that class as an abstract class.

```
interface I{
        public abstract void m1();
        public abstract void m2();
}
```

Invalid syntax:

```
class A implements I{
        public void m2(){
        }
```

```
        }
```

Valid syntax:

```
        abstract class A implements I{
                public void m2(){
                }
                public abstract void m1();
        }
abstract class  MM extends java.lang.Object{
        MM(){
           super();
           System.out.println("MM class constructor");
        }
        void m1(){
           System.out.println("MM class m1 method");
        }
        void m2() {
           System.out.println("MM class m2 method");
        }
        abstract void m3();
}
class NN extends MM{
        NN(){
                super();
                System.out.println("NN constructor");
```

```
	}
	void m3(){
		System.out.println("NN class m3 method");
	}
}
public class AbstractDemo {
	public static void main(String[] args) {
		//MM obj = new MM();
		MM obj;
		obj = new NN();
		obj.m1();
		obj.m2();
		obj.m3();
	}
}
```

Note:

Relation between object and constructor:

Whenever we create object with the help of new keyword, we can execute the constructor.

Without object creation we can execute the constructor in abstract class.

Object is created but not constructor is executed.

B clone(), deserialization.

```
interface MM{
	void m1();//public abstract void m1();
	void m2();//public abstract void m2();
	void m3();//public abstract void m3();
}
abstract class NN implements MM{
	public void m1(){
	   System.out.println("NN class m1 method");
	}
	public void m2(){
	   System.out.println("NN class m2 method");
	}
}
class OO extends NN{
	public void m3(){
	   System.out.println("OO class m3 method");
	}
}
public class AbstractDemo {
	public static void main(String[] args) {
		OO obj = new OO();
		obj.m1();
		obj.m2();
		obj.m3();
```

```
    }
}
```

If any class which is not implements all the abstract methods of interface, we should be mentioning that class as an abstract class.

final means nobody inherit the functionalities.

By default constructors are not inherit, so declare constructor as final meaningless.

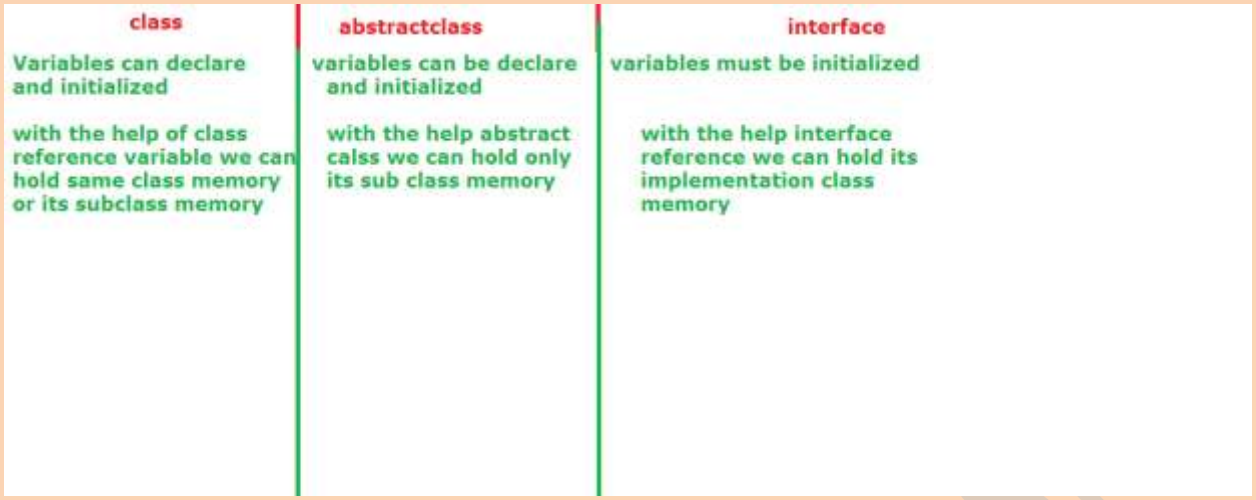
We cannot declare constructor as static.

static means class data, but constructor is purely related to object.

We cannot mention constructor as abstract, for this we have two reasons

The reason is whenever we create an object constructor automatically called so constructor must be having the body and also abstract means no body we should be use inheritance for providing body, but constructors are not participated in inheritance.

class	abstractclass	interface
•used to provides implementation/logic	•used to provides implementation and abstraction	• used to provides rules
•100% logic/implement ation	•some part of logic and some part of rule	•100% abstraction/rule (upto java 1.7)
•we have super classe for class	•we have super classe for abstract class	•we dont have super class
•we can write any type of variables and methods	•we can write any type of variables and methods	•we can write public static final variables and public abstract methods
• we can write blocks,constructors •we can create object •we can create reference • class can communicate with other class,ac,interface	• we can write blocks and constructors • we cannot create object •we can create reference • AC can communicate with class,AC,interface	we can not write block and constructors •we can not create object • we can create reference • interface can communicate with another interfa



With the help of Interface variable we can hold it implementation class memory.

Like bellow

Interface Animal{

}

Class Tiger implements Animal{

}

Animal obj = new Tiger();

Implementation class: Tiger

Interface reference variable: obj.

Abstract class reference variable can hold its sub class memory

Abstract class AC{

}

Class SAC extends AC{

}

AC obj = new SAC();//upcasting

Subclass: SAC

Abtract class reference variable: obj

Class reference variable can hold its own memory or its subclass memory.

```
Class  A{
}
Class B extends A{
}
A obj = new A();
B obj1 = new B();
A obj = new B();//upcasting
```

Q) How can we call constructor of abstract class in java?
A) By creating object of it subclass.

```
abstract class  A{
    A(){
        s.o.p("absract class");
    }
}
class B extends A{
    psvm(-){
        new B();
    }
}
```

Q)Is the following syntax valid or not?

```
abstract class A{
    A(int x){    }
}
class B extends A{
}
```

Answer: NO

Q) how to resolve about problem?

We have two alternatives to resolve the problem.

1) Write default constructor and in B class and super class constructor, with the help of this syntax

```
B(){
    super(111);
}
```

(OR)

2) Write default constuctor in class A.

```
class A{
    A(int x){}
    A(){}
}
```

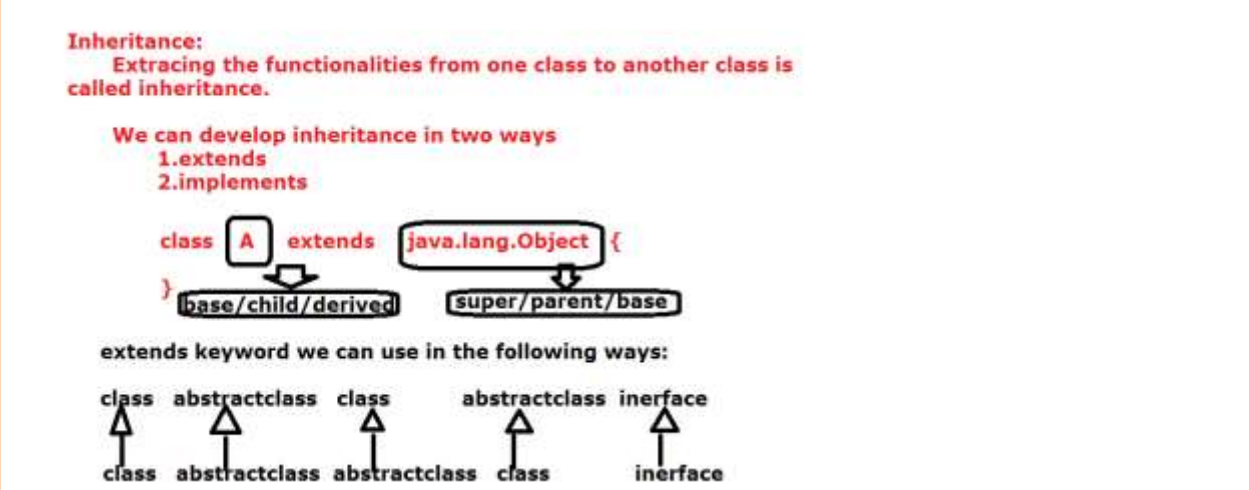
Inheritance:

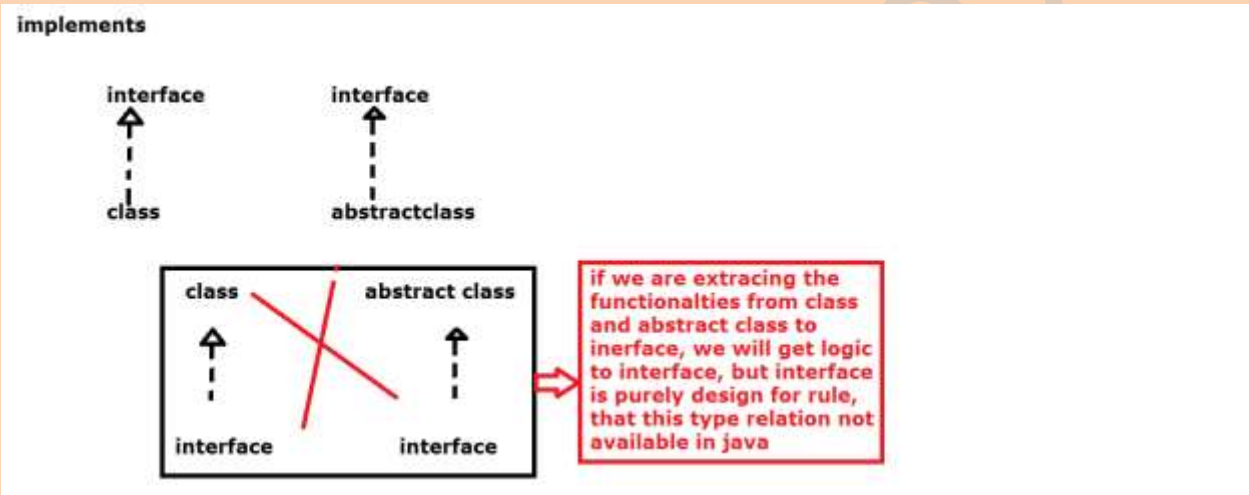
Extracting the functionalities(data and methods) from one class to another class is called inheritance without creating object. (static).

Without inheritance we need to face following problems.

1. Duplicate code/boilerplate of code.
2. Size of the code will be increasing
3. Compilation time will be increases.
4. Low performance
5. Development time will be increases.
6. Project cost will be increases.
7. Productivity will be decreases.

To avoiding above problems java uses one of the object oriented programming system principle that is "INHERITANCE".





What is relation in java?

A: it is communication /process to access the data from one class to anther class.

we have two types of relations.

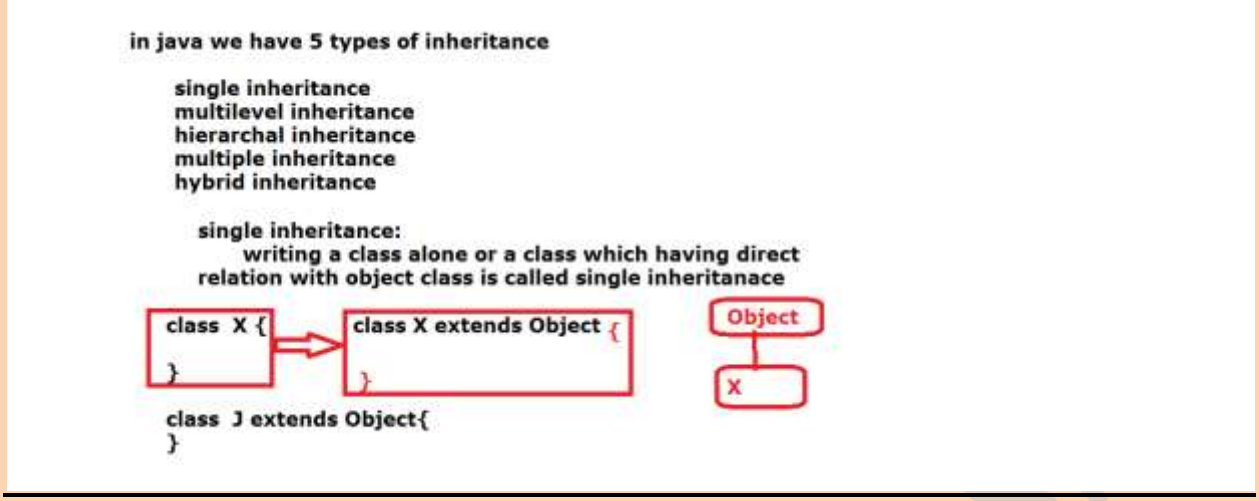
1. Is-a (by using inheritance access the data)

2. Has-a (by using object creation access the data)

```
class A{
    int add (int a, int b){
        return a+b;
    }
```

```
}
class B extends A{  //inheritance (IS-A)
	public static void main(String[] s){
		B obj = new B();
		System.out.println(obj.add(10,20));
	}
}
class C extends java.lang.Object{
	public static void main(String[] s){
		/*C obj = new C();
		System.out.println(obj.add(30,40));
		*/
		A obj = new A(); //HAS-A
		System.out.println(obj.add(30,40));
	}

}
```

In java we have 5 types inheritances.



a. Single Inheritance:

Extending the functionalities from java.lang.Object class is called single level inheritance.

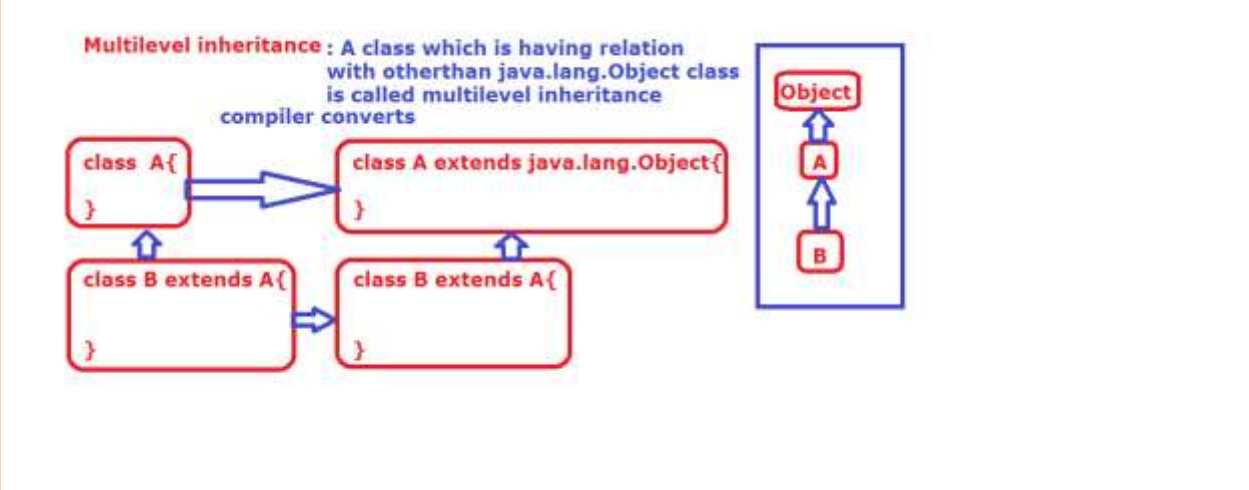
```
class A  extends java.lang.Object{

}

Class B{

}

Javac B.java

Class B extends java.lang.Object{//compiler provides

}
```

b. Multi level inheritance:

Extends the functionalities from other than java.lang.Object class is called multilevel inheritance.

```
class A{

}

class B extends A{
```

}



Note:

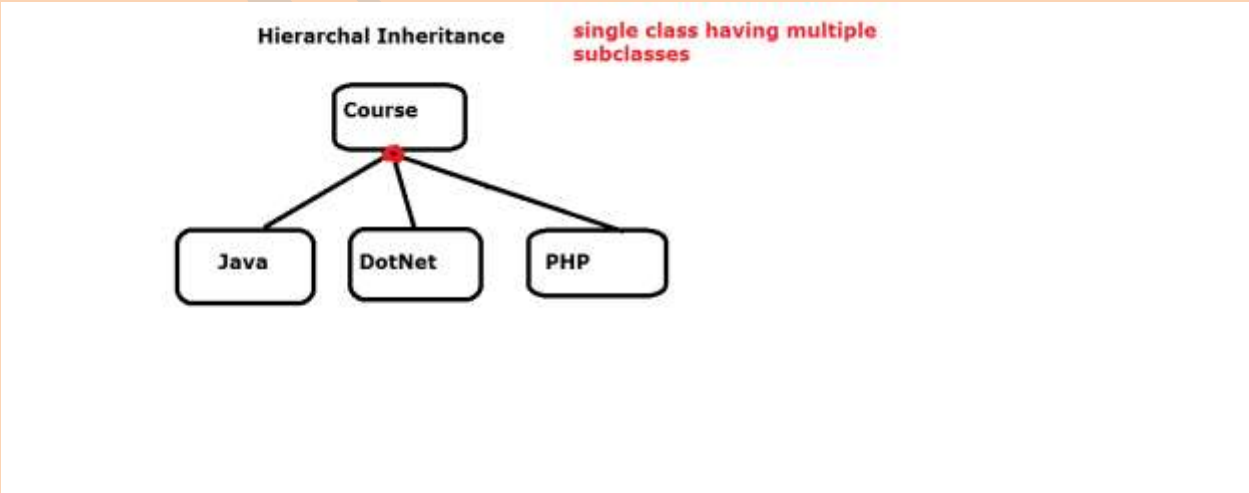
Every class is the subclass of java.lang.Object either directly or indirectly.

c. <u>Hierarchal inheritance:</u>

Derived the one class functionalities into more than

one subclass is called hierarchal inheritance.

```
class A{}
class B extends A{}
class C extends A{}
```



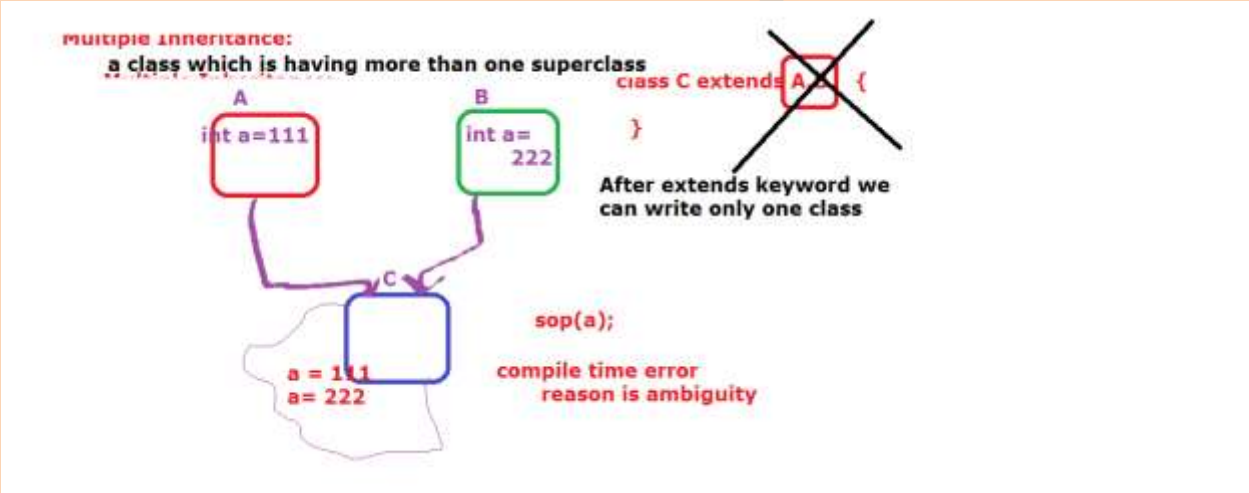
d. <u>Multiple inheritance:</u>

Extracting the functionalities from more than one super class to only one subclass is called multiple.

Java does not support multiple inheritances through classes. The reason is ambiguous problem.

```
class A{

}

class B{

}

class C extends A,B{

}
```

After extends keyword we can't write more than one class name.



e. Hybrid Inheritance:

Combination of all the inheritance is called hybrid inheritance. In hybrid inheritance also, we have multiple inheritance, so we can't implements hybrid inheritance in java.

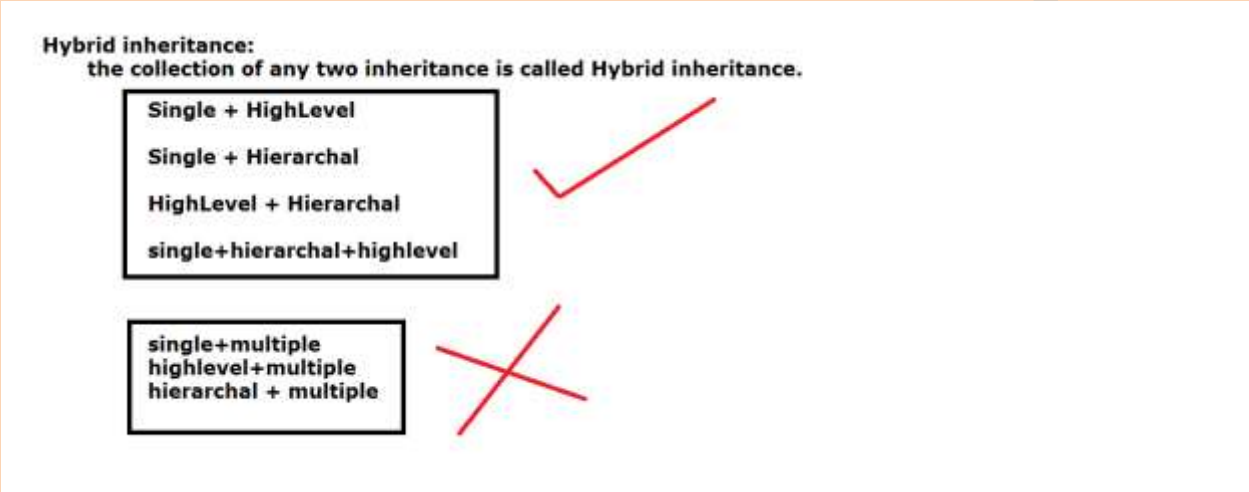
Single Level + Multi Level

Single Level + Hierarchal

Single Level + Hierarchal + Multi Level

The above three combinations are comes under Hybrid. These type of combinations are, we can implement in java.

But if we add multiple inheritances to above combinations, then we cannot implement.



Note:

We can achieve multiple inheritances through interfaces. (only for methods not for variables).

We can develop inheritance with the help of two keywords.

a. extends

b. implements

"extends" can be used in the following scenarios relation between

a. class to class

b. abstract class to abstract class

c. interface to interface

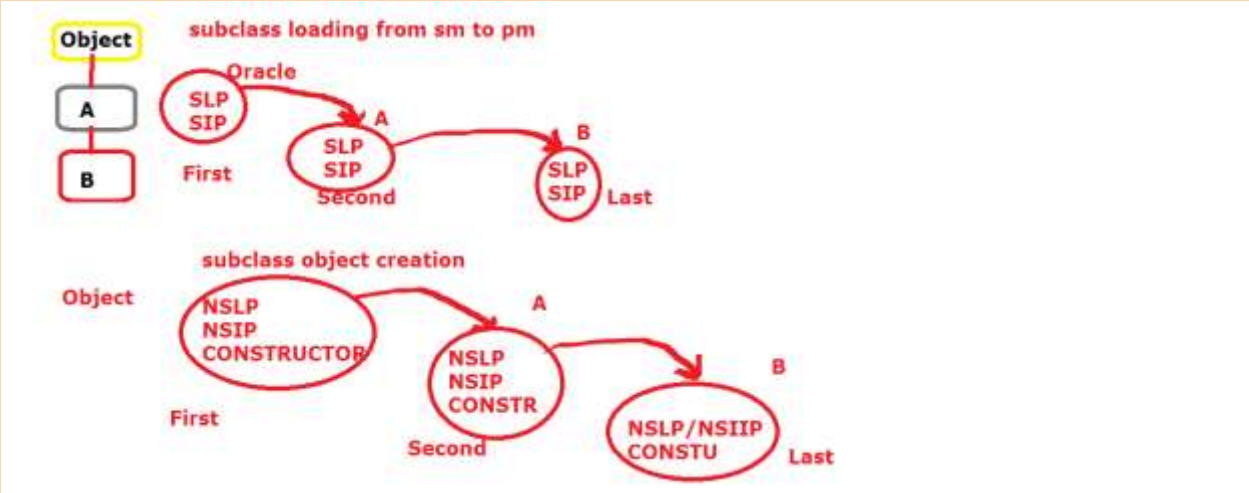
d. class to abstract class

e. abstract class to class

"implements" can be used in the following scenarios relation between

a. class to interface

b. abstract class to interface




```
class A{
        int a = 10;
        A(){
           super();
           System.out.println("A class constructor");
        }
}
class B extends A{
        B(){
          super();
          System.out.println("B class constructor");
        }
        int b = 20;
}
public class InheritanceDemo {
```

```
	public static void main(String[] args) {
		B obj = new B();
		System.out.println(obj.a);
		System.out.println(obj.b);
	}
}
```

abstract class to class:

```
class A{
	void m1(){
		System.out.println("A m1()");
	}
}
abstract class B extends A{
	abstract void m2();
}
class C extends B{
	void m2(){
		System.out.println("C m2() ");
	}
}
public class AbstractDemo1 {
	public static void main(String[] args) {
		C obj = new C();
```

```
            obj.m1();
            obj.m2();
        }
}
```

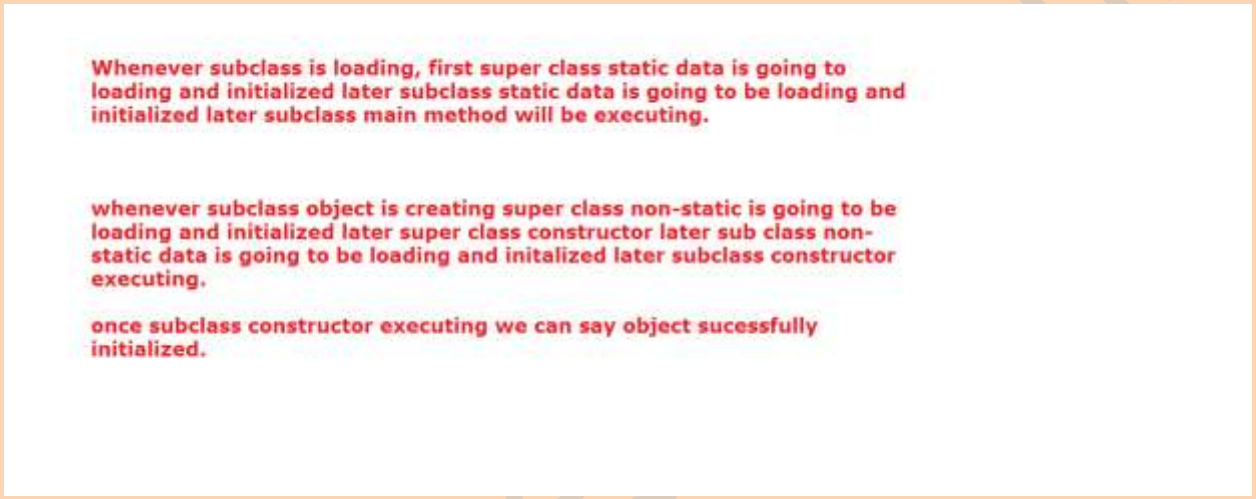
Note: If super class having parameterized constructor, in sub class programmer must be write one constructor, in that constructor we need to write one super(--) with matching argument syntax.

```
package inheritance;
class A{
        A(int x){
            System.out.println("A class constructor");
        }
}
class B extends A{
        B(){
            super(100);
            System.out.println("B class constructor");
        }
}
public class InheritanceDemo {
        public static void main(String[] args) {
            B obj = new B();
            System.out.println(obj);
        }
}
```

Note: When ever Sub class is loading from secondary memory to primary memory.

First super class static variable and static block will be loaded after that sub class static variable and static blocks will be loaded.

When we create subclass object, then first super class non-static variables, non-static block, constructors are loaded after that sub class non-static variables, non-static blocks, constructors are loaded after that JVM will provides memory for subclass only.



Note: If we are loading byte code of any class by using Java command then only main method will be executes, other only Static loading phase and Static initialization phases will be executes from super class to sub class.

```
class A extends java.lang.Object{
        static int a = m1();
        static{
                System.out.println("super class static block");
                System.out.println("---------------------------------");
        }
        static int m1(){
                System.out.println("super class static m1 method");
```

```
		return 111;
	}
	public static void main(String[] s){
		System.out.println("super class main method");
	}
	int b = m2();
	{
		System.out.println("super class non-static block");
		System.out.println("---------------------------------");
	}
	int m2(){
		System.out.println("super class non-static m2 method");
		return 222;
	}
	A(){
		System.out.println("super class constructor");
		System.out.println("---------------------------------");
	}
}
class B extends A{
	static int c = m3();
	static{
		System.out.println("sub class static block");
		System.out.println("---------------------------------");
```

```
	}
	static int m3(){
		System.out.println("sub class static m1 method");
		return 333;
	}
	public static void main(String[] s){
		System.out.println("sub class main method");
		System.out.println("---------------------------------");
		B obj = new B();
	}
	int d = m4();
	{
		System.out.println("sub class non-static block");
		System.out.println("---------------------------------");
	}
	int m4(){
		System.out.println("sub class non-static m4 method");
		return 444;
	}
	B(){
		System.out.println("sub class constructor");
	}
}
class Test extends Object{
```

```
	public static void main(String[] s){
		B.m3();
	}
}
```

If we are executing above program like below

	java B

then only B class main method will be executes

If we executes above program like below

	java Test

in this time B class main method not executes only B class static loading phase and static initialization phases will executes

---

```
class A{
	static int a = m1();
	static int m1(){
	System.out.println(a);
System.out.println("super class  static m1 method");
	return 111;
	}
	static{
	System.out.println("super class static block");
	}
	int b = m2();
```

```
        int m2(){
                System.out.println("------------");
                System.out.println(b);
                System.out.println("super class non-static m2 method");
                return 222;
        }
        {
           System.out.println("super class non-static block");
        }
        A(){
                 System.out.println("super class constructor");
        }
}
class B extends A{
        static int c = m3();
        static int m3(){
                System.out.println("-----------------");
                System.out.println(c);
           System.out.println("sub class static m3 method");
                return 333;
        }
        static{
        System.out.println("sub class static block");
        }
```

```
        int d = m4();
        int m4(){
                System.out.println("--------------------");
                System.out.println(d);
                System.out.println("sub class non-static m4 method");
                return 444;
        }
        B(){
                System.out.println("sub class constructor");
        }
        {
                System.out.println("sub non-static block");
        }
}
public class InheritanceDemo {
        public static void main(String[] args) {
                B obj = new B();
                System.out.println(obj);
        }
}
```

Note:

Whenever we create object for sub class jvm only allocate the memory for subclass not for its super class.

```
class A{
        A(){
```

```
        System.out.println("super class constructor");
        }
}
class B extends A{
        B(){
        System.out.println("sub class constructor");
        System.out.println(this.hashCode() +"..."+super.hashCode());
        System.out.println(this.getClass().getNam()
        +"..."+super.getClass().getName());
        }
}
public class InheritanceDemo {
        public static void main(String[] args)throws InterruptedException {
                B obj= new B();
                Thread.sleep(365*1000*24*7);
                System.out.println("**************");
        }
}
```

Java Visual VM:

It is a tool, which is coming to our system mean while of java software installation.

It is available in C:\Program Files\Java\Jdk\bin\jvisualvm.

With the help of this tool, we can recognize the number of instances created by JVM for our application.

Meanwhile of working with this jvisualvm tool our program must be in the running mode.

Steps to working Jvisualvm tool:

1. Go to C:\Program Files\Java\Jdk\bin folder.
2. Search jvisualvm tool.
3. Double click on jvisualvm

   Note: program must be in running mode.
4. In the left side of tool, select our present running program.
5. Right click on our program
6. Click on open
7. Click on monitor
8. Click on heap dump
9. Click on classes
10. Search our program (type our package name and (.) in the bottom of jvisualvm tool search box)

Example on multilevel and hierarchal inheritance:

```
class A{
	int a = 111;
}
class B extends A{
	int b = 222;
}
class C extends B{
	int c = 333;
```

```
}
class D extends A{
	int d = 444;
}
public class InheritanceDemo {
	public static void main(String[] args){
		C obj= new C();
		System.out.println(obj.a);
		System.out.println(obj.b);
		System.out.println(obj.c);
		D obj1 = new D();
		System.out.println(obj1.d);;
		System.out.println(obj1.a);
	}
}
```

Ex:

```
class MM{
}
class NN extends MM{
}
interface Vehicle{
	public abstract void windows();
	public abstract void ac();
}
```

```
class Car implements Vehicle{
	public  void ac(){
		System.out.println("AC required-car");
	}
	public void windows(){
		System.out.println("No. of windows are: "+6+"-car");
	}
}
class Lorry implements Vehicle{
	public void ac(){
		System.out.println("AC not required=lorry");
	}
	public void windows(){
		System.out.println("No. of windows are: 4"+"-lorry");
	}
}
abstract class Bus implements Vehicle{
	public void windows(){
		System.out.println("No. of windows are: 50-Bus");
	}
	public abstract void ac();
}
class CityBus extends Bus{
	public void ac(){
```

```
            System.out.println("ac not required-citybus");
        }
}
class StateBus extends Bus{
        public void ac(){
            System.out.println("ac required-statebus");
        }
}
class Test{
        public static void main(String[] s){
            Car c = new Car();
            c.ac();
            c.windows();
            Lorry l = new Lorry();
            l.ac();
            l.windows();
            CityBus cb = new CityBus();
            cb.windows();
            cb.ac();
            StateBus sb = new StateBus();
            sb.windows();
            sb.ac();
        }
}
```

1. If any class interact with interface, that class must and should provide body for all abstract methods which are available in interface

2. The above rule is common for all classes, if not follows those classes comes under abstract class.

Create object for subclass:

If we create object for subclass both sub and super class data can be access.

NN obj = new NN();

If we create an object for super class only super class data type can be access.

Up-casting or generalization or widening:

converting different type of objects to common object.

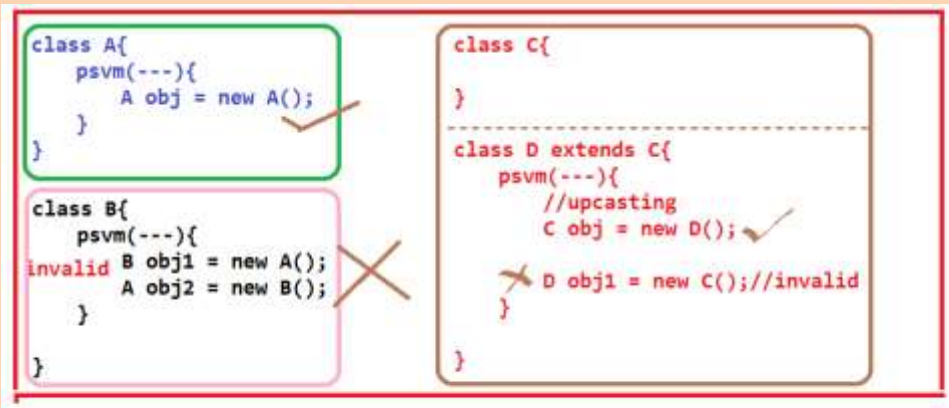
why should go for upcasting?
instead of writing individual logic for individual objects for , we should write single logic for different object and sending data from one place to another palcee.

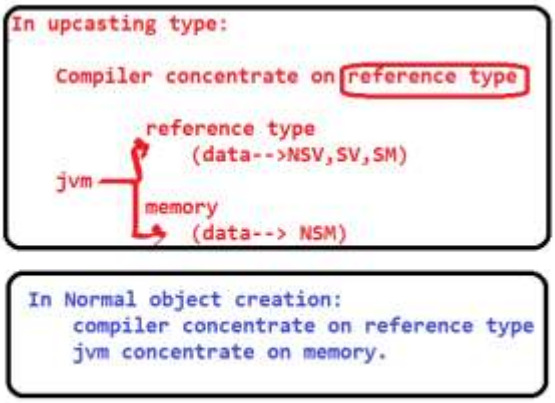
Placing sub class memory into super reference is called up-casting.

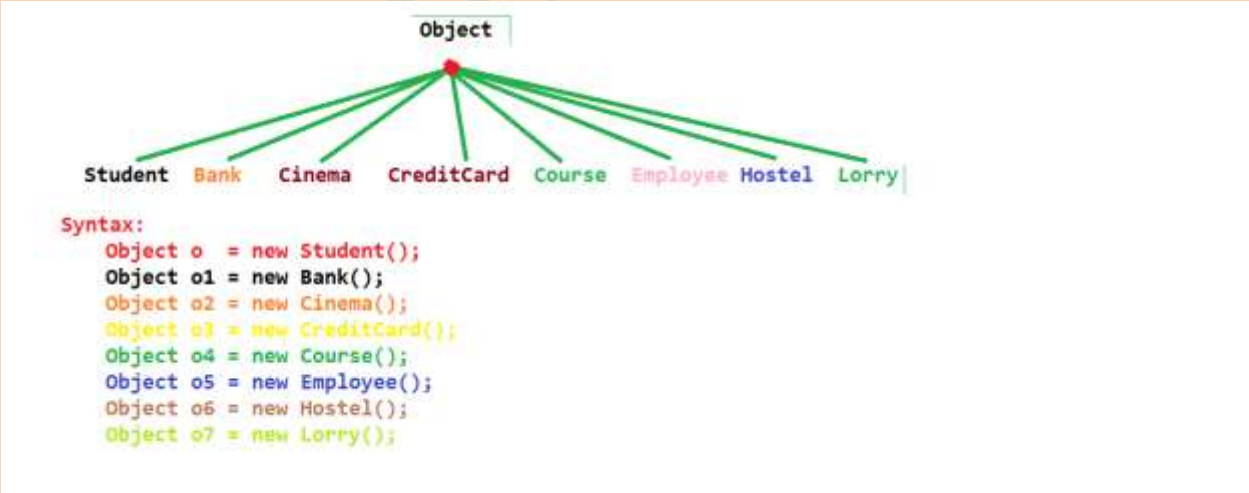
MM obj = new NN ();

In the up-casting procedure, static variable, non-static variable, static methods are executed from reference type (super class and its super class).

Only non-static method are executed from memory (new NN ()--subclass and its super class).


class A{
psvm(---){
A obj = new A();
}
}
class B{
psvm(---){
invalid B obj1 = new A();
A obj2 = new B();
}
}
class C{
}
class D extends C{
psvm(---){
//upcasting
C obj = new D();
D obj1 = new C();//invalid
}
}


In upcasting type:
Compiler concentrate on reference type
reference type
(data-->NSV,SV,SM)
jvm
memory
(data--> NSM)
In Normal object creation:
compiler concentrate on reference type
jvm concentrate on memory.


Object
Student
Bank
Cinema
CreditCard
Course
Employee
Hostel
Lorry
Syntax:
Object o = new Student();
Object o1 = new Bank();
Object o2 = new Cinema();
Object o3 = new CreditCard();
Object o4 = new Course();
Object o5 = new Employee();
Object o6 = new Hostel();
Object o7 = new Lorry();

```
class A{

}
```

```
class B{
}
class C extends A{
}
class UpCastingDemo{
	static public void main(String[] r){
		//A obj = new B();//incompatiable types
		//C obj1 = new A();//incompatiable types
		/*A obj1 = new A();
		C obj2 = (C) obj1;	java.lang.ClassCastException
		*/
		A obj1 = new C();   //Upcasting
		C obj2 = (C)obj1;   // Downcasting
	}
}
class MM{
	static int a = 111;
	int b = 222;
	static void m3(){
		System.out.println("MM m3 method");
	}
	void m4(){
		System.out.println("NN m4 method");
	}
```

```
	void m1(){
		System.out.println("MM m1 method");
	}
}
class NN extends MM{
	static int a = 333;
	int b = 444;
	static void m3(){
		System.out.println("NN m3 method");
	}
	void m4(){
		System.out.println("NN m4 method");
	}
	void m2(){
		System.out.println("NN m2 method");
	}
}
public class InheritanceDemo {
	public static void main(String[] args){
		NN obj = new NN();
		obj.m1();
		obj.m2();
		System.out.println("-----------");
		MM obj1 = new MM();
```

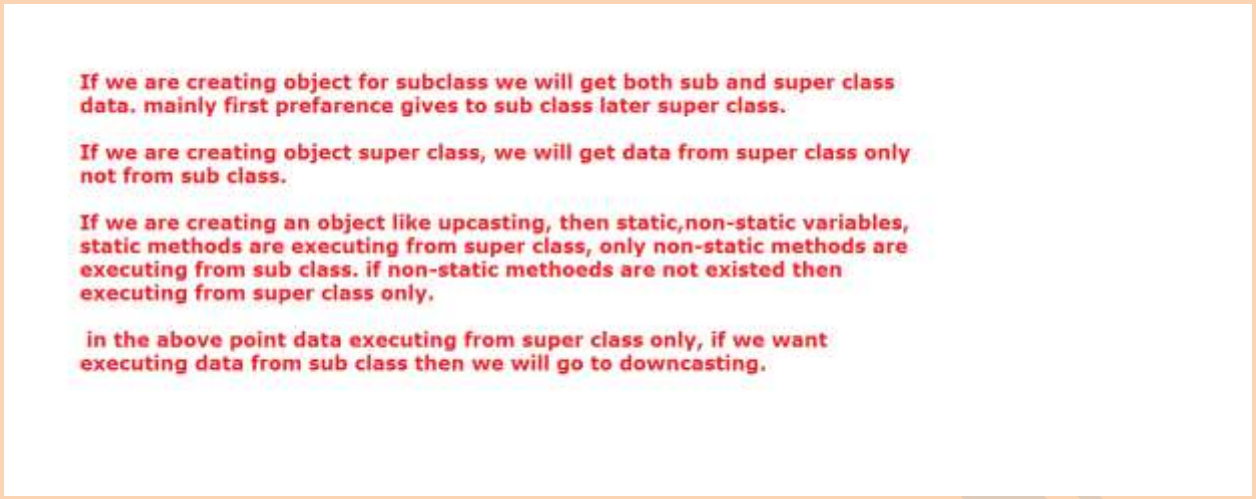
```
        obj1.m1();
        System.out.println("------------");
        MM obj2 = new NN();//upcasting
        System.out.println(obj2.a);
        System.out.println(obj2.b);
        obj2.m3();
        obj2.m4();
        //NN obj3 = new MM();
        System.out.println("--------");
        NN obj4 = (NN)obj2;
        System.out.println(obj4.a);
        System.out.println(obj4.b);
        obj4.m3();
        obj4.m4();
    }
}
```

Down-casting or specialization or narrowing:

Converting super class memory (subclass) into sub class type is called down-casting.

If we want do the down-casting first we need to up-casting.

```
MM obj  = new NN ();//up-casting
NN obj1 = (NN) obj; //down-casting.
```



In java we have three types of relations.

1. Is-A relation
2. Has-A relation
3. Uses-A relation.

Is-A:

Accessing all functionalities from one class to another class, then we can go fro Is-a relation.

In java we can develop is a relation with the help of extends keyword.

```
class MM{}

class NN extends MM{}
```

In the above relation we have a small drawback. That is if super class having 1000 variables, if we want to use some part of data (variables), then Is-A is not a best choice, the reason unnecessarily wasting of memory.

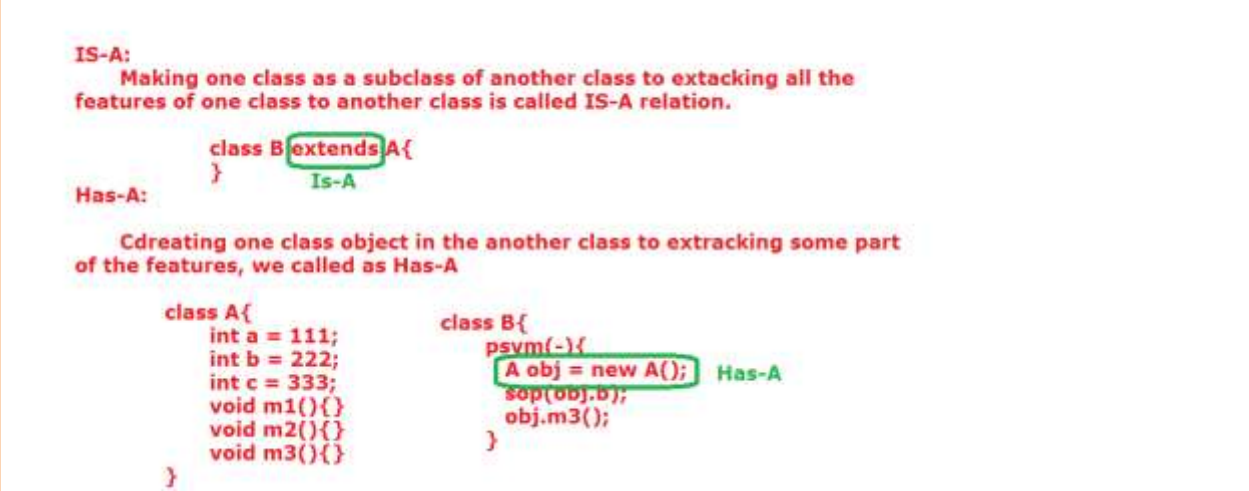
To avoid this drawback we can go for has-a relation.

Has-A:

Creating one class object within the another class is called has-a relation

```
class MM{}

class NN{

        psvm(){

        MM obj = new MM();

        }

}
```

Uses-A: using one class reference in another class as a method parameter is called uses-a relation.



This concept is comes under call by reference.

If we want use same memory data within the different location (method), then we need to forward that memory into different location(method) as argument.

```
abstract class AC{

        AC(int x){



        }

}

class SAC extends AC{
```

}

In the above coding we will get compiler time error.

The reason is if we are not writing any constructor in SAC then compiler will provide the following code.

```
class SAC extends AC{
        SAC(){
                super();
        }
}
```

super () will call super class (AC) zero argument constructor but there is no zero/default constructor in AC.

To overcome this problem we will for the following remedies.

1. Write zero argument constructors in AC (super class).

   or

2. Write constructor in SAC (subclass) and call argument constructor of AC (AC(int x){})

```
import java.util.Scanner;

interface Bill{
        double getRate();
        public double calculateBill(int units);
}
abstract class Plan implements Bill{
        double rate =0;
        public abstract double getRate();
        public double calculateBill(int units){
                rate = getRate()*units;
                return rate;
        }
}
class DemesticPlan extends Plan{
        public double getRate(){
                return 3;
        }
}
```

```
class CommercialPlan extends Plan{
	public double getRate(){
		return 5;
	}
}
class PlanFactory{
	public Plan getPlan(String type){
		if(type == null){
			return null;
		}
		else if(type.equalsIgnoreCase("DemesticPlan")){
			return new DemesticPlan();
		}
		else if(type.equalsIgnoreCase("CommercialPlan")){
			return new CommercialPlan();
		}
		return null;
	}
}
public class ElectricityBill {
	public static void main(String[] args) {
		Scanner scan = new Scanner(System.in);
		System.out.println("enter your plan");
		String planName = scan.next();
		PlanFactory factory = new PlanFactory();
		Plan plan = factory.getPlan(planName);
		System.out.println("enter units");
		int units = scan.nextInt();
		if(plan !=null){
			double rate = plan.calculateBill(units);
			System.out.println("currentBill: "+rate);
		}
		else
			System.out.println("your plan type is not existed in the world");
	}
}
```