01-03-2025

Relationship between the classes :

In java, In between the classes we have 2 types of relation :

1) IS-A Relation
2) HAS-A Relation

We can achieve IS-A relation by using Inheritance Concept.
We can achieve HAS-A relation by using Association Concept.

IS-A Relation Example :

```
public class Vehicle
{
}
public class Car extends Vehicle  //[Car IS-A Vehicle]
{
}
public class Toyota extends Car  //[Toyota IS-A Car]
{
}
```

HAS-A Relation Example :

```
public class Address
{
}

public class Student
{
   private Address address; //HAS-A Relation [Student has
}                                    Address]
```

Inheritance (IS-A Relation) :

Deriving a new class(Developer2) from the existing class (Developer1) in such a way that the new class (Developer2) will acquire all the properties and features (except private properties) from existing class (Developer1).

The main purpose of inheritance to provide "Code Reusability".

In java, We provide inheritance by using extends keyword.

By using inheritance, The relationship between the classes would be parent and child,
According to Java, Parent class is called super class and child class is called sub class.

In inheritance, sub class need not to start the process from
begning onwards because sub class has already all the features and properties are available.

It provides "tightly couple" relation that menas, if we modify anything in the super class then it will automatically reflect to all the sub classes.

It provides hierarchical classification of classes, In this hierarchy, If we move towards upward direction then more generalized properties will occur but if we move towards downward direction then more specialied properties will
occur.

Types of Inheritance in java :

We have 5 types of Inheritance in java :

1) Single Level Inheritance
2) Multilevel Inheritance
3) Hierarchical Inheritance
4) Multiple Inheritance [Not supported by using class]
5) Hybrid Inheritance.

Note : a) Inheritance follows top to bottom approach.
b) In inheritance, It is always better to create the
Object for more specialized class (child class)

//Program on Single Level Inheritance :

```
package com.ravi.inheritance;

public class Parent
{
  public void house()
  {
            System.out.println("3 BHK HOUSE");
  }
}
```

```
package com.ravi.inheritance;

public class Child extends Parent
{
  public void car()
  {
            System.out.println("BMW Car");
  }
}



package com.ravi.inheritance;

public class SingleLevelDemo {

        public static void main(String[] args)
        {
                Child c1 = new Child();
                c1.house();
                c1.car();
        }

}
```

//Program on Single Level Inheritance :

```
package com.ravi.inheritance;

class Super
{
        private int x, y;

        public void setData(int x, int y)
        {
                this.x = x;
                this.y = y;
        }

        public int getX()
        {
                return x;
```

```
	}

	public int getY()
	{
		return y;
	}


}
class Sub extends Super
{
	public void showData()
	{
		System.out.println("x value is :"+getX());
		System.out.println("y value is :"+getY());
	}
}
public class SingleLevelExample {

	public static void main(String[] args)
	{
      Sub s1 = new Sub();
      s1.setData(100, 200);
      s1.showData();
	}

}
```

03-03-2025

How to initialize the super class properties (super class instance variable) through sub class object :

super keyword is used to access the member or to access the memory of super class.

In order to initialize the super class properties we should use super keyword in the sub class as a first line of constructor.

super keyword always referes to its immediate super class.

Just like this keyword, super keyword (non static member) also we can't use inside static context.

super keyword we can use 3 ways in java :
---
1) To access super class variable (Variable Hiding)
2) To access super class method (Method Overriding)
3) To access super class constructor. (Constructor Chaining)

1) To access the super class variable (Variable Hiding) :

Whenever super class variable name and sub class variable name both are same then it is called variable Hiding, Here sub class variable hides super class variable.

In order to access super class variable i.e super class memory, we should use super keyword as shown in the program.[03-MARCH]

VariableHidingDemo1.java

```
package com.ravi.variable_hiding;

class Father
{
	protected double balance = 50000;
}

class Son extends Father
{
	protected double balance = 20000;  //Variable Hiding

	public void printBalance()
	{
		System.out.println("Son balance is :"+this.balance);
		System.out.println("Father balance is :"+super.balance);
	}
}
public class VariableHidingDemo1
{
	public static void main(String[] args)
	{
		Son raj = new Son();
		raj.printBalance();
	}

}
```

==========
Method Hiding = It is only possible with static method.
Method Overriding = It is only possible with non-static method.

2) To call the method super class (Method Overriding)

If the super class non static method name and sub class non static method name both are same (Method Overriding) and if we create an object for sub class then sub class method will be executed (bottom to top), if we want to call super class method from sub class method body then we we should use super keyword as shown in the program.

```
package com.ravi.inheritance_demo;

class Alpha
{
	@Override
	public String toString()
	{
		return "Alpha []";
	}
}

class Beta extends Alpha
{
	@Override
	public String toString()
	{
		return "Beta []";
	}
}

class Gamma extends Beta
{
	@Override
	public String toString()
	{
		return "Gamma []";
	}
}

public class MethodOverridingDemo1 {

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

```
	{
		Gamma g = new Gamma();
		System.out.println(g);
	}

}
```

```
package com.ravi.inheritance_demo;

class Base
{
	public void show()
	{
		System.out.println("Base class show method");
	}
}
class Derived extends Base
{
	public void show()
	{
		System.out.println("Derived class show method");
		super.show();
	}
}

public class MethodOverridingDemo2 {

	public static void main(String[] args)
	{
		Derived d = new Derived();
		d.show();
	}

}
```
==========

3) To call super class Constructor (Constructor Chaining)

Whenever we write a class in java and we don't write any kind of constructor to the class then the java compiler will automatically add one default constructor to the class.

THE FIRST LINE OF ANY CONSTRUCTOR IS RESERVERD EITHER FOR super() or this() keyword that means first line of any constructor is used to call another constructor of either same class OR super class.

In the first line of any constructor if we don't specify either super() or this() then the compiler will automatically add super() to the first line of constructor.

Now the purpose of this super() [added by java compiler], to call the default constructor or No-Argument constructor of the super class.

In order to call the constructor of super class as well as same class, we have total 4 cases.

04-03-2025

Case 1:

---

super() : Automatically added by compiler to maintain the hierarchy in the first line of the Constructor. It is used to call default OR no argument constructor of super class.

```
package com.ravi.constructor_chaining;

class Alpha
{
	public Alpha()
	{
		super(); //No Argument OR Default constructor of super class
		System.out.println("Alpha class Constructor");
	}
}
class Beta extends Alpha
{
	public Beta()
	{
		super();//No Argument OR Default constructor of super class
		System.out.println("Beta class Constructor");
	}
}

public class ConstructorChainingDemo1
{
	public static void main(String[] args)
	{
		new Beta();
	}
```

}

Case 2 :
---
super("Scott"); : User has to write explicitly to the first line
of the Constructor. It is used to call
Parameterized Constructor (Pramaterized constructor of super class which accepts a String value as a parameter) of super class.

```
package com.ravi.constructor_chaining;

class Super
{
	public Super(String name)
	{
		super();
		System.out.println("My name is :"+name);
	}
}
class Sub extends Super
{
	public Sub()
	{
		super("Scott");
		System.out.println("No Argument constructor of sub class");
	}
}

public class ConstructorChainingDemo2 {

	public static void main(String[] args)
	{
		new Sub();

	}

}
```

IQ :
---
class A
{

```
	public A()
	{
		System.out.println("A");
	}
}
class B extends A
{
}

class C extends B
{
	public C()
	{
		System.out.println("C");
	}
}
public class Test
{
  public static void main(String [] args)
  {
     new C();
  }
}
```

Output is : AC
So it is clear that, compiler adds default constructor in B class as well as in the first line compiler is adding super().

Case 3 :

---

this() : Written by user in the first line of constructor. It is used to call no argument constructor of current class OR Same class.

```
package com.ravi.constructor_chaining;

class Base
{
	public Base()
	{
		System.out.println("No Arg. constructor of super class ");
	}
	public Base(int x)
```

```
	{
		this();
		System.out.println("Parameterized constructor of super class :"+x);
	}
}

class Derived extends Base
{
	public Derived()
	{
		super(15);
		System.out.println("No Arg. constructor of Sub class ");
	}
}

public class ConstructorChaningDemo3
{
	public static void main(String[] args)
	{
		new Derived();

	}

}
```

Case 4 :

---

this("Smith"); : It is explicitly written by user in the first
line of the constructor. It is used to call
parameterized constructor of current class.

```
package com.ravi.constructor_chaining;

class Parent
{
	public Parent()
	{
		this("Smith");
		System.out.println("No Argument Constructor of Parent class");
	}

	public Parent(String name)
	{
```

```
		System.out.println("Parameterized Constructor :"+name);
	}
}
class Child extends Parent
{
	public Child()
	{
		System.out.println("No Argument Constructor of Child class");
	}
}

public class ConstructorChainingDemo4
{
	public static void main(String[] args)
	{
		new Child();
	}

}
```

==========

WAP to show hierarchical Inheritance :

```
package com.ravi.hierarchical;

class Shape
{
	protected int x;

	public Shape(int x)
	{
		this.x = x;
		System.out.println("x value is :"+x);
	}
}
class Square extends Shape
{
	public Square(int side)
	{
		super(side);
	}

	public void getAreaOfSquare()
	{
```

```
		double area = x * x;
		System.out.println("Area of Square is :"+area);
	}

}

class Rectangle extends Shape
{
	protected int breadth;

	public Rectangle(int length, int breadth)
	{
		super(length);
		this.breadth = breadth;
	}

	public void getAreaOfRectangle()
	{
		double area = x * breadth;
		System.out.println("Area of Rectangle is :"+area);
	}

}
public class HierarchicalDemo1
{
	public static void main(String[] args)
	{
		Square ss = new Square(10);
		ss.getAreaOfSquare();

		Rectangle rr = new Rectangle(8, 9);
		rr.getAreaOfRectangle();

	}

}
```

Program on Hierarchical Inheritance :

```
package com.ravi.hierarchical;

class Employee
{
```

```
	protected double salary;

	public Employee(double salary)
	{
		super();
		this.salary = salary;
	}
}
class Developer extends Employee
{
	public Developer(double salary)
	{
		super(salary);
	}

	@Override
	public String toString()
	{
		return "Developer [salary=" + salary + "]";
	}

}

class Designer extends Employee
{
	public Designer(double salary)
	{
		super(salary);
	}

	@Override
	public String toString()
	{
		return "Designer [salary=" + salary + "]";
	}
}
public class HierarchicalDemo2 {

	public static void main(String[] args)
	{
		Developer d1 = new Developer(60000);
		System.out.println(d1);
```

```
		Designer d2 = new Designer(35000);
		System.out.println(d2);

	}

}
```

//Program on Single Level Inheritance :

```
package com.ravi.single_level;

class Emp
{
	protected int employeeId;
	protected String employeeName;
	protected double employeeSalary;

	public Emp(int employeeId, String employeeName, double employeeSalary)
	{
		super();
		this.employeeId = employeeId;
		this.employeeName = employeeName;
		this.employeeSalary = employeeSalary;
	}
}
class Pemp extends Emp
{
	protected String department;
	protected String designation;

	public Pemp(int employeeId, String employeeName, double employeeSalary, String
department, String designation)
	{
		super(employeeId, employeeName, employeeSalary);
		this.department = department;
		this.designation = designation;
	}

	@Override
	public String toString() {
		return "Pemp [employeeId=" + employeeId + ", employeeName=" +
employeeName + ", employeeSalary="
				+ employeeSalary + ", department=" + department + ",
```

```
designation=" + designation + "]";
	}
}

public class SingleLevelDemo1 {

	public static void main(String[] args)
	{
		Pemp p = new Pemp(1, "Scott", 90000, "IT", "Programmer");
		System.out.println(p);

	}

}
```

05-03-2025

//Program on Multilevel Inheritance :

```
package com.ravi.multilevel_inheritance;

class Student
{
	protected int id;
	protected String name;
	protected String address;

	public Student(int id, String name, String address)
	{
		super();
		this.id = id;
		this.name = name;
		this.address = address;
	}




}

class Science extends Student
{
	protected int phyMarks;
```

```
	protected int cheMarks;

	public Science(int id, String name, String address, int phyMarks, int cheMarks)
	{
		super(id, name, address);
		this.phyMarks = phyMarks;
		this.cheMarks = cheMarks;
	}




}
class PCM extends Science
{
	protected int mathMarks;

	public PCM(int id, String name, String address, int phyMarks, int cheMarks, int
mathMarks) {
		super(id, name, address, phyMarks, cheMarks);
		this.mathMarks = mathMarks;
	}

	@Override
	public String toString()
	{
		return "PCM [id=" + id + ", name=" + name + ", address=" + address + ",
phyMarks=" + phyMarks + ", cheMarks="
				+ cheMarks + ", mathMarks=" + mathMarks + "]";
	}

	public void totalMarks()
	{
		int total = this.phyMarks + this.cheMarks + this.mathMarks;
		System.out.println("Total Marks is :"+total);
	}
}

public class MultiLevelInheritance
{
	public static void main(String[] args)
	{
		PCM p = new PCM(111, "Scott", "Ameerpet", 90, 80, 92);
```

```
		System.out.println(p);
		p.totalMarks();
	}

}
```

==========

The following program describes that method execution is always
from bottom to top only.

```
package com.ravi.multilevel_inheritance;

class Vehicle
{
	protected String name;

	public Vehicle(String name)
	{
		this.name = name;
	}

	public void run()
	{
		System.out.println(name+" Vehicle is Running");
	}
}

class Car extends Vehicle
{
	public Car(String name)
	{
		super(name);
	}

	public void run()
	{
		System.out.println(name+" Car is Running");
	}

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

```
	{
		Car car = new Car("Naxon");
		car.run();

	}

}
```

Note : Naxon Car is running.

==========

**Why java does not support multiple Inheritance ?

Multiple Inheritance is a situation where a sub class wants to inherit the properties two or more than two super classes.

In every constructor we have super() or this(). When compiler will add super() to the first line of the constructor then we have an ambiguity issue that super() will call which super class constructor as shown in the diagram [05-03]

It is also known as Diamond Problem in java so the final conclusion is we can't achieve multiple inheritance using classes but same we can achieve by using interface [interface does not contain any constructor]

HOW MANY WAYS WE CAN INITIALIZE THE OBJECT PROPERTIES ?

The following are the ways to initialize the object properties

---

1) At the time of declaration :

Example :

```
public class Test
{
   int x = 10;
   int y = 20;
}

Test t1 = new Test();   [x = 10  y = 20]
Test t2 = new Test();   [x = 10  y = 20]
```

Here the drawback is all objects will be initialized with same value.

---

2) By using Object Reference :

```
public class Test
{
  int x,y;
}

Test t1 = new Test();   t1.x=10;   t1.y=20;
Test t2 = new Test();   t2.x=30;   t2.y=40;
```

Here we are getting different values with respect to object but here the program becomes more complex.

---

3) By using methods :

A) First Approach (Method without Parameter)

```
public class Test
{
  int x,y;

  public void setData()
  {
    x = 100;  y = 200;
  }
}

Test t1 = new Test();  t1.setData();  [x = 100   y = 200]
Test t2 = new Test();  t2.setData();  [x = 100   y = 200]
```

Here also, all the objects will be iniatilized with same value.

B) Second Approach (Method with Parameter)

```
public class Test
{
  int x,y;

  public void setData(int x, int y)
  {
    this.x = x;
      this.y = y;
  }
```

```
}

 Test t1 = new Test();  t1.setData(12,78);  [x = 12  y = 78]
 Test t2 = new Test();  t2.setData(15,29);  [x = 15  y = 29]
```

Here the Drawback is initialization and re-initialization both are done in two different lines so Constructor introduced.

---

4) By using Constructor

A) First Approach (No Argument Constructor)

```
public class Test
{
  int x,y;

  public Test()  //All the objects will be initialized with
  {                                same value
    x = 0;  y = 0;
  }

}

 Test t1 = new Test();    [x = 0   y = 0]
 Test t2 = new Test();    [x = 0   y = 0]
```

B) Second Approach (Parameterized Constructor)

```
 public class Test
 {
  int x,y;

  public Test(int x, int y)
  {
    this.x = x;
       this.y = y;
  }
}

 Test t1 = new Test(12,78);    [x = 12   y = 78]
 Test t2 = new Test(15,29);    [x = 15   y = 29]
```

This is the best way to initialize our instance variable because variable initialization and variable re-initialization both will be done in the same line as well as all the objects will be initialized with different values.

C) Third Approach (Copy Constructor)

```
public class Manager
{
  private int managerId;
    private String managerName;

    public Manager(Employee emp)
    {
      this.managerId = emp.getEmployeeId();
      this.managerName = emp.getEmployeeName();
  }
}
```

Here with the help of Object reference (Employee class) we are initializing the properties of Manager class. (Copy Constructor)

d) By using instance block (Instance Initializer)

```
public class Test
{
  int x,y;

public Test()
  {
    System.out.println(x); //100
    System.out.println(y); //200
  }

//Instance block
  {
     x = 100;
     y = 200;
  }
}
```

---

5) By using super keyword :

```
class Super
{
  int x,y;

  public Super(int x , int y)
  {
    this.x = x;
       this.y = y;
  }
}
class Sub extends Super
{
   Sub()
   {
     super(100,200); //Initializing the properties of super class
   }
}

new Sub();
```

==========

Assignment :

//Program on Hybrid Inheritance

```
class Vehicle
{
}
class Car extends Vehicle
{}
```

Maruti and Ford

==========

Access modifiers in java :

In order to define the accessibility level of the class as well as member of the class we have 4 access modifiers :

1) private (Within the same class)
2) default (Within the same package/folder)
3) protected (Within the same package OR even from another
   package by using Inheritance)

4) public (No Restriction)
---
private :

It is the most restrictive access modifier because the member declared as private can't be accessible from outside of the class.
In Java we can't declare an outer class as a private or protected. Generally we should declare the data member(variables) as private [data hiding].

In java outer class can be declared as public, abstract, final, sealed and non-sealed only.

default :-

It is an access modifier which is less restrictive than private. It is such kind of access modifier whose physical existance is not avaialble that means when we don't specify any kind of access modifier before the class name, variable name or method name then by default it would be default.

As far as its accessibility is concerned, default members are accessible within the same folder(package) only. It is also known as private-package modifier.

06-03-2025
---
protected :

It is an access modifier which is less restrictive than default because the member declared as protected can be accessible from the outside of the package (folder) too but by using inheritance concept.

Program

Access.java [Available in com.ravi.m1 package]

```
package com.ravi.m1;

public class Access
{
   protected int x = 500;
}
```

ELC.java[This program is available in another package com.ravi.m2]

```
package com.ravi.m2;

import com.ravi.m1.Access;

public class ELC extends Access
{
	public static void main(String[] args)
	{
		ELC e = new ELC();
		System.out.println(e.x);

	}

}
```

---

public :

It is an access modifier which does not contain any kind of restriction that is the reason the member declared as public can be accessible from everywhere without any restriction.

According to Object Oriented rule we should declare the classes and methods as public where as variables must be declared as private or protected according to the requirement.

Note : If a method is used for internal purpose only (like validation) then we can declare that method as private method.It is called Helper method.

==========

JVM Architecture with class loader sub system :

The entire JVM Architecture is divided into 3 sections :

1) Class Loader sub system
2) Runtime Data areas (Memory Areas)
3) Execution Engine

Class Loader Sub System :

The main purpose of Class Loader sub system to load the required .class file into JVM Memory from different memory loactions.

In order to load the .class file into JVM Memory, It uses an algorithm called "Delegation Hierarchy Algoroithm".

Internally, Class Loader sub system performs the following Task

1) LOADING
2) LINKING
3) INITIALIZATION

LOADING :

In order to load the required .class file, JVM makes a request to class loader sub system. The class loader sub system follows delegation hierarchy algorithm to load the required .class files from different areas.

To load the required .class from different area, we have 3 different kinds of class loaders.

1) Bootstrap/Primordial class Loader

2) Platform/Extension class Loader

3) Application/System class Loader

Bootstrap/Primordial class Loader :-

It is responsible for loading all the predefined .class files that means all API(Application Programming Interface) level predefined classes are loaded by Bootstrap class loader.

It has the highest priority becuase Bootstrap class loader is the super class for Platform class loader.

It loads the classes from the following path
C -> Program files -> Java -> JDK -> lib -> jrt-fs.jar

Platform/Extension class loader :

It is responsible to load the required .class file which is given by some 3rd party in the form of jar file.

It is the sub class of Bootstrap class loader and super class of Application class loader so it has more priority than Application class loader.

It loads the required .class file from the following path.
C -> Program files -> Java -> JDK -> lib -> ext -> ThirdParty.jar

Command to create the jar file :
jar cf FileName.jar FileName.class [*.class]

[If we want to compile more than one java file at a time then the command is : javac *.java]

07-03-2025

Application/System class loader :

It is responsible to load all userdefined .class file into JVM memory.

It has the lowest priority because it is the sub class Platform class loader.

It loads the .class file from class path level or environment
variable.

Note : If all the class loaders are failed to load the .class file
into JVM memory then JVM will generate an exception i.e
java.lang.ClassNotFoundException.

How Delegation Hierarchy algorithm works :-

Whenever JVM makes a request to class loader sub system to load the required .class file into JVM memory, first of all, class loader sub system makes a request to Application class loader, Application class loader will delegate(by pass) the request to the platform class loader, platform class loader will also delegate the request to Bootstrap class loader.

Bootstrap class loader will load the .class file from lib folder(jrt-fs.jar) and then by pass the request back to platform class loader, platform class loader will load the .class file from ext folder(*.jar) and by pass the request back to Application class loader, It will load the .class file from environment variable into JVM memory.

Note : java.lang.Object is the first class to be loaded into JVM Memory.

What is Method Chaning in java ?

It is a technique through which we call multiple methods in a single statement.

In this method chaining, always for calling next method we depend upon last method return type.

The final return type of the method depends upon last method call as shown in the program.

MethodChainingDemo1.java

```
package com.ravi.method_chaining_demo1;

public class MethodChainingDemo1 {

	public static void main(String[] args)
	{
		String str = "india";
		char ch = str.toUpperCase().concat(" is great").charAt(0);
	System.out.println(ch);
	}

}
```

MethodChaingDemo2.java

```
package com.ravi.method_chaining_demo1;

public class MethodChaingDemo2 {

	public static void main(String[] args)
	{
		String str = "Hyderabad";
		int length = str.concat(" is an IT city").toUpperCase().length();
		System.out.println("Length is :"+length);
	}

}
```

Role of java.lang.Class class in class loading :

There is a predefined class called Class available in java.lang pacakge.

In JVM memory whenever we load a class then it is loaded in special memory area called Method Area and retutn type is java.lang.Class class object.

```
java.lang.Class cls = AnyClass.class
```

java.lang.Class class contains a predefined non static method called getName() through which we can get the fully qualified name [Package Name + class Name]

public String getName() : Provide fully qualified name of the class.

```
package com.ravi.method_chaining_demo1;

class A{}

class B{}

class Customer{}

class Sample{}


public class MethodAreaDemo {

        public static void main(String[] args)
        {
      Class c1 = A.class;
      System.out.println(c1.getName());

      c1 = B.class;
      System.out.println(c1.getName());

      c1 = Customer.class;
      System.out.println(c1.getName());

      c1 = Sample.class;
      System.out.println(c1.getName());
        }

}
```

//WAP that describes Application class loader is responsible to load our user-defined .class

java.lang.Class class has provided a predefined non static method
called getClassLoader(), the return type of this method is ClassLoader class.

```
   public ClassLoader getClassLoader();

package com.ravi.loading;

public class Test {
```

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

	 System.out.println("Test.class file is loaded by :"+Test.class.getClassLoader());

		//OR

	 Class cls = Test.class;
	 System.out.println("Class Loader name is :"+cls.getClassLoader());
	}

}
```

08-03-2025

WAP to show the Platform class loader is super class of
Application class loader.

```
class Hello
{
}
public class Test
{
	public static void main(String[] args)
	{
		System.out.println("Super class of Application class loader is " +
Hello.class.getClassLoader().getParent());
	}
}
```

getClassLoader() is a predefined non static method of java.lang.Class class and the return type of this method is java.lang.ClassLoader

```
 public ClassLoader getClassLoader()
```

ClassLoader which is an abstract class has provided a predefined non static methdo called getParent(), the return type of this methdo is ClassLoader.

```
public ClassLoader getParent() :
```

It will provide the name of the super class and will ClassLoader object.

```
class Hello
{
}
public class Test
{
	public static void main(String[] args)
	{
		System.out.println("Super class of Platform class loader is " +
Hello.class.getClassLoader().getParent().getParent()); //null
	}
}
```

Note :- Here we will get the output as null because it is built in class loader for JVM which is used for internal purpose (loading only predefined .class file) so implementation is not provided hence we are getting null.

Linking :

In linking phase we have 3 modules :

a) Verify b) Prepare c) Resolve

Verify :-

It ensures the correctness of the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing a runtime error i.e java.lang.VerifyError.

There is something called ByteCodeVerifier(Component of JVM), responsible to verify the loaded .class file i.e byte code. Due to this verify module JAVA is highly secure language.

java.lang.VerifyError is the sub class of java.lang.linkageError.

prepare :

[Static variable memory allocation + static variable initialization with default value even the variable is final]

It will allocate the memory for all the static data members, here all the static data member will get the default values so if we have static int x = 100; then for variable x memory will be allocated (4 bytes) and now it will be initialized with default value i.e 0, even the variable is final.

static Test t = new Test();

Here, t is a static reference variable so for t variable (reference variable) memory will be allocated as per JVM implementation i.e for 32 bit JVM (4 bytes of Memory) and for 64 bit (8 bytes of memory) and initialized with null.

Memory allocation and initialzation with default value for NSV :

For NSV memory allocation and initialization with default value will be done at the time of Object creation by using new keyword with the help of Java Compiler.

Memory allocation and initialzation with default value for SV :

For SV memory allocation and initialization with default value will be done at the time of CLASS LOADING in the prepare phase.

Resolve :

All the symbolic references (#7) will be converted into direct references OR actual reference.

javap -verbose FileName.class

Note :- By using above command we can read the internal details of .class file.

10-03-2025

Initialization :

Here class initialization will takes place. All the static data member will get their actual/original value and we can also use static block for static data member initialization.

Here, In this class initialization phase static variable and static block is having same priority so it will executed according to the order.(Top to bottom)

Static Block OR Static Initializer :

It is a special block in java which is automatically executed at the time of loading the .class file.

Example :

```
static
{

}
```

Static blocks are executed only once because in java we can load the .class files only once.

If we have more than one static blocks in a class then it will be executed according to the order [Top to bottom]

The main purpose of static block to initialize the static data member of the class so it is also known as static initializer.

In java, a class is not loaded automatically, it is loaded based on the user request so static block will not be executed everytime, It depends upon whether class is loaded or not.

static blocks are executed before the main or any static method.

A static blank final field must be initialized inside the static block only.

```
static final int A;   //static blank final field

static
{
  A = 100;
}
```

A static blank final field also have default value.

We can't write any kind of return statement inside static block.

If we don't declare static variable before static block body execution then we can perform write operation(Initialization is possible due to prepare phase) but read operation is not possible directly otherwise we will get an error Illegal forward reference, It is possible with class name bacause now compiler knows that variable is coming from class area OR Method area.

Example :

```
static
{
  x = 100; //Valid
  System.out.println(x); //Invalid
  System.out.println(ClassName.x); //valid
}
static int x;
```

If we directly perform direct read and write operation inside a static method then it is valid.

```
public static void m1()
{
   y = 200; //Valid
   System.out.println(y); //Valid
}

static int y;
```

```
//static block
class Foo
{
	Foo()
	{
		System.out.println("No Argument constructor..");
	}

	{
		System.out.println("Non static Block..");
	}

	static
	{
		System.out.println("Static block...");
	}

}
public class StaticBlockDemo
{
	public static void main(String [] args)
	{
		System.out.println("Main Method Executed ");

	}
}
```

Note : Here Foo.class file is not loaded hence static block will not be executed.

```
class Test
{
	static int x;

	static
```

```
	{
		x = 100;
		System.out.println("x value is :"+x);
	}

	static
	{
		x = 200;
		System.out.println("x value is :"+x);
	}

	static
	{
		x = 300;
		System.out.println("x value is :"+x);
	}
}
public class StaticBlockDemo1
{
	public static void main(String[] args)
	{
		System.out.println("Main Method");
		System.out.println(Test.x);
	}
}
```

Note : If a class contains more than 1 static block then it will be executed from top to bottom.

```
class Foo
{
	static int x;

   static
	{
		System.out.println("x value is :"+x);
	}
}

public class StaticBlockDemo2
{
	public static void main(String[] args)
	{
		 new Foo();
	}
```

```
}
```

Note : static variables are also having default value.

```
class Demo
{
	final static int a ;		//Blank static final field

	static
	{
		m1();
		a = 100;
		System.out.println("User Value :"+a);
	}

	public static void m1()
	{
		System.out.println("Default Value :"+a);
	}


}
public class  StaticBlockDemo3
{
	public static void main(String[] args)
		{
		System.out.println("a value is :"+Demo.a);
	   }
}
```

Note : A statitc blank final field also have default value.

```
class  A
{
	static
	{
		System.out.println("A");
	}

	{
		System.out.println("B");
	}
```

```
	A()
	{
		super();
		System.out.println("C");
	}
}
class B extends A
{
	static
	{
		System.out.println("D");
	}

	{
		System.out.println("E");
	}

	B()
	{
		super();
		System.out.println("F");
	}

}
public class StaticBlockDemo4
{
	public static void main(String[] args)
	{
		new B();  //class loading + Object Creation
	}
}
```

Note : Always Parent class will be loaded first before Child.

```
//illegal forward reference

class Demo
{
   static
	{
		i = 100;
	}
```

```
    static int i;
}

public class StaticBlockDemo5
{
    public static void main(String[] args)
          {
        System.out.println(Demo.i);
          }
}
-------------------------------------------------------------
class Demo
{
    static
          {
                i = 100;
                //System.out.println(i); //Illegal forward reference
                System.out.println(Demo.i);
    }

    static int i;
}

public class StaticBlockDemo6
{

    public static void main(String[] args)
          {
                System.out.println(Demo.i);
          }
}
-------------------------------------------------------------
class StaticBlockDemo7
{
          static
          {
      System.out.println("Static Block");
            return;
          }

          public static void main(String[] args)
          {
                System.out.println("Main Method");
```

```
        }
}
-------------------------------------------------------------
public class StaticBlockDemo8
{
   final static int x; //Blank static final field

   static
        {
                m1();
                x = 15;
        }

   public static void m1()
        {
                System.out.println("Default value of x is :"+x);
        }

        public static void main(String[] args)
        {
                System.out.println("After initialization :"+StaticBlockDemo8.x);
        }
}
-------------------------------------------------------------
11-03-2025
----------
class Test
{
        public static final Test t1 = new Test();          //t1 = null

        static
        {
          System.out.println("static block");
        }

        {
          System.out.println("Non static block");
        }

        Test()
        {
                System.out.println("No Argument Constructor");
        }
```

```
}

public class StaticBlockDemo9
{
	public static void main(String[] args)
	{
		new Test();  //2 steps (class loading + Object creation)
	}
}
-------------------------------------------------------------
class Sample
{
	static
	{
		System.out.println("Static Block");
		x = m1();
		System.out.println("x from static block :"+Sample.x);
	}

	public static int m1()
	{
	  System.out.println("Static Method");
	  return 100;
	}

	static int x;
}


public class StaticBlockDemo10
{
	public static void main(String[] args)
	{
		System.out.println("x from main method :"+Sample.x);
	}
}
-------------------------------------------------------------
class Demo
{
	public static void print()
	{
```

```
                x = 120;
                System.out.println("x value is :"+x);
        }

        static int x;
}

public class StaticBlockDemo11
{
        public static void main(String[] args)
        {
                System.out.println(Demo.x);
                Demo.print();
        }
}
```

==========

Compile time constant :

A compile time constant is a constant that is evaluated and replaced with its value at compile time rather than runtime.

It must be declared with static and final modifier as well as initialized with constant expression. (Must not be initialized by static method call)

At compile, time constant value will be converted by compiler at the time of compilation itself so, at runtime JVM can see the constant value but not the class name so class will not be loaded as shown in the program.

Example : public static final int A = 100; //Valid
[Class will not be loaded]

public static final int A = m1(); //valid [Here class will be loaded by JVM]

```
class Test
{
        public static final int MAX_VALUE = 120;

        static
        {
                System.out.println("Static Block");
        }
}
```

```
public class CompileTimeConstantDemo1
{
	public static void main(String[] args)
	{
		System.out.println(Test.MAX_VALUE);
	}
}
```

Note : class is not loaded hence static block will not be executed.

```
 class Demo
{
	public static final int MIN_VALUE = m1();

	public static int m1()
	{
		return 1;
	}

	static
	{
		System.out.println("Static Block");
	}
}

public class CompileTimeConstantDemo2
{
	public static void main(String[] args)
	{
		System.out.println(Demo.MIN_VALUE);
	}
}
```

Foo.java

```
public class  Foo
{
	public static final int MAX_PRIORITY = 72;
}
```

CompileTimeConstantDemo3.java

```
//ELC
public class CompileTimeConstantDemo3
{
	public static void main(String[] args)
	{
		System.out.println(Foo.MAX_PRIORITY);
	}
}
```

Note : If we compile and execute both programs we will get 72 value, now change the 72 to 95, Re- compile only Foo.java so a new .class file will be created which will hold 95 value. Now without re-compilation of CompileTimeConstantDemo3.java if we execute CompileTimeConstantDemo3.java then we wil get 72 so it takes the value at compile time.

==========

Can we write a java program without main method ?

It looks like we can write a java program by using static block (without main method) but this facility was available till JDK 1.6.

```
class WithoutMain
{
	static
	{
	System.out.println("Hello world");
	System.exit(0);
	}
}
```

From JDK 1.7V onwands we can't write a java program without main method because at the time of class loading JVM will verify the presence of main method.

Variable Memory Allocation, Initialization and life cycle :

---

1) static field OR Class variable :

Memory allocation done at prepare phase of class loading and initialized with default value even variable is final.

It will be initialized with Original value (If provided by user at the time of declaration) at class

initialization phase.

When JVM will shutdown then during the shutdown phase class will be un-loaded from JVM memory so static data members are destroyed. They have long life.

2) Non static field OR Instance variable

Memory allocation done at the time of object creation using new keyword (Instantiation) and initialized as a part of Constructor with default values even the variable is final. [Object class-> at the time of declaration -> instance block -> constructor]

When object is eligible for GC then object is destroyed and all the non static data memebers are also destroyed with corresponding object. It has less life in comparison to static data members becuase they belongs to object.

3) Local Variable

Memory allocation done at stack area (Stack Frame) and developer is responsible to initialize the variable before use. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

4) Parameter variable

Memory allocation done at stack area (Stack Frame) and end user is responsible to pass the value at runtime. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

Note : We can done validation only one parameter variables.

==========
How many ways we can load the .class file into JVM memory :

There are so many ways to load the .class file into JVM memory but the following are the common examples :

1) By using java command

```
public class Test
{
}

javac Test.java
java Test
```

Here we are making a request to JVM to load Test.class file into JVM memory

2) By using Constructor (new keyword at the time of creating object).

3) By accessing static data member of the class.

4) By using inheritance

5) By using Reflection API

//Program that describes we can load a .class file by using new keyword (Object creation) OR by accessing static data member of the class.

```
class Demo
{
	static int x = 10;
	static
	{
		System.out.println("Static Block of Demo class Executed!!! :"+x);
	}
}
public class ClassLoading
{
	public static void main(String[] args)
	{
		System.out.println("Main Method");
		//new Demo();
	   System.out.println("Hello "+Demo.x);
	}

}
```

//Program that describes whenever we try to load sub class, first of all super class will be loaded. [before parent, child can't exist]

```
class Alpha
{
	static
	{
		System.out.println("Static Block of super class Alpha!!");
	}
```

```
}
class Beta extends Alpha
{
	static
	{
		System.out.println("Static Block of Sub class Beta!!");
	}
}
class InheritanceLoading
{
	public static void main(String[] args)
	{
		 new Beta();
	}
}
------------------------------------------------------------
package iq;
class Alpha
{
	int x = 100;
}
class Beta extends Alpha
{
	int x = 200; //Variable Hiding

	public static void access()
	{
	  Beta b1 = new Beta();
	  System.out.println("x value is :"+b1.x);

	  Alpha a1 = b1;
	  System.out.println("x value is :"+a1.x);


	}
}
public class Main
{
	public static void main(String[] args)
	{
		Beta.access();
      }
```

```
}
```

12-03-2025

Loading the .class file by using Reflection API :
---
java.lang.Class class has provided a predefined static factory method called forName(String className), It is mainly used to load the given .class file at runtime, The return type of this method is java.lang.Class

public static java.lang.Class forName(String className) throws ClassNotFoundException

Note : This method throws a checked execption i.e ClassNotFoundException

```
class Foo
{
	static
	{
		System.out.println("Static Block!!!!");
	}
}
public class ClassLoading
{
	public static void main(String[] args) throws Exception
	{
		Class.forName("Foo");
	}
}
```

Note : Here Foo class will be loaded as a result static block will be executed.

```
package com.ravi.class_loading;

class Test
{
	 static
	 {
		 System.out.println("Static Block");
	 }
}
public class ClassLoading
{
```

```
	public static void main(String[] args) throws Exception
	{
		Class.forName("com.ravi.class_loading.Test"); //FQN(Fully Qualified Name)
	}
}
```

Note : While working with Eclipse IDE FQN (Full Qualified Name) is required.

*** What is the difference between java.lang.ClassNotFoundException
and java.lang.NoClassDefFoundError

java.lang.ClassNotFoundException :-

It occurs when we try to load the required .class file at RUNTIME by using Class.forName(String className) statement or loadClass() static of ClassLoader class and if the required .class file is not available at runtime then we will get an exception i.e java.lang.ClassNotFoundException

Note :- It does not have any concern at compilation time, at run time, JVM will simply verify whether the required .class file is available or not available.

```
class Test
{
	static
	{
		System.out.println("Static Block");
	}
}

public class ClassNotFoundExceptionDemo
{
	public static void main(String[] args) throws Exception
	{
		Class.forName("Sample");
	}
}
```

Note : We will get java.lang.ClassNotFoundException because at runtime JVM will try to load Sample.class file but Sample.class file is not available at runtime.

---

java.lang.NoClassDefFoundError :

It occurs when the class was present at the time of COMPILATION but at runtime the required .class file is not available(manualy deleted by user) Or it is not available in the current directory (Misplaced) then we will get a runtime error i.e java.lang.NoClassDefFoundError.

```
class Student
{
	public void greet()
	{
		System.out.println("Hello Everyone!!!");
	}
}

public class NoClassDefFoundErrorDemo
{
	public static void main(String[] args)
	{
		Student s1 = new Student();
		s1.greet();
	}
}
```

Note : After compilation, delete Student.class file from the current folder and exceute the program.

==========

** A static method does not act on instance variable directly why?

All the static members (static variable, static block, static method, static nested inner class) are loaded/executed at the time of loading the .class file into JVM Memory.

At class loading phase object is not created because object is created in the 2nd phase i.e Runtime data area so at the TIME OF EXECUTION OF STATIC METHOD AT CLASS LOADING PAHSE, NON STATIC VARIABLE WILL NOT BE AVAILABLE BY DEFAULT henec we can't access non static variable from static context[static block, static method and static nested inner class] without creating the object.

```
public class Test
{
	int x = 100;

	public static void main(String[] args)
	{
		System.out.println("x value is :"+x); //error
	}
```

}

Runtime Data areas :

Once a class is loaded into JVM memory then it contains verious types of variable (SV, NSV), methods and blocks.

Based on the type of variable, methods and blocks, It is divided into different memory section i.e different memory
areas which are as follows :

1) Method Area
2) HEAP Area
3) Stack Area
4) PC Register
5) Native Method Stack

1) Method Area :

All the static members are available inside Method Area/Class Area.

Whenever we load .class into JVM memory then class is loaded inside Method area and return java.lang.Class class Object.

Any .class file is loaded into Method Area so, We can get complete information of loaded .class file from the Method area like name of the class, name of package, name of all the static and non static variables, methods and so on.

We have only one Method Area per JVM that means for a single JVM we have only Method area.

WAP to show that we can get complate class information from Method Area.

Methods of java.lang.Class class :
---
getDeclaredMethods() is a predefined non static method available in java.lang.Class class , the return type of this method is Method array where Method is a predefined class available in java.lang.reflect sub package.

public Method[] getDeclaredMethods()

getDeclaredFields() is a predefined non static method available in java.lang.Class class , the return type of this method is Field array where Field is a predefined class available in

java.lang.reflect sub package.

public Field[] getDeclaredFields()

Field and Method both the classes are providing getName() method to get the name of the field and Method.

//WAP to get the information of any class by using command line argument

```
package com.ravi.class_loading;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ClassDescription
{
	public static void main(String[] args) throws Exception
	{
		Class cls = Class.forName(args[0]);

		System.out.println("Class Name is :"+cls.getName());

		System.out.println("Package Name is :"+cls.getPackageName());

		System.out.println("All the fields declared in the class :");

		Field  []fields = cls.getDeclaredFields();

		int count = 0;

		for(Field field : fields)
		{
			System.out.println(field.getName());
			count++;
		}

		System.out.println("Total Number of Fields are :"+count);

		System.out.println("All the methods declared in the class :");

	    Method methods[] = cls.getDeclaredMethods();

	    count = 0;
```

```
            for(Method method : methods)
            {
                  System.out.println(method.getName());
                  count++;
            }

            System.out.println("Total Number of methods are :"+count);


      }

}
```

HEAP AREA :

All the Objects and its member (NSV + NSM) are available in HEAP memory. The Objects which are stored in the HEAP Memory
are eligible for GC when object does not contain any references.

We have only one HEAP Area per JVM.

STACK AREA :

All the methods are executed in Stack Area.
All the local and parameter variables are stored in Stack Area.
Every time we call a method in java then a separate Stack Frame will be created and each stack frame contains 3 parts :

a) Local Variable Array
b) Frame Data
c) Operand Stack

We have multiple Stack Area per JVM.

PC Register :

It stands for Program Counter Register.
In java environment, we can craete multiple threads.
In order to hold the currently executing instruction of each thread we have a separate PC register. [13-MAR]

Native Method Stack :

Native method means, the java methods which are written by using native languages like C and C++. In order to write native method we need native method library support.

Native method stack will hold the native method information in a separate stack.

Execution Engine : [Interpreter + JIT Compiler]

Interpreter

In java, JVM contains an interpreter which executes the program line by line. Interpreter is slow in nature because at the time of execution if we make a mistake at line number 9 then it will throw the execption at line number 9 and after solving the execption again it will start the execution from line number 1 so it is slow in execution that is the reason to boost up the execution java software people has provided JIT compiler.

JIT Compiler :

It stands for just in time compiler. The main purpose of JIT compiler to boost up the execution so the execution of the program will be completed as soon as possible.

JIT compiler holds the repeated instruction like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.

==========

HAS-A Relation in java :

HAS-A Relation :

```
class Order
{
  private int orderId;
  private String itemName;
  private double itemPrice;
}

class Customer
{
  private int customerId;
  private String customerName;
  private String customerAddress;
  private Order order; //HAS-A relation
}
```

If we use a class as a property to another class then it is called HAS-A Relation.

The limitation of IS-A relation is, it is tightly coupled relation i.e IS-A type relation so if we modify the content of super class then it will automatically, It will reflect to all the sub classes.

While working with HAS-A relation we can access the property of another class so HAS-A relation provides accessibility
feature.

HAS-A relation we can achieve by using Association concept.

Association is divided into two types :

1) Composition (Strong Reference)
2) Aggregation (Weak Reference)

Association :

Association is a connection between two separate classes that can be built up through their Objects.

The association builds a relationship between the classes and describes how much a class knows about another class.

This relationship can be unidirectional or bi-directional. In Java, the association can have one-to-one, one-to-many, many-to-one and many-to-many relationships.

Example:-
One to One: A person can have only one PAN card
One to many: A Bank can have many Employees
Many to one: Many employees can work in single department
Many to Many: A Bank can have multiple customers and a customer can have multiple bank accounts.
---
3 files :

```
package com.ravi.association;

public class Student
{
	private int roll;
	private String name;
```

```
private int marks;

public Student(int roll, String name, int marks)
{
	super();
	this.roll = roll;
	this.name = name;
	this.marks = marks;
}

@Override
public String toString()
{
	return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";
}

public int getRoll()
{
	return roll;
}

public void setRoll(int roll)
{
	this.roll = roll;
}

public String getName()
{
	return name;
}

public void setName(String name)
{
	this.name = name;
}

public int getMarks()
{
	return marks;
}

public void setMarks(int marks)
{
```

```
		this.marks = marks;
	}

}

package com.ravi.association;

import java.util.Scanner;

public class Trainer
{
	public static void viewStudentProfile(Student obj)
	{
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter the Student Roll :");
		int roll = sc.nextInt();

		if(roll == obj.getRoll())
		{
			System.out.println(obj);
		}
		else
		{
			System.err.println("Sorry!!! Student data is not available...");
		}

	}


}


package com.ravi.association;

public class AssociationDemo
{
	public static void main(String[] args)
	{
		Student s1 = new Student(101, "Scott", 90);
		Trainer.viewStudentProfile(s1);
	}
}
-----------------------------------------------------------
```

15-03-2025

Composition (Strong reference) :

Composition in Java is a way to design classes such that one class contains an object of another class. It is a way of establishing a "HAS-A" relationship between classes.

Composition represents a strong relationship between the containing class and the contained class.If the containing object (Car object) is destroyed, all the contained objects (Engine object) are also destroyed.

A car has an engine. Composition makes strong relationship between the objects. It means that if we destroy the owner object, its members will be also destroyed with it. For example, if the Car is destroyed the engine will also be destroyed as well.

Program Guidelines :
---
1) One object can't exist without another object
2) We will not create two separate objects, during the creation of Car object, Engine object should be automatically created.
3) We can declare blank final field.

3 files :

Engine.java(C)

```
package com.ravi.composition;

public class Engine
{
  private String engineType;
  private int horsePower;

        public Engine(String engineType, int horsePower)
        {
                super();
                this.engineType = engineType;
                this.horsePower = horsePower;
        }

        @Override
        public String toString()
        {
```

```
		return "Engine [engineType=" + engineType + ", horsePower=" + horsePower +
"]";
	}
}
```

Car.java

```
---------
package com.ravi.composition;

public class Car
{
	private String carName;
	private int carModel;
	private final Engine engine; // HAS-A relation [Blank final Field]

	public Car(String carName, int carModel)
	{
		super();
		this.carName = carName;
		this.carModel = carModel;
		this.engine = new Engine("Battery", 1200);  //Composition
	}

	@Override
	public String toString()
	{
		return "Car [carName=" + carName + ", carModel=" + carModel + ", engine=" +
engine + "]";
	}

}
```

CompositionDemo.java

```
--------------------
package com.ravi.composition;

public class CompositionDemo {

	public static void main(String[] args)
	{
		Car naxon = new Car("Tata Naxon", 2025);
		System.out.println(naxon);
	}
```

}

Aggregation (Weak Referance) :

Aggregation in Java is another form of association between classes that represents a "HAS-A" relationship, but with a weaker bond compared to composition.

In aggregation, one class contains an object of another class, but the contained object can exist independently of the container. If the container object is destroyed, the contained object can still exist.

3 files :

College.java

```
package com.ravi.aggregation;

public class College
{
	private String collegeName;
	private String collegeLocation;

	public College(String collegeName, String collegeLocation)
	{
		super();
		this.collegeName = collegeName;
		this.collegeLocation = collegeLocation;
	}

	@Override
	public String toString()
	{
		return "College [collegeName=" + collegeName + ", collegeLocation=" +
collegeLocation + "]";
	}

}
```

Student.java

```
package com.ravi.aggregation;

public class Student
```

```
{
	private int studentId;
	private String studentName;
	private String studentAddress;
	private College college; // HAS-A Relation

	public Student(int studentId, String studentName, String studentAddress, College
college)
	{
		super();
		this.studentId = studentId;
		this.studentName = studentName;
		this.studentAddress = studentAddress;
		this.college = college;
	}

	@Override
	public String toString() {
		return "Student [studentId=" + studentId + ", studentName=" + studentName + ",
studentAddress=" + studentAddress
				+ ", college=" + college + "]";
	}

}
```

AggregationDemo.java

```
package com.ravi.aggregation;

public class AggregationDemo
{
	public static void main(String[] args)
	{
		College clg = new College("VIT", "Vellore");

		//clg = new College("NIT", "Hyd");

		Student s1 = new Student(1, "Scott", "S R Nagar", clg);
		System.out.println(s1);

		Student s2 = new Student(2, "Smith", "Ameerpet", clg);
		System.out.println(s2);
```

```
                Student s3 = new Student(3, "Alen", "Panjagutta", clg);
                System.out.println(s3);

        }

}
```

Description of System.out.println() :

```
public class System
{
   public final static java.io.PrintStream out = null;  //HAS-A Relation
}
```

System.out.println();

Internally System.out.println() creates HAS-A relation because System class contains a predefined class called java.io.PrintStream as shown in the above example.

The following program describes that how System.out.println() works internally :

```
package com.ravi.has_a_reln;

class Test
{
        static String out = "Hyderabad";  //HAS-A Relation
}

public class HasARelationDemo
{

         public static void main(String[] args)
         {
        int length = Test.out.length();
        System.out.println(length);
         }

}
```

==========