Object-Oriented Programming (OOP)

Types of Programming Languages:

1. Unstructured Programming Languages (USPLs)

- o Outdated, used in early computing.
- No standard structure.
- o Uses mnemonic codes (low-level).
- \circ No functions \rightarrow No code reusability.
- Uses only goto statements for flow control.
- **Examples:** BASIC, FORTRAN.

2. Structured Programming Languages (SPLs)

- o Modern and still relevant.
- Standard structure with functions.
- Uses high-level syntax.
- o Provides better flow control (loops, conditionals).
- Examples: C, PASCAL.

SPL vs OOP

Feature	Structured Programming	Object-Oriented Programming
Approach	Difficult	Simple
Modularization	Less	More
Abstraction	Less	High
Security	Less	High
Shareability	Low	High
Reusability	Low	High

Aspect-Oriented Programming (AOP)

- Not a programming language but a methodology applied to OOP.
- Improves **Shareability** and **Reusability**.
- Separates service logic from business logic.
- Steps:
 - 1. Identify service logic & declare as aspects.
 - 2. Find join points.
 - 3. Inject aspects at runtime.

Object-Oriented Features:

- 1. Class
- 2. Object
- 3. Encapsulation
- 4. Abstraction
- 5. Inheritance

- 6. Polymorphism
- 7. Message Passing

Object-Oriented vs Object-Based Languages

Feature Object-Oriented Languages Object-Based Languages

Supports Inheritance ⋄ Yes **★** No

Examples C++, Java, Python JavaScript

Class vs Object

Feature Class Object

Definition Group of elements with common An individual entity with real

properties & behavior. properties & behavior.

Nature Virtual (blueprint). Real (instance of class).

Encapsulation Virtual encapsulation. Physical encapsulation.

• Multiple objects can be created from a single class.

Encapsulation vs Abstraction

Feature Encapsulation Abstraction

Definition Binding data and code. Hiding unnecessary details, showing only required

functionality.

Purpose Data security and integrity. Simplicity and usability.

Relation Both together provide **Formula:** Encapsulation + Abstraction = Security

security.

Key OOP Concepts

Inheritance

- Transfers variables and methods from one class to another.
- Advantage: Code Reusability.

Polymorphism

- **Definition:** One thing exists in multiple forms.
- Advantage: Flexibility in designing applications.

Message Passing

• Sending data along with execution flow between entities.

• Advantage: Communication & Data Navigation.

Class in Java

Purpose:

• Represents **Entities** such as Employee, Student, Customer, Account, Product, etc.

Components:

- Attributes (Data): Represented as variables inside the class.
 - Example (Student class): studentId, studentName, studentAddress, studentEmailId
- **Behaviors** (Activities): Represented as methods.
 - o Example (Transaction class): deposit(), withdraw(), transferFunds()
 - o Example (Account class): createAccount(), updateAccount(),
 deleteAccount()

Class Syntax:

```
java
CopyEdit
[Access Modifiers] class ClassName [extends SuperClassName]
[implements InterfaceList] {
    // ---- Variables ----
    // ---- Constructors ----
    // ---- Methods -----
    // ---- Blocks -----
    // ---- Classes -----
    // ---- Abstract Classes -----
    // ---- Interfaces -----
    // ---- Enums -----
}
```

Access Modifiers

Purpose:

- 1. **Define Scope:**
 - o **private (Restricted):** Accessible only within the same class.
 - o **default>** (**Package**): Accessible throughout the same package.
 - o **protected:** Accessible in the same package and in subclasses from other packages
 - o **public (Global):** Accessible throughout the application.
- 2. **Provide Extra Nature:** Such as static, final, abstract, native, volatile, transient, synchronized, and strictfp.

Usage in Classes:

- Outer classes can only be declared as public or <default>.
- Inner classes can have all modifiers including public, protected, <default>, and private.
- *Note:* When an access modifier is applied to a class, it affects its members (inner classes, methods, variables) and not the class itself.
 - o *Example:* Declaring a class as private is invalid for outer classes but valid for inner classes.

Additional Examples:

```
java
CopyEdit
class A {
    int i = 10;
    static int j = 20;
    static class B {
    }
}
A a = new A();
System.out.println(a.i);
System.out.println(A.j); // Valid: Accessing static member via ClassName.
```

Inheritance & Interface Implementation in Class Syntax

- **extends:** Represents inheritance (only one superclass allowed).
- **implements:** Allows implementation of one or more interfaces.

Rules:

- A class can be declared:
 - o With extends and without implements
 - o Without extends and with implements
 - o With both, where extends must come before implements.

Valid & Invalid Class Syntax Examples

Syntax	Valid/Invalid	Explanation
class A { }	Valid	Default access is allowed.
<pre>public class A { }</pre>	Valid	Public class is allowed.
<pre>protected class A { }</pre>	Invalid	Outer classes cannot be protected.
<pre>private class A { }</pre>	Invalid	Outer classes cannot be private.
<pre>class A { private class B { } }</pre>	Valid	Inner class can be private.

Syntax	Valid/Invalid	Explanation
<pre>class A { public class B { } }</pre>	Valid	Inner class can be public.
<pre>class A { protected class B { } }</pre>	Valid	Inner class can be protected.
static class A { }	Invalid	Outer classes cannot be static.
final class A { }	Valid	Final classes are allowed.
abstract class A { }	Valid	Abstract classes are allowed.
native class A { }	Invalid	Native is not allowed for classes.
<pre>class A { abstract class B { } }</pre>	Valid	Inner abstract class is allowed.
<pre>class A { static class B { } }</pre>	Valid	Inner static class is allowed.
<pre>class A { volatile class B { } }</pre>	Invalid	Volatile not allowed for classes.
class A extends B { }	Valid	Inheritance is allowed.
<pre>class A extends B, C { }</pre>	Invalid	Multiple inheritance not allowed.
<pre>class A implements I { }</pre>	Valid	Single interface implementation.
<pre>class A implements I1, I2 { }</pre>	Valid	Multiple interface implementations.
<pre>class A implements I extends B { }</pre>	Invalid	Incorrect combination of extends and implements.
<pre>class A extends B implements I { }</pre>	Valid	Correct order: extends first, then implements.
<pre>class A extends B implements I1, I2 { }</pre>	Valid	Correct order with multiple interfaces.
class A extends A { }	Invalid	A class cannot extend itself.
<pre>class A extends B { } class B extends A { }</pre>	Invalid	Circular inheritance is not allowed.

Steps to Utilize Classes in Java Applications

1. Declare a Class:

Use the class keyword.

2. Define Variables & Methods:

Inside the class, add variables and methods as per requirements.

3. Create an Object:

In the main class (within main ()), instantiate the user-defined class.

4. Access Class Members:

Use the created object to access variables and methods.

Example 1: Single Class Usage

java CopyEdit

```
class Employee {
    int eno = 111;
    String ename = "Durga";
    float esal = 50000.0f;
    String eemailId = "durga123@durgasoft.com";
    String emobileNo = "91-9988776655";
    String eaddr = "Hyderabad";
    public void displayEmpDetails() {
         System.out.println("Employee Details");
         System.out.println("----");
        System.out.println("Employee Number : " + eno);
System.out.println("Employee Name : " + ename);
System.out.println("Employee Salary : " + esal);
         System.out.println("Employee Email Id : " +
eemailId);
         System.out.println("Employee Mobile No : " +
emobileNo);
        System.out.println("Employee Address : " + eaddr);
    }
}
class Test {
    public static void main(String[] args) {
         Employee emp = new Employee();
         emp.displayEmpDetails();
    }
}
```

Example 2: Multiple Classes (Student & Customer)

Student.java:

```
java
CopyEdit
public class Student {
    String sid = "S-111";
    String sname = "Durga";
    String saddr = "Hyd";

    public void getStudentDetails() {
        System.out.println("Student Details");
        System.out.println("------");
        System.out.println("Student Id : " + sid);
        System.out.println("Student Name : " + sname);
        System.out.println("Student Address : " + saddr);
    }
}
```

Customer.java:

```
java
CopyEdit
public class Customer {
    String cid = "C-111";
    String cname = "Anil";
    String caddr = "Chennai";
    public void getCustomerDetails() {
        System.out.println("Customer Details");
        System.out.println("----");
        System.out.println("Customer Id : " + cid);
System.out.println("Customer Name : " + cname);
        System.out.println("Customer Address : " + caddr);
    }
}
Test.java:
java
CopyEdit
public class Test {
    public static void main(String[] args) {
        Student student = new Student();
        student.getStudentDetails();
        System.out.println();
        Customer customer = new Customer();
        customer.getCustomerDetails();
    }
}
Expected Output:
markdown
CopyEdit
Student Details
_____
Student Id : S-111
Student Name : Durga
Student Address : Hyd
Customer Details
Customer Id : C-111
Customer Name : Anil
Customer Address : Chennai
```

Q&A: Concrete Methods vs. Abstract Methods

Q) What are the differences between Concrete methods and Abstract methods?

Ans:

- Concrete Method:
 - o A normal Java method with **both** method declaration and implementation.
 - o Can be defined in **classes** and **abstract classes**.
 - o No special keyword is required.
 - o **Shareability:** Provides less shareability.
- Abstract Method:
 - o A Java method with **only** the method declaration (no implementation).
 - o Can be declared in **abstract classes** and **interfaces**.
 - o Must use the abstract keyword.
 - o **Shareability:** Provides more shareability.

Examples:

```
Concrete Method Example:
```

```
java
CopyEdit
class Employee {
    void display() {
        System.out.println("Concrete Method");
    }
}
Abstract Method Example:

java
CopyEdit
abstract class Employee {
    abstract void display();
}
```

Abstract Classes:

- Abstract classes allow both concrete and abstract methods.
- Declared using the abstract keyword.
- Cannot create objects, only reference variables.
- Steps to use:
 - 1. Declare an abstract class using abstract class ClassName {}.
 - 2. Define variables and methods as required.
 - 3. Provide a subclass that implements abstract methods.
 - 4. In main (), create an object of the subclass and access members.

Example:

```
abstract class A{
  void m1() {
      System.out.println("m1-A");
  }
```

```
abstract void m2();
   abstract void m3();
class B extends A{
   void m2(){
       System.out.println("m2-B");
   }
   void m3(){
       System.out.println("m3-B");
   }
}
public class Test {
   public static void main(String[] args) {
       A = new B();
       a.m1();
       a.m2();
       a.m3();
       B b = new B();
       b.m1();
       b.m2();
       b.m3();
   }
}
```

Difference Between Classes and Abstract Classes:

Feature	Classes	Abstract Classes
Methods	Only concrete	Concrete & Abstract
Declaration	class keyword	$\verb"abstract class" keyword"$
Object Creation	Allowed	Not Allowed
Shareability	Less	More

Interfaces:

- Allows only abstract methods.
- Declared using the interface keyword.
- Cannot create objects, only reference variables.
- All variables are public static final by default.
- All methods are public abstract by default.
- Steps to use:
 - 1. Declare an interface using interface InterfaceName {}.
 - 2. Declare variables and methods.
 - 3. Implement the interface in a class.
 - 4. Provide implementations for all abstract methods.
 - 5. In main (), create an object of the implementation class and access members.

Example:

```
interface I{
   int x = 10;
   void m1();
   void m2();
   void m3();
}
class A implements I{
   public void m1(){
       System.out.println("m1-A");
   public void m2(){
       System.out.println("m2-A");
   public void m3(){
       System.out.println("m3-A");
public class Test {
   public static void main(String[] args) {
       I i = new A();
       i.m1();
       i.m2();
       i.m3();
       A = new A();
       a.m1();
       a.m2();
       a.m3();
       System.out.println(I.x);
   }
}
```

Differences Between Classes, Abstract Classes, and Interfaces:

Feature	Classes	Abstract Classes	Interfaces
Methods	Only concrete	Concrete & Abstract	Only abstract
Declaration	class keyword	abstract class keyword	interface keyword
Object Creation	Allowed	Not Allowed	Not Allowed
Variable Defaults	None	None	<pre>public static final</pre>
Method Defaults	None	None	public abstract
Shareability	Low	Medium	High
Constructors	Allowed	Allowed	Not Allowed
Static Blocks	Allowed	Allowed	Not Allowed
Inner Classes	Regular	Regular	Static by default

Implement Services	Partial Implementation	Define Services
DCI VICES		

Methods in Java

Purpose

• Definition:

A method is a set of instructions representing a particular action of an entity. It is executed when the method is invoked.

• Basic Syntax:

```
java
CopyEdit
[Access_Modifiers] [Optional_Modifiers] ReturnType
methodName([Parameter_List]) [throws Exception_List] {
    // method body
}
```

Key Elements:

Access Modifiers:

public, protected, (default), private

Optional Modifiers:

static, final, abstract, native, synchronized, strictfp

o Return Type:

Any primitive type, user-defined type, or void (if no value is returned)

Method Name:

The identifier to recognize and invoke the method

o Parameter List:

Inputs to the method (can be empty)

o Throws Clause:

Used to pass exceptions to the caller

Validity of Method Syntax Examples

Syntax Example	Validity	Explanation
<pre>public void m1(){ }</pre>	Valid	Public access, no return value, no parameters.
<pre>protected void m1(){ }</pre>	Valid	Protected access.
<pre>void m1() { }</pre>	Valid	Default (package-private) access.
<pre>private void m1(){ }</pre>	Valid	Private access.
<pre>static void m1(){ }</pre>	Valid	Static method; belongs to the class.
abstract void m1(){}	Invalid	Abstract methods cannot have a body; must be declared without { }.
abstract void m1();	Valid	Correct abstract method declaration (no body).

Syntax Example	Validity	Explanation
<pre>native void m1();</pre>	Valid	Native method declaration; implemented in external code.
<pre>final void m1(){ }</pre>	Valid	Final method; cannot be overridden.
<pre>final abstract void m1(){ }</pre>	Invalid	Cannot combine final and abstract (final prevents overriding, abstract requires overriding).
<pre>volatile void m1(){ }</pre>	Invalid	volatile is not allowed for methods.
<pre>transient void m1(){ }</pre>	Invalid	transient is not applicable to methods.
<pre>synchronized void m1(){ }</pre>	Valid	Synchronized method; controls access in multi-threaded environments.
<pre>synchronized final void m1(){ }</pre>	Valid	Combination is acceptable.
<pre>strictfp void m1(){ }</pre>	Valid	Ensures strict floating-point calculations.
int m1(){ return 10; }	Valid	Returns an int value.
<pre>float m1() { return 22.22f; }</pre>	Valid	Returns a float value.
<pre>long m1() { return 10; }</pre>	Valid	Returns a long value.
<pre>double m1() { return 22.22f; }</pre>	Valid	Implicit conversion from float to double.
<pre>float m1() { return 22.222; }</pre>	Invalid	The literal 22.222 is considered double; must append f to denote a float literal (e.g. 22.221).
int m1(){ }	Invalid	A non-void method must return a value.
<pre>void m1() { }</pre>	Valid	Void method with no return value.
<pre>void m1() { return 10; }</pre>	Invalid	Void methods cannot return a value.
A m1() { A a = new A(); return a; }	Valid	Returns an object of type A.
A m1() { B b = new B(); return b; }	Invalid	Cannot return an object of type B if method signature expects type A (unless B is a subclass of A).
<pre>void m1(int i, float f){ }</pre>	Valid	Method with two parameters of types int and float.
<pre>void m1(A a) { }</pre>	Valid	Method with a parameter of user-defined type A.
<pre>void m1(void) { }</pre>	Invalid	In Java, void cannot be used as a parameter type.
<pre>void m1() { void m2() { } }</pre>	Invalid	Methods cannot be nested inside other methods.
<pre>void m1() { class A{ } }</pre>	Valid	Local inner classes are allowed inside methods.
<pre>void m1() throws Exception { }</pre>	Valid	Declares that the method may throw an Exception.

void m1()throws
ArithmeticException,
NullPointerException { }

Valid

Declares multiple exceptions in the throws clause (note the correct exception names: ArithmeticException, etc.).

Example: Using Methods in a Java Application

```
java
CopyEdit
class B {
    String convertMessage(String message) {
        String newVal = message.toUpperCase();
        System.out.println(newVal);
        return newVal;
    }
}
class A {
    void m1(String str) {
        System.out.println(str);
    String m2 (String name) {
        String newString = name + "@durgasoft.com";
        return newString;
    }
    int[] m3(int[] values) {
        int[] newValues = new int[values.length];
        for (int i = 0; i < values.length; i++) {</pre>
            newValues[i] = values[i] * 2;
        }
        return newValues;
    }
    String m4(B b) {
        String str = b.convertMessage("durga software
solutions");
        return str;
    }
}
public class Main {
    public static void main(String[] args) {
        A = new A();
        a.m1("Hello m1() Method");
        String str = a.m2("Durga");
        System.out.println(str);
```

```
int[] values = {10, 20, 30, 40, 50};
int[] newValues = a.m3(values);
for (int value : newValues) {
        System.out.print(value + " ");
}
System.out.println();

B b = new B();
String str1 = a.m4(b);
System.out.println(str1);
}
```

Output:

scss
CopyEdit
Hello m1() Method
Durga@durgasoft.com
20 40 60 80 100
DURGA SOFTWARE SOLUTIONS
DURGA SOFTWARE SOLUTIONS

Method Description: Signature vs. Prototype

- Method Signature:
 - o **Definition:**

Includes only the method name and the parameter list.

o Example:

forName(Class cls)

- Method Prototype:
 - o **Definition:**

A more detailed description that includes the access modifiers, return type, method name, parameter list, and the throws clause.

Example:

public static Class forName(Class cls) throws ClassNotFoundException

Methods Based on Object State Manipulation

Mutator Methods

• Purpose:

To modify or set the data in an object.

• Example:

In Java Bean classes, methods like setName(), setAge(), etc., are mutator methods.

Accessor Methods

• Purpose:

To access or retrieve data from an object.

• Example:

In Java Bean classes, methods like getName(), getAge(), etc., are accessor methods.

Example: Java Bean for Encapsulation

Encapsulation is achieved by declaring properties as private and providing public getter and setter methods.

```
java
CopyEdit
class User {
   private String uname;
   private String upwd;
    // Mutator Methods (Setters)
    public void setUname(String userName) {
        this.uname = userName;
   public void setUpwd(String userPassword) {
        this.upwd = userPassword;
    }
    // Accessor Methods (Getters)
    public String getUname() {
        return this.uname;
    public String getUpwd() {
        return this.upwd;
}
public class Main {
    public static void main(String[] args) {
        User user = new User();
        user.setUname("durga");
        user.setUpwd("durga123");
        System.out.println("User Details");
        System.out.println("----");
```

Example: Employee Java Bean

```
java
CopyEdit
class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    // Getter and Setter methods
    public int getEno() {
        return eno;
    public void setEno(int enoParam) {
        eno = enoParam;
    public String getEname() {
        return ename;
    public void setEname(String enameParam) {
        ename = enameParam;
    public float getEsal() {
        return esal;
    public void setEsal(float esalParam) {
        esal = esalParam;
    public String getEaddr() {
        return eaddr;
    }
    public void setEaddr(String eaddrParam) {
        eaddr = eaddrParam;
}
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.setEno(111);
        emp.setEname("Durga");
        emp.setEsal(5000.0f);
```

Output:

```
yaml
CopyEdit
Employee Details
-----
Employee Number : 111
Employee Name : Durga
Employee Salary : 5000.0
Employee Address : Hyd
```

Methods in Java

Key Concepts

• Definition:

A method is a set of instructions representing a particular action of an entity, executed when the method is called.

• Method Components:

Access Modifiers:

public, protected, (default), private
(These define the visibility of the method.)

Optional Modifiers:

static, final, abstract, native, synchronized, strictfp (They modify method behavior, e.g., whether it belongs to a class, can be overridden, or follows strict floating-point rules.)

Return Type:

Specifies the data type of the value returned. Can be any primitive type, user-defined type, or void (if no value is returned).

Method Name:

The identifier used to invoke the method.

Parameter List:

Provides input data to the method. Java methods accept all primitive and user-defined data types.

Throws Clause:

Used to pass exceptions from the method to the caller for handling.

Basic Syntax

```
java
CopyEdit
[Access_Modifiers] [Optional_Modifiers] ReturnType
methodName([Parameter_List]) [throws Exception_List] {
    // method body
}
```

Validity of Method Declarations

Below is a table summarizing various method declaration examples along with their validity:

Syntax Example	Validity	Explanation
<pre>public void m1(){ }</pre>	Valid	Public method with no return value and no parameters.
<pre>protected void m1(){ }</pre>	Valid	Protected access.
<pre>void m1() { }</pre>	Valid	Package-private (default) access.
<pre>private void m1(){ }</pre>	Valid	Private access.
<pre>static void m1(){ }</pre>	Valid	Static method, belongs to the class.
<pre>abstract void m1(){ }</pre>	Invalid	Abstract methods cannot have a body; they should end with a semicolon.
abstract void m1();	Valid	Correct abstract method declaration (no body provided).
<pre>native void m1();</pre>	Valid	Native method declaration (implemented externally).
<pre>final void m1(){ }</pre>	Valid	Final method; cannot be overridden.
<pre>final abstract void m1(){ }</pre>	Invalid	A method cannot be both final (preventing override) and abstract (requiring override).
<pre>volatile void m1(){ }</pre>	Invalid	volatile is applicable to variables, not methods.
<pre>transient void m1(){ }</pre>	Invalid	transient is applicable to variables, not methods.
<pre>synchronized void m1(){ }</pre>	Valid	Synchronized method to control thread access.
<pre>synchronized final void m1() { }</pre>	Valid	Combination of synchronized and final is allowed.
<pre>strictfp void m1(){ }</pre>	Valid	Ensures consistent floating-point calculations.
int m1(){ return 10; }	Valid	Method returning an int value.
<pre>float m1() { return 22.22f; }</pre>	Valid	Method returning a float value.
<pre>long m1() { return 10; }</pre>	Valid	Method returning a long value.

```
Syntax Example
                                        Validity
                                                           Explanation
                                                Returns a float literal, which is
double m1() { return 22.22f; }
                                        Valid
                                                implicitly converted to double.
                                                The literal 22.222 is considered a
float m1() { return 22.222; }
                                        Invalid double; must use 22.222f for a
                                                float return type.
                                                A non-void method must return a
int m1(){}
                                        Invalid
                                                value.
void m1() { }
                                        Valid
                                                Void method with no return value.
void m1() { return 10; }
                                        Invalid Void methods cannot return a value.
A m1() \{ A a = new A(); return \}
                                        Valid
                                                Returns an object of type A.
a; }
                                                Cannot return an object of type B
A m1() \{ B b = new B(); return \}
                                                when the method is declared to
                                        Invalid
b; }
                                                return type A (unless B is a subclass
                                                Method with two parameters: an int
void m1(int i, float f) { }
                                        Valid
                                                and a float.
                                                Method with a parameter of user-
void m1(A a) { }
                                        Valid
                                                defined type A.
                                                In Java, void cannot be used as a
void m1(void) { }
                                        Invalid
                                                parameter type.
                                                Methods cannot be nested inside
void m1() { void m2() { } }
                                        Invalid
                                                other methods.
                                                Local inner classes are allowed
void m1() { class A { } }
                                        Valid
                                                inside a method body.
                                                Declares that the method may throw
void m1() throws Exception { } Valid
                                                an Exception.
void m1() throws
                                                Declares multiple exceptions in the
ArithmeticException,
                                        Valid
                                                throws clause. (Make sure exception
NullPointerException { }
                                                names are correctly capitalized.)
```

Example: Using Methods in a Java Application

```
java
CopyEdit
class B {
    String convertMessage(String message) {
        String newVal = message.toUpperCase();
        System.out.println(newVal);
        return newVal;
    }
}
class A {
    void m1(String str) {
```

```
System.out.println(str);
    }
    String m2 (String name) {
        String newString = name + "@durgasoft.com";
        return newString;
    }
    int[] m3(int[] values) {
        int[] newValues = new int[values.length];
        for (int i = 0; i < values.length; i++) {</pre>
            newValues[i] = values[i] * 2;
        return newValues;
    }
    String m4(B b) {
        String str = b.convertMessage("durga software
solutions");
        return str;
    }
}
public class Main {
    public static void main(String[] args) {
        A = new A();
        a.m1("Hello m1() Method");
        String str = a.m2("Durga");
        System.out.println(str);
        int[] values = {10, 20, 30, 40, 50};
        int[] newValues = a.m3(values);
        for (int value : newValues) {
            System.out.print(value + " ");
        System.out.println();
        B b = new B();
        String str1 = a.m4(b);
        System.out.println(str1);
    }
}
Output:
```

```
scss
CopyEdit
Hello m1() Method
Durga@durgasoft.com
20 40 60 80 100
```

Method Description: Signature vs. Prototype

- Method Signature:
 - o **Definition:** Includes only the method name and its parameter list.
 - o **Example:**

```
forName(Class cls)
```

- Method Prototype:
 - o **Definition:** A detailed description that includes the access modifiers, return type, method name, parameter list, and the throws clause.
 - Example:

```
public static Class forName(Class cls) throws
ClassNotFoundException
```

Types of Methods Based on Object State Manipulation

Mutator Methods

• Purpose:

To modify or set the state (data) of an object.

• Example:

All setXXX () methods in Java Bean classes are mutator methods.

Accessor Methods

• Purpose:

To access or retrieve the state (data) of an object.

• Example:

All getXXX () methods in Java Bean classes are accessor methods.

Example: Java Bean for Encapsulation

User Bean Example:

```
java
CopyEdit
class User {
    private String uname;
    private String upwd;

    // Mutator Methods (Setters)
    public void setUname(String userName) {
        this.uname = userName;
```

```
public void setUpwd(String userPassword) {
       this.upwd = userPassword;
    }
    // Accessor Methods (Getters)
   public String getUname() {
       return this.uname;
   public String getUpwd() {
      return this.upwd;
    }
}
public class Main {
   public static void main(String[] args) {
       User user = new User();
       user.setUname("durga");
       user.setUpwd("durga123");
       System.out.println("User Details");
       System.out.println("----");
       System.out.println("User Name : " +
user.getUname());
       System.out.println("User Password : " +
user.getUpwd());
   }
}
```

Example: Employee Java Bean

```
java
CopyEdit
class Employee {
   private int eno;
   private String ename;
   private float esal;
   private String eaddr;
    // Getter and Setter methods
    public int getEno() {
        return eno;
    public void setEno(int enoParam) {
        eno = enoParam;
    public String getEname() {
        return ename;
    public void setEname(String enameParam) {
        ename = enameParam;
```

```
public float getEsal() {
       return esal;
   public void setEsal(float esalParam) {
       esal = esalParam;
    }
   public String getEaddr() {
       return eaddr;
   public void setEaddr(String eaddrParam) {
       eaddr = eaddrParam;
}
public class Main {
   public static void main(String[] args) {
       Employee emp = new Employee();
       emp.setEno(111);
       emp.setEname("Durga");
       emp.setEsal(5000.0f);
       emp.setEaddr("Hyd");
       System.out.println("Employee Details");
       System.out.println("----");
       System.out.println("Employee Number : " +
emp.getEno());
       System.out.println("Employee Name : " +
emp.getEname());
       System.out.println("Employee Salary
emp.getEsal());
       System.out.println("Employee Address
emp.getEaddr());
    }
```

Output:

```
yaml
CopyEdit
Employee Details
-----
Employee Number : 111
Employee Name : Durga
Employee Salary : 5000.0
Employee Address : Hyd
```

Definition:

A Java Bean is a reusable software component that is used to manage the state of an entity. They are widely used in enterprise applications for tasks such as managing user data, performing validations, handling persistence, and transferring data between application layers (e.g., Controller to View in MVC).

Guidelines for Preparing Java Bean Components

• Class Declaration:

- o Must be declared as public, non-abstract, and non-final.
 - Reasoning:
 - **Public:** To make the bean accessible throughout the application.
 - **Non-abstract:** To allow object creation.
 - **Non-final:** To permit inheritance and improve reusability.

• Serialization:

- Every Java Bean class should implement the java.io.Serializable interface.
 - *Reasoning:* This enables the bean objects to be transported over a network or stored (e.g., session persistence).

• Properties:

- Define variables corresponding to the entity (for example, fields on a user form or columns in a database table).
- All properties must be declared as private to enforce encapsulation.
- o Provide a separate public **setter** and **getter** method for each property.

• Constructor:

o If a constructor is provided, it should be a public, zero-argument constructor.

• Overriding equals () and hashCode ():

 Optionally, override these methods if you need custom comparison or hashcode behavior for bean objects.

Example: Java Bean for a User

```
java
CopyEdit
class User implements java.io.Serializable {
    private String uname;
   private String upwd;
    // Zero-argument constructor
    public User() { }
    // Mutator Methods (Setters)
    public void setUname(String userName) {
        this.uname = userName;
    public void setUpwd(String userPassword) {
        this.upwd = userPassword;
    // Accessor Methods (Getters)
    public String getUname() {
        return this.uname;
    public String getUpwd() {
        return this.upwd;
```

```
public class Main {
    public static void main(String[] args) {
        User user = new User();
        user.setUname("durga");
        user.setUpwd("durga123");
        System.out.println("User Details");
        System.out.println("User Name : " +
        user.getUname());
        System.out.println("User Password : " +
        user.getUpwd());
    }
}
```

Example: Java Bean for an Employee

```
java
CopyEdit
class Employee implements java.io.Serializable {
    private int eno;
   private String ename;
   private float esal;
    private String eaddr;
    // Zero-argument constructor
    public Employee() { }
    // Getter and Setter methods
    public int getEno() {
        return eno;
    public void setEno(int enoParam) {
        eno = enoParam;
    public String getEname() {
        return ename;
    public void setEname(String enameParam) {
        ename = enameParam;
    public float getEsal() {
        return esal;
    public void setEsal(float esalParam) {
        esal = esalParam;
    public String getEaddr() {
```

```
return eaddr;
   public void setEaddr(String eaddrParam) {
       eaddr = eaddrParam;
}
public class Main {
   public static void main(String[] args) {
       Employee emp = new Employee();
       emp.setEno(111);
       emp.setEname("Durga");
       emp.setEsal(5000.0f);
       emp.setEaddr("Hyd");
       System.out.println("Employee Details");
       System.out.println("----");
       System.out.println("Employee Number : " +
emp.getEno());
       System.out.println("Employee Name
emp.getEname());
       System.out.println("Employee Salary
emp.getEsal());
       System.out.println("Employee Address
emp.getEaddr());
    }
}
```

Var-Arg Methods

Definition:

A Var-Arg (variable-argument) method allows you to pass a variable number of arguments to a method. When invoked, the JVM automatically converts the variable arguments into an array.

Key Points

• Syntax:

The var-arg parameter is declared by appending . . . to the data type:

```
java
CopyEdit
void m1(int... numbers) {
    // numbers is an array of int
}
```

• Invocation Examples:

```
    m1(); → Valid
    m1(10); → Valid
    m1(10, 20); → Valid
```

```
o m1 (10, 20, 30); \rightarrow Valid
o m1 (22.22f); \rightarrow Invalid (if the var-arg is declared as int)
```

• JVM Conversion:

When you call a var-arg method, the arguments are collected into an array of the declared type.

Example: Var-Arg Method for Addition

```
java
CopyEdit
class A {
   void add(int... ints) { // ints is an int[] array
       int addResult = 0;
       System.out.println("No Of Arguments : " +
ints.length);
       System.out.print("Argument Values : ");
       for (int value : ints) {
          System.out.print(value + " ");
          addResult += value;
       }
       System.out.println();
       System.out.println("Arguments SUM : " +
addResult);
       System.out.println("----");
   }
}
class Test {
   public static void main(String[] args) {
       A = new A();
                         // 0 arguments
       a.add();
      }
}
```

Expected Output:

```
Argument Values : 10 20
Arguments SUM : 30

No Of Arguments : 3
Argument Values : 10 20 30
Arguments SUM : 60
```

Questions and Answers on Var-Arg Methods

Q: Is it possible to provide normal parameters in a Var-Arg method?

Yes, it is possible to include normal parameters along with a var-arg parameter. However, the normal parameters must come **before** the var-arg parameter since the var-arg must always be the last parameter.

Q: Is it possible to provide more than one Var-Arg parameter in a single method? **Ans:**

No, a method cannot have more than one var-arg parameter because only the last parameter can be variable. If you try to declare more than one, the compiler will generate an error indicating that the var-arg parameter must be the last parameter.

Examples:

• Incorrect Declaration (More than one var-arg parameter):

```
java
CopyEdit
class A {
    // Error: varargs parameter must be the last
parameter
    void m1(int... i, float... f) {
        System.out.println("Var-Arg Method");
    }
}
```

• Correct Declaration (Normal parameter before var-arg):

```
java
CopyEdit
class A {
    void m1(float f, int... i) {
        System.out.println("Var-Arg Method");
    }
}
public class Test {
    public static void main(String[] args) {
        A a = new A();
```

```
a.m1(22.22f);
a.m1(22.22f, 10);
a.m1(22.22f, 10, 20);
a.m1(22.22f, 10, 20, 30);
}
```

In the above code, the normal parameter float f comes first, followed by the var- arg int... i.

Object Creation Process in Java

Why Create Objects?

• Java is Object Oriented:

Every operation in a Java application is performed via objects.

• Data Storage:

Objects store entity data temporarily during program execution.

• Member Access:

To access instance members (variables and methods), an object is required.

Object Creation Syntax

```
java
CopyEdit
ClassName refVar = new ClassName([ParamValues]);
```

- Explanation:
 - o new ClassName ([ParamValues]) calls the constructor of the class.
 - o **Example:**

```
java
CopyEdit
class A {
    // class members...
}
A a = new A();
```

JVM Actions During Object Creation

When the statement new ClassName ([ParamValues]) is executed, the JVM performs the following steps:

1. Class Loading:

- o JVM locates the class (via the current directory, predefined libraries, or locations in the classpath).
- o The class bytecode is loaded into the **Method Area**.

 A corresponding java.lang.Class object is created in the heap containing metadata (class name, modifiers, superclass, interfaces, variables, methods, etc.).

2. Memory Allocation:

- The JVM determines the minimal object size based on the instance variables and their data types.
- o The Heap Manager allocates memory for the new object.
- o A unique identity (hashcode) is generated as an integer.

3. Reference Creation:

- The integer hashcode is converted to its hexadecimal form, which becomes the **reference value**.
- o This reference value is assigned to the reference variable.

4. Instance Variable Initialization:

- o Memory is allocated for all instance variables.
- Variables are initialized using:
 - Explicit initializations at the class level,
 - Initialization within the constructor,
 - **Default values** based on data type if no explicit initialization is provided.

Working with Object Methods

Obtaining the Hashcode and Reference String

- hashCode():
 - o Method: public native int hashCode()
 - o **Purpose:** Returns a hashcode (unique identity) for the object.
 - o *Note:* A native method declared in Java and implemented in a non-Java language.

• toString():

- o Method: public String toString()
- o **Purpose:** Returns a string representation of the object.
- **Default Behavior:**

In the default implementation (from java.lang.Object), it returns a string in the form:

css
CopyEdit
ClassName@HexHashCode

Usage:

When an object is passed to System.out.println(), the JVM automatically calls toString() on that object.

Example Code: Using hashCode() and toString()

java CopyEdit

```
class A {
    // Class definition (no members for simplicity)
}

public class Main {
    public static void main(String[] args) {
        A a = new A();
        int hc = a.hashCode();
        System.out.println("Hashcode : " + hc);
        String ref = a.toString();
        System.out.println("Ref Value : " + ref);
    }
}

Sample Output:

sql
CopyEdit
Hashcode : 2055281021
Ref Value : A@7a81197d
```

Overriding toString() for Custom Output

If you do not want the default object reference display, you can override the toString() method in your class.

Example: Custom toString() in an Account Class

```
java
CopyEdit
class Account {
   String accNo = "abc123";
   String accHolderName = "Durga";
   String accType = "Savings";
   long balance = 50000L;
   public String toString() {
       System.out.println("Account Details");
       System.out.println("----");
       System.out.println("Account Number : " + accNo);
       System.out.println("Account Holder Name : " +
accHolderName);
       System.out.println("Account Type : " + accType);
System.out.println("Account Balance : " + balance);
       return "----";
   }
}
public class Main {
```

Inheritance and Object Creation

• Default Superclass:

Every class in Java implicitly extends java.lang.Object if no other superclass is specified.

- Multiple Inheritance Concern:
 - o Scenario:

If a class explicitly extends a superclass, it still indirectly inherits from java.lang.Object.

o Clarification:

This is **multi-level inheritance**, not multiple inheritance. Java does not support multiple inheritance (i.e., extending two classes directly).

Predefined Classes and Their toString() Methods

Some predefined Java classes override toString() to provide more meaningful output:

- **String:** Displays the string value.
- **StringBuffer:** Displays the content.
- Exception classes: Display exception details.
- Thread, Wrapper classes, and Collection classes: Provide their own specific implementations.

Example:

```
java
CopyEdit
import java.util.ArrayList;
```

```
public class Main {
   public static void main(String[] args) {
       String str = new String("Welcome To String
Manipulations");
       System.out.println(str);
       ArithmeticException exception = new
ArithmeticException ("My Arithmetic Exception");
       System.out.println(exception);
       Thread thread = new Thread();
       System.out.println(thread);
       ArrayList list = new ArrayList();
       list.add(10);
       list.add(20);
       list.add(30);
       list.add(40);
       System.out.println(list);
   }
}
Sample Output:
mathematica
CopyEdit
Welcome To String Manipulations
java.lang.ArithmeticException: My Arithmetic Exception
Thread[Thread-0,5,main]
[10, 20, 30, 40]
```

Types of Objects in Java

Immutable Objects

• Definition:

Once created, the data inside an immutable object cannot be changed.

o Behavior:

If modifications are attempted, a new object is created instead.

- Examples:
 - o **String class** objects are immutable.
 - o Wrapper classes objects (e.g., Integer, Double) are immutable.

Mutable Objects

• Definition:

The content of mutable objects can be changed directly.

• Examples:

- o **StringBuffer class** objects are mutable.
- By default, many user-defined objects are mutable unless explicitly made immutable.

Code Example: Immutable vs. Mutable Objects

```
java
CopyEdit
public class Main {
   public static void main(String[] args) {
       // Immutable Example with String
       String str1 = new String("Durga ");
       String str2 = str1.concat("Software ");
       String str3 = str2.concat("Solutions");
       System.out.println(str1); // Durga
       System.out.println(str2); // Durga Software
       System.out.println(str3); // Durga Software Solutions
       System.out.println(str1 == str2); // false
       System.out.println(str2 == str3); // false
       System.out.println(str3 == str1); // false
       System.out.println();
       // Mutable Example with StringBuffer
       StringBuffer sb1 = new StringBuffer("Durga ");
       StringBuffer sb2 = sb1.append("Software ");
       StringBuffer sb3 = sb2.append("Solutions");
       System.out.println(sb1); // Durga Software Solutions
       System.out.println(sb2); // Durga Software Solutions
       System.out.println(sb3); // Durga Software Solutions
       System.out.println(sb1 == sb2); // true
       System.out.println(sb2 == sb3); // true
       System.out.println(sb3 == sb1); // true
   }
}
```

Observations:

- Immutable (String): Each change creates a new object.
- Mutable (StringBuffer): Changes modify the same object.

Difference Between Object and Instance

• Object:

A block of memory that holds data (state) and behavior (methods). It is the actual entity created using the new keyword.

• Instance:

A specific copy or occurrence of an object at a particular point in time. An object may

have multiple instances over its lifetime (for example, after state changes), but when you refer to an object, you always access its latest state.

• Clarification:

An object represents the blueprint in memory, whereas an instance represents the concrete manifestation (current state) of that object.

Constructors in Java

Definition and Role

• Definition:

A constructor is a special block of code used to initialize new objects.

• Role in Object Creation:

- o **Initialization:** Provides initial values to instance (class-level) variables.
- Timing: Executed exactly at the time an object is created (not before or after).
- o **Naming:** Must have the same name as the class.
- o **Return Type:** Do not have any return type (not even void).

Constructor Rules and Syntax

Basic Syntax

```
java
CopyEdit
[Access_Modifier] ClassName([Parameter_List]) [throws
Exception_List] {
    // instructions (initializations)
}
```

Key Points

Access Modifiers Allowed:

public, protected, (default), private *They control the visibility of the constructor.*

• Modifiers Not Allowed:

Cannot use static, final, abstract, etc., with constructors.

• Throws Clause:

A constructor can include a throws clause to pass exceptions to the caller.

• Naming Requirement:

The constructor's name must exactly match the class name; otherwise, the compiler treats it as a method (which then must have a return type).

Examples

Valid Constructor

```
java
CopyEdit
class A {
    // A valid, no-argument constructor
    A() {
        System.out.println("A-Con");
    }
}

public class Main {
    public static void main(String[] args) {
        A a = new A(); // Output: A-Con
    }
}
```

Invalid Constructor Name Example

```
java
CopyEdit
class A {
    // Incorrect: Constructor name does not match the class
name
    // This will raise "invalid method declaration; return type
required"
    B() {
        System.out.println("A-Con");
    }
}
```

Constructor with Return Type (Treated as a Normal Method)

```
java
CopyEdit
class A {
   // This is not a constructor but a method named A (with a
void return type)
   void A() {
       System.out.println("A-Con");
   }
}
public class Test {
   public static void main(String[] args) {
       A a = new A(); // Creates object with default
constructor (if available)
                       // Calls the method A(), not the
       a.A();
constructor
   }
}
```

Modifiers Not Allowed with Constructors

```
java
CopyEdit
class A {
    // Error: static modifier not allowed with constructor
    static A() {
        System.out.println("A-Con");
    }
}
```

Default vs. User-Defined Constructors

Default Constructor

• Definition:

If no constructor is explicitly provided, the compiler adds a 0-argument constructor (default constructor).

• Example:

```
java
CopyEdit
public class Test {
    // No constructor provided by the developer.
}
// Compiler adds: public Test() { }
```

• Note:

If at least one constructor is explicitly defined, the compiler does not create a default constructor.

User-Defined Constructor

Types:

0-Arg Constructor:

A constructor with no parameters.

Parameterized Constructor:

A constructor that accepts one or more parameters to initialize objects with custom values.

Example: Parameterized Constructors

```
java
CopyEdit
class Account {
   String accNo;
   String accHolderName;
   String accType;
   long accBalance;
```

```
// Parameterized constructor
   Account (String acc No, String acc Holder Name, String
acc Type, long acc Balance) {
       accNo = acc No;
       accHolderName = acc Holder Name;
       accType = acc Type;
       accBalance = acc Balance;
   }
   public void getAccountDetails() {
       System.out.println("Account Details");
       System.out.println("----");
System.out.println("Account Number : " + accNo);
       System.out.println("Account Holder Name : " +
accHolderName);
       System.out.println("Account Type : " + accType);
System.out.println("Account Balance : " +
accBalance);
  }
public class Main {
   public static void main(String[] args) {
       Account account1 = new Account ("a111", "Durga",
"Savings", 50000L);
       account1.getAccountDetails();
       System.out.println();
       Account account2 = new Account ("a222", "Venkat",
"Savings", 60000L);
       account2.getAccountDetails();
       System.out.println();
       Account account3 = new Account ("a333", "Ramana",
"Savings", 40000L);
       account3.getAccountDetails();
   }
}
Sample Output:
yaml
CopyEdit
Account Details
_____
Account Number : a111
Account Holder Name : Durga
Account Type : Savings
Account Balance : 50000
```


Reasoning:

Using parameterized constructors allows providing object-specific data at creation time. This is especially important in applications (like banking) where account details must be set during object creation, rather than later using setter methods.

Constructor Overloading

Definition:

Constructor overloading is the concept of having more than one constructor in a class with the same name (which is the class name) but different parameter lists.

Advantages:

- Provides flexibility in object creation.
- Allows initializing objects in different ways based on the available data.

Example: Constructor Overloading with a Student Class

```
java
CopyEdit
class Student {
   String sid;
   String sname;
   String semail;
   String smobile;

   // 0-Arg Constructor
   Student(String studentId) {
      sid = studentId;
   }
   // 2-Arg Constructor
   Student(String studentId, String studentName) {
      sid = studentId;
   }
```

```
sname = studentName;
   }
   // 3-Arg Constructor
   Student(String studentId, String studentName, String
studentEmail) {
       sid = studentId;
        sname = studentName;
        semail = studentEmail;
   }
   // 4-Arg Constructor
   Student(String studentId, String studentName, String
studentEmail, String studentMobile) {
        sid = studentId;
       sname = studentName;
        semail = studentEmail;
        smobile = studentMobile;
   }
   public void getStudentDetails() {
        System.out.println("Student Details");
        System.out.println("----");
       System.out.println("Student Id : " + sid);
System.out.println("Student Name : " + sname);
System.out.println("Student Email : " + semail);
System.out.println("Student Mobile No : " + smobile);
   }
}
public class Main {
   public static void main(String[] args) {
        Student s1 = new Student("S-111");
        s1.getStudentDetails();
        System.out.println();
        Student s2 = new Student("S-222", "Durga");
        s2.getStudentDetails();
       System.out.println();
        Student s3 = new Student("S-333", "Anil",
"anil@dss.com");
        s3.getStudentDetails();
        System.out.println();
        Student s4 = new Student ("S-444", "Ramesh",
"ramesh@dss.com", "91-9988776655");
        s4.getStudentDetails();
   }
}
```

Another Example with a Simple Class

```
java
CopyEdit
class A {
   int i, j, k;
   // Default 0-Arg Constructor
   A() { }
   // Overloaded Constructors
   A(int x) {
      i = x;
   }
   A(int x, int y) {
      i = x;
       j = y;
   A(int x, int y, int z) {
       i = x;
       j = y;
```

```
k = z;
   }
   public void add() {
       System.out.println("ADD : " + (i + j + k));
   }
}
public class Main {
   public static void main(String[] args) {
       A a1 = new A();
       al.add(); // Output: ADD
                                : 0
       A = a2 = new A(10);
       a2.add(); // Output: ADD
                                   : 10
       A = a3 = new A(10, 20);
       a3.add(); // Output: ADD
                                   : 30
       A = 4 = 10, 20, 30;
                                   : 60
       a4.add(); // Output: ADD
   }
}
```

Summary

- Constructors are special methods for initializing new objects.
- They must share the same name as the class, lack a return type, and cannot be static or final.
- **Default Constructors** are provided by the compiler if no constructor is explicitly declared.
- **User-Defined Constructors** allow customization—either with no parameters (0-arg) or with parameters (parameterized).
- **Constructor Overloading** enables multiple constructors with different parameter lists, increasing flexibility in object creation.

The this Keyword in Java

Definition:

this is a keyword that represents the current class object. It is used to differentiate between class-level members and local variables, access methods, invoke constructors, and return the current object.

1. Referring to Current Class Variables

• Purpose:

Use this to access instance variables when local variables (e.g., parameters) have the same names.

• Syntax:

```
java
CopyEdit
this.varName;
```

• Example:

```
java
CopyEdit
class A {
   int i = 10;
   int j = 20;
   A(int i, int j) { // Local parameters: i = 50, j = 60
        System.out.println("Local Vars : " + i + "
" + j);
        // 50
                  60
        System.out.println("Class Vars : " + this.i +
      " + this.j); // 10
                            20
    }
public class Main {
   public static void main(String[] args) {
       A = new A(50, 60);
    }
}
```

• Java Bean Context:

In setter methods, this assigns the parameter value to the class-level variable.

```
java
CopyEdit
class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;

    public void setEno(int eno) {
        this.eno = eno;
    }
    // Similar setters for ename, esal, eaddr...
}
```

2. Referring to Current Class Methods

• Purpose:

Call other instance methods from the same class using this.

• Syntax:

```
java
CopyEdit
this.methodName([ParamValues]);
```

• Example:

```
java
CopyEdit
class A {
    void m1() {
        System.out.println("m1-A");
        m2();
                       // Direct call
                      // Using 'this'
        this.m2();
    }
    void m2() {
        System.out.println("m2-A");
    }
}
public class Main {
    public static void main(String[] args) {
        A = new A();
        a.m1();
    }
}
```

Output:

```
css
CopyEdit
m1-A
m2-A
m2-A
```

3. Referring to Current Class Constructors

• Purpose:

Invoke another constructor of the same class from within a constructor (constructor chaining).

• Syntax:

- o Must be the **first statement** in the constructor.
- **Example of Constructor Chaining:**

```
java
CopyEdit
class A {
   A() {
        this(10); // Calls the int-parameter constructor
        System.out.println("A-Con");
    A(int i) {
        this(22.22f); // Calls the float-parameter
constructor
        System.out.println("A-int-param-con");
   A(float f) {
        this (33.3333); // Calls the double-parameter
constructor
        System.out.println("A-float-param-con");
   A(double d) {
        System.out.println("A-double-param-con");
    }
}
class Test {
   public static void main(String[] args) {
        A = new A();
    }
}
```

```
css
CopyEdit
A-double-param-con
A-float-param-con
A-int-param-con
A-Con
```

- Rules for Using this with Constructors:
 - o The this () call must be the **first statement** in the constructor.
 - o It can be used only within a constructor (not in regular methods).
- Q: Is it possible to access more than one constructor using this in a single constructor?

A:

No. Since the this () call must be the first statement, you cannot have more than one such call in a constructor. Any additional call will result in a compile-time error.

Invalid Example:

```
java
CopyEdit
```

```
class A {
    A() {
        this(10);
        this(22.22f); // Error: call to this must be the
first statement
        System.out.println("A-Con");
    }
    A(int i) {
        System.out.println("A-int-param-con");
    }
    A(float f) {
        System.out.println("A-float-param-con");
    }
}
```

4. Returning the Current Class Object

• Purpose:

Return the current class object from a method, typically used for method chaining.

• Syntax:

```
java
CopyEdit
return this;
```

• Example:

```
java
CopyEdit
class A {
   A getRef1() {
        A a = new A();
        return a;
    }
    A getRef2() {
        return this;
    }
}
public class Main {
    public static void main(String[] args) {
        A = new A();
        System.out.println(a);
                                          // Prints:
A@<hashcode>
        System.out.println(a.getRef1());  // New object
each call: different hashcode
        System.out.println(a.getRef1());
                                         // Same
        System.out.println(a.getRef2());
object: same hashcode
        System.out.println(a.getRef2());
```

```
}
```

Explanation:

- o **getRef1()** creates and returns a new object on each call.
- o **getRef2()** returns the same object reference (this), reducing duplicate objects.

Method Chaining

• Definition:

The process of invoking multiple methods on the same object in a single statement by having each method return the current object (this).

• Example:

```
java
CopyEdit
class A {
   A m1() {
        System.out.println("m1-A");
        return this;
    }
    Am2()
        System.out.println("m2-A");
        return this;
   A m3() {
        System.out.println("m3-A");
        return this;
    }
public class Main {
   public static void main(String[] args) {
        A = new A();
        a.m1().m2().m3(); // Chained method calls on the
same object
    }
}
```

Output:

```
css
CopyEdit
m1-A
m2-A
m3-A
```

Static Keyword in Java

In Java, the **static** keyword is used to increase shareability by allowing a member to belong to the class rather than to any individual instance. It has four primary applications:

- 1. Static Variables
- 2. Static Methods
- 3. Static Blocks
- 4. Static Import

Each use case contributes to reducing memory overhead and improves the efficiency of code sharing.

1. Static Variables

Definition & Initialization:

- **Static variables** (or class variables) are initialized when the class bytecode is loaded into memory.
- They exist as a single copy, shared among all instances (objects) of the class.

Key Characteristics:

- Memory Location: Stored in the Method Area.
- **Shareability:** Since they are part of the class definition, any modification reflects across all objects.
- **Access:** They can be accessed by both class reference and object reference, although using the class name is the recommended practice.
- No Instance-Dependent: Even if an object reference is null, accessing a static variable will not trigger a NullPointerException.

Code Example:

```
java
CopyEdit
class User {
    String uname;
    String uaddr;
    int uage;
    String uemail;
    String umobile;
    public static final int MIN_AGE = 18;
    public static final int MAX_AGE = 25; // Constants are
typically static
```

```
public User (String uname, String uaddr, int uage, String
uemail, String umobile) {
        this.uname = uname;
        this.uaddr = uaddr;
        this.uage = uage;
        this.uemail = uemail;
        this.umobile = umobile;
    public void getUserDetails() {
        System.out.println("User Details");
        System.out.println("----");
        System.out.println("User Name
                                               : " + uname);
        System.out.println("User Address : " + uaddr);
                                               : " + uage);
        System.out.println("User Age
                                         : " + uemail);
        System.out.println("User Email
        System.out.println("User Mobile Number : " + umobile);
        System.out.println("User Min Age : " + MIN_AGE);
System.out.println("User Max Age : " + MAX_AGE);
    }
public class Main {
    public static void main(String[] args) {
        User user1 = new User("Durga", "Hyd", 23,
"durga@gmail.com", "91-9988776655");
        user1.getUserDetails();
        System.out.println();
        User user2 = new User("Venkat", "Chennai", 22,
"venkat@gmail.com", "91-5566778899");
       user2.getUserDetails();
    }
}
```

Explanation:

- The constants MIN_AGE and MAX_AGE are declared as static. This ensures they are loaded once and shared among every User instance.
- The method getUserDetails () prints out both instance-specific and static data, highlighting the difference in how they are accessed.

2. Static Methods

Key Points:

- **Static methods** belong to the class and can be called without creating an instance.
- They can access static variables directly, but cannot directly access instance variables or methods (since instance data is tied to specific objects).
- Useful for utility or helper methods where instance data is not required.

Example:

```
java
CopyEdit
class MathUtil {
    public static int add(int a, int b) {
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        int sum = MathUtil.add(5, 10);
        System.out.println("Sum: " + sum);
    }
}
```

3. Static Blocks

Purpose:

- Static blocks are used for initializing static variables that require more complex logic.
- They execute only once when the class is loaded.

Example:

Explanation:

- The static block runs before any instance of the class is created.
- It ensures that CONFIG_VALUE is properly initialized regardless of when or how the class is referenced.

4. Static Import

Purpose:

- **Static import** allows direct access to static members (variables and methods) of a class without qualifying them with the class name.
- This can improve readability in some contexts but should be used judiciously to avoid confusion.

Example:

```
java
CopyEdit
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
public class Main {
    public static void main(String[] args) {
        System.out.println("Value of PI: " + PI);
        System.out.println("Square root of 16: " + sqrt(16));
    }
}
```

Explanation:

• With static import, you can use PI and sqrt directly, enhancing code clarity in math-intensive computations.

Differences Between Static and Instance Variables

The following table outlines the main differences:

Feature	Static Variable	Instance Variable
Initialization	At class loading time	When each object is created
Declaration	Must use the static keyword	No static keyword required
Memory Allocation	Stored in the Method Area	Stored in the Heap (within object)
Sharing	One copy shared across all objects	Each object has its own copy

Feature	Static Variable	Instance Variable
Modification Impact	A change affects all instances	A change affects only that specific instance
Access Method	Accessible using class name or object reference	Only accessible via object reference
Null Reference	Accessing via a null reference does not throw an exception	Accessing via a null reference results in a NullPointerException
Usage in Methods	Available in both static and instance methods	Available only in instance methods

Detailed Reasoning

1. Shareability & Memory Efficiency:

- Static Variables: Shareability is one of their main benefits. Since there's a single copy per class, it reduces memory overhead. For example, constants (like MIN_AGE and MAX_AGE) should be static because their value is universal across instances.
- Instance Variables: While they allow each object to maintain its own state, this comes at the cost of memory as each object holds a separate copy.

2. Access Patterns:

- Static Methods: Can be called without an instance, making them ideal for utility functions (like mathematical operations).
- Instance Methods: Must be called on an object and operate on instance data, making them suitable for behaviors that depend on the state of an object.

3. Initialization Order:

- Static Blocks: Ensure that complex static variable initialization happens only once when the class is loaded.
- Instance Initialization: Instance variables are set up when the constructor is invoked, which allows them to be initialized based on input or context at runtime.

4. Error Handling:

 Null Reference Behavior: Accessing static members through a null object reference still works because the member is associated with the class. In contrast, trying to access an instance variable on a null reference will lead to a NullPointerException.

Conclusion

Understanding the differences between static and instance variables, as well as the various ways to use the static keyword, is crucial in Java for writing efficient and maintainable code. By knowing when to use each type of variable or method, developers can better manage memory usage, avoid common errors, and structure their programs in a way that promotes clarity and reusability.

Static Methods, Blocks, and Import in Java

Java provides several ways to use the static keyword to control execution timing, improve shareability, and simplify code access. The following sections detail the usage and behavior of static methods, blocks, and imports along with some interesting questions and answers.

1. Static Methods

Definition & Behavior

- Static Method: A regular Java method marked with the static keyword.
- When Executed: Recognized and executed as soon as it is accessed (load time or runtime).
- Access Limitations:
 - Can directly access only static members (variables or other static methods)
 of the same class.
 - Cannot directly access instance members; to do so, you must create an object.
 - Cannot use the this keyword inside its body.

Access Pattern:

- o Can be called using the class name or an object reference.
- o If accessed using a reference that is null, no NullPointerException is raised (unlike instance methods).

Example

```
java
CopyEdit
class A {
   int i = 10;
   static int j = 20;

   static void m1() {
       System.out.println("m1-A");
       // System.out.println(i); // Error: Cannot access
instance variable directly
```

```
System.out.println(j);
        A = new A();
        System.out.println(a.i);
        // System.out.println(this.j); // Error: 'this' not
allowed in static context
   void m2() {
        System.out.println("m2-A");
                      // Calling static method from an
       m1();
instance method is allowed.
       this.ml(); // Also allowed.
}
public class Main {
   public static void main(String[] args) {
        A a1 = new A();
        al.ml(); // Calling using object reference
                 // Recommended: Calling using class name
        A.m1();
        a1.m2();
       A a2 = null;
        // a2.m2(); // Would throw
java.lang.NullPointerException because m2 is an instance
method.
        a2.m1(); // Works fine because m1() is static.
    }
}
```

Reasoning & Key Points

- Access Restrictions: Static methods cannot use this because there's no instance context.
- **Null Reference Safety:** Calling a static method via a null reference is safe because static methods are tied to the class, not the instance.

2. Displaying Text Without the main () Method

A. Using Static Variable and Static Method Combination (Valid in Java 6)

Q: Is it possible to display a line of text on the command prompt without using the main() method?

Ans: Yes, by combining a static variable and a static method. **Mechanism in Java 6:**

- When the class is loaded, static variables are initialized.
- The static method is called during this initialization.
- If the static method calls System.exit(0), the program terminates immediately after displaying the message.

```
Example
```

```
java
CopyEdit
class Test {
    static int i = m1();
    static int m1() {
        System.out.println("Welcome to Durgasoft!");
        System.exit(0); // Terminates the program immediately return 10;
    }
}
```

Note:

- **Java 6 Behavior:** JVM loads the class bytecode without checking for the main () method. The static initialization executes and displays the message.
- Java 7 Onward: JVM first checks for the existence of main (). Without a main method, it displays an error:

```
typescript
CopyEdit
Error: Main method not found in class Test, please define
the main method as:
   public static void main(String[] args)
```

B. Using a Static Block

Q: Is it possible to display a line of text on the console without using the main() method, static variable, and static method?

Ans: Yes, by using a **static block**.

Mechanism in Java 6:

- The static block executes when the class is loaded.
- Using System.exit(0) in the static block terminates the program after displaying the message.

Example

```
java
CopyEdit
class Test {
    static {
```

```
System.out.println("Welcome to Durgasoft!");
System.exit(0);
}
```

Note:

- Java 6: The static block executes as part of class loading.
- **Java 7 Onward:** The absence of a main() method prevents the class from loading, and JVM reports an error about the missing main method.

C. Using Static Anonymous Inner Classes of the Object Class

Q: Is it possible to display a line of text on the command prompt without using the main() method, the static variable-static method combination, and the static block?

Ans: Yes, by using **static anonymous inner classes** of the Object class.

Note:

- **Java 6:** The class loads and the anonymous inner class initializer runs, printing the message.
- **Java 7 Onward:** Without a main method, the JVM does not load the class, and it results in an error regarding the missing main method.

3. Static Blocks

Definition & Behavior

- **Static Block:** A block of code marked by the static keyword that executes when the class bytecode is loaded.
- Usage Limitations:

- Cannot directly access instance variables; to access instance members, you must create an object.
- Cannot use the this keyword.

Example

```
java
CopyEdit
class A {
   int i = 10;
    static int j = 20;
    static {
        System.out.println("SB-A");
        // System.out.println(i); // Error: Cannot access
instance variable
        System.out.println(j);
        A = new A();
        System.out.println(a.i);
        // System.out.println(this.j); // Error: 'this' not
allowed
    }
}
public class Main {
   public static void main(String[] args) {
        A a = new A(); // Static block has already executed at
class loading time.
}
```

Key Points

- Execution Timing: Static blocks execute only once when the class is loaded.
- **Purpose:** They are used for complex initialization of static variables.

4. Static Import

Definition & Benefits

- **Static Import:** Enables direct usage of static members (variables or methods) of a class without qualifying them with the class name.
- Syntax:
 - Import all static members:

```
java
CopyEdit
import static packageName.ClassName.*;
```

o Import a specific static member:

```
java
CopyEdit
import static packageName.ClassName.memberName;
```

Usage Scenario

• **Typical Use:** To improve code readability, especially when many static members are used (e.g., constants in java.lang.Thread).

Example:

```
java
CopyEdit
import static java.lang.Thread.*; // Imports all static
members from Thread
import static java.lang.System.out; // Imports the static
member 'out' from System

class Test {
    public static void main(String[] args) {
        out.println(MIN_PRIORITY);
        out.println(NORM_PRIORITY);
        out.println(MAX_PRIORITY);
    }
}
```

Explanation:

- With the static import, constants like MIN_PRIORITY, NORM_PRIORITY, and MAX_PRIORITY from java.lang. Thread can be accessed directly without the class name prefix.
- Similarly, System.out is imported statically, allowing direct usage of out.println().

Summary Table: Key Differences and Behaviors

Feature	Description & Behavior
Static Method	- Executes when accessed (load or runtime)- Can only directly access static members- this keyword is not allowed
Static Variable & Static Method Combo	 Used to display text without a main () method in Java 6 JVM loads class bytecode, executes static initializer, then exits

Feature	Description & Behavior	
Static Block	Executes when class is loadedUsed for complex initializationCannot access instance variables without creating an object	
Static Anonymous Inner Class	 An alternative technique for executing code during class loading in Java 6 Similar limitations as static block, relies on initializer 	
Static Import	 Simplifies code by allowing direct access to static members Improves readability by omitting the class name in code references 	

Final Reasoning

- Java 6 vs. Java 7 Onward:
 - In Java 6, the JVM loads the class bytecode without checking for a main ()
 method, so static initializations (via variables, blocks, or anonymous inner
 classes) execute and can display output.
 - o From Java 7 onward, the JVM checks for the main () method first. If it isn't found, the class isn't loaded and an error is thrown.
- When to Use Each Technique:
 - Static Methods & Blocks: Use them for shared functionality and one-time initialization.
 - Static Import: Use when many static members are required for clarity or to reduce redundancy.
 - Static Anonymous Inner Classes: A creative (but less common) way to execute code on class load.

Static Context / Static Flow of Execution in Java

When a Java class is loaded into memory, a special environment called the **Static Context** is created. This context is responsible for handling all loading-time activities and consists of three elements:

- 1. Static Variables
- 2. Static Methods
- 3. Static Blocks

Key Points:

• Static Variables & Static Blocks:

- Are recognized and executed during the class-loading phase.
- o Execute only once regardless of the number of objects created.

• Static Methods:

 Are recognized during class loading but are executed only when they are explicitly accessed.

How Static Context Works

When a class is loaded:

- The JVM initializes static variables and executes static blocks in the order they appear in the class.
- The order of execution among static members (variables and blocks) is as they appear in the source code.
- Once the static context is set up, instance variables, instance blocks, and the constructor execute later at the time of object creation.

Example 1

```
java
CopyEdit
class A {
   static {
       System.out.println("SB-A");
   static int i = m1();
   static int m1(){
       System.out.println("m1-A");
       return 10;
   }
}
public class Main {
   public static void main(String[] args) {
       A = new A();
   }
}
```

Output:

```
css
CopyEdit
SB-A
m1-A
```

Explanation:

- Static Block: Executes first and prints SB-A.
- Static Variable i: Is then initialized by calling m1 (), which prints m1-A.
- **Note:** Even though an object is created in main(), the static context (block and variable initialization) executes only once when the class is loaded.

Example 2

```
java
CopyEdit
class A {
   static int m1(){
       System.out.println("m1-A");
       return 10;
   }
   static {
       System.out.println("SB1-A");
   static int i = m1();
   static int m2(){
       System.out.println("m2-A");
       return 20;
   }
   static {
       System.out.println("SB2-A");
   }
   static int j = m2();
public class Main {
   public static void main(String[] args) {
       A = new A();
   }
}
```

Output:

```
css
CopyEdit
SB1-A
m1-A
SB2-A
m2-A
```

Explanation:

- 1. **Static Block 1:** Executes first and prints SB1-A.
- 2. Static Variable i: Calls m1 (), printing m1-A.
- 3. Static Block 2: Executes next, printing SB2-A.

4. **Static Variable j: Calls** m2 (), printing m2-A.

Example 3

```
java
CopyEdit
class A {
   static {
       System.out.println("SB-A");
   static int m1(){
       System.out.println("m1-A");
       return 10;
   }
   static int i = m1();
}
public class Main {
   public static void main(String[] args) {
       A = new A();
       A = a2 = new A();
   }
}
```

Output:

```
css
CopyEdit
SB-A
m1-A
```

Explanation:

• The static block and static variable initialization execute only once when the class is loaded, regardless of the number of objects created later.

Example 4

This example demonstrates the interplay between static and instance components.

```
java
CopyEdit
class A {
    // Instance Block (runs for every object)
    {
        System.out.println("IB-A");
```

```
}
   static int m1(){
       System.out.println("m1-A");
       return 10;
   }
   // Instance variable initialization (runs per object)
   int i = m2();
   // Constructor (runs per object)
   A() {
       System.out.println("A-Con");
   // Static variable initialization (runs once)
   static int j = m1();
   int m2(){
       System.out.println("m2-A");
       return 20;
   }
   // Static Block (runs once)
   static{
       System.out.println("SB-A");
public class Main {
   public static void main(String[] args) {
       A = new A();
   }
}
```

css CopyEdit m1-A SB-A IB-A m2-A A-Con

Explanation:

1. Static Members:

- o m1 () is called as part of static variable j initialization, printing m1-A.
- o The static block then executes, printing SB−A.

2. Instance Members (When Object is Created):

- o Instance block executes, printing IB−A.
- o Instance variable i is initialized by calling m2 (), printing m2−A.
- o Constructor executes, printing A-Con.

Example 5

Another variation of instance and static flow:

```
java
CopyEdit
class A {
   static {
       System.out.println("SB-A");
   static int m1(){
       System.out.println("m1-A");
       return 10;
   }
   static int i = m1();
   A(){
       System.out.println("A-Con");
   }
   {
       System.out.println("IB-A");
   int m2(){
       System.out.println("m2-A");
       return 20;
   int j = m2();
public class Main {
   public static void main(String[] args) {
       A a1 = new A();
       System.out.println();
       A = a2 = new A();
}
```

Output:

```
css
CopyEdit
SB-A
m1-A
IB-A
m2-A
A-Con
IB-A
m2-A
A-Con
```

Explanation:

• Static Context:

- o SB-A is printed from the static block.
- o m1-A is printed from static variable i initialization.

For a1 (first object):

- o Instance block executes, printing IB-A.
- o Instance variable j is initialized via m2 (), printing m2-A.
- o Constructor executes, printing A-Con.

For a2 (second object):

 Since the static context was already set up, only the instance block, instance variable initialization, and constructor execute again.

Summary Table: Static vs. Instance Flow

Phase	Executed Once Per Class Loading	Executed for Every Object Creation
Static Blocks	Yes	No
Static Variables	Yes (in order of appearance)	No
Static Methods	Recognized at load time; executed on demand	N/A (they run only when explicitly called)
Instance Blocks	No	Yes (executes every time an object is created)
Instance Variables	No	Yes (each object has its own copy)
Constructor	No	Yes (executes during object creation)

Final Reasoning

• Static Context:

- Provides a way to perform class-level initialization (via static blocks and variables) before any object is created.
- Ensures that certain code is executed only once (e.g., configuration or resource allocation).

• Order of Execution:

 The order in which static members appear in the code determines their initialization sequence.

- Once the class is loaded, these static components remain in memory, and any further object creation only triggers the instance-related components.
- Usage Considerations:
 - Static Blocks and Variables: Ideal for one-time setup activities.
 - Instance Blocks and Constructors: Ensure that each object is properly initialized with its unique state.

Class.forName() Method

The Class.forName() method is used to load a class's bytecode into memory without necessarily creating an object. It belongs to the java.lang.Class class and is especially useful in situations such as JDBC driver loading.

How It Works

When you call:

```
java
CopyEdit
Class cls = Class.forName("Employee");
```

JVM performs the following steps:

1. Class Lookup:

 Searches for the class named "Employee" in the current location, Java's predefined libraries, or locations specified by the classpath environment variable.

2. Exception Handling:

o If the .class file is not found, a ClassNotFoundException is thrown.

3. Class Loading:

o If found, the JVM loads the class bytecode into memory.

4. Metadata Creation:

o A java.lang.Class object is created in the heap containing metadata (class name, modifiers, superclass info, methods, constructors, fields, etc.).

5. Return Value:

o The method returns a reference to the Class object.

Examples

Example 1: Class Not Found

```
java
CopyEdit
public class Main {
    public static void main(String[] args) throws Exception {
        Class cls = Class.forName("Employee");
    }
}
```

```
cpp
CopyEdit
Exception in thread "main" java.lang.ClassNotFoundException:
Employee

Example 2: Proper Class Loading

java
CopyEdit
class Employee {
   static {
      System.out.println("Employee class loading....");
   }
   Employee() {
```

System.out.println("Employee class Object

Class cls = Class.forName("Employee");

System.out.println("Super Class Name : " +

public static void main(String[] args) throws Exception {

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

Output:

}

}

}

Creating....");

public class Main {

```
vbnet
CopyEdit
Employee class loading....
Class Name : Employee
Super Class Name : java.lang.Object
```

cls.getSuperclass().getName());

Usage in JDBC:

In JDBC, drivers are loaded using Class.forName(), for example:

```
java
CopyEdit
Class.forName("oracle.jdbc.OracleDriver");
Class.forName("com.mysql.cj.jdbc.Driver");
```

This loads the driver class without creating an object, making it ready for use.

newInstance() Method

After a class is loaded, the newInstance() method (from the java.lang.Class class) can be used to create an object of that class explicitly.

How It Works

When you call:

```
java
CopyEdit
Object obj = cls.newInstance();
```

JVM does the following:

1. Constructor Lookup:

Searches for a zero-argument (default) constructor that is non-private.

2. **Object Creation:**

 If found, it invokes the constructor to create an instance and returns an Object reference.

3. Exceptions:

- o If no O-arg constructor is available, it throws InstantiationException.
- o If the constructor is private, it throws IllegalAccessException.
- For a private parameterized constructor, you get an InstantiationException (not an IllegalAccessException).

Examples

Example 1: Using a Valid Constructor

```
java
CopyEdit
class Employee {
    static {
        System.out.println("Employee class loading....");
    }
    Employee() {
        System.out.println("Employee class Object
Creating....");
    }
}
public class Main {
    public static void main(String[] args) throws Exception {
        Class cls = Class.forName("Employee");
        Object obj = cls.newInstance();
    }
}
```

Output:

```
kotlin
CopyEdit
Employee class loading....
Employee class Object Creating.....
Example 2: No Zero-Arg Constructor
java
CopyEdit
class Employee {
   static {
       System.out.println("Employee class loading....");
   Employee(int i) {
       System.out.println("Employee class Object
Creating....");
   }
}
public class Main {
   public static void main(String[] args) throws Exception {
       Class cls = Class.forName("Employee");
       Object obj = cls.newInstance();
   }
}
Output:
срр
CopyEdit
Employee class loading.....
Exception in thread "main" java.lang.InstantiationException:
Employee
Example 3: Private Constructor
java
CopyEdit
class Employee {
   static {
       System.out.println("Employee class loading....");
   private Employee() {
       System.out.println("Employee class Object
Creating....");
   }
```

public static void main(String[] args) throws Exception {

Class cls = Class.forName("Employee");

Object obj = cls.newInstance();

}

public class Main {

```
}
```

```
ruby
CopyEdit
Employee class loading....
Exception in thread "main" java.lang.IllegalAccessException:
class Main cannot access a member of class Employee with
modifiers "private"
```

Factory Method

A **Factory Method** is a design pattern method that returns an object reference. It abstracts the instantiation process so that the caller does not need to know the exact class of the object that is being created.

Key Points:

- Purpose:
 - o To encapsulate object creation and allow flexibility in which object is created.
- Return Type:
 - The method returns an object reference that may be of the same class or another class.
- Types of Factory Methods:
 - 1. Static Factory Methods:
 - Defined as static methods that return an object.
 - Examples include NumberFormat.getInstance(),
 DateFormat.getDateInstance(), and
 DriverManager.getConnection().
 - 2. Instance Factory Methods:
 - Non-static methods that return an object.
 - Many methods in the String class (like concat(), trim(), etc.)
 behave as instance factory methods.

Example

```
java
CopyEdit
class A {
    // Private constructor prevents direct object creation
    private A() {
        System.out.println("A-Con");
    }
    void m1() {
        System.out.println("m1-A");
}
```

```
}
// Static Factory Method
public static A getInstance() {
    // Could include logic to decide which object to create
    A a = new A();
    return a;
}

public class Main {
    public static void main(String[] args) {
        A a = A.getInstance();
        a.m1();
}
```

css CopyEdit A-Con m1-A

Singleton Class

A **Singleton Class** is designed to allow only one object of that class to be created throughout the lifecycle of an application.

Steps to Create a Singleton Class:

- 1. Private Constructor:
 - Prevents other classes from instantiating the singleton class.
- 2. Static Instance Variable:
 - Holds the sole instance of the class.
- 3. Static Factory Method:
 - o Returns the single instance, creating it if it does not exist.

Example: Lazy Initialization

```
java
CopyEdit
class A {
   private static A a = null; // Initially null

private A() {
     System.out.println("A-Con");
}

// Static factory method with lazy initialization
```

```
static A getInstance() {
       if(a == null) {
           a = new A();
       return a;
   }
}
public class Main {
   public static void main(String[] args) {
       A a1 = A.getInstance(); // First call creates the
object
       A a2 = A.getInstance(); // Subsequent calls return the
same object
       A = A.getInstance();
       System.out.println(a1);
       System.out.println(a2);
       System.out.println(a3);
   }
}
```

less CopyEdit A-Con A@7a81197d A@7a81197d A@7a81197d

Optimized Singleton Implementations

Optimized Code 1: Using a Static Block

```
java
CopyEdit
class A {
    private static A a = null;
    static {
        a = new A();
    }
    private A() {
            System.out.println("A-Con");
    }
    static A getInstance() {
            return a;
        }
}
public class Main {
    public static void main(String[] args) {
```

```
A a1 = A.getInstance();
A a2 = A.getInstance();
A a3 = A.getInstance();

System.out.println(a1);
System.out.println(a2);
System.out.println(a3);
}
```

Optimized Code 2: Inline Initialization

```
java
CopyEdit
class A {
   private static A a = new A(); // Inline creation
   private A() {
      System.out.println("A-Con");
   public static A getInstance() {
      return a;
}
public class Main {
   public static void main(String[] args) {
      A = A.getInstance();
      A = A.getInstance();
      A = A.getInstance();
       System.out.println(a1);
       System.out.println(a2);
       System.out.println(a3);
   }
}
```

Usage in MVC Applications:

• In many MVC-based applications, the controller, service, and repository classes are implemented as singleton classes. This ensures that only one instance is created, reducing resource usage and maintaining consistent state.

Final Thoughts

Class.forName() and newInstance():

These methods help in dynamic class loading and object instantiation. They
are especially useful in frameworks like JDBC and server-side containers
where lifecycle management is critical.

• Factory Methods:

 They abstract the object creation process, promoting loose coupling and flexibility in code.

Singleton Classes:

 By ensuring that only one instance exists, singletons help manage shared resources efficiently, a principle especially important in web applications and design patterns.

The final Keyword in Java

The final keyword in Java is used to restrict modifications. It is commonly used to create constant expressions and to prevent further inheritance or method overriding. There are three primary uses:

- 1. Final Variables
- 2. Final Methods
- 3. Final Classes

1. Final Variables

A **final variable** is one whose value cannot be modified after it has been assigned. Once declared and initialized, you cannot reassign it.

Key Points:

- Immutability: After assignment, its value is fixed.
- Common Use: Used for constants (e.g., configuration values, fixed parameters).

Example:

```
java
CopyEdit
final int i = 10;
i = i + 10; // Error: cannot assign a value to final variable i
i = 20; // Error: cannot assign a value to final variable i
```

Practical Scenario:

In banking applications, an account number should not change once assigned. Thus, it is declared as final:

```
java
CopyEdit
public class BankAccount {
    private final long accountNumber;
    // other details like account holder's name, address etc. are non-final
    public BankAccount(long accountNumber) {
        this.accountNumber = accountNumber;
}
```

```
}

// Getter method (no setter for accountNumber)
public long getAccountNumber() {
    return accountNumber;
}
```

Note on Loop Variables:

Loop counters or variables used for iterations must not be declared as final since they need to be updated (incremented/decremented).

2. Final Methods

A **final method** is one whose implementation cannot be changed by subclasses. This means that if a method is declared as final, it cannot be overridden in any subclass.

Key Points:

- **Prevents Overriding:** Ensures the behavior of the method remains unchanged.
- **Usage Scenario:** Used when you want to lock down the functionality provided by a superclass method.

Example:

```
java
CopyEdit
class A {
   final void m1() {
        System.out.println("X-Functionality");
    }
}
class B extends A {
   // Attempting to override m1() will cause a compilation error:
    // void m1() { System.out.println("Y-Functionality"); } // Error!
}
public class Main {
   public static void main(String[] args) {
       A = new B();
        a.m1(); // Always calls A's m1()
    }
}
```

Note:

• While a superclass's final method cannot be overridden, a subclass may declare its own new final methods.

3. Final Classes

A final class is one that cannot be subclassed. No class can extend a final class.

Key Points:

- Prevents Inheritance: The class is locked down so that its implementation is not modified by subclassing.
- Usage Scenario: Often used for security reasons or to maintain immutability (e.g., java.lang.String is a final class).

Example:

```
java
CopyEdit
final class A {
   void m1() {
       System.out.println("X-Functionality");
   void m2() {
       System.out.println("Y-Functionality");
    }
}
// Trying to extend final class A results in a compilation error:
class B extends A { // Error!
public class Main {
   public static void main(String[] args) {
       A = new A();
       a.m1();
   }
}
```

Note:

 While a superclass should not be final if it is meant to be extended, subclasses may be declared as final.

Constant Variables: Convention & Enums

Using public static final Convention

In many applications, constant values are declared using the combination of public, static, and final modifiers. This makes the constants accessible application-wide and ensures they cannot be changed.

Example: Thread Priority Constants

```
java
CopyEdit
class Thread {
    public static final int MIN_PRIORITY = 1;
    public static final int NORM_PRIORITY = 5;
    public static final int MAX_PRIORITY = 10;
}
```

Usage Example:

```
java
CopyEdit
public class Main {
    public static void main(String[] args){
        System.out.println(Thread.MIN_PRIORITY);
        System.out.println(Thread.NORM_PRIORITY);
        System.out.println(Thread.MAX_PRIORITY);
    }
}
```

Output:

```
CopyEdit

1

5

10
```

Problems with the Conventional Approach:

- 1. Repetition: You must explicitly write public static final for every constant.
- 2. **Type Safety:** Constants can be of different data types, reducing type safety.
- 3. Clarity: The constant's name may not clearly convey its intention if its value is used directly.

Solution: Enums

Enums (enumerations) provide a robust alternative:

- **Default Modifiers:** All enum constants are implicitly public static final.
- **Type Safety:** All constants within an enum are of the same type.
- Intention-Revealing: When printed, they display their name rather than a numeric value.

Enum Example: User Status

```
java
CopyEdit
enum UserStatus {
    AVAILABLE, BUSY, IDLE;
}

public class Main {
    public static void main(String[] args) {
        System.out.println(UserStatus.AVAILABLE);
        System.out.println(UserStatus.BUSY);
        System.out.println(UserStatus.IDLE);
}
```

```
}
```

Output:

```
objectivec
CopyEdit
AVAILABLE
BUSY
IDLE
```

Behind the Scenes: How Enums Work

When compiled, an enum is converted to a final class that extends <code>java.lang.Enum</code>. For example, the <code>UserStatus</code> enum is translated to a class similar to:

```
java
CopyEdit
final class UserStatus extends java.lang.Enum<UserStatus> {
   public static final UserStatus AVAILABLE;
   public static final UserStatus BUSY;
   public static final UserStatus IDLE;

   public static UserStatus[] values();
   public static UserStatus valueOf(String name);
   // static initialization block...
}
```

Implications:

- Every enum is final (cannot be subclassed).
- Each constant is a public static final member.
- All constants are of the same type, ensuring type safety and clearer usage.

Final Thoughts

- Final Variables: Lock down the value after assignment, ideal for constants.
- Final Methods: Prevent changes in method behavior by disallowing overriding.
- Final Classes: Ensure a class cannot be subclassed, preserving its implementation.
- Constant Declarations: Using public static final ensures global accessibility and immutability, but enums offer a more type-safe and intention-revealing approach for constants.

Enums, Final, and Main() Method in Java

This note explains several advanced Java concepts, including how enums can include variables, methods, and constructors (like normal classes), and the detailed requirements for the main() method. It is organized in Q&A style to cover every point.

1. Enums with Normal Variables, Methods, and Constructors

Q: In general, in Java applications we will use enums to declare constant variables. In this context, is it possible to provide normal variables, normal methods, constructors inside the enum like normal classes?

A:

Yes, it is possible. An enum can have:

- Instance variables (to hold additional information),
- Constructors (to initialize the constant objects),
- Methods (to expose behavior).

Example: Apple Enum

```
java
CopyEdit
enum Apple {
     A(500), B(300), C(100); // Constant objects with associated prices
     // Instance variable
     private int price;
     // Enum constructor (automatically invoked for each constant)
     Apple(int price) {
          this.price = price;
     // Instance method to get the price
     public int getPrice() {
          return price;
     }
}
public class Main {
     public static void main(String[] args) {
          System.out.println("A-Grade Apple : " + Apple.A.getPrice());
System.out.println("B-Grade Apple : " + Apple.B.getPrice());
System.out.println("C-Grade Apple : " + Apple.C.getPrice());
}
```

Translated by the compiler, the Apple enum becomes similar to:

```
java
CopyEdit
final class Apple extends Enum<Apple> {
   public static final Apple A = new Apple(500);
   public static final Apple B = new Apple(300);
   public static final Apple C = new Apple(100);
   private int price;
   private Apple(int price) {
```

```
this.price = price;
}

public int getPrice() {
    return price;
}

// Additional methods: values(), valueOf(String) are generated automatically.
}
```

Q: Write a Java program to represent Notebook and its details like number of pages and price value by using enum.

A:

The following program defines a NoteBook enum with two properties: number of pages and price.

```
java
CopyEdit
enum NoteBook {
   A(300, 150), B(200, 100), C(100, 50);
   private int pages;
   private int price;
    NoteBook(int pages, int price) {
       this.pages = pages;
       this.price = price;
   public int getPages() {
       return pages;
   public int getPrice() {
       return price;
    }
}
public class Main {
   public static void main(String[] args) {
        System.out.println("A-Grade Notebook : Pages :" +
NoteBook.A.getPages() + " Price : " + NoteBook.A.getPrice());
        System.out.println("B-Grade Notebook : Pages :" +
NoteBook.B.getPages() + " Price : " + NoteBook.B.getPrice());
       System.out.println("C-Grade Notebook : Pages :" +
NoteBook.C.getPages() + " Price : " + NoteBook.C.getPrice());
   }
}
```

Expected Output:

```
mathematica
CopyEdit
A-Grade Notebook : Pages :300     Price : 150
B-Grade Notebook : Pages :200     Price : 100
C-Grade Notebook : Pages :100     Price : 50
```

2. Importance and Rules of the main() Method

The main () method serves as the entry point for a Java application. Let's explore its importance and requirements in detail.

Q: What is the requirement of the main() method in Java applications?

A:

- The main() method is the starting point for the JVM to begin application execution.
- It contains the application logic that the JVM executes automatically.
- It defines the start and end of the application's execution.

Syntax:

```
java
CopyEdit
public static void main(String[] args) {
    // Application logic
}
```

Note:

 Although not a predefined method in the language specification, its conventional signature must be followed exactly.

Q: What is the requirement to declare the main() method as public?

A:

- Declaring main() as public makes it accessible to the JVM.
- If main() is private, default, or protected, its scope is limited, and the JVM (which resides outside the application's package or class) cannot access it.
- Examples:
 - o Private or default main() leads to an error:
 - Java 6: "Main method not public."
 - Java 7: "Main method not found in class Test, please define the main method as: public static void main(String[] args)"

Q: What is the requirement to declare the main() method with static?

A:

• The JVM uses the class name to access the main() method.

- Only static methods can be called without creating an instance.
- If main() is not static:
 - o Java 6: Throws java.lang.NoSuchMethodError: main
 - Java 7: "Main method is not static in class Test, please define the main method as: public static void main(String[] args)"

Q: What is the requirement to declare the main() method with the void return type?

A:

- The main() method must not return any value.
- It must have a void return type so that the application terminates after the main() method completes.
- If declared with a non-void return type:
 - o Java 6: Throws java.lang.NoSuchMethodError: main
 - o Java 7: "Main method must return a value of type void in class Test..."

Q: What is the requirement for main() method parameters?

A:

- The main() method must accept an array of String values (String[] args).
- This array holds the command-line arguments passed to the application.
- Why String?
 - o Command-line input can be of varied types, but all are received as strings.
- Why an Array?
 - The number of command-line arguments may vary, and an array can hold multiple values.

Example:

```
java
CopyEdit
class Test {
    public static void main(String[] args) {
        for (String str : args) {
            System.out.println(str);
        }
    }
}
```

Command Line Execution Example:

```
arduino
CopyEdit
D:\java830>java Test 10 "abc" 'A' 50000L false
10
```

```
abc
A
50000L
false
```

- If the main() method is declared without a String[] parameter, the JVM will not find it and will throw an error:
 - o Java 6: "java.lang.NoSuchMethodError: main"
 - Java 7: "Main method is not found in class Test, please define the main method as: public static void main(String[] args)"

Q: What are the valid syntaxes of the main() method?

A:

Below are the syntaxes along with their validity:

```
    public static void main(String[] args) — Valid
    public static void main(String[] abc) — Valid (parameter name can be any identifier)
    public static void main(String args[]) — Valid
    public static void main(String []args) — Valid
```

- 5. public static void main(String margs) Valid (using varargs)
- 6. public static void main(string[] args) Invalid (type is case-sensitive; must be String)
- 7. public static void main(String[][] args) Invalid (expects a one-dimensional array)
- public static void main(String args1, String args2) Invalid (incorrect parameter list)
- 9. public static void Main(String[] args) Invalid (method name is casesensitive; must be main)

```
10. public static int main(String[] args) — Invalid (return type must be void)
```

- 11. public static final void main(String[] args) Valid
- 12. public void main(String[] args) Invalid (must be static)
- 13. static void main(String[] args) Invalid (must be public for JVM access)
- 14. static public void main(String[] args) Valid

Q: Is it possible to provide more than one main() method in a single Java application?

A:

Yes, you can have more than one main() method as long as they are in different classes. The JVM executes the main() method in the class specified at runtime.

```
Example: Multiple Classes with main()
java
CopyEdit
class A {
```

```
public static void main(String[] args) {
        System.out.println("main()-A");
}
class B {
    public static void main(String[] args) {
        System.out.println("main()-B");
class C {
    public static void main(String[] args) {
        System.out.println("main()-C");
}
Running commands:
mathematica
CopyEdit
D:\JAVA7>java A
main()-A
D:\JAVA7>java B
main()-B
```

You can also call one class's main() from another by passing a String[] parameter.

```
Example: Chained main() Calls
```

D:\JAVA7>java C

main()-C

```
java
CopyEdit
class A {
    public static void main(String[] args) {
        System.out.println("main()-A");
        String[] str = {};
       B.main(str);
    }
}
class B {
    public static void main(String[] args) {
        System.out.println("main()-B");
        C.main(args);
    }
}
class C {
    public static void main(String[] args) {
        System.out.println("main()-C");
}
```

Output:

```
CopyEdit main()-A main()-B main()-C
```

Q: Is it possible to overload the main() method?

A:

Yes, you can overload the main() method by providing additional versions with different parameter lists. However, the JVM always calls the one with the String[] parameter.

```
Example: Overloading main()
java
CopyEdit
class Test {
    public static void main(String[] args) {
        System.out.println("String[]-Param main()");
    public static void main(int[] args) {
        System.out.println("int[]-Param main()");
    public static void main(float[] args) {
        System.out.println("float[]-Param main()");
}
When executing:
vbnet
CopyEdit
D:\JAVA7>javac Test.java
D:\JAVA7>java Test
String[]-Param main()
```

Only the main(String[] args) version is executed by the JVM.

Final Summary

- **Enums:** Can include normal variables, methods, and constructors. They are a powerful way to create type-safe constants with behavior.
- Final Keyword:
 - Final Variables prevent value modification.
 - o Final Methods prevent method overriding.
 - Final Classes prevent inheritance.
 - Enums provide an alternative for defining constants with type safety.
- main() Method Requirements:
 - o Must be declared as public static void main(String[] args) (or valid variants) to be recognized by the JVM.
 - Its parameters allow the passing of command-line arguments as an array of strings.

- o Overloading is allowed, but only the String[] version is used as the entry point.
- Proper declaration (public, static, void, and correct parameter type) is essential;
 otherwise, runtime errors occur.

Java Class Relationships and Dependency Injection

In Java application development, defining relationships between classes is essential for communication, code reusability, and efficient memory and execution time management. Relationships enable classes to interact and share data, forming the backbone of a modular and maintainable codebase.

1. Class Relationships

There are three primary relationships between classes in Java:

Relationship Definition		Mechanism	Purpose	
IS-A	Represents inheritance. A subclass is a specialized form of its superclass.	extends keyword	Code reuse, polymorphism, hierarchical modeling	
HAS-A	Represents composition or aggregation. A class contains a reference to another class instance.	Member variables (fields)	Code reuse, object composition, data navigation	
USES-A	Represents dependency. A class uses another class temporarily, typically via method parameters.	Method parameters or locals	Loose coupling, single- use dependencies	

Key Differences: HAS-A vs. IS-A

Feature	HAS-A Relationship	IS-A Relationship
Definition	One class contains or aggregates another as a member.	One class inherits from another class.
Syntax	Field declaration: private Account account;	Class declaration: class SavingsAccount extends Account
Coupling	Composition/Aggregation (stronger for composition)	Inheritance (tight coupling)
Lifecycle Management	Contained object may have independent lifecycle	Subclass lifecycle tied to superclass
Use Case	Use when one class needs to own or use another instance.	Use when a class is a subtype of another.

2. Associations

Associations describe how instances of one class relate to instances of another. Java supports four types of associations, commonly implemented via Dependency Injection (DI).

Association Type	Description	Example Scenario
One-to-One	One instance of class A relates to exactly one instance of class B.	$\begin{array}{c} Employee \leftrightarrow Bank \\ Account \end{array}$
One-to-Many	One instance of class A relates to multiple instances of class B.	Department → Employees
Many-to-One	Multiple instances of class A relate to one instance of class B.	Employees → Department
Many-to- Many	Multiple instances of class A relate to multiple instances of class B.	Students ↔ Courses

3. Dependency Injection (DI)

Dependency Injection is a design pattern used to inject dependent objects into a class, improving modularity and testability.

Types of DI:

- 1. **Constructor Injection**: Dependencies are provided through a class constructor.
- 2. **Setter Injection**: Dependencies are provided through setter methods after object creation.

3.1 Constructor Dependency Injection

Injecting a dependent object via the constructor. Dependencies are resolved at object creation.

```
class Account { /* fields and methods omitted */ }

class Employee {
    private Account account;

    // Constructor DI
    public Employee(Account account) {
        this.account = account;
    }
    // ... other fields and methods
}

public class Test {
    public static void main(String[] args) {
        Account account = new Account();
        Employee emp = new Employee(account);
    }
}
```

3.2 Setter Method Dependency Injection

Injecting a dependent object via a setter method. Dependencies can be set or changed post-construction.

```
class ReviewAndRating { /* fields and methods omitted */ }

class Movie {
    private ReviewAndRating reviewAndRating;

    // Setter DI
    public void setReviewAndRating(ReviewAndRating reviewAndRating) {
        this.reviewAndRating = reviewAndRating;
    }
    // ... other fields and methods
}

public class Test {
    public static void main(String[] args) {
        ReviewAndRating rr = new ReviewAndRating();
        Movie m = new Movie();
        m.setReviewAndRating(rr);
    }
}
```

4. Examples of One-to-One Association

4.1 Constructor Injection Example (Employee ← Account)

Account.java

```
public class Account {
    private String accountNumber;
    private String accountHolderName;
    private String accountType;
    private long accountBalance;

    public Account(String accountNumber, String accountHolderName, String accountType, long accountBalance) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.accountType = accountType;
        this.accountBalance = accountBalance;
    }
    // Getters omitted for brevity
}
```

Employee.java

```
public class Employee {
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;
    private Account account;

    public Employee(int eno, String ename, float esal, String eaddr,
Account account) {
```

```
this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
        this.account = account;
    public void getEmployeeDetails() {
        System.out.println("Employee Details");
        // print details including account fields
}
Main.java
public class Main {
    public static void main(String[] args) {
        Account account = new Account("abc123", "Durga", "Savings", 50000);
        Employee employee = new Employee(111, "Durga", 25000, "Hyd",
account);
        employee.getEmployeeDetails();
    }
```

4.2 Setter Injection Example (Person ← DrivingLicence)

DrivingLicence.java

```
public class DrivingLicence {
    private String licenceNo;
    private String licenceHolderName;
    private String licenceHolderAddress;
    private String licenceType;
    private String licenceDate;
    private String licenceExprDate;

    // Getters and setters omitted
}
```

Person.java

```
public class Person {
    private String aadharNo;
    private String personName;
    private String personDob;
    private String personEmailId;
    private String personMobileNo;
    private DrivingLicence drivingLicence;

    // Setters and getters omitted for brevity

    public void getPersonDetails() {
        System.out.println("Person Details");
        // print personal and driving licence details
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        DrivingLicence dl = new DrivingLicence();
        // set DL fields

        Person person = new Person();
        // set person fields
        person.setDrivingLicence(dl);

        person.getPersonDetails();
    }
}
```

Summary:

- IS-A: Inheritance, use extends for subtyping.
- **HAS-A**: Composition/Aggregation, use member fields for ownership.
- **USES-A**: Temporary dependency, use method parameters.
- **Associations**: One-to-one, one-to-many, many-to-one, many-to-many, implemented via DI.
- **Dependency Injection**: Constructor vs. setter injection for decoupled, testable code.

One-To-Many Association in Dependency Injection

Definition:

A one-to-many association is a relationship between two entities where a single instance of one entity (the "one" side) is linked to multiple instances of another entity (the "many" side). In the context of Dependency Injection (DI), this means injecting a collection (array, list, etc.) of dependent objects into a single consumer object.

Key Characteristics:

- Uni-directional mapping: The "one" entity holds references to multiple "many" entities
- **Multiplicity**: One instance can have zero, one, or many related instances on the "many" side.
- **Dependency management**: DI frameworks can supply collections of dependencies automatically.

1. Constructor-Based DI Example

Scenario: A Movie has multiple Actor instances.

1.1 Actor.java

```
public class Actor {
    private String name;
```

```
private String role;
private String address;
private int noOfMovies;

public Actor(String name, String role, String address, int noOfMovies)

{
    this.name = name;
    this.role = role;
    this.address = address;
    this.noOfMovies = noOfMovies;
}

// Getters
public String getName() { return name; }
public String getRole() { return role; }
public String getAddress() { return address; }
public int getNoOfMovies() { return noOfMovies; }
}
```

1.2 Movie.java

```
public class Movie {
   private String movieName;
   private String releaseDate;
   private String directorName;
   private String producerName;
   private Actor[] actors;
   public Movie (String movieName,
               String releaseDate,
               String directorName,
               String producerName,
               Actor[] actors) {
       this.movieName = movieName;
       this.releaseDate = releaseDate;
       this.directorName = directorName;
       this.producerName = producerName;
       this.actors = actors;
   }
   public void getMovieDetails() {
       System.out.println("Movie Details");
       System.out.println("----");
      System.out.println();
       System.out.println("Actors:");
       System.out.printf("%-10s %-10s %-10s %-10s%n", "Name", "Role",
"Address", "Movies");
       System.out.println("-----
");
       for (Actor actor : actors) {
          System.out.printf("%-10s %-10s %-10s %-10d%n",
                          actor.getName(),
                           actor.getRole(),
                           actor.getAddress(),
                           actor.getNoOfMovies());
       }
```

```
}
```

1.3 Main.java (Test)

```
public class Main {
    public static void main(String[] args) {
        Actor actor1 = new Actor("Prabhas", "Hero", "Hyd", 15);
        Actor actor2 = new Actor("Anushka", "Heroin", "Hyd", 25);
        Actor actor3 = new Actor("Rana", "Villain", "Hyd", 13);
        Actor[] actors = { actor1, actor2, actor3 };
        Movie movie = new Movie(
            "Bahubali",
            "10-July-2015",
            "S S Rajamouli",
            "Shobu Yarlagadda",
            actors
        movie.getMovieDetails();
}
Output Table
 Name Role Address Movies
Prabhas Hero Hyd
Anushka Heroin Hyd
       Villain Hyd
Rana
                    13
```

Reasoning:

- Constructor DI enforces immutability: all required dependencies (Actor[]) are provided at object creation.
- Ensures Movie cannot exist without its actors collection.

2. Setter-Based DI Example

Scenario: A Movie has multiple ReviewAndRating instances.

2.1 ReviewAndRating.java

```
package com.durgasoft.entities;

public class ReviewAndRating {
    private String reviewId;
    private String reviewerName;
    private String reviewDescription;
    private String reviewDate;
    private float rating;
```

```
// Getters and setters
public String getReviewId() { return reviewId; }
public void setReviewId(String reviewId) { this.reviewId = reviewId; }
public String getReviewerName() { return reviewerName; }
public void setReviewerName(String reviewerName) { this.reviewerName =
reviewerName; }
public String getReviewDescription() { return reviewDescription; }
public void setReviewDescription(String reviewDescription) {
this.reviewDescription = reviewDescription; }
public String getReviewDate() { return reviewDate; }
public void setReviewDate(String reviewDate) { this.reviewDate =
reviewDate; }
public float getRating() { return rating; }
public void setRating(float rating) { this.rating = rating; }
}
```

2.2 Movie.java

```
package com.durgasoft.entities;
public class Movie {
    private String movieName;
    private String releaseDate;
    private String directorName;
    private String producerName;
    private ReviewAndRating[] reviewAndRatings;
    // Getters and setters
    public String getMovieName() { return movieName; }
    public void setMovieName(String movieName) { this.movieName =
movieName; }
    public String getReleaseDate() { return releaseDate; }
    public void setReleaseDate(String releaseDate) { this.releaseDate =
releaseDate; }
    public String getDirectorName() { return directorName; }
    public void setDirectorName(String directorName) { this.directorName =
directorName; }
    public String getProducerName() { return producerName; }
    public void setProducerName(String producerName) { this.producerName =
producerName; }
    public ReviewAndRating[] getReviewAndRatings() { return
reviewAndRatings; }
    public void setReviewAndRatings(ReviewAndRating[] reviewAndRatings) {
this.reviewAndRatings = reviewAndRatings; }
    public void getMovieDetails() {
        System.out.println("Movie Details");
        System.out.println("----");
        System.out.println("Movie Name : " + movieName);
System.out.println("Release Date : " + releaseDate);
        System.out.println("Director Name : " + directorName);
        System.out.println("Producer Name : " + producerName);
        System.out.println();
        System.out.println("Review And Ratings");
        for (int i = 0; i < reviewAndRatings.length; i++) {</pre>
            ReviewAndRating rr = reviewAndRatings[i];
            System.out.println("Review#" + (i+1));
```

```
System.out.println("\tReview ID
                                                      : " +
rr.getReviewId());
                                                     : " +
            System.out.println("\tReviewer Name
rr.getReviewerName());
            System.out.println("\tReview Description : " +
rr.getReviewDescription());
           System.out.println("\tReview Date
                                                     : " +
rr.getReviewDate());
                                                      : " +
            System.out.println("\tRating
rr.getRating());
           System.out.println();
        }
    }
2.3 Test.java (Usage)
package com.durgasoft.test;
import com.durgasoft.entities.Movie;
import com.durgasoft.entities.ReviewAndRating;
public class Test {
    public static void main(String[] args) {
        ReviewAndRating r1 = new ReviewAndRating();
        r1.setReviewId("R111"); r1.setReviewerName("Suresh Kondeti");
        r1.setReviewDescription("Block Buster"); r1.setReviewDate("10-11-
2024"); r1.setRating(3.5f);
        ReviewAndRating r2 = new ReviewAndRating();
        r2.setReviewId("R222"); r2.setReviewerName("Sairaj");
        r2.setReviewDescription("Superhit"); r2.setReviewDate("10-11-
2024"); r2.setRating(3.0f);
        ReviewAndRating r3 = new ReviewAndRating();
        r3.setReviewId("R333"); r3.setReviewerName("Venkat N");
        r3.setReviewDescription("Average"); r3.setReviewDate("10-11-2024");
r3.setRating(2.5f);
        ReviewAndRating[] reviews = { r1, r2, r3 };
        Movie movie = new Movie();
        movie.setMovieName("Amaran");
        movie.setReleaseDate("09-11-2024");
        movie.setDirectorName("Rajkumar");
        movie.setProducerName("Kamalhasan");
        movie.setReviewAndRatings(reviews);
        movie.getMovieDetails();
    }
Output Structure
Movie Details
_____
Movie Name : Amaran Release Date : 09-11-2024
Director Name : Rajkumar
Producer Name : Kamalhasan
```

Review And Ratings

Review#1

Review ID

Review ID : R111
Reviewer Name : Suresh Kondeti Review Description : Block Buster Review Date : 10-11-2024

: 3.5 Rating

Reasoning:

Setter DI provides flexibility: dependencies can be changed after object creation.

• Useful when dependencies are optional or can be updated dynamically.

3. Comparison Table

Aspect	Constructor DI	Setter DI
Injection Timing	At object creation	After object creation
Immutability	Enforces mandatory dependencies	Allows optional/mutable dependencies
Null-Safety	Ensures no nulls if constructor enforces checks	Risk of unset dependencies if setter not called
Use Case	Required dependencies, mandatory collections	Optional collections, configurable at runtime

Conclusion:

One-to-many associations in DI help manage collections of related entities cleanly and declaratively. Choosing between constructor and setter injection depends on whether the collection is mandatory and immutable or optional and dynamic.

4. Many-To-One Association in DI

Definition:

A many-to-one association is a relationship where multiple instances of one entity (the "many" side) are linked to a single instance of another entity (the "one" side). In DI, this means injecting a single dependency into multiple consumer objects.

4.1 Constructor-Based DI Example

Scenario: Multiple Order instances share one Customer.

Customer.java

```
CopyEdit
public class Customer {
    private String cid;
    private String cname;
    private String mobile;
    private String email;
    private String caddr;
    public Customer (String cid, String cname, String mobile, String email,
String caddr) {
        this.cid = cid;
        this.cname = cname;
        this.mobile = mobile;
        this.email = email;
        this.caddr = caddr;
    // Getters...
Order.java
java
CopyEdit
public class Order {
    private String orderID;
    private String itemName;
    private String deliveryDate;
    private String deliveryAddress;
    private Customer customer;
    public Order(String orderID, String itemName, String deliveryDate,
                 String deliveryAddress, Customer customer) {
        this.orderID = orderID;
        this.itemName = itemName;
        this.deliveryDate = deliveryDate;
        this.deliveryAddress = deliveryAddress;
        this.customer = customer;
    public void getOrderDetails() {
        System.out.println("Order Details");
        // ... prints order fields ...
        System.out.println("Customer Details");
        // ... prints customer via getters ...
    }
}
Main.java (Test)
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Customer customer = new Customer("C111", "Durga", "9988776655",
"durga@gmail.com", "Hyd");
        Order o1 = new Order ("0-111", "Samsung Galaxy-22", "20-11-2024",
                              "360/3r, S R Nagar, Hyd", customer);
        Order o2 = new Order("0-222", "Lenovo Laptop", "20-11-2024",
                              "360/3r, S R Nagar, Hyd", customer);
```

Reasoning:

- Constructor DI ensures every Order is created with its Customer dependency.
- All orders share the same Customer instance, reducing duplication and ensuring consistency.

4.2 Setter-Based DI Example

Scenario: Multiple Item instances share one Customer.

Customer.java

```
java
CopyEdit
public class Customer {
    private String cid;
    private String cname;
    private String caddr;
    private String cemail;
    private String cmobile;

    // Getters and setters...
}
```

Item.java

```
java
CopyEdit
public class Item {
    private String itemId;
    private String itemName;
    private int itemPrice;
    private String mfgDate;
    private String exprDate;
    private Customer customer;

    // Getters, setters, and getItemDetails()...
}
```

Main.java (Test)

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
```

```
Customer customer = new Customer();
        customer.setCid("C-111");
        // ... set other customer fields ...
        Item i1 = new Item();
        // ... set item fields ...
        i1.setCustomer(customer);
        Item i2 = new Item();
        // ... set item fields ...
        i2.setCustomer(customer);
        Item i3 = new Item();
        // ... set item fields ...
        i3.setCustomer(customer);
        i1.getItemDetails();
        i2.getItemDetails();
        i3.getItemDetails();
    }
}
```

Reasoning:

- Setter DI allows sharing a single Customer across multiple Item instances without recreating it.
- Provides flexibility to change the associated Customer at runtime if needed.

Comparison (Many-To-One vs One-To-Many):

Association	One-To-Many	Many-To-One
Direction	Single holds many	Many share one
DI Pattern	Inject collection into one class	Inject single dependency into many
Use Case	Movie → Actor[]	$Order[] \rightarrow Customer$
Immutability	Constructor enforces full list	Constructor ensures each consumer has dependency
Flexibility	Setter allows updating collection	Setter allows reassigning shared dependency

Let me know if this looks good, and I can try integrating it directly into the canvas notes again!

Many-to-Many Association in Java (with Constructor & Setter Dependency Injection)

What is Many-to-Many Association?

It is a relationship between entities where:

• Multiple instances of one entity are related to multiple instances of another entity.

Real-World Examples:

Example Entities Involved

Authors & Books Many Authors → Many Books

Students & Courses Many Students → Many Courses

1. Constructor Dependency Injection Example

Example: Multiple Books prepared by Multiple Authors

Class Diagram:

```
vbnet
CopyEdit
Author --> Book (Many-to-Many using Constructor Injection)
```

Author.java

```
java
CopyEdit
public class Author {
   private String authorName;
   private String authorQualification;
   private String authorExperience;
    public Author(String authorName, String authorQualification, String
authorExperience) {
        this.authorName = authorName;
        this.authorQualification = authorQualification;
        this.authorExperience = authorExperience;
    }
    public String getAuthorName() {
       return authorName;
    public String getAuthorQualification() {
        return authorQualification;
    public String getAuthorExperience() {
       return authorExperience;
```

Book.java

```
java
CopyEdit
public class Book {
   private String bookId;
   private String bookName;
   private String bookType;
   private int bookPrice;
   private Author[] authors;
   public Book(String bookId, String bookName, String bookType, int
bookPrice, Author[] authors) {
       this.bookId = bookId;
       this.bookName = bookName;
       this.bookType = bookType;
       this.bookPrice = bookPrice;
       this.authors = authors;
   public void getBookDetails() {
       System.out.println("Book Details");
       System.out.println("----");
       System.out.println("Book ID : " + bookId);
                                       : " + bookName);
       System.out.println("Book Name
                                         : " + bookType);
       System.out.println("Book Type
       System.out.println("Book Price : " + bookPrice);
       System.out.println("Authors Details:");
       int count = 0;
       for(Author author: authors){
           count++;
           System.out.println("Author#" + count + ":");
           System.out.println("\tAuthor Name
author.getAuthorName());
           System.out.println("\tAuthor Qualification : " +
author.getAuthorQualification());
           System.out.println("\tAuthor Experience : " +
author.getAuthorExperience());
       System.out.println();
    }
```

Main.java

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Author author1 = new Author("Anshul", "BTech", "5 Books");
        Author author2 = new Author("Sujeet", "BTech", "3 Books");
        Author author3 = new Author("Pranav", "BTech", "7 Books");
        Author[] authors1 = {author1, author2, author3};
```

```
Author[] authors2 = {author1, author2};
Author[] authors3 = {author2, author3};

Book book1 = new Book("B-111", "JAVA", "Software", 700, authors1);
Book book2 = new Book("B-222", "PYTHON", "Software", 500,
authors2);
Book book3 = new Book("B-333", "CCNA", "Hardware", 400, authors3);
book1.getBookDetails();
book2.getBookDetails();
book3.getBookDetails();
}
```

OUTPUT:

Output for Book Details is given exactly in your example and shows proper Many-to-Many relationship between Book & Author.

2. Setter Dependency Injection Example

Example: Multiple Students joined Multiple Courses

Class Diagram:

```
vbnet
CopyEdit
Student --> Course (Many-to-Many using Setter Injection)
```

Course.java

```
java
CopyEdit
public class Course {
    private String cid;
    private String cname;
    private int ccost;

public String getCid() {
        return cid;
    }

public void setCid(String cid) {
        this.cid = cid;
    }

public String getCname() {
        return cname;
    }

public void setCname(String cname) {
```

```
this.cname = cname;
}

public int getCcost() {
    return ccost;
}

public void setCcost(int ccost) {
    this.ccost = ccost;
}
```

Student.java

```
java
CopyEdit
public class Student {
    private String sid;
    private String sname;
    private String saddr;
    private Course[] courses;
    public String getSid() {
       return sid;
    public void setSid(String sid) {
        this.sid = sid;
    public String getSname() {
       return sname;
    public void setSname(String sname) {
       this.sname = sname;
    public String getSaddr() {
       return saddr;
    public void setSaddr(String saddr) {
        this.saddr = saddr;
    public Course[] getCourses() {
       return courses;
    public void setCourses(Course[] courses) {
        this.courses = courses;
    public void getStudentDetails(){
        System.out.println("Student Details");
        System.out.println("----");
        System.out.println("Student Id : " + sid);
System.out.println("Student Name : " + sname);
```

```
System.out.println("Student Address : " + saddr);
System.out.println("CID\tCNAME\tCCOST");
System.out.println("-----");

for(Course c : courses){
    System.out.print(c.getCid() + "\t");
    System.out.print(c.getCname() + "\t");
    System.out.println(c.getCcost());
}
System.out.println();
}
```

Main.java

```
java
CopyEdit
public class Main {
    public static void main(String[] args) {
        Course course1 = new Course();
        course1.setCid("C111");
        course1.setCname("JAVA");
        course1.setCcost(50000);
        Course course2 = new Course();
        course2.setCid("C222");
        course2.setCname("PYTHON");
        course2.setCcost(40000);
        Course course3 = new Course();
        course3.setCid("C333");
        course3.setCname("AWS");
        course3.setCcost(30000);
        Course[] courses1 = {course1, course2, course3};
        Course[] courses2 = {course1, course2};
        Course[] courses3 = {course2, course3};
        Student student1 = new Student();
        student1.setSid("S-111");
        student1.setSname("Swapnil");
        student1.setSaddr("Pune");
        student1.setCourses(courses1);
        Student student2 = new Student();
        student2.setSid("S-222");
        student2.setSname("Priya");
        student2.setSaddr("Guntur");
        student2.setCourses(courses2);
        Student student3 = new Student();
        student3.setSid("S-333");
        student3.setSname("Pranita");
        student3.setSaddr("Odisha");
        student3.setCourses(courses3);
        student1.getStudentDetails();
        student2.getStudentDetails();
        student3.getStudentDetails();
```

OUTPUT:

Output matches your example clearly showing Many-to-Many relationship between Student & Course.

Composition vs Aggregation

Feature	Composition	Aggregation
Type of Relation	Strong Association	Weak Association
Dependency	Contained object life depends on container	Contained object independent of container
Example	Library & Books (If Library closes → Books gone)	Library & Students (Even if Library closes → Students exist)

Reasoning:

Concept	Reason
Concept	itcason

Constructor Injection Used when dependency is mandatory and fixed at object creation

Setter Injection Used when dependency is optional or can change after object creation

Composition Tight Coupling between classes

Aggregation Loose Coupling between classes

Inheritance in Java — Complete Notes

Definition:

Inheritance is a relationship between classes where one class (child class) acquires the properties (variables) and behaviors (methods) of another class (parent class).

Terminologies:

Term Meaning

Superclass / Parent Class / Base Class Class whose properties & methods are inherited

Subclass / Child Class / Derived Class Class which inherits from another class

Advantage of Inheritance:

- → Code Reusability
- → Reduces code duplication
- → Enhances maintainability

Reason: If variables and methods are written in a superclass, they can be reused in all subclasses without redefining them.

Types of Inheritance in Java:

S.No	Type of Inheritance	Supported in Java	Reason
1	Single Inheritance	Yes	Simple parent-child relation
2	Multiple Inheritance	No (using classes)	Ambiguity Problem
3	Multilevel Inheritance	Yes	Chain of Inheritance
4	Hierarchical Inheritance	Yes	Multiple children inherit from one parent
5	Hybrid Inheritance	No (using classes)	Combination of Single & Multiple inheritance (Ambiguity problem)

Note:

→ Java supports Multiple & Hybrid Inheritance using *Interfaces* (Not with classes).

1. Single Inheritance

Definition:

Inheritance from one superclass to one subclass.

Diagram:

Example:

```
java
CopyEdit
class A {
  void m1() {
       System.out.println("m1-A");
   }
}
class B extends A {
   void m2() {
      System.out.println("m2-B");
public class Main {
  public static void main(String[] args) {
      A a = new A();
       a.m1();
       B b = new B();
      b.ml(); // Inherited from A
       b.m2();
   }
}
```

Output:

```
css
CopyEdit
m1-A
m1-A
m2-B
```

2. Multiple Inheritance

Definition:

Inheritance from more than one superclass to a subclass.

Diagram:



Java Restriction: Not Supported (Compilation Error) Reason: Ambiguity Problem → Diamond Problem.

Example:

```
java
CopyEdit
class A {
    void m1() {
        System.out.println("m1-A");
    }
}
class B {
    void m2() {
        System.out.println("m2-B");
    }
}
// Error: Cannot inherit from both A and B class C extends A, B {
    void m3() {
        System.out.println("m3-C");
    }
}
```

Output:

```
javascript
CopyEdit
Compilation Error
```

3. Multilevel Inheritance

Definition:

Inheritance in multiple levels (Grandparent → Parent → Child)

Diagram:

```
css
CopyEdit
A \rightarrow B \rightarrow C
```

Example:

```
java
CopyEdit
```

```
class A {
  void m1() {
      System.out.println("m1-A");
}
class B extends A {
  void m2() {
      System.out.println("m2-B");
class C extends B {
  void m3() {
      System.out.println("m3-C");
public class Main {
  public static void main(String[] args) {
      A = new A();
       a.m1();
       B b = new B();
       b.m1();
       b.m2();
       C c = new C();
       c.m1();
       c.m2();
       c.m3();
   }
}
```

Output:

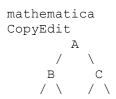
css CopyEdit m1-A m1-A m2-B m1-A m2-B m3-C

4. Hierarchical Inheritance

Definition:

One superclass having multiple subclasses.

Diagram:



Example:

```
java
CopyEdit
class A {
   void m1() {
       System.out.println("m1-A");
class B extends A {
  void m2() {
       System.out.println("m2-B");
}
class C extends A {
  void m3() {
       System.out.println("m3-C");
class D extends B {
  void m4() {
      System.out.println("m4-D");
class E extends B {
  void m5() {
       System.out.println("m5-E");
class F extends C {
   void m6() {
      System.out.println("m6-F");
class G extends C {
  void m7() {
       System.out.println("m7-G");
public class Main {
   public static void main(String[] args) {
       A = new A(); a.m1();
       B b = new B(); b.m1(); b.m2();
       C c = new C(); c.m1(); c.m3();
       D d = new D(); d.m1(); d.m2(); d.m4();
       E = new E(); e.m1(); e.m2(); e.m5();
       F f = new F(); f.m1(); f.m3(); f.m6();
       G g = new G(); g.m1(); g.m3(); g.m7();
   }
```

Output:

```
CSS
CopyEdit
m1-A
m1-A
m2-B
m1-A
m3-C
m1-A
m2-B
m4-D
m1-A
m2-B
m5-E
m1-A
m3-C
m6-F
m1-A
m3-C
m7-G
```

5. Hybrid Inheritance

Definition:

Combination of Single & Multiple Inheritance.

Diagram:

```
ini
CopyEdit
Hybrid = Single + Multiple
```

Java Restriction: Not Supported (with classes)

Reason: Ambiguity problem → Same method from multiple parents.

Example:

```
java
CopyEdit
class A {
    void m1() {
        System.out.println("m1-A");
    }
}
class B extends A {
    void m2() {
        System.out.println("m2-B");
    }
}
class C extends A {
    void m3() {
        System.out.println("m3-C");
    }
}
// Error: Cannot extend both B and C
```

```
class D extends B, C {
   void m4() {
       System.out.println("m4-D");
   }
}
```

Output:

```
javascript
CopyEdit
Compilation Error
```

Important Point:

Access Accessible in Subclass Accessible in Superclass

Variables/Methods in Superclass Yes

Variables/Methods in Subclass No

Not accessible in Superclass

Example:

```
java
CopyEdit
class A {
   int i = 10;
   void m1() {
       System.out.println("m1-A");
       // System.out.println(j); → Error
       // m2(); \rightarrow Error
   }
}
class B extends A {
  int j = 20;
   void m2() {
       System.out.println("m2-B");
       System.out.println(i); // Access Superclass Variable
       m1();
                               // Access Superclass Method
   }
public class Main {
   public static void main(String[] args) {
      A = new A();
       a.m1();
       B b = new B();
       b.m2();
   }
}
```

Output:

CopyEdit m1-A m2-B 10

m1-A

Summary Table:

Type of InheritanceJava SupportReasonSingle InheritanceYesSimple Parent → ChildMultiple InheritanceNoAmbiguity ProblemMultilevel InheritanceYesChain of InheritanceHierarchical InheritanceYesOne Parent → Many ChildrenHybrid InheritanceNoCombination of Single & Multiple → Ambiguity