**UNIT - 1**

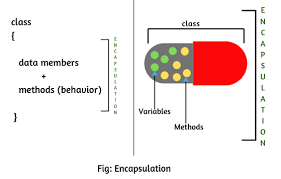
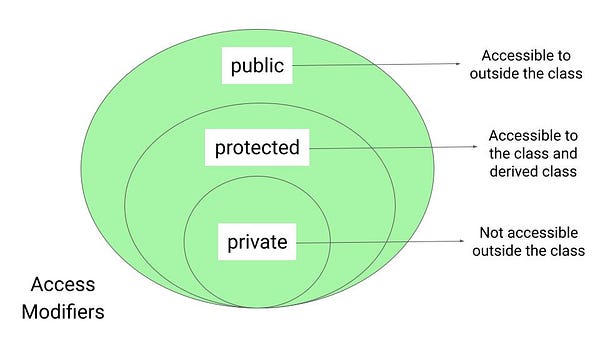
***INTRODUCTION TO OOP CONCEPTS***

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods. Objects are instances of classes, which serve as blueprints that define an object’s behavior (methods) and properties (data fields). The four fundamental concepts of OOP are **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction**. OOP promotes better design through modularity, code reusability, and scalability, making complex systems easier to develop and maintain. By encapsulating data and behaviors in objects, OOP mimics real-world scenarios, encouraging a natural and organized approach to software design.



***1. ENCAPSULATION***

Encapsulation is the practice of bundling an object's data (variables) and methods (functions) into a single unit or class and restricting access to the inner workings of that object. The main goal of encapsulation is to protect an object’s integrity by preventing outside interference and misuse of its data. In Java, encapsulation is implemented using **access modifiers** such as private, public, protected, and the default. This ensures that only certain methods are available for accessing or modifying the object's data. Encapsulation makes the code more secure and easier to maintain, as changes to the internal implementation do not affect external code. Additionally, it promotes better control over data by exposing only what is necessary through getter and setter methods. For example, a Person class might store a person's age as a private variable, but provide public methods to safely access or modify it, ensuring valid input.

**Code Example:**

class Person {

private String name; // private variable

private int age; // private variable

// Public getter method for name

public String getName() {

return name;

}

// Public setter method for name

public void setName(String name) {

this.name = name;

}

// Public getter method for age

public int getAge() {

return age;

}

// Public setter method for age

public void setAge(int age) {

if (age > 0) { // Encapsulation ensures valid data

this.age = age;

}

}

}

public class Main {

public static void main(String[] args) {

Person person = new Person();

person.setName("Alice");

person.setAge(25);

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

System.out.println("Age: " + person.getAge());

}

}

**Output:**

Name: Alice

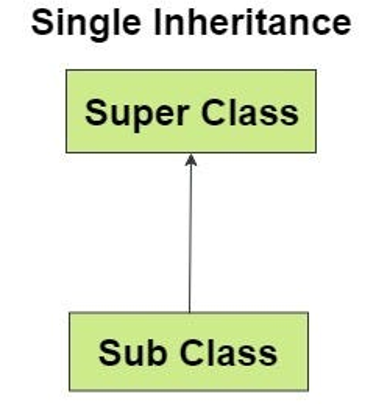
Age: 25

***2. INHERITANCE***

Inheritance is an OOP concept that allows a new class (subclass or child class) to inherit properties and methods from an existing class (superclass or parent class). It provides a way to achieve **code reusability** and promotes a hierarchical relationship between classes. In Java, the keyword extends is used to establish an inheritance relationship. Through inheritance, a subclass can use or override the methods of the superclass while adding its own specialized methods and properties. This enables a modular design and reduces redundancy. For example, a Vehicle class might define common features like speed and fuelCapacity, and specific classes such as Car or Bike can inherit these attributes while implementing their own specialized features. Inheritance also supports the **is-a** relationship, meaning a Car is a Vehicle, but it also has unique attributes like the number of doors. However, Java doesn’t support **multiple inheritance** with classes to avoid ambiguity but allows it through interfaces.

Inheritance in Object-Oriented Programming (OOP) allows a class (subclass) to inherit properties and methods from another class (superclass). It promotes code reusability and establishes a relationship between classes. There are several types of inheritance in Java:

**1. Single Inheritance**

In single inheritance, a subclass inherits from only one superclass. This is the simplest form of inheritance and helps avoid complexity.

**Example:**

class Animal {

void eat() {

System.out.println("Eating...");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark(); // Subclass method

}

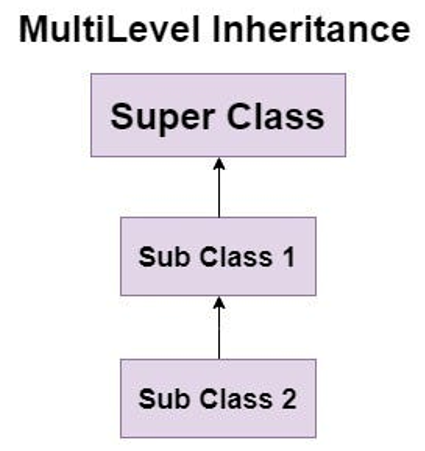
}

**Output:**

Eating...

Barking...

**2. Multilevel Inheritance**

In multilevel inheritance, a class can inherit from another class, which in turn inherits from another class. This creates a hierarchy of classes.

**Example:**

class Animal {

void eat() {

System.out.println("Eating...");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

class Puppy extends Dog {

void weep() {

System.out.println("Weeping...");

}

}

public class Main {

public static void main(String[] args) {

Puppy puppy = new Puppy();

puppy.eat(); // Inherited from Animal

puppy.bark(); // Inherited from Dog

puppy.weep(); // Puppy method

}

}

**Output:**

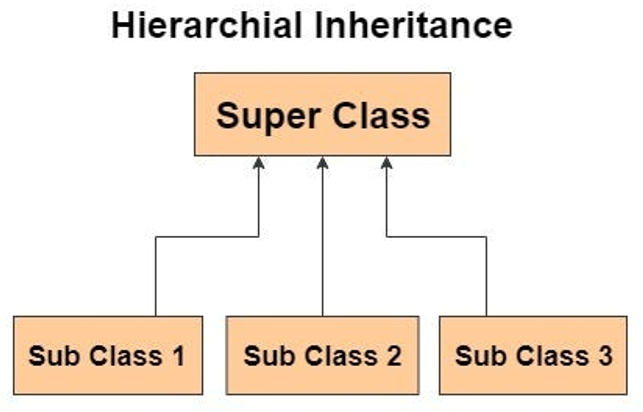
Eating...

Barking...

Weeping...

**3. Hierarchical Inheritance**

In hierarchical inheritance, multiple subclasses inherit from a single superclass. This structure is useful when several classes share common functionality.

**Example:**

class Animal {

void eat() {

System.out.println("Eating...");

}

}

class Dog extends Animal {

void bark() {

System.out.println("Barking...");

}

}

class Cat extends Animal {

void meow() {

System.out.println("Meowing...");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited from Animal

dog.bark(); // Dog method

Cat cat = new Cat();

cat.eat(); // Inherited from Animal

cat.meow(); // Cat method

}

}

**Output:**

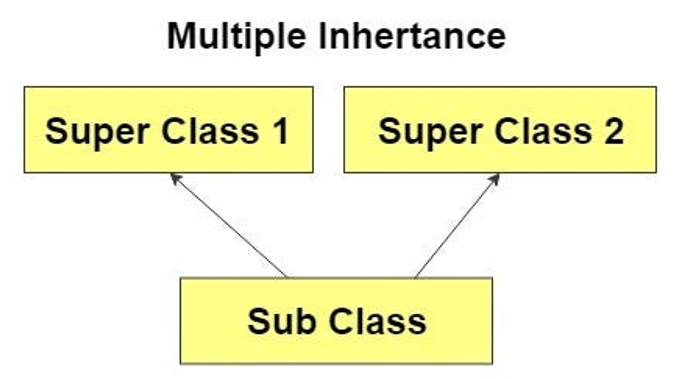
Eating...

Barking...

Eating...

Meowing...

**4. Multiple Inheritance (via Interfaces)**

Java does not support multiple inheritance through classes to avoid ambiguity (the "Diamond Problem"). However, it allows a class to implement multiple interfaces, which can be seen as a form of multiple inheritance.

**Example:**

interface CanRun {

void run();

}

interface CanBark {

void bark();

}

class Dog implements CanRun, CanBark {

public void run() {

System.out.println("Dog is running...");

}

public void bark() {

System.out.println("Dog is barking...");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.run(); // Implementing CanRun

dog.bark(); // Implementing CanBark

}

}

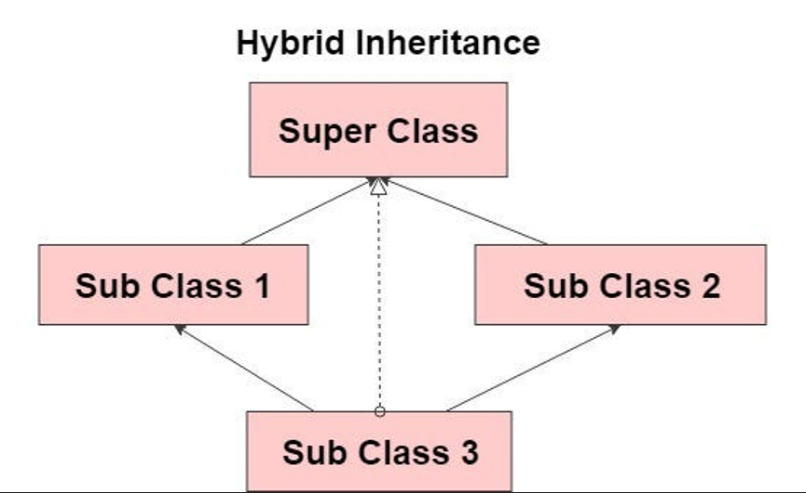
**Output:**

Dog is running...

Dog is barking...

**5. Hybrid Inheritance**

Hybrid inheritance is a combination of two or more types of inheritance (e.g., single, multilevel, hierarchical). While Java does not support hybrid inheritance through classes directly due to complexity, it can be achieved using interfaces.

**Example:**

interface CanFly {

void fly();

}

class Animal {

void eat() {

System.out.println("Eating...");

}

}

class Bird extends Animal implements CanFly {

public void fly() {

System.out.println("Bird is flying...");

}

}

class Sparrow extends Bird {

void chirp() {

System.out.println("Sparrow is chirping...");

}

}

public class Main {

public static void main(String[] args) {

Sparrow sparrow = new Sparrow();

sparrow.eat(); // Inherited from Animal

sparrow.fly(); // Implemented from CanFly

sparrow.chirp(); // Sparrow method

}

}

**Output:**

Eating...

Bird is flying...

Sparrow is chirping...

**Summary of Inheritance Types:**

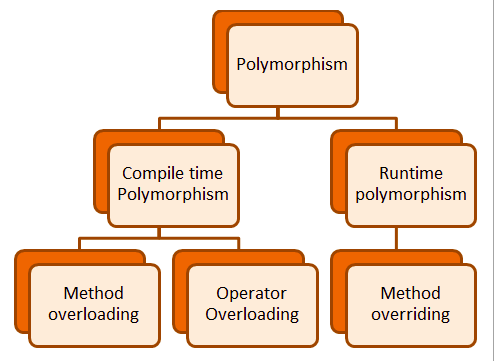
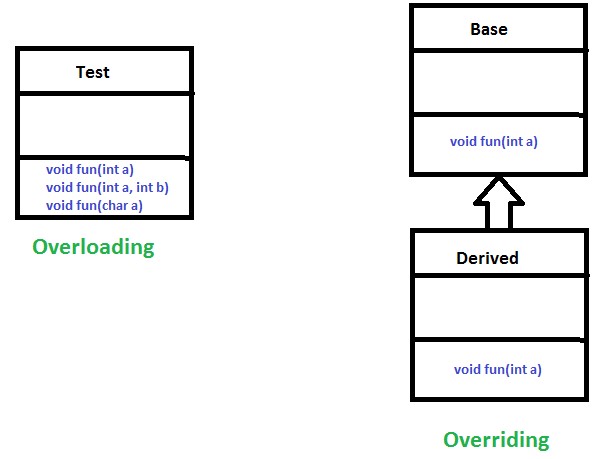
| **TYPE OF INHERITANCE** | **DESCRIPTION** | **EXAMPLE CLASSES** |
| --- | --- | --- |
| Single Inheritance | A subclass inherits from one superclass. | Dog extends Animal |
| Multilevel Inheritance | A class inherits from another class, creating a hierarchy. | Puppy extends Dog, which extends Animal |
| Hierarchical Inheritance | Multiple subclasses inherit from a single superclass. | Dog, Cat extends Animal |
| Multiple Inheritance | A class implements multiple interfaces. | Dog implements CanRun, CanBark |
| Hybrid Inheritance | Combination of two or more types of inheritance using interfaces. | Sparrow extends Bird and implements CanFly |

***3. POLYMORPHISM***

The word Poly means **“Many”** and morphism means **“Forms”.** Polymorphism is the ability of a single function, method, or object to take multiple forms. It allows one interface to be used for a general class of actions, promoting **flexibility** in code. There are two types of polymorphism in Java:

* **Compile-time (Method Overloading)**
* **Run-time (Method Overriding)**.

**Method overloading** allows multiple methods with the same name but different parameter lists to coexist within the same class, enabling different behaviors based on the argument types. **Method overriding** allows a subclass to provide a specific implementation of a method that is already defined in the parent class. Polymorphism is crucial for creating **generic code**, as the same method can behave differently depending on the object type at runtime. For example, a draw() method may behave differently when invoked on a Circle object versus a Rectangle object, while sharing the same method name in a superclass Shape.

**Code Example**

**(Method Overriding - Run-time Polymorphism):**

class Shape {

public void draw() {

System.out.println("Drawing a shape.");

}

}

class Circle extends Shape {

@Override

public void draw() {

System.out.println("Drawing a circle.");

}

}

class Square extends Shape {

@Override

public void draw() {

System.out.println("Drawing a square.");

}

}

public class Main {

public static void main(String[] args) {

Shape shape1 = new Circle(); // Polymorphism: Shape reference to Circle object

Shape shape2 = new Square(); // Polymorphism: Shape reference to Square object

shape1.draw(); // Calls the draw method in Circle

shape2.draw(); // Calls the draw method in Square

}

}

**Output:**

Drawing a circle.

Drawing a square.

**Method Overloading (Compile-time Polymorphism)**

class Calculator {

// Method to add two integers

public int add(int a, int b) {

return a + b;

}

// Overloaded method to add three integers

public int add(int a, int b, int c) {

return a + b + c;

}

// Overloaded method to add two double values

public double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

// Calling overloaded methods

System.out.println("Addition of two integers: " + calc.add(10, 20));

System.out.println("Addition of three integers: " + calc.add(10, 20, 30));

System.out.println("Addition of two doubles: " + calc.add(5.5, 3.7));

}

}

**Output:**

Addition of two integers: 30

Addition of three integers: 60

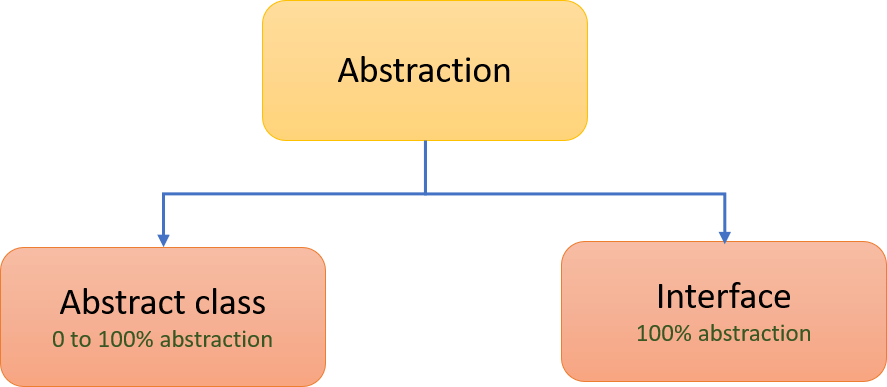
Addition of two doubles: 9.2

***4. CLASSES AND DATA ABSTRACTION***

Data abstraction in Object-Oriented Programming (OOP) refers to the concept of exposing only essential details and hiding unnecessary implementation details from the user. It allows users to interact with objects at a higher level of abstraction, while ignoring complex underlying logic. Abstraction enables a simpler interface for users while keeping the internal structure and working of the object hidden.

In Java, **abstract classes** and **interfaces** are tools that can achieve data abstraction. With abstract classes, you define methods that must be implemented by subclasses while also allowing for some method implementation. This helps in creating a structured hierarchy and enforces a certain level of functionality in subclasses.

For example, consider a class Shape, which defines an abstract method draw(). This method must be implemented by any class that extends Shape. Subclasses like Circle and Rectangle provide specific implementations of draw(). This allows the program to handle different shapes uniformly through abstraction, while each shape can have its own unique behavior.



**Abstract Classes in Java**

An **abstract class** is a class that cannot be instantiated directly and is meant to be subclassed. It allows defining abstract methods that must be implemented by the subclass. Abstract classes can also have non-abstract methods, which contain implementations that subclasses can inherit.

**Advantages of Data Abstraction:**

* **Simplifies code:** Users interact with a simple interface rather than complex internal logic.
* **Reduces complexity:** Unnecessary details are hidden, making code easier to maintain.
* **Enhances security:** Internal data is hidden and can only be accessed through exposed methods.
* **Modularity:** Code becomes modular, allowing changes to the implementation without affecting users.

**Example of Data Abstraction Using Abstract Class:**

// Abstract class

abstract class Vehicle {

// Abstract method (no implementation)

public abstract void start();

// Regular method (implemented method)

public void fuel() {

System.out.println("Fueling the vehicle.");

}

}

// Subclass that extends the abstract class

class Car extends Vehicle {

@Override

public void start() {

System.out.println("Starting the car.");

}

}

public class Main {

public static void main(String[] args) {

Vehicle myCar = new Car(); // Abstract class reference, Car object

myCar.start(); // Calls the start method in Car

myCar.fuel(); // Calls the fuel method from Vehicle class

}

}

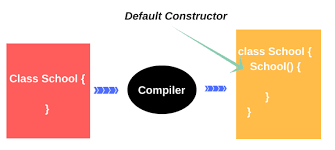
**Output:**

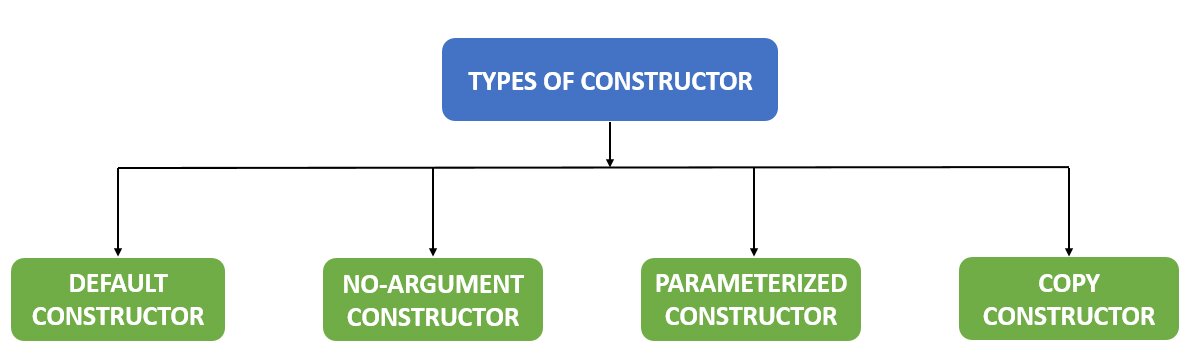
Starting the car.

Fueling the vehicle.

***5. CONSTRUCTORS***

A **constructor** is a special method in a class that is called when an object is **instantiated**. At the time of calling constructor, memory for the object is allocated in the memory. It initializes the object’s properties and does not return a value. Constructors have the same name as the class and can be **default** (no parameters) or **parameterized** (with parameters). Java provides a default constructor if none is specified, but you can define your own constructors to initialize object attributes with specific values. Constructors are crucial in object creation and provide a controlled way to initialize an object in a consistent and valid state. For instance, a Person class can have a constructor that accepts name and age as parameters, ensuring each new Person object is initialized with meaningful values. Unlike regular methods, constructors do not have a return type and are automatically called when an object is created.





**Code Example:**

class Student {

String name;

int age;

// Constructor with parameters

public Student(String name, int age) {

this.name = name;

this.age = age;

}

// Method to display student info

public void displayInfo() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

// Creating objects with parameterized constructor

Student student1 = new Student("John", 20);

student1.displayInfo(); // Output: Name: John, Age: 20

}

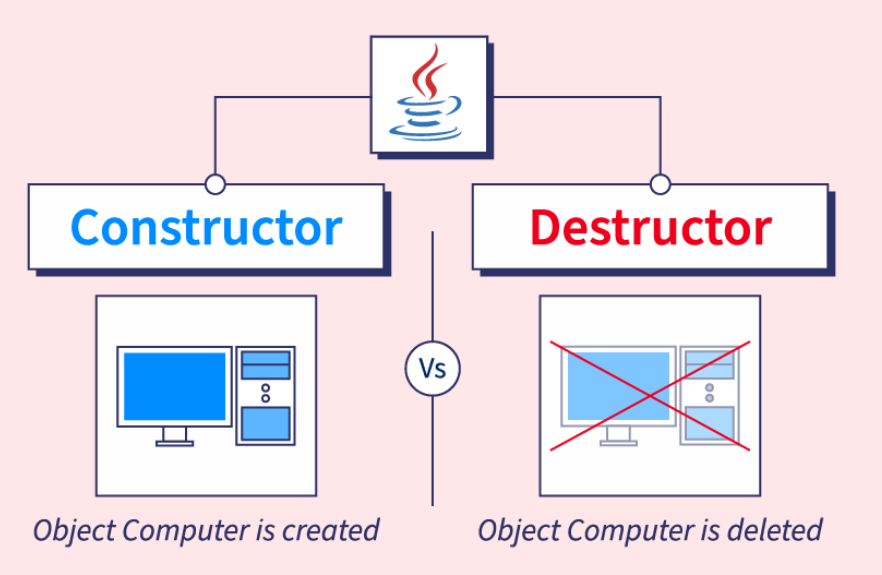
}

**Output:**

Name: John, Age: 20

***6. DESTRUCTORS (FINALIZE METHOD)***

Java doesn’t support destructors like C++, but it has a mechanism for resource cleanup known as **Garbage Collection (GC)**. Garbage collection automatically manages memory by deleting unused objects to free up space. Java’s alternative to destructors is the finalize() method, which is called by the garbage collector just before an object is destroyed. However, the finalize() method is rarely used today, as garbage collection is handled automatically and developers are encouraged to release resources (like closing files or network connections) explicitly rather than relying on finalize(). The unpredictability of the garbage collection process and its timing makes finalize() unreliable for critical resource management. Java provides better mechanisms like try-with-resources and finally blocks to handle resource deallocation.



**Destructors (Using finalize() in Java)**

**Code Example:**

class Resource {

// Constructor

public Resource() {

System.out.println("Resource created.");

}

// Called before object is destroyed

protected void finalize() {

System.out.println("Resource is being destroyed.");

}

}

public class Main {

public static void main(String[] args) {

Resource res = new Resource();

res = null; // Dereferencing the object

System.gc(); // Request garbage collection

}

}

**Output:**

Resource created.

Resource is being destroyed.

| **Basis** | **Constructor** | **Destructor** |
| --- | --- | --- |
| Purpose | Used to initialize the object of a class in Java. | There is no explicit destructor in Java; memory is handled by the Garbage Collector. |
| Syntax | Declared as className(parameters) {}. | Java does not have destructors; the finalize() method (deprecated) was used but is replaced by GC. |
| Parameters | Can accept parameters to initialize the object. | There are no arguments for a destructor since it does not exist in Java. |
| Invocation | Invoked when a new object is created using the new keyword. | Java’s garbage collector handles object destruction automatically; finalize() was invoked before. |
| Memory Management | Allocates memory and initializes object properties. | Memory deallocation is handled automatically by the garbage collector in Java. |
| Overloading | Can be overloaded by defining multiple constructors with different parameters. | No destructor in Java, hence no overloading concept for it. |
| Naming Convention | Constructor’s name must be the same as the class name. | No destructors in Java, so no naming convention applies. |
| Multiple Versions | Multiple constructors with different parameters (constructor overloading). | Java uses automatic garbage collection; no destructor concept or variations. |
| Copy Mechanism | Supports the concept of a copy constructor, though it must be explicitly defined. | No concept of a copy destructor, as Java manages object destruction through garbage collection. |
| Order of Execution | Invoked during object creation in the order defined. | The garbage collector determines the order of object destruction automatically. |