Java Oops

✅ Class and Object in Java – Beginner-Friendly Explanation

In **Java**, **Classes** and **Objects** are fundamental building blocks of **Object-Oriented Programming (OOP)**.

🔷 1. What is a Class?

A **class** is a **blueprint or template** for creating objects. It defines:

- **Fields (variables)** – to hold data.
- **Methods (functions)** – to perform actions.

> Think of a class like a **blueprint for a car**. It defines what a car has (wheels, engine) and what it can do (drive, stop), but it's not an actual car.

👉 Syntax:

```
java

class ClassName {
    // Fields (Variables)
    int num;
    String name;

    // Methods (Functions)
    void display() {
        System.out.println("Hello from a method!");
    }
}
```

🔷 2. What is an Object?

An **object** is an **instance** of a class. It is a **real-world entity** created from the blueprint (class).

> If the class is the **blueprint**, the object is the **actual car** built from it.

👉 Creating an object:

```
java

ClassName obj = new ClassName();
```

✅ Full Example

```java
// Define the class
class Car {
    // Fields
    String color;
    int speed;

    // Method
    void drive() {
        System.out.println("The car is driving.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Create object of Car class
        Car myCar = new Car();

        // Assign values to object fields
        myCar.color = "Red";
        myCar.speed = 120;

        // Access object fields and method
        System.out.println("Car color: " + myCar.color);
        System.out.println("Car speed: " + myCar.speed + " km/h");
        myCar.drive();
    }
}
```

🖨️ Output:

```yaml
Car color: Red
Car speed: 120 km/h
The car is driving.
```

🔑 Key Points

Concept	Meaning
`class`	Defines structure
`object`	Instance of a class
`new` keyword	Used to create an object
`.` (dot operator)	Access object members

Would you like a real-world example (like Student, BankAccount, etc.) or want to learn about constructors next?

Great! Let's learn about **Constructors in Java**, including **Default** and **Parameterized** constructors with simple examples.

🔷 What is a Constructor?

A **constructor** is a **special method** in Java that is used to **initialize objects**.

✅ Key Points:

- It has the **same name as the class**.
- It **does not have a return type** (not even `void`).
- It is **automatically called** when an object is created.

🔹 1. Default Constructor

A **default constructor** is the **no-argument** constructor provided by Java **automatically** if no constructor is defined by the user.
But you can also define it yourself.

👉 Example:

```
java

class Student {
    // Fields
    String name;
    int age;

    // Default constructor
    Student() {
        name = "Unknown";
        age = 0;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Default constructor is called
        s1.display();
    }
}
```

🖨️ Output:

```
yaml

Name: Unknown, Age: 0
```

🔹 2. Parameterized Constructor

A **parameterized constructor** takes arguments to **initialize the object with specific values**.

👉 Example:

```
java

class Student {
    String name;
    int age;

    // Parameterized constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s2 = new Student("John", 20);  // Passing values
        s2.display();
    }
}
```

🖨️ Output:

```
yaml

Name: John, Age: 20
```

🔸 Constructor Overloading

You can define **multiple constructors** in a class with different parameters. This is called **constructor overloading**.

👉 Example:

```
java

class Student {
    String name;
    int age;

    // Default constructor
    Student() {
        name = "Not Set";
        age = 0;
    }

    // Parameterized constructor
    Student(String n, int a) {
        name = n;
        age = a;
```

```
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

✅ Summary Table:

Type	Parameters	Purpose
Default Constructor	No	Initialize with default values
Parameterized	Yes	Initialize with given values
Overloaded Constructor	Varies	Multiple constructors with different arguments

Would you like to learn about **`this` keyword** or **constructor chaining (`this()`)** next?

Absolutely! Let's learn about the **`this` keyword** in Java — it's a very important concept in object-oriented programming.

🔷 What is `this` keyword in Java?

The **`this`** keyword is a **reference variable** in Java that **refers to the current object** of the class.

✅ Why is `this` used?

1. To **refer to current object** variables (when local and instance variable names are the same).
2. To **call one constructor from another** (constructor chaining).
3. To **pass current object as a parameter** to a method or constructor.
4. To **return the current object** from a method.

🔹 1. `this` to Refer to Instance Variables

When local variables (like parameters) **shadow** instance variables (same name), use **`this`** to refer to the instance variable.

👉 Example:

```
java

class Student {
    String name;
    int age;

    // Constructor
    Student(String name, int age) {
        this.name = name;  // Refers to instance variable
        this.age = age;    // Refers to instance variable
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Ravi", 22);
        s1.display();
    }
}
```

🖨️ Output:

```
yaml

Name: Ravi, Age: 22
```

Without **`this`**, the constructor would assign local variables to themselves — and instance variables remain uninitialized.

🔹 2. `this()` to Call Another Constructor (Constructor Chaining)

You can call another constructor of the same class using **`this()`**.

👉 Example:

```
java

class Student {
    String name;
    int age;

    // Default constructor
    Student() {
        this("Unknown", 0); // Calls parameterized constructor
    }

    // Parameterized constructor
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
}
```

🖨️ Output:

```
yaml

Name: Unknown, Age: 0
```

🔹 3. `this` to Pass Current Object as Parameter

You can pass the current object to another method.

👉 Example:

```
java

class Student {
    void display(Student s) {
        System.out.println("Method called with object: " + s);
    }

    void call() {
        display(this);  // Passing current object
    }
}
```

🔹 4. `this` to Return Current Object

👉 Example:

```java
class Student {
    Student getObject() {
        return this;
    }
}
```

✅ Summary Table

Usage	Purpose
`this.var = var`	Resolves naming conflict
`this()`	Calls another constructor
`this.method()`	Calls a method on current object
`return this`	Returns current object
`method(this)`	Pass current object as argument

Would you like to move next to **inheritance**, **super keyword**, or **method overriding**?

Great! Let's explore **Inheritance** in Java — a core concept of Object-Oriented Programming (OOP).

🔷 What is Inheritance?

Inheritance is the process by which one class (**child/subclass**) **inherits the properties and behaviors (fields and methods)** of another class (**parent/superclass**).

> Think of it like a child inheriting traits from their parents.

✅ Benefits of Inheritance:

- **Code reusability** – reuse common code in the parent class.
- **Method Overriding** – redefine inherited methods.
- **Polymorphism** – use parent reference to refer to child object.

🔸 Syntax

```
java

class Parent {
    // fields and methods
}

class Child extends Parent {
    // additional fields and methods
}
```

🔹 Example: Single Inheritance

```
java

// Parent class
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

// Child class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();   // Inherited method
        d.bark();  // Own method
    }
}
```

🖨️ Output:

```
nginx


This animal eats food.
Dog barks.
```

🔹 Types of Inheritance in Java

Type	Supported in Java?	Example
Single	✅ Yes	One child inherits one parent
Multilevel	✅ Yes	Child → Parent → Grandparent
Hierarchical	✅ Yes	Multiple children inherit one parent
Multiple	❌ No (with classes)	Java doesn't allow multiple inheritance with classes due to ambiguity (but supports it using **interfaces**)

🔹 Example: Multilevel Inheritance

```
java


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 p = new Puppy();
        p.eat();   // from Animal
        p.bark();  // from Dog
        p.weep();  // from Puppy
    }
}
```

🔹 `super` Keyword (Used in Inheritance)

The **`super`** keyword is used to:

1. Call the **parent class constructor**.

2. Access **parent class methods or variables**.

```java
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    void sound() {
        super.sound();  // Calls parent class method
        System.out.println("Dog barks");
    }
}
```

✅ Summary:

Term	Meaning
`extends`	Used to inherit from a class
`super`	Refers to parent class
`override`	Redefine method from parent class
Code reuse	Child class reuses parent logic

Would you like to go deeper into **`super` keyword**, **method overriding**, or learn about **interfaces and multiple inheritance using interfaces** next?

Excellent! Let's dive into **Method Overriding** in Java — a key concept in **runtime polymorphism** and inheritance.

🔷 What is Method Overriding?

Method Overriding occurs when a **subclass provides a specific implementation** of a method that is **already defined in its parent class**.

✅ Key Points:

- Same **method name**, **return type**, and **parameters**.
- The method in the child class **overrides** the method in the parent class.
- Enables **runtime polymorphism**.
- Use **`@Override` annotation** (optional but recommended).

🔹 Example of Method Overriding

```
java

class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // Polymorphism
        a.sound();             // Calls Dog's overridden method
    }
}
```

🖨️ Output:

```
nginx

Dog barks
```

🔹 Why Use Method Overriding?

- To change behavior of an inherited method.
- To implement **runtime polymorphism** (i.e., method call is resolved at **runtime** based on the object).

🔶 Rules for Method Overriding

Rule	Description
1. Method Signature	Must have the **same name, return type, and parameters**
2. Access Modifier	Cannot reduce visibility (e.g., **`public`** → **`private`** ❌)
3. Final Methods	Cannot override methods declared as **`final`**
4. Static Methods	**Cannot** override static methods (they are class-level, not instance-level)
5. Constructors	**Cannot** be overridden
6. Use **`@Override`**	Optional but helps detect errors at compile-time

🔹 Example: Access Modifier Rule

```
java


class Parent {
    public void show() {
        System.out.println("Parent show");
    }
}

class Child extends Parent {
    // private void show() { ❌ Not allowed: reduced visibility
    public void show() {
        System.out.println("Child show");
    }
}
```

🔹 Using `super` to Call Parent Method

You can still access the **parent class method** using the **`super`** keyword.

```
java


class Dog extends Animal {
    @Override
    void sound() {
        super.sound();  // Calls parent class method
        System.out.println("Dog barks");
    }
}
```

🔹 Real-World Example

```
java


class Bank {
    int getInterestRate() {
        return 0;
    }
}
```

```
class SBI extends Bank {
    int getInterestRate() {
        return 5;
    }
}

class ICICI extends Bank {
    int getInterestRate() {
        return 6;
    }
}

public class Main {
    public static void main(String[] args) {
        Bank b1 = new SBI();
        Bank b2 = new ICICI();
        System.out.println("SBI Interest Rate: " + b1.getInterestRate());
        System.out.println("ICICI Interest Rate: " + b2.getInterestRate());
    }
}
```

🖨️ Output:

```
yaml


SBI Interest Rate: 5
ICICI Interest Rate: 6
```

✅ Summary

Concept	Description
Method Overriding	Redefining parent class method in child
`@Override`	Indicates overriding is intentional
`super.method()`	Calls parent method
Runtime Polymorphism	Behavior decided at runtime

Would you like to now learn about **method overloading**, **runtime vs compile-time polymorphism**, or **abstract classes and methods**?

Great choice! Let's break down **Polymorphism** in Java — one of the **four pillars of Object-Oriented Programming (OOP)**.

🔷 What is Polymorphism?

Polymorphism means **"many forms"**. In Java, polymorphism allows you to **perform a single action in different ways**.

> One interface, many implementations.

🔸 Types of Polymorphism in Java:

Type	Also Called	When Decided
Compile-time	Method Overloading	At compile-time
Runtime	Method Overriding	At runtime

✅ 1. Compile-Time Polymorphism (Method Overloading)

Occurs when **multiple methods** in the **same class** have the **same name but different parameters**.

🔹 Example:

```
java

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

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));          // 5
        System.out.println(calc.add(2.5, 3.5));      // 6.0
        System.out.println(calc.add(1, 2, 3));       // 6
    }
}
```

✅ Key Points:

- Resolved by **compiler**.
- Method name is same but **parameter list differs** (type, number, order).
- Also known as **static binding** or **early binding**.

✅ 2. Runtime Polymorphism (Method Overriding)

Occurs when a **child class provides a specific implementation** of a method already defined in its **parent class**.

🔹 Example:

```
java

class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a;
        a = new Dog();
        a.sound();  // Dog barks

        a = new Cat();
        a.sound();  // Cat meows
    }
}
```

✅ Key Points:

- Resolved at **runtime** using the **object's actual type**.
- Requires **inheritance** and **method overriding**.
- Also called **dynamic binding** or **late binding**.

🔸 Differences Between Compile-Time and Runtime Polymorphism

Feature	Compile-Time (Overloading)	Runtime (Overriding)
Resolved at	Compile-time	Runtime
Binding type	Static binding	Dynamic binding
Class relationship	Same class	Parent-child (Inheritance)
Method signature	Different	Same

Feature	Compile-Time (Overloading)	Runtime (Overriding)
Flexibility	Less flexible	More flexible (polymorphic)

✅ Summary

Polymorphism Type	Mechanism	Example
Compile-time	Method Overloading	**`add(int, int)`** and **`add(double, double)`**
Runtime	Method Overriding	**`sound()`** in **`Dog`**, **`Cat`** overriding **`Animal`**

Would you like to explore **interfaces and polymorphism**, or maybe go deeper into **abstract classes vs interfaces** next?

Great! Let's now understand **Encapsulation** — another key pillar of **Object-Oriented Programming (OOP)** in Java.

🔷 What is Encapsulation?

Encapsulation is the process of **wrapping data (variables)** and **methods (functions)** into a single unit (class), and **restricting direct access** to some of the object's components.

> Think of it like a **capsule** where everything is packed and protected inside.

✅ Main Goals of Encapsulation:

1. **Data hiding** – prevent external code from accessing internal details.
2. **Controlled access** – access/modification through public methods (getters/setters).
3. **Improved security** and **maintainability**.

🔹 How to Achieve Encapsulation in Java?

Step	Description
Make variables **`private`**	To **hide** data from outside access
Use **`public`** getter/setter	To provide **controlled access**
Keep methods inside class	Keep implementation details hidden

🔸 Example: Encapsulation in Action

```
java

class Student {
    // Private data members (data hiding)
    private String name;
    private int age;

    // Public getter method
    public String getName() {
        return name;
    }

    // Public setter method
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age > 0) {
            this.age = age;
        } else {
            System.out.println("Age must be positive.");
        }
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();

        // Setting values using setters
        s.setName("Ravi");
        s.setAge(22);

        // Getting values using getters
        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());
    }
}
```

🖨️ Output:

```
makefile

Name: Ravi
Age: 22
```

🔹 Benefits of Encapsulation

✅ Improved data protection
✅ Code maintainability
✅ Control over how data is accessed/modified
✅ Can add validation logic inside setters

🔹 Real-Life Analogy

Think of a **TV remote**:

- The inner circuits (data) are hidden from you.
- You interact with it using **buttons** (methods).
- You don't need to know how it works internally — just what buttons to press.

✅ Summary:

Concept	Explanation
Encapsulation	Binding data + methods & hiding internal details
Access Modifier	Use **`private`** for variables
Getter/Setter	Use **`public`** methods to access data
Benefit	Security, control, flexibility

Would you like to now move to **Abstraction**, or explore **Access Modifiers (private, public, protected)** in detail?

Great! Let's now learn about **Abstraction** in Java — another fundamental concept of **Object-Oriented Programming (OOP)**.

🔷 What is Abstraction?

Abstraction means **hiding internal implementation details** and **showing only the essential features** of an object.

> Think of abstraction as **showing only what is necessary** and hiding complexity.

✅ Real-Life Analogy:

- A **car** has a **steering wheel, pedals, and buttons** — you use them to drive.
- You **don't need to know how the engine, brake system, or sensors work** internally.
- That's abstraction — **hide complexity, show functionality**.

🔹 In Java, Abstraction is Achieved Using:

Tool	Description
Abstract Class	Partial abstraction (0% to 100%)
Interface	Full abstraction (100%)

🔸 1. Abstract Class

An **abstract class**:

- Cannot be instantiated (no objects can be created).
- Can have **abstract methods** (without body) and **concrete methods** (with body).
- Is used when some shared behavior is needed but specific parts should be defined by subclasses.

👉 Syntax:

```
java

abstract class Animal {
    abstract void sound(); // abstract method (no body)

    void eat() { // concrete method
        System.out.println("Animals eat food");
    }
}
```

✅ Example:

```java
abstract class Animal {
    abstract void sound();  // Abstract method

    void sleep() {
        System.out.println("Animal sleeps");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();  // Polymorphism
        a.sound();  // Dog barks
        a.sleep();  // Animal sleeps
    }
}
```

🔸 2. Interface (100% Abstraction)

An **interface**:

- Only contains **abstract methods** (until Java 7).
- From Java 8+, it can have **default** and **static** methods.
- **Cannot have instance variables**, only **constants** (`public static final`).
- A class **implements** an interface.

👉 Example:

```java
interface Shape {
    void draw();  // abstract method
}

class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

✅ Summary Table:

Feature	Abstract Class	Interface
Inheritance	**`extends`**	**`implements`**
Abstraction Level	Partial (0-100%)	Full (100%)
Method Types	Abstract + Concrete	Only abstract (default/static allowed from Java 8)

Feature	Abstract Class	Interface
Multiple Inheritance	Not supported	Supported via multiple interfaces
Variables	Instance variables allowed	Only constants (**`public static final`**)

✅ Why Use Abstraction?

- **Hide complex code** from the user.
- **Improve code maintainability**.
- **Focus on essential features**.
- **Standardize structure** via interfaces and abstract classes.

✅ Quick Example Comparison

```
java

abstract class Vehicle {
    abstract void run();
}

interface Engine {
    void start();
}
```

Would you like to dive deeper into **interface vs abstract class**, or see examples of **multiple interfaces** and **default methods** in action?

Great! Let's understand the difference between **static** and **non-static** (also called **instance**) members in Java — a very important concept.

🔷 What are Members in Java?

In Java, **members** refer to:

- **Variables** (fields)
- **Methods**
- **Blocks**
- **Nested classes**

They can be either:

- **Static** (class-level)
- **Non-static** (object-level or instance-level)

✅ Static Members

- Belong to the **class**, not to any specific object.
- Shared among **all instances** of the class.
- Can be accessed **without creating an object**.
- Declared using the keyword **`static`**.

🔸 Static Variable Example:

```
java

class Employee {
    int id;                  // Non-static
    String name;             // Non-static
    static String company = "TechCorp";  // Static

    Employee(int i, String n) {
        id = i;
        name = n;
    }

    void display() {
        System.out.println(id + " " + name + " " + company);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee(101, "Alice");
        Employee e2 = new Employee(102, "Bob");

        e1.display();
        e2.display();
    }
}
```

🖨️ Output:

```
101 Alice TechCorp
102 Bob TechCorp
```

✅ **`company`** is shared across all objects.

🔸 Static Method Example:

```java
class Utility {
    static void greet() {
        System.out.println("Welcome to Java!");
    }
}

public class Main {
    public static void main(String[] args) {
        Utility.greet();  // Called without object
    }
}
```

✅ **`greet()`** is called directly from the class.

✅ Non-static (Instance) Members

- Belong to a **specific object**.
- Each object has its **own copy** of the non-static fields.
- Can be accessed **only through an object**.
- No **`static`** keyword is used.

🔸 Differences Between Static and Non-static Members

Feature	Static Members	Non-static Members
Belongs to	Class	Instance (object)
Accessed by	Class name or object	Object only
Memory	Shared among all objects	Each object has its own copy
Initialization	When class is loaded	When object is created
Keyword	Declared with **`static`**	No **`static`** keyword
Example	**`static int count;`**	**`int age;`**

🔹 Example Showing Both Together

```java
class Student {
    static String college = "ABC College";  // static member
    String name;                            // non-static member

    Student(String name) {
        this.name = name;
    }

    void show() {
        System.out.println(name + " studies in " + college);
    }

    static void changeCollege(String newCollege) {
        college = newCollege;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("John");
        Student s2 = new Student("Emma");

        s1.show();  // John studies in ABC College
        s2.show();  // Emma studies in ABC College

        Student.changeCollege("XYZ University");

        s1.show();  // John studies in XYZ University
        s2.show();  // Emma studies in XYZ University
    }
}
```

✅ Summary

Concept	Static	Non-static
Belongs to	Class	Object
Shared?	Yes	No
Access	Directly via class	Only via object
Memory use	Less (shared)	More (individual copies)
Example	**`static int count;`**	**`int age;`**

Would you like to explore **static blocks**, **static nested classes**, or practice some real-world questions using static vs non-static concepts?

Great! Let's explore the **`final` keyword** in Java — it's simple but very powerful.

🔷 What is `final` in Java?

The **`final`** keyword is used to **restrict the user**. Once something is declared **`final`**, it **cannot be changed** or **overridden**.

You can use **`final`** with:

1. **Variables**
2. **Methods**
3. **Classes**

✅ 1. `final` Variable (Constant)

A **`final`** variable is a **constant**. Its value **cannot be changed** after it is initialized.

👉 Example:

```
java

public class Main {
    public static void main(String[] args) {
        final int x = 10;
        // x = 20; // ❌ Error: cannot assign a value to final variable
        System.out.println(x);
    }
}
```

- You **must initialize** a final variable **once**, either at declaration or inside the constructor.

✅ 2. `final` Method

A **`final`** method **cannot be overridden** by a subclass.

👉 Example:

```
java

class Animal {
    final void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    // void sound() { ❌ Error: Cannot override final method
    //     System.out.println("Dog barks");
    // }
}
```

✅ Use it when you want to **lock** the method's behavior in subclasses.

✅ 3. `final` Class

A **`final`** class **cannot be inherited** (extended).

👉 Example:

```java
final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

// class Car extends Vehicle { ❌ Error: Cannot inherit from final class }

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.run();
    }
}
```

✅ Summary Table

Usage	Restriction
`final` variable	Cannot change the value (constant)
`final` method	Cannot override in subclass
`final` class	Cannot be extended (no inheritance)

🔹 Additional Notes:

- **`final`** with objects: you **can't reassign** the object, but **you can change its fields**.

```java
final Student s = new Student();
s.name = "Ravi";    // ✅ allowed
// s = new Student(); // ❌ not allowed
```

Would you like to move next to **`static final` (constants)**, **`finalize()` method**, or **difference between final, finally, and finalize**?