Java OOPs Intro:

🔷 Introduction to OOPs in Java

Java follows the **Object-Oriented Programming (OOPs) paradigm**, which is based on the concept of **objects** interacting with each other.

🔶 Key Principles of OOPs

1. **Class & Object** → (Building blocks of OOP)
2. **Encapsulation** → (Data hiding using access modifiers)
3. **Inheritance** → (Code reuse between parent & child classes)
4. **Polymorphism** → (Method Overloading & Overriding)
5. **Abstraction** → (Hiding unnecessary details)

🔷 Class and Object in Java

🔶 What is a Class?

A **class** is a blueprint or template that defines **properties (variables) and behavior (methods)** for objects.

🔶 Syntax of a Class

```
class ClassName {
    // Fields (Variables)
    dataType variableName;

    // Methods
    returnType methodName() {
        // Method body
    }
}
```

Example of a Class

```
class Car {
    // Properties (Attributes)
    String brand;
    int speed;

    // Behavior (Method)
    void accelerate() {
        speed += 10;
        System.out.println(brand + " is accelerating. Speed: " + speed);
    }
}
```

🔶 What is an Object?

An **object** is an instance of a class that contains **real data** and can call methods.

Creating Objects

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

        // Assigning values to object properties
        myCar.brand = "Toyota";
        myCar.speed = 50;

        // Calling the method
        myCar.accelerate();
    }
}
```

🔷 **Output:**

```
Toyota is accelerating. Speed: 60
```

🔶 Relationship Between Class and Object

Feature	Class	Object
Definition	A blueprint for creating objects	An instance of a class
Contains	Variables & Methods	Real data
Memory Allocation	No memory until object is created	Memory allocated at runtime
Example	`Car` (General idea)	`Toyota`, `Ford` (Real-world instances)

🔷 Access Modifiers in Java

Access modifiers define the **visibility** and **accessibility** of classes, methods, and variables.

Types of Access Modifiers

Modifier	Class	Package	Subclass	World
public	✅	✅	✅	✅
protected	✅	✅	✅	❌
default (no modifier)	✅	✅	❌	❌
private	✅	❌	❌	❌

Examples of Access Modifiers

1. Public Modifier (Accessible Everywhere)

```
public class PublicExample {
    public int number = 10;
    public void show() {
        System.out.println("Public method called");
```

```
    }

    public static void main(String[] args) {
        PublicExample obj = new PublicExample();
        obj.show();
    }
}
```

🔷 **Output:**

```
Public method called
```

2. Private Modifier (Accessible Only Within the Same Class)

```
class PrivateExample {
    private int number = 20;
    private void display() {
        System.out.println("Private method called");
    }

    public static void main(String[] args) {
        PrivateExample obj = new PrivateExample();
        obj.display(); // Accessible within the class
    }
}
```

🔷 **Output:**

```
Private method called
```

3. Protected Modifier (Accessible Within the Same Package and Subclasses)

```
class Parent {
    protected void show() {
```

```
        System.out.println("Protected method called");
    }
}

class Child extends Parent {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.show(); // Accessible because it's inherited
    }
}
```

🔷 **Output:**

```
Protected method called
```

4. Default (No Modifier) - Accessible Only Within the Same Package

```
class DefaultExample {
    void show() {
        System.out.println("Default method called");
    }
}

class Test {
    public static void main(String[] args) {
        DefaultExample obj = new DefaultExample();
        obj.show(); // Accessible within the same package
    }
}
```

🔷 **Output:**

```
Default method called
```

📌 Methods and Functions in Java

🔷 1. Introduction to Methods

A **method** in Java is a block of code designed to perform a specific task. It enhances **code reusability, modularity, and maintainability**.

💡 Why Use Methods?

✅ **Code Reusability** – Write once, use multiple times.

✅ **Modularity** – Breaks code into smaller, manageable parts.

✅ **Readability & Maintainability** – Easier to debug and modify.

✅ **Encapsulation** – Hides implementation details from users.

🔷 2. Types of Methods in Java

Java provides different types of methods based on their functionality and purpose:

1️⃣ Predefined Methods (Built-in Methods)

- Java provides a set of built-in methods that can be directly used.
- Found in Java libraries like `Math`, `System`, `String`, `Arrays`, etc.

Example: Using Built-in Methods

```
java
CopyEdit
public class PredefinedExample {
    public static void main(String[] args) {
        String str = "Java Programming";
        System.out.println("Length: " + str.length());  // Built-in method length()
        System.out.println("Square Root of 25: " + Math.sqrt(25));  // Math class
method
    }
```

```
}
```

Output:

```
mathematica
CopyEdit
Length: 16
Square Root of 25: 5.0
```

2️⃣ User-Defined Methods

- These are custom methods created by programmers.
- Useful for performing **specific tasks** in an organized manner.

Example: Creating a User-Defined Method

```
java
CopyEdit
public class UserDefinedExample {
    public void greet() {
        System.out.println("Hello, welcome to Java!");
    }

    public static void main(String[] args) {
        UserDefinedExample obj = new UserDefinedExample();
        obj.greet();
    }
}
```

Output:

```
css
CopyEdit
Hello, welcome to Java!
```

3️⃣ Parameterized Methods

- These methods accept **input values (parameters)**.
- Parameters help in making methods more **dynamic and reusable**.

Example: Using Parameters in Methods

```
java
CopyEdit
public class ParameterExample {
    public void add(int a, int b) {
        int sum = a + b;
        System.out.println("Sum: " + sum);
    }

    public static void main(String[] args) {
        ParameterExample obj = new ParameterExample();
        obj.add(5, 10); // Passing values to the method
    }
}
```

Output:

```
makefile
CopyEdit
Sum: 15
```

4️⃣ Methods with Return Type

- These methods **return a value** using the `return` keyword.
- The return type must be specified (e.g., `int`, `String`, `boolean`).

Example: Method Returning an Integer

```
java
CopyEdit
public class ReturnExample {
    public int square(int num) {
        return num * num;
    }

    public static void main(String[] args) {
        ReturnExample obj = new ReturnExample();
        int result = obj.square(6);
        System.out.println("Square: " + result);
    }
}
```

Output:

```
makefile
CopyEdit
Square: 36
```

5️⃣ Static vs. Non-Static Methods

🔶 Static Methods

- Declared using the `static` keyword.
- **Can be called without creating an object**.

- Belongs to the **class, not an instance**.

Example: Static Method

```
java
CopyEdit
public class StaticExample {
    public static void showMessage() {
        System.out.println("This is a static method.");
    }

    public static void main(String[] args) {
        StaticExample.showMessage(); // Direct call without an object
    }
}
```

Output:

```
csharp
CopyEdit
This is a static method.
```

🔷 Non-Static Methods

- Require an **object** to be called.
- Belong to an **instance of a class**.

Example: Non-Static Method

```
java
CopyEdit
public class NonStaticExample {
    public void display() {
```

```
        System.out.println("This is a non-static method.");
    }

    public static void main(String[] args) {
        NonStaticExample obj = new NonStaticExample(); // Object creation
        obj.display();
    }
}
```

Output:

```
csharp
CopyEdit
This is a non-static method.
```

🔷 3. Method Overloading

- **Multiple methods with the same name but different parameters**.
- Helps in increasing the **readability and reusability** of code.

Example: Method Overloading

```
java
CopyEdit
public class OverloadingExample {
    public void display(int num) {
        System.out.println("Integer: " + num);
    }

    public void display(String text) {
        System.out.println("String: " + text);
    }
```

```
    public static void main(String[] args) {
        OverloadingExample obj = new OverloadingExample();
        obj.display(10);
        obj.display("Hello Java");
    }
}
```

Output:

```
makefile
CopyEdit
Integer: 10
String: Hello Java
```

🔷 4. Return Type vs. Non-Return Type Methods

Feature	Return Type Method	Non-Return Type Method (void)
Definition	Returns a value using `return`	Performs an action without returning
Return Type	Must specify (`int` , `String` , `boolean`)	Must use `void`
Return Statement	Required (`return value;`)	Not required
Example	`public int getNumber() { return 10; }`	`public void display() { System.out.println("Hello"); }`

🔷 5. Key Differences: Static vs. Non-Static Methods

Feature	Static Method	Non-Static Method
Belongs To	Class	Object (Instance)
Called Using	Class name (`ClassName.method()`)	Object (`obj.method()`)
Requires Object?	❌ No	✅ Yes

Example	`public static void show() {}`	`public void display() {}`

🔷 6.Pass by Value in Java

- **Java always uses "pass by value"** when passing arguments to methods.
- This means a **copy** of the value is passed, not the original variable.

Example: Pass by Value (Primitive Data Types)

```
java
CopyEdit
public class PassByValue {
    public void modify(int num) {
        num = num + 10; // Modifying the copied value
    }

    public static void main(String[] args) {
        PassByValue obj = new PassByValue();
        int x = 50;
        obj.modify(x);
        System.out.println("Original Value: " + x); // Value remains unchanged
    }
}
```


Output:

```
yaml
CopyEdit
Original Value: 50
```

➡️ The value **does not change** because **Java does not pass the actual variable, only its copy**.

Method Modifiers & Return Types in Java

1. Method Modifiers in Java

Java methods can have multiple access modifiers, but only in valid combinations. Below are the valid and invalid method modifier combinations:

✅ Valid Combinations

1. `static final`
   - A method can be both `static` and `final`, meaning it belongs to the class and cannot be overridden.

```
class Example {
    static final void show() {
        System.out.println("Static Final Method");
    }

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

Output:

```
Static Final Method
```

2. `static synchronized`
   - A `static` method can also be `synchronized`, meaning it is thread-safe for class-level operations.

```
class Example {
    static synchronized void show() {
        System.out.println("Static Synchronized Method");
    }
```

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

Output:

```
Static Synchronized Method
```

3. `final synchronized`
   - A method can be both `final` and `synchronized`, meaning it cannot be overridden and is thread-safe.

```
class Example {
    final synchronized void show() {
        System.out.println("Final Synchronized Method");
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.show();
    }
}
```

Output:

```
Final Synchronized Method
```

❌ Invalid Combinations

1. `static abstract` ❌
   - A method **cannot** be both `static` and `abstract` because:
     - `static` methods **belong to the class** and cannot be overridden.

- `abstract` methods **must be overridden** in subclasses.

```
abstract class Example {
    static abstract void show(); // ❌ ERROR
}
```

Compilation Error:

```
Modifier 'static' not allowed here
```

2. **`final abstract`** ❌
   - A method **cannot** be both `final` and `abstract` because:
     - `final` methods **cannot be overridden**.
     - `abstract` methods **must be overridden** in subclasses.

```
abstract class Example {
    final abstract void show(); // ❌ ERROR
}
```

Compilation Error:

```
Modifier 'final' not allowed here
```

---

2. Return Types in Java Methods

A method in Java can return different types of values:

✅ Valid Return Types

1. **Primitive Data Types (`int`, `char`, `double`, etc.)**

```
class Example {
    int getNumber() {
        return 42;
```

```
    }

    public static void main(String[] args) {
        Example obj = new Example();
        System.out.println("Returned Number: " + obj.getNumber());
    }
}
```

Output:

```
Returned Number: 42
```

2. **User-Defined Data Types (Objects)**

```
class Example {
    String message;

    Example(String msg) {
        this.message = msg;
    }

    Example getObject() {
        return new Example("Returning an Object");
    }

    public static void main(String[] args) {
        Example obj = new Example("Hello");
        Example newObj = obj.getObject();
        System.out.println(newObj.message);
    }
}
```

Output:

```
Returning an Object
```

3. `void` **(No Return Value)**

```
class Example {
    void display() {
        System.out.println("Void Method Called");
    }

    public static void main(String[] args) {
        Example obj = new Example();
        obj.display();
    }
}
```

Output:

```
Void Method Called
```

---

3. Summary Notes

- **Methods can have multiple access modifiers, but only in valid combinations.**
- ✅ **Valid Combinations:**
  - `static final` → Belongs to class and cannot be overridden.
  - `static synchronized` → Class-level thread safety.
  - `final synchronized` → Instance-level thread safety and cannot be overridden.
- ❌ **Invalid Combinations:**
  - `static abstract` → `static` methods cannot be abstract.
  - `final abstract` → `final` methods cannot be abstract.
- **Return Types Allowed:**
  - **Primitive types** (`int`, `char`, `boolean`, etc.)
  - **Objects** (User-defined classes)

- `void` (No return value)

- **Modifiers Meaning:**
  - `static` → Method belongs to the class, not objects.
  - `final` → Method cannot be overridden.
  - `synchronized` → Ensures thread safety.
  - `abstract` → Method must be overridden in a subclass.

Explanation of Method Information & Types in Java

1. Describing Method Information

In Java, there are two ways to describe a method:

✅ Method Signature

- **Definition**: A method signature consists of:
  - **Method name**
  - **Parameter list** (types and order of parameters)
- **Example**:

```
java
CopyEdit
void display(int a, String name)
```

  - Method name: `display`
  - Parameter list: `(int a, String name)`

✅ Method Prototype

- **Definition**: A method prototype includes:
  - **Access Modifiers** (public, private, protected, etc.)

- **Return Type** (void, int, String, Object, etc.)
- **Method Name**
- **Parameter List**
- **Throws Clause** (if the method throws exceptions)

- **Example**:

```
java
CopyEdit
public int getData(String name) throws IOException
```

- Access Modifier: `public`
- Return Type: `int`
- Method Name: `getData`
- Parameter List: `(String name)`
- Exception: `throws IOException`

---

2. Types of Methods in Java (Based on Object State Manipulation)

✅ 1. Mutator Methods (Setter Methods)

- **Definition**: Mutator methods modify or set the values of object properties.
- **Example**: `setXXX()` methods in Java Bean classes.
- **Example in Java**:

```
java
CopyEdit
public class Student {
    private String name;
```

```
    // Mutator method
    public void setName(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Alice"); // Modifying object data
    }
}
```

✅ 2. Accessor Methods (Getter Methods)

- **Definition**: Accessor methods retrieve or access the values of object properties.
- **Example**: `getXXX()` methods in Java Bean classes.
- **Example in Java**:

```
java
CopyEdit
public class Student {
    private String name;

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

    // Accessor method
    public String getName() {
        return name;
    }
```

```
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Alice");
        System.out.println("Student Name: " + s.getName()); // Retrieving obj
ect data
    }
}
```

Output:

```
yaml
CopyEdit
Student Name: Alice
```

---

3. Java Bean Class

- A **Java Bean** is a simple Java class that follows these rules:
  1. **Private properties** (variables).
  2. **Public getter and setter methods**.
  3. **A no-argument constructor**.
- **Example of a Java Bean Class**:

```
java
CopyEdit
public class Employee {
    private int empId;
    private String empName;

    // No-argument constructor
    public Employee() {}
```

```
    // Mutator methods (Setters)
    public void setEmpId(int empId) {
        this.empId = empId;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    // Accessor methods (Getters)
    public int getEmpId() {
        return empId;
    }

    public String getEmpName() {
        return empName;
    }

    public static void main(String[] args) {
        Employee e = new Employee();
        e.setEmpId(101);
        e.setEmpName("John Doe");

        System.out.println("Employee ID: " + e.getEmpId());
        System.out.println("Employee Name: " + e.getEmpName());
    }
}
```


Output:

```
yaml
CopyEdit
Employee ID: 101
```

```
Employee Name: John Doe
```

📌 Summary Notes

Describing Method Information

Approach	Includes
Method Signature	Method Name + Parameter List
Method Prototype	Access Modifiers + Return Type + Method Name + Parameter List + Throws Clause

Types of Methods in Java

Type	Purpose	Example Methods
Mutator Methods	Modify/set data in objects	`setXXX()`
Accessor Methods	Retrieve/access data from objects	`getXXX()`

Java Bean Class Rules

1. Private instance variables.
2. Public getter (`getXXX()`) and setter (`setXXX()`) methods.
3. No-argument constructor.

Java Memory Management and how JVM Works:-

Step 1: Loading a Class into Memory

When you run a Java program, the **Java Virtual Machine (JVM)** loads the required class into memory.

For example, if you have a class named `TestMain`, its **bytecode** (compiled `.class` file) is loaded into a special memory area called the **Method Area**.

At this stage, the JVM automatically creates an internal **metadata object** for the class in **Heap Memory**, which contains details like:

✔️ Class variables

✔️ Methods

✔️ Implementations

Step 2: Creating the `main()` Method Stack

- The JVM looks for the `main()` method inside the `TestMain` class.
- If found, a **Main Thread Stack** is created in **Stack Memory**.
- The stack is used to manage method calls and local variables during execution.

Step 3: Creating an Object (e.g., `Test t = new Test();`)

When we write:

```
java
CopyEdit
Test t = new Test();
```

JVM performs the following steps:

✅ Step 3.1: Loading the `Test` Class

- If the `Test` class is not already loaded, JVM loads its **bytecode** into the **Method Area**.
- Metadata (class variables, methods) is stored in **Heap Memory**.

✅ Step 3.2: Allocating Memory for the Object

- JVM calculates **how much memory** is needed for the object based on its **instance variables and their data types**.
- It informs the **Heap Manager** to create the object.

✅ Step 3.3: Object Creation in Heap Memory

- The **Heap Manager** creates the object as a **single unit** inside the **Heap Memory**.
- A **unique integer ID (hashCode)** is assigned to the object.

✅ Step 3.4: Assigning a Reference Variable

- The **hashCode** (unique ID) is converted into a **hexadecimal reference value**.
- This reference value is stored in the **reference variable (`t`)**.
- Now, `t` points to the object in Heap Memory.

Example Code with Explanation

```
java
CopyEdit
class Test {
    int x = 10; // Instance variable
}

public class TestMain {
    public static void main(String[] args) {
        Test t = new Test(); // Object creation
        System.out.println(t); // Printing reference value
    }
}
```

✅ Step-by-Step Execution

1. `TestMain` class is loaded into **Method Area**.
2. JVM creates a **Main Thread Stack**.
3. `new Test();`
    - JVM loads `Test` class (if not already loaded).
    - **Heap Manager** creates an object and assigns it a unique **hashCode**.
    - The **hashCode** is converted to **reference value** and stored in `t`.

Sample Output

```
css
CopyEdit
Test@1b6d3586
```

(The reference value is the hexadecimal form of the hashCode.)

Summary

Step	What Happens?
1. Load Class	Class bytecode is loaded into **Method Area**.
2. Create Main Stack	If `main()` is found, a **Main Thread Stack** is created.
3. Load Test Class	`Test` class bytecode is loaded into the **Method Area** (if needed).
4. Calculate Memory	JVM calculates object size based on instance variables.
5. Create Object	The **Heap Manager** allocates memory for the object.
6. Assign Reference	Object gets a **unique hashCode**, converted into a **reference value** and stored in a variable.

In Simple Words 🧠

- **JVM loads the class.**
- **It creates a stack for method execution.**

- **When an object is created (`new Test();`), memory is allocated in the Heap.**
- **Each object gets a unique ID (hashCode), converted to a reference variable (`t`).**

🚀 Java Program Execution & JVM Memory Management (Diagram)

Code Example:

```
java
CopyEdit
class Test {
    int x = 10; // Instance variable
}

public class TestMain {
    public static void main(String[] args) {
        Test t = new Test(); // Object creation
        System.out.println(t); // Printing reference value
    }
}
```


🔷 Step 1: JVM Loads `TestMain` Class

📌 Method Area:

- `TestMain` class bytecode is loaded.
- Metadata (method details, variables, etc.) is stored.

📝 Diagram Representation:

```
markdown
CopyEdit
Method Area
```

```
---------------
| TestMain Class |
---------------
```

🔷 Step 2: JVM Creates the Main Thread Stack

📌 **Stack Memory:**

- JVM checks if `main()` is present in `TestMain`.
- A **Main Thread Stack** is created.

📝 **Diagram Representation:**

```
markdown
CopyEdit
Stack Memory
--------------------
|  Main Thread Stack |
--------------------
```

🔷 Step 3: JVM Executes `Test t = new Test();`

📌 **Class Loading:**

- `Test` class bytecode is loaded into the **Method Area** (if not already loaded).

📌 **Heap Memory:**

- JVM calculates memory size for `Test` object.
- The **Heap Manager** creates an object.
- JVM assigns a unique **hashCode** to the object.

📌 **Reference Value:**

- The **hashCode** is converted into a **hexadecimal reference value** (e.g., `1b6d3586`).

- This reference is stored in variable `t` in **Stack Memory**.

📝 **Diagram Representation:**

```
pgsql
CopyEdit
Method Area            Heap Memory           Stack Memory
-------------------    --------------------  --------------------
| TestMain Class |     | Test Object        |  | Main Thread Stack |
| Test Class     | --> | x = 10             |  |-------------------|
-------------------    | hashCode: 123456   |  | t = 0x1b6d3586    |
                       --------------------  --------------------
```

🔷 `t` **(reference variable) in Stack Memory points to the object in Heap Memory.**

🔷 **Object hashCode (** `123456` **) is converted to** `0x1b6d3586` **(hex reference).**

🔷 Step 4: JVM Executes `System.out.println(t);`

📌 **JVM prints the reference value:**

```
css
CopyEdit
Test@1b6d3586
```

📌 Summary in Diagram

```
pgsql
CopyEdit
1️⃣ JVM loads TestMain class into Method Area.
2️⃣ JVM creates a Main Thread Stack in Stack Memory.
3️⃣ JVM loads Test class and calculates object memory.
4️⃣ Heap Manager creates the Test object and assigns hashCode.
5️⃣ Reference variable `t` in Stack points to the object.
```

6️⃣ JVM prints reference value of object.

Understanding `hashCode()` and `toString()` in Java

1. `hashCode()` Method

What is `hashCode()`?

The `hashCode()` method returns an integer (a unique number) for each object. It is used when storing objects in **hash-based collections** like `HashMap` or `HashSet`.

Why Do We Need `hashCode()`?

Imagine you have a **box of chocolates**, and each chocolate has a unique ID. If someone asks for a chocolate with ID `123`, instead of checking every chocolate, you directly look for `123`. This is exactly what `hashCode()` does in collections.

Example Without Overriding `hashCode()`

```
java
CopyEdit
class Student {
    int id;
    String name;

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

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

```
        Student s1 = new Student(1, "Alice");
        Student s2 = new Student(1, "Alice");

        System.out.println(s1.hashCode());  // Random hash
        System.out.println(s2.hashCode());  // Different random hash
    }
}
```

Output (Default `hashCode()` Behavior)

```
CopyEdit
12345678
87654321
```

> Even though s1 and s2 have the same id and name, their hashCode() is different because they are different objects.

Example With Overridden `hashCode()`

```
java
CopyEdit
import java.util.Objects;

class Student {
    int id;
    String name;

    Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(id, name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1, "Alice");
        Student s2 = new Student(1, "Alice");

        System.out.println(s1.hashCode());  // Same hash for s1 and s2
        System.out.println(s2.hashCode());  // Same hash for s1 and s2
    }
}
```

Output (After Overriding `hashCode()`)

```
CopyEdit
356573597
356573597
```

> Now, s1 and s2 have the same hashCode() because they have the same id and name.

---

2. `toString()` Method

What is `toString()`?

The `toString()` method returns a **text representation of an object**.

Why Do We Need `toString()`?

By default, when you print an object, Java prints something like:

```
css
CopyEdit
Student@1a2b3c4
```

This is not meaningful. If we override `toString()`, we can print meaningful information.

Example Without Overriding `toString()`

```
java
CopyEdit
class Student {
    int id;
    String name;

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1, "Alice");

        System.out.println(s1);  // Default output (not readable)
    }
}
```

Output (Default `toString()` Behavior)

```
css
CopyEdit
Student@1a2b3c4
```

> This is not useful because it prints the class name and hash code.

Example With Overridden `toString()`

```
java
CopyEdit
class Student {
    int id;
    String name;

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

    @Override
    public String toString() {
        return "Student{id=" + id + ", name='" + name + "'}";
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1, "Alice");

        System.out.println(s1);  // Now prints useful information
```

```
    }
}
```

Output (After Overriding `toString()`)

```
bash
CopyEdit
Student{id=1, name='Alice'}
```

> Now, the output is readable and meaningful.

Final Summary

Method	Purpose	Default Behavior	After Overriding
`hashCode()`	Generates an integer for fast lookup in collections	Returns an integer (memory-based)	Returns same hash for equal objects
`toString()`	Returns object information as text	Prints `ClassName@HashCode`	Prints useful object details

---

Key Takeaways

✅ Always override `hashCode()` when working with `HashMap`, `HashSet`, etc.

✅ Always override `toString()` for better debugging.

✅ Use `Objects.hash(id, name)` for easy `hashCode()` implementation.

✅ Use a **clear format** for `toString()` output (like JSON or key-value pairs).

Understanding Constructors and `this` Keyword in Java

1. What is a Constructor in Java?

A **constructor** is a **special method** that is called **automatically** when an object of a class is created.

It is used to **initialize** object properties.

Key Points About Constructors

✔️ The **name** of the constructor **must be the same** as the class name.

✔️ **No return type** (not even `void`).

✔️ **Called automatically** when an object is created.

✔️ Used to set **initial values** for object properties.

Example of a Constructor

```
java
CopyEdit
class Student {
    String name;
    int age;

    // Constructor
    Student() {
        System.out.println("A new student object is created!");
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Constructor is automatically called
```

```
    }
}
```

Output:

```
csharp
CopyEdit
A new student object is created!
```

> 🔷 Explanation: The constructor is automatically executed when we create an object.

---

2. Types of Constructors in Java

There are **three types** of constructors in Java:

1️⃣ **Default Constructor** (No parameters)

2️⃣ **Parameterized Constructor** (With parameters)

3️⃣ **Copy Constructor** (Creates a copy of another object)

1️⃣ Default Constructor (No Parameters)

A constructor **without parameters** is called a **default constructor**.

```
java
CopyEdit
class Student {
    String name;
    int age;

    // Default Constructor
    Student() {
```

```
        name = "Unknown";
        age = 18;
        System.out.println("Default values assigned: " + name + ", " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(); // Calls the default constructor
    }
}
```

Output:

```
sql
CopyEdit
Default values assigned: Unknown, 18
```

> 🔷 Explanation: The default constructor assigns predefined values.

2️⃣ Parameterized Constructor (With Parameters)

A constructor **with parameters** is called a **parameterized constructor**.

```
java
CopyEdit
class Student {
    String name;
    int age;

    // Parameterized Constructor
```

```
    Student(String studentName, int studentAge) {
        name = studentName;
        age = studentAge;
        System.out.println("Student Created: " + name + ", " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20);
        Student s2 = new Student("Bob", 22);
    }
}
```

Output:

```
yaml
CopyEdit
Student Created: Alice, 20
Student Created: Bob, 22
```

> 🔷 Explanation: The constructor initializes the values using parameters.

3️⃣ Copy Constructor (Copies Values from Another Object)

A **copy constructor** is used to **copy values from one object to another**.

```
java
CopyEdit
class Student {
    String name;
```

```
    int age;

    // Parameterized Constructor
    Student(String studentName, int studentAge) {
        name = studentName;
        age = studentAge;
    }

    // Copy Constructor
    Student(Student s) {
        name = s.name;
        age = s.age;
    }

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20);
        Student s2 = new Student(s1);  // Copying s1 into s2

        s1.display();
        s2.display();
    }
}
```


Output:

```
yaml
CopyEdit
Student: Alice, Age: 20
```

```
Student: Alice, Age: 20
```

> 🔷 Explanation: s2 is a copy of s1.

---

3. Understanding `this` Keyword

The `this` keyword refers to the **current object** of the class.

It is mainly used in three cases:

1️⃣ **To refer to the current object's variables** (when variable names are the same).

2️⃣ **To call another constructor** from a constructor (constructor chaining).

3️⃣ **To return the current object** (for method chaining).

1️⃣ `this` to Refer to Instance Variables

If local and instance variables have the same name, `this` helps to distinguish them.

```
java
CopyEdit
class Student {
    String name;
    int age;

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

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

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 20);
        s1.display();
    }
}
```

Output:

```
yaml
CopyEdit
Name: Alice, Age: 20
```

> 🔷 Explanation: this.name refers to the instance variable, while name refers to the constructor parameter.

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

We can call one constructor from another using `this()`.

```
java
CopyEdit
class Student {
    String name;
    int age;

    // Constructor 1 (default)
    Student() {
        this("Unknown", 18);  // Calls Constructor 2
    }
```

```
    // Constructor 2 (parameterized)
    Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();  // Calls default constructor, which calls para
meterized constructor
        s1.display();
    }
}
```

Output:

```
yaml
CopyEdit
Student: Unknown, Age: 18
```

> 🔷 Explanation: this("Unknown", 18) calls the second constructor.

3️⃣ `this` to Return Current Object (Method Chaining)

The `this` keyword can **return the current object** to enable **method chaining**.

```
java
CopyEdit
class Student {
    String name;

    Student setName(String name) {
        this.name = name;
        return this;  // Returns the current object
    }

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

public class Main {
    public static void main(String[] args) {
        new Student().setName("Alice").display(); // Method Chaining
    }
}
```

Output:

```
yaml
CopyEdit
Student Name: Alice
```

> 🔷 Explanation: setName() returns this, allowing us to call display() in a single line.

Final Summary

Concept	Description	Example
Constructor	Initializes an object when it is created	`Student s = new Student();`
Default Constructor	No parameters, assigns default values	`Student() { name = "Unknown"; }`
Parameterized Constructor	Takes arguments to initialize values	`Student(String n) { name = n; }`
Copy Constructor	Copies values from another object	`Student(Student s) { name = s.name; }`
`this` **Keyword**	Refers to the current object	`this.name = name;`
`this()` **Constructor Chaining**	Calls another constructor	`this("Unknown", 18);`
`this` **for Method Chaining**	Returns the current object	`return this;`

1. `extends` Keyword in Java

Example:

```
java
CopyEdit
// Parent class
class Animal {
    void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Child class inheriting from Animal
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.makeSound(); // Calls parent class method
        d.bark();      // Calls child class method
    }
}
```

Expected Output:

```
css
CopyEdit
Animal makes a sound
Dog barks
```

Explanation:

- `Dog` class extends `Animal`, so it inherits `makeSound()`.
- The `bark()` method is defined in the `Dog` class.
- The `Main` class creates an object of `Dog` and calls both methods.

---

2. Method Overriding in Java

Example:

```
java
CopyEdit
```

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

// Child class overrides run() method
class Car extends Vehicle {
    @Override
    void run() {
        System.out.println("Car is running safely");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.run(); // Calls overridden method from Car class
    }
}
```


Expected Output:

```
arduino
CopyEdit
Car is running safely
```

Explanation:

- The `Car` class overrides the `run()` method from `Vehicle`.
- The overridden method in `Car` is executed instead of the parent class method.

- **Polymorphism:** Even though `Car` inherits `Vehicle`, its own method is executed.

3. Method Overloading in Java

Example:

```
java
CopyEdit
class Calculator {
    // Overloaded method with 2 int parameters
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method with 2 double parameters
    double add(double a, double b) {
        return a + b;
    }

    // Overloaded method with 3 int parameters
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println(calc.add(5, 10));       // Calls add(int, int)
        System.out.println(calc.add(5.5, 2.5));    // Calls add(double, double)
        System.out.println(calc.add(1, 2, 3));     // Calls add(int, int, int)
    }
```

```
}
```

Expected Output:

```
CopyEdit
15
8.0
6
```

Explanation:

- The `add()` method is **overloaded** with different parameter lists.
- The correct method is selected at compile time based on the arguments provided.

Bonus: Overriding vs. Overloading in One Program

```
java
CopyEdit
class Parent {
    void show() {  // Overridden method
        System.out.println("Parent class show() method");
    }
}

class Child extends Parent {
    @Override
    void show() { // Overriding Parent's show()
        System.out.println("Child class show() method");
    }
```

```
    // Overloaded method
    void show(String msg) {
        System.out.println("Overloaded show() method: " + msg);
    }
}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.show();        // Calls overridden method in Child
        obj.show("Hello");  // Calls overloaded method
    }
}
```

Expected Output:

```
pgsql
CopyEdit
Child class show() method
Overloaded show() method: Hello
```

Explanation:

- `show()` is **overridden** in `Child`, so it replaces the method in `Parent`.
- `show(String msg)` is **overloaded**, meaning it coexists with `show()`.

Final Summary

Feature	Method Overriding	Method Overloading
Definition	Redefining a parent class method in a subclass	Same method name but different parameters in the same class

Polymorphism Type	Runtime polymorphism	Compile-time polymorphism
Method Signature	Must be same	Must be different (parameters change)
Return Type	Must be same or covariant	Can be different
Scope	Happens in child class	Happens inside the same class
Example	`void run()` in both parent & child	`void add(int, int)` and `void add(double, double)`

Understanding Static Context and Instance Context in Java

In Java, the **static context** and **instance context** define how variables and methods are accessed and executed. These two concepts are fundamental for understanding **memory allocation, method execution, and object-oriented behavior** in Java.

---

1. Static Context in Java

What is Static Context?

- The **static context** refers to class-level variables and methods that **belong to the class itself rather than any instance (object)**.
- Static members are **shared across all instances** of a class.
- **No object creation is needed** to access static members.
- Static methods **cannot access instance variables or methods directly** because they belong to the class, not an instance.

Key Points About Static Context:

✔ **Memory Allocation:** Static variables and methods are stored in the **Method Area (Class Area)** of JVM memory.

✔ **Accessing:** Static members can be accessed using the **class name** (e.g., `ClassName.methodName()`).

✔ **No Object Needed:** They can be used without creating an object of the class.

✔ **Cannot Use `this` or `super`:** Since `this` refers to the current object and `super` refers to the parent class object, they cannot be used in a static context.

✔ **Can Access Only Static Members Directly:** Static methods can **only access static variables and other static methods** directly.

Syntax Example:

```
java
CopyEdit
class Example {
    static int count = 0; // Static variable

    static void display() { // Static method
        System.out.println("Static method called. Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Example.display(); // Calling static method without object
    }
}
```


Expected Output:

```
sql
CopyEdit
Static method called. Count: 0
```

Why Can't Static Methods Access Non-Static Variables?

```
java
CopyEdit
class Test {
    int num = 10;  // Instance variable

    static void printNum() {
        // System.out.println(num); // ❌ Compilation Error: Cannot access instan
ce variable
    }
}
```

- Since `num` is an **instance variable**, it exists **only when an object is created**.
- But `printNum()` is static, and it exists **before any object is created**.
- That's why static methods **cannot directly access** non-static (instance) variables.

---

2. Instance Context in Java

What is Instance Context?

- The **instance context** refers to variables and methods that **belong to individual objects** of a class.
- Each object of the class has its **own copy** of instance variables.
- **Instance methods can access both instance and static members.**

Key Points About Instance Context:

✔ **Memory Allocation:** Instance variables and methods are stored in the **Heap Memory** of JVM.

✔ **Accessing:** Instance members require an **object** to be accessed (`objectName.methodName()`).

✔ **Each Object Has Its Own Copy:** Unlike static members, instance variables **are unique to each object**.

✔ **Instance Methods Can Access Both Static and Non-Static Members:** They can call **both instance and static variables/methods**.

Syntax Example:

```
java
CopyEdit
class Example {
    int count = 0; // Instance variable

    void increment() { // Instance method
        count++;
        System.out.println("Instance method called. Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Example obj1 = new Example();
        Example obj2 = new Example();

        obj1.increment(); // Count = 1
        obj2.increment(); // Count = 1 (separate instance)
    }
}
```

Expected Output:

```
pgsql
CopyEdit
Instance method called. Count: 1
```

```
Instance method called. Count: 1
```

Explanation:

- `count` is an **instance variable**, so each object (`obj1`, `obj2`) has **its own copy**.
- When `increment()` is called, it modifies the respective object's `count`.

---

3. Static Context vs. Instance Context in Java

Feature	Static Context	Instance Context
Belongs To	Class (shared by all objects)	Object (each object has its own copy)
Memory Location	**Method Area (Class Area)**	**Heap Memory**
Access	Can be accessed without an object (`ClassName.method()`)	Requires object (`object.method()`)
Can Access	**Only static members**	Both static and non-static members
Use of `this` and `super`	❌ Not allowed	✅ Allowed
Common Use Cases	Utility methods (e.g., `Math.pow()`, `Collections.sort()`)	Instance-specific behavior

---

4. Example Comparing Static and Instance Context

```
java
CopyEdit
class Demo {
    static int staticVar = 10; // Static variable
    int instanceVar = 20; // Instance variable

    static void staticMethod() {
        System.out.println("Static method called. StaticVar: " + staticVar);
        // System.out.println(instanceVar); // ❌ Error: Cannot access instance va
```

```
riable
    }

    void instanceMethod() {
        System.out.println("Instance method called. StaticVar: " + staticVar + ", In
stanceVar: " + instanceVar);
    }
}

public class Main {
    public static void main(String[] args) {
        // Calling static method using class name
        Demo.staticMethod();

        // Creating an instance and calling instance method
        Demo obj = new Demo();
        obj.instanceMethod();
    }
}
```


Expected Output:

```
pgsql
CopyEdit
Static method called. StaticVar: 10
Instance method called. StaticVar: 10, InstanceVar: 20
```

Explanation:

1. `staticMethod()` **is called without an object**, and it can only access the `staticVar`.
2. `instanceMethod()` **is called using an object (** `obj` **)**, so it can access **both static and instance variables**.

---

5. Real-World Example of Static and Instance Context

Bank Account Example

```
java
CopyEdit
class BankAccount {
    static double interestRate = 5.0; // Static variable shared by all accounts
    double balance; // Instance variable unique to each account

    BankAccount(double balance) {
        this.balance = balance;
    }

    void showAccountDetails() {
        System.out.println("Balance: $" + balance + ", Interest Rate: " + interestR
ate + "%");
    }

    static void changeInterestRate(double newRate) {
        interestRate = newRate;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount acc1 = new BankAccount(1000);
        BankAccount acc2 = new BankAccount(2000);

        acc1.showAccountDetails();
        acc2.showAccountDetails();

        // Changing interest rate (affects all accounts)
        BankAccount.changeInterestRate(6.5);
```

```
        acc1.showAccountDetails();
        acc2.showAccountDetails();
    }
}
```

Expected Output:

```
yaml
CopyEdit
Balance: $1000.0, Interest Rate: 5.0%
Balance: $2000.0, Interest Rate: 5.0%
Balance: $1000.0, Interest Rate: 6.5%
Balance: $2000.0, Interest Rate: 6.5%
```

Explanation:

- `interestRate` is **static**, so **all accounts share the same rate**.
- `balance` is an **instance variable**, so each account has **its own balance**.

Conclusion

- **Static Context:** Belongs to the class, shared among all objects, does not require an instance.
- **Instance Context:** Belongs to an object, each instance has its own copy, requires object creation.
- **Best Practices:**
  ✔️ Use static for
  utility methods, constants, and shared data.
  ✔️ Use instance variables for
  object-specific data.

✔️ Avoid excessive use of static variables to maintain **object-oriented principles**.

Inheritance in Java

Definition:

Inheritance is a mechanism in Java where one class acquires the properties and behaviors (methods) of another class. It allows **code reusability** and establishes a **parent-child relationship** between classes.

Key Points:

1. The **child (subclass)** inherits fields and methods from the **parent (superclass)**.
2. The keyword `extends` is used to inherit a class.
3. Inheritance follows the **IS-A relationship** (i.e., a subclass **is-a** type of superclass).
4. Helps in **code reusability and maintainability**.

Types of Inheritance in Java:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance***(Java does not support multiple inheritance with classes, but it can be achieved using interfaces.)*

---

1. Single Inheritance

One class inherits from another class.

Example:

```
java
CopyEdit
// Parent class
class Animal {
    void eat() {
        System.out.println("Animals can eat.");
    }
}

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

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


Output:

```
nginx
CopyEdit
Animals can eat.
Dogs can bark.
```

2. Multilevel Inheritance

A class is derived from another derived class.

Example:

```
java
CopyEdit
// Parent class
class Animal {
    void eat() {
        System.out.println("Animals eat food.");
    }
}

// Child class
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammals can walk.");
    }
}

// Grandchild class
class Human extends Mammal {
    void speak() {
        System.out.println("Humans can speak.");
    }
}

// Main class
public class MultilevelInheritance {
    public static void main(String[] args) {
        Human h = new Human();
        h.eat();   // Inherited from Animal
        h.walk();  // Inherited from Mammal
        h.speak(); // Own method
```

```
    }
}
```

Output:

```
css
CopyEdit
Animals eat food.
Mammals can walk.
Humans can speak.
```

---

3. Hierarchical Inheritance

A single parent class has multiple child classes.

Example:

```
java
CopyEdit
// Parent class
class Vehicle {
    void run() {
        System.out.println("Vehicles can run.");
    }
}

// Child class 1
class Car extends Vehicle {
    void fourWheels() {
        System.out.println("Cars have four wheels.");
    }
}
```

```
// Child class 2
class Bike extends Vehicle {
    void twoWheels() {
        System.out.println("Bikes have two wheels.");
    }
}

// Main class
public class HierarchicalInheritance {
    public static void main(String[] args) {
        Car c = new Car();
        c.run();
        c.fourWheels();

        Bike b = new Bike();
        b.run();
        b.twoWheels();
    }
}
```

Output:

```
arduino
CopyEdit
Vehicles can run.
Cars have four wheels.
Vehicles can run.
Bikes have two wheels.
```

`super` Keyword in Java

The `super` keyword is used to refer to the **parent class**. It is used for:

1. **Accessing the parent class constructor**
2. **Calling the parent class method**
3. **Accessing the parent class variable**

---

1. Using `super` to Call Parent Class Method

When a method in the child class has the same name as the method in the parent class, `super` helps to call the parent's version.

Example:

```
java
CopyEdit
class Parent {
    void display() {
        System.out.println("This is the Parent class.");
    }
}

class Child extends Parent {
    void display() {
        super.display();  // Calls Parent's display() method
        System.out.println("This is the Child class.");
    }
}

public class SuperMethodExample {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

Output:

```
csharp
CopyEdit
This is the Parent class.
This is the Child class.
```

---

2. Using `super` to Access Parent Class Variables

If both child and parent class have a variable with the same name, `super` helps to differentiate them.

Example:

```
java
CopyEdit
class Parent {
    String name = "Parent Class";
}

class Child extends Parent {
    String name = "Child Class";

    void display() {
        System.out.println("Child name: " + name);
        System.out.println("Parent name: " + super.name); // Access parent's vari
able
    }
}

public class SuperVariableExample {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
```

```
    }
}
```

Output:

```
pgsql
CopyEdit
Child name: Child Class
Parent name: Parent Class
```

---

3. Using `super()` to Call Parent Class Constructor

The `super()` is used to call the **parent class constructor** from the child class.

Example:

```
java
CopyEdit
class Parent {
    Parent() {
        System.out.println("Parent class constructor called.");
    }
}

class Child extends Parent {
    Child() {
        super(); // Calls Parent class constructor
        System.out.println("Child class constructor called.");
    }
}

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

```
        Child c = new Child();
    }
}
```

Output:

```
kotlin
CopyEdit
Parent class constructor called.
Child class constructor called.
```

---

1. Polymorphism in Java

Definition:

Polymorphism is one of the four pillars of Object-Oriented Programming (OOP) that allows a single interface to represent different underlying data types. It means **"one name, many forms."**

Types of Polymorphism:

1. **Compile-time Polymorphism (Method Overloading)**
2. **Run-time Polymorphism (Method Overriding)**

1.1 Compile-time Polymorphism (Method Overloading)

- Achieved using **method overloading**.
- Multiple methods in the same class have the **same name but different parameters** (number, type, or sequence).
- The compiler determines which method to execute based on the method signature.

Example: Method Overloading

```
java
CopyEdit
class Calculator {
    // Method with two parameters
    int add(int a, int b) {
        return a + b;
    }

    // Method with three parameters (different number of arguments)
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with double type parameters (different data type)
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));      // Calls add(int, int)
        System.out.println(calc.add(5, 10, 15));  // Calls add(int, int, int)
        System.out.println(calc.add(5.5, 2.5));   // Calls add(double, double)
    }
}
```


✅ **Key Points:**

✔️ Achieved by changing method parameters (not return type).

✔️ Performed at **compile time** (method is resolved before execution).

1.2 Run-time Polymorphism (Method Overriding)

- Achieved using **method overriding**.
- A subclass provides a **specific implementation** of a method defined in its superclass.
- It is determined at **runtime**.

Example: Method Overriding

```
java
CopyEdit
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 myAnimal = new Dog(); // Upcasting
        myAnimal.sound(); // Calls Dog's sound() method
    }
}
```

Output:

```
nginx
CopyEdit
```

```
Dog barks
```

✅ **Key Points:**

✔ Achieved using **inheritance** (subclass redefines superclass method).

✔ Uses `@Override` annotation for better readability.

✔ **Method resolution happens at runtime** (Dynamic Method Dispatch).

---

2. `final` Keyword in Java

The `final` keyword in Java is used to **restrict modifications**. It can be applied to:

1. **Variables** – Prevents reassignment.
2. **Methods** – Prevents overriding.
3. **Classes** – Prevents inheritance.

2.1 `final` Variable

A `final` variable **cannot be changed** after initialization.

```
java
CopyEdit
class Test {
    final int MAX_VALUE = 100; // Constant variable

    void display() {
        // MAX_VALUE = 200;  // ❌ Error: Cannot change final variable
        System.out.println("Max Value: " + MAX_VALUE);
    }
}

public class Main {
    public static void main(String[] args) {
        Test obj = new Test();
        obj.display();
```

```
    }
}
```

✅ **Key Points:**

✔ `final` variables must be initialized at declaration or in the constructor.

✔ Cannot be reassigned a new value.

2.2 `final` Method

A method declared as `final` **cannot be overridden** in a subclass.

```
java
CopyEdit
class Parent {
    final void show() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // void show() { ❌ Error: Cannot override final method
    //     System.out.println("Trying to override");
    // }
}

public class Main {
    public static void main(String[] args) {
        Parent obj = new Parent();
        obj.show();
    }
}
```

✅ **Key Points:**

✔️ Prevents method modification in subclasses.

✔️ Useful for maintaining security and design constraints.

2.3 `final` Class

A `final` class **cannot be inherited**.

```
java
CopyEdit
final class Vehicle {
    void show() {
        System.out.println("This is a final class.");
    }
}

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

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


✅ **Key Points:**

✔️ Used to prevent inheritance of a class.

✔️ Often used in utility classes like `java.lang.String`.

Wrapper Classes in Java

1. Introduction

In Java, **wrapper classes** are used to convert primitive data types into objects. They belong to the `java.lang` package and provide a way to treat primitive types as objects.

For example, the primitive type `int` can be wrapped inside an `Integer` object.

2. Why Use Wrapper Classes?

- Java collections (like `ArrayList`) do not support primitive types; they only store objects.
- Provide utility methods for primitive types (e.g., `Integer.parseInt()` to convert a String to an int).
- Enable **autoboxing and unboxing** (automatic conversion between primitives and wrapper objects).
- Useful in synchronization and multithreading.

3. List of Wrapper Classes in Java

Each primitive data type has a corresponding wrapper class:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

4. Example: Using Wrapper Classes

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

```
        // Creating wrapper objects
        Integer intObj = Integer.valueOf(100);
        Double doubleObj = Double.valueOf(10.5);
        Boolean boolObj = Boolean.valueOf(true);

        // Converting wrapper objects to primitive types
        int intValue = intObj.intValue();
        double doubleValue = doubleObj.doubleValue();
        boolean boolValue = boolObj.booleanValue();

        // Displaying values
        System.out.println("Integer value: " + intValue);
        System.out.println("Double value: " + doubleValue);
        System.out.println("Boolean value: " + boolValue);
    }
}
```

Output:

```
Integer value: 100
Double value: 10.5
Boolean value: true
```

5. Autoboxing and Unboxing

Autoboxing: Automatic conversion of primitive types into their corresponding wrapper objects.

```
public class AutoBoxingExample {
    public static void main(String[] args) {
        int num = 50;
        Integer obj = num;  // Autoboxing
        System.out.println("Wrapper Object: " + obj);
```

```
    }
}
```

Output:

```
Wrapper Object: 50
```

Unboxing: Automatic conversion of wrapper objects back into primitive types.

```
public class UnboxingExample {
    public static void main(String[] args) {
        Integer obj = Integer.valueOf(30);
        int num = obj;  // Unboxing
        System.out.println("Primitive value: " + num);
    }
}
```

Output:

```
Primitive value: 30
```


6. Important Methods in Wrapper Classes

Method	Description
`parseInt(String s)`	Converts a String to an int (`Integer.parseInt("123") → 123`).
`valueOf(String s)`	Converts a String to a wrapper object (`Integer.valueOf("123") → Integer object`).
`xxxValue()`	Converts a wrapper object to its primitive form (`doubleValue()` , `intValue()` , etc.).
`toString()`	Converts wrapper objects to String (`Integer.toString(10) → "10"`).
`compareTo()`	Compares two wrapper objects (`x.compareTo(y)`).

`equals(Object obj)`	Checks if two wrapper objects are equal.

Example: Using Wrapper Class Methods

```
public class WrapperMethodsExample {
    public static void main(String[] args) {
        // Convert String to int
        int num1 = Integer.parseInt("123");
        System.out.println("Parsed Integer: " + num1);

        // Convert int to String
        String str = Integer.toString(456);
        System.out.println("String representation: " + str);

        // Compare two Integer objects
        Integer a = 10;
        Integer b = 20;
        System.out.println("Comparison result: " + a.compareTo(b));

        // Check equality
        Integer c = 50;
        Integer d = 50;
        System.out.println("Are c and d equal? " + c.equals(d));
    }
}
```


Output:

```
Parsed Integer: 123
String representation: 456
Comparison result: -1
Are c and d equal? true
```

7. Advantages of Wrapper Classes

✔ Used in collections (like `ArrayList<Integer>`).

✔ Provide utility methods for conversion and comparison.

✔ Enable autoboxing and unboxing for easy coding.

✔ Allow `null` values (useful in databases where fields may be empty).

8. Disadvantages of Wrapper Classes

✘ **More memory usage**: Objects consume more memory than primitives.

✘ **Slower performance**: Extra processing is required compared to primitive types.

9. When to Use Wrapper Classes?

- When working with **collections (ArrayList, HashMap, etc.)**.
- When **storing null values** in a variable.
- When using **utility functions** like `Integer.parseInt()` or `Double.toString()`.
- When performing **object-oriented programming** with Java's APIs.

10. Summary

- Wrapper classes convert **primitive data types** into **objects**.
- Java provides **8 wrapper classes** (`Integer`, `Double`, `Boolean`, etc.).
- **Autoboxing** and **Unboxing** allow easy conversion between primitives and wrapper objects.
- **Useful in collections, utility functions, and database handling.**

Encapsulation, Abstract Keyword, Interface, Getters, and Setters

1. Encapsulation

Definition

Encapsulation is one of the core principles of Object-Oriented Programming (OOP). It refers to the bundling of data (variables) and methods (functions) into a single unit (class) and restricting direct access to some of the object's details.

Key Features of Encapsulation

- **Data Hiding:** Variables of a class are made `private` to prevent direct access.
- **Access Control:** We provide `public` getter and setter methods to access and update private variables.
- **Security & Validation:** We can add validation logic inside setter methods to control data modification.
- **Flexibility:** The internal implementation of a class can be changed without affecting external code.

Example of Encapsulation

```
class Student {
    private String name;  // Private variable (cannot be accessed directly)

    // Setter method to assign a value
    public void setName(String newName) {
        name = newName;
    }

    // Getter method to retrieve the value
    public String getName() {
        return name;
    }
}

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

```
        Student s = new Student();   // Create an object
        s.setName("John");          // Using setter
        System.out.println(s.getName());  // Using getter
    }
}
```

Output:

```
John
```

---

2. Getters and Setters in Java

Why Use Getters and Setters?

- **Encapsulation:** Protects private variables from direct modification.
- **Validation:** Allows control over what values are assigned.
- **Read-Only / Write-Only Access:** Provides control over data access.

Example with Validation

```
class Person {
    private int age;

    // Setter with validation
    public void setAge(int newAge) {
        if (newAge > 0) {  // Ensuring age is positive
            age = newAge;
        } else {
            System.out.println("Age cannot be negative!");
        }
    }

    // Getter method
    public int getAge() {
```

```
        return age;
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setAge(25);   // Setting age
        System.out.println("Age: " + p.getAge());

        p.setAge(-5);   // Invalid age
    }
}
```

Output:

```
Age: 25
Age cannot be negative!
```

Read-Only Property (Only Getter, No Setter)

```
class Student {
    private final int id = 101;

    // Only Getter
    public int getId() {
        return id;
    }
}
```

Write-Only Property (Only Setter, No Getter)

```
class Account {
    private String password;
```

```
    // Only Setter
    public void setPassword(String newPassword) {
        if (newPassword.length() >= 6) {
            password = newPassword;
        } else {
            System.out.println("Password must be at least 6 characters long!");
        }
    }
}
```

Abstract Keyword & Interface in Java

1. Abstract Keyword in Java

Definition:

The `abstract` keyword in Java is used to define an **abstract class** or **abstract method**. An abstract class cannot be instantiated, and it serves as a blueprint for other classes. An abstract method does not have a body and must be implemented in a subclass.

1.1 Abstract Class

- A class declared using the `abstract` keyword is called an **abstract class**.
- It **may or may not contain abstract methods**.
- It **can have constructors, static methods, and instance variables**.
- It **cannot be instantiated directly**.
- It can contain both **abstract and concrete methods**.

Syntax of an Abstract Class:

```
abstract class Vehicle {
    String brand;
```

```
    // Constructor
    Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method (must be implemented by subclasses)
    abstract void start();

    // Concrete method (can be used by subclasses)
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

1.2 Abstract Method

- An abstract method is declared **without a body**.
- It is declared using the `abstract` keyword inside an abstract class.
- Subclasses **must** override the abstract method, or they also become abstract.

Implementing an Abstract Class:

```
class Car extends Vehicle {
    // Constructor
    Car(String brand) {
        super(brand);
    }

    // Implementing the abstract method
    @Override
    void start() {
        System.out.println("Car is starting with a key...");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota");
        myCar.start();         // Calls the overridden method
        myCar.displayBrand();  // Calls the concrete method
    }
}
```

Output:

```
Car is starting with a key...
Brand: Toyota
```

Key Points About Abstract Classes:

Feature	Abstract Class
Object creation	Cannot be instantiated directly
Abstract methods	Can have abstract and concrete methods
Constructors	Can have constructors
Variables	Can have instance and static variables
Inheritance	Used as a base class for inheritance
Multiple Inheritance	Not supported in Java

2. Interface in Java

Definition:

An `interface` in Java is a blueprint of a class that contains **only abstract methods** (before Java 8). It is used for achieving **full abstraction** and **multiple inheritance** in Java.

2.1 Characteristics of an Interface

- An interface **only contains abstract methods** (before Java 8).
- In Java 8+, interfaces **can have default and static methods**.
- It cannot have constructors.
- All variables in an interface are **implicitly public, static, and final**.
- A class implements an interface using the `implements` keyword.
- It supports **multiple inheritance**.

Syntax of an Interface:

```
interface Animal {
    // Abstract method (implicitly public and abstract)
    void makeSound();
}
```

2.2 Implementing an Interface

Example:

```
interface Animal {
    void makeSound();  // Abstract method
}

class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();  // Calls the implemented method
```

```
    }
}
```

Output:

```
Dog barks: Woof woof!
```

2.3 Default & Static Methods in Interface (Java 8+)

```
interface Vehicle {
    void start();

    // Default method (can be overridden)
    default void stop() {
        System.out.println("Vehicle is stopping...");
    }

    // Static method (cannot be overridden)
    static void show() {
        System.out.println("Static method in interface.");
    }
}

class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike is starting...");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike myBike = new Bike();
        myBike.start();  // Calls implemented method
        myBike.stop();   // Calls default method
```

```
        Vehicle.show();  // Calls static method
    }
}
```

Output:

```
Bike is starting...
Vehicle is stopping...
Static method in interface.
```

Key Points About Interfaces:

Feature	Interface
Object creation	Cannot be instantiated directly
Methods	Only abstract (before Java 8), default & static methods allowed (Java 8+)
Variables	Implicitly public, static, and final
Multiple Inheritance	Supports multiple inheritance
Constructors	Cannot have constructors
Inheritance	Implemented using the `implements` keyword

3. Differences Between Abstract Class and Interface

Feature	Abstract Class	Interface
Object Creation	Cannot be instantiated	Cannot be instantiated
Methods	Can have both abstract and concrete methods	Only abstract methods (before Java 8), can have default & static methods (Java 8+)
Variables	Can have instance and static variables	Only public, static, and final variables
Constructors	Can have constructors	Cannot have constructors

Multiple Inheritance	Does not support multiple inheritance	Supports multiple inheritance
Implementation	Extended using `extends`	Implemented using `implements`
Use Case	Used when classes share a common behavior	Used when multiple unrelated classes need a common functionality

4. When to Use Abstract Class vs Interface?

- **Use abstract class when:**
  - You need to share **code** (concrete methods) among related classes.
  - You want to provide **constructors** and **non-final** fields.
  - The classes share a common **base functionality**.
- **Use interface when:**
  - You need **multiple inheritance**.
  - You want to enforce **contract-based** design (all implementing classes must provide specific behavior).
  - You want to achieve **full abstraction**.

Conclusion

- **Abstract classes** are best for **hierarchical relationships** (e.g., Vehicle → Car, Bike).
- **Interfaces** are best for defining **capabilities** that multiple unrelated classes can implement (e.g., Flyable, Swimmable).
- Use **abstract classes** when you need to share code, and use **interfaces** when you need multiple inheritance and full abstraction.

Abstraction and Interface in Java

1. Abstraction in Java

Definition:

Abstraction is a fundamental concept in Object-Oriented Programming (OOP) that focuses on **hiding implementation details** and only exposing essential features to the user. It allows developers to design systems where the internal workings remain hidden while ensuring usability.

Key Features of Abstraction:

- Hides the implementation details from the user.
- Focuses on what an object does rather than how it does it.
- Achieved using **abstract classes** and **interfaces** in Java.
- Improves **code reusability** and **maintainability**.

---

2. Abstract Classes in Java

Definition:

An abstract class in Java is a class that **cannot be instantiated** and serves as a blueprint for other classes. It can contain both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation).

Characteristics of Abstract Classes:

- Can have both **abstract** and **non-abstract** methods.
- Can have **constructors, fields, and static methods**.
- Cannot be instantiated directly; it must be **inherited**.
- Allows code reuse through **concrete methods**.
- Used when multiple classes share a common structure but require specific implementations.

Syntax of an Abstract Class:

```
abstract class Vehicle {
    String brand;
```

```
    Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method (must be implemented by subclasses)
    abstract void start();

    // Concrete method
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

Implementing an Abstract Class:

```
class Car extends Vehicle {
    Car(String brand) {
        super(brand);
    }

    @Override
    void start() {
        System.out.println("Car is starting with a key...");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota");
        myCar.start();          // Calls the overridden method
        myCar.displayBrand();   // Calls the concrete method
    }
}
```

Output:

```
Car is starting with a key...
Brand: Toyota
```

When to Use Abstract Classes:

- When multiple classes share a common structure.
- When you need to provide partial implementation.
- When you want to define fields and non-abstract methods.

---

3. Interface in Java

Definition:

An **interface** in Java is a blueprint that **only contains abstract methods** (before Java 8). It provides full abstraction and allows multiple inheritance.

Characteristics of Interfaces:

- All methods are **implicitly public and abstract** (before Java 8).
- **Cannot have constructors**.
- **Cannot have instance variables** (only public, static, and final variables).
- **Supports multiple inheritance**.
- Provides full abstraction.

Syntax of an Interface:

```
interface Animal {
    void makeSound(); // Abstract method
}
```

Implementing an Interface:

```
class Dog implements Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks: Woof woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeSound();  // Calls the implemented method
    }
}
```

Output:

```
Dog barks: Woof woof!
```

Default & Static Methods in Interface (Java 8+)

```
interface Vehicle {
    void start();

    // Default method
    default void stop() {
        System.out.println("Vehicle is stopping...");
    }

    // Static method
    static void show() {
        System.out.println("Static method in interface.");
    }
}
```

```
class Bike implements Vehicle {
    @Override
    public void start() {
        System.out.println("Bike is starting...");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike myBike = new Bike();
        myBike.start();  // Calls implemented method
        myBike.stop();   // Calls default method
        Vehicle.show();  // Calls static method
    }
}
```

Output:

```
Bike is starting...
Vehicle is stopping...
Static method in interface.
```

When to Use Interfaces:

- When multiple unrelated classes need a common behavior.
- When you need to achieve full abstraction.
- When you want to support multiple inheritance.

---

4. Differences Between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Object Creation	Cannot be instantiated	Cannot be instantiated

Methods	Can have abstract and concrete methods	Only abstract methods (before Java 8), default & static methods allowed (Java 8+)
Variables	Can have instance and static variables	Only public, static, and final variables
Constructors	Can have constructors	Cannot have constructors
Multiple Inheritance	Does not support multiple inheritance	Supports multiple inheritance
Inheritance	Extended using `extends`	Implemented using `implements`
Use Case	Used when classes share a common behavior	Used when multiple unrelated classes need a common functionality

5. Advanced Concepts

Multiple Interfaces in a Single Class

A class can implement multiple interfaces, allowing it to inherit behaviors from different sources.

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Bird implements Flyable, Swimmable {
    @Override
    public void fly() {
        System.out.println("Bird is flying...");
    }

    @Override
    public void swim() {
```

```
        System.out.println("Bird is swimming...");
    }
}

public class Main {
    public static void main(String[] args) {
        Bird myBird = new Bird();
        myBird.fly();
        myBird.swim();
    }
}
```

Output:

```
Bird is flying...
Bird is swimming...
```

Interface Inheritance

Interfaces can extend other interfaces.

```
interface Animal {
    void eat();
}

interface Mammal extends Animal {
    void walk();
}

class Human implements Mammal {
    @Override
    public void eat() {
        System.out.println("Human is eating...");
    }
```

```
    @Override
    public void walk() {
        System.out.println("Human is walking...");
    }
}
```

6. Conclusion

- **Abstraction** hides the implementation details and provides only the essential functionality.
- **Abstract classes** allow partial abstraction and code reuse.
- **Interfaces** provide full abstraction and support multiple inheritance.
- Use **abstract classes** when you need a common structure and **interfaces** when you need to enforce a contract across unrelated classes.