DAY 4

Methods:

What is a Method?

- A method is a block of code in Java that performs a specific task.
- It helps to reuse code and makes the program organized and readable.
- In other programming languages, it is called a function.

Syntax of a Method:

```
returnType methodName(parameters) {
    // code to execute
}
```

Types of Methods in Java:

Instance Method

- Belongs to an object.
- You must create an object of the class to call it.

- **Example:**

```
class Example {
    void show() {
        System.out.println("Instance Method");
    }

    public static void main(String[] args) {
        Example obj = new Example();  // Creating object
        obj.show();  // Calling instance method
    }
}
```

Output:

Instance Method

Static Method

- Belongs to the class, not to any object.
- Called using the class name.
- **Example**:

```
class Demo {
    static void display() {
        System.out.println("Static Method");
    }

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

```
}
```
Output:
Static Method

Abstract Method

- Declared without a body (no code inside).
- Defined in an abstract class.
- Implemented in child classes (subclasses).
- **Example**:

```
abstract class Shape {
    abstract void draw();  // Abstract method
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing Circle");
    }

    public static void main(String[] args) {
        Circle c = new Circle();
        c.draw();  // Calls the implemented method
    }
}
```

Output:
Drawing Circle

Final Method

- A method that cannot be overridden by subclasses.
- Useful to prevent modification of important logic.
- **Example**:

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

class Child extends Parent {
    // void show() {}  // ✕ Error! Cannot override final method
    public static void main(String[] args) {
        Child c = new Child();
        c.show();  // Calls the final method from Parent
    }
}
```

Output:

```
Final Method
```

Synchronized Method

- Used in multithreading.
- Ensures only one thread accesses the method at a time.
- Prevents data inconsistency.
- Example:

```
class Counter {
    int count = 0;
```

```
synchronized void increment() {
    count++;
}

public static void main(String[] args) {
    Counter c = new Counter();

    // Thread 1
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            c.increment();
        }
    });

    // Thread 2
    Thread t2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            c.increment();
        }
    });

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
    } catch (Exception e) {}
```

```
        System.out.println("Final Count: " + c.count);
    }
}
```

Output (example):

Final Count: 2000

Without synchronized, the output might be less than 2000 due to data inconsistency.

In Short:

Method Type	Object Needed?	Can Override?	Purpose
Instance Method	Yes	Yes	Normal behavior with objects
Static Method	No	No	Shared behavior across class
Abstract Method	Yes	Must override	To force subclass to define method
Final Method	Yes	✕ No	Prevent changes in logic
Synchronized	Yes	Yes	Thread safety

Introduction to OOP's Concepts:

1) Class

- A class is a user-defined data type or a blueprint that defines the structure and behavior (data and methods) of objects.
- It acts like a template that allows us to create multiple objects having similar properties and behaviors.
- Think of a class as a drawing of a car. You can make many real cars (objects) from that drawing.
- **Example**:

```
class Car {
    String brand;
    String model;

    void showDetails() {
        System.out.println("Brand: " + brand + ",
Model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        car1.brand = "Toyota";
        car1.model = "Fortuner";
        car1.showDetails();
    }
}
```

Output:

```
Brand: Toyota, Model: Fortuner
```

2) Object

- An object is a real-world entity created from a class. It holds specific values for the class's attributes and can call the methods of the class.
- If a class is a plan, an object is the actual building constructed from that plan.
- **Example**:

```
class Car {
    String brand;
    String model;

    void showDetails() {
        System.out.println("Brand: " + brand + ",
Model: " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        car1.brand = "Toyota";
        car1.model = "Fortuner";

        Car car2 = new Car();
        car2.brand = "Honda";
        car2.model = "Civic";

        car1.showDetails();
```

```
        car2.showDetails();
    }
}
```

Output:

Brand: Toyota, Model: Fortuner

Brand: Honda, Model: Civic

3) Encapsulation

- Encapsulation is the concept of wrapping (or binding) variables (data) and methods (code) together in a single unit called a class.
- It also involves hiding the internal data of an object using the private keyword, allowing access only through public methods (getters and setters).
- It's like locking sensitive data in a box and giving access only through keys (methods).
- **Example**:

```
class Student {
    private String name;
    private int age;

    public Student(String n, int a) {
        name = n;
        age = a;
    }

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

```
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Ayaan", 21);
        s1.display();
        // System.out.println(s1.age); // ✕ Error: 'age'
has private access
    }
}
```

Output:

Name: Ayaan, Age: 21

4) Inheritance

- Inheritance is a mechanism where one class (child or subclass) can reuse or extend the properties and methods of another class (parent or superclass).
- This promotes code reusability and reduces duplication.
- Like a child inheriting traits from parents.
- **Example**:

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

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

public class Main {
    public static void main(String[] args) {
        Dog dog1 = new Dog();
        dog1.sound();  // Inherited from Animal
        dog1.bark();   // Defined in Dog
    }
}
```

Output:

Animals make sound

Dog barks

5) Polymorphism

- Polymorphism means "many forms". It allows the same method name to behave differently based on the object or number/type of arguments.
- There are two types:
  - Compile-time polymorphism → **Method Overloading**
  - Runtime polymorphism → **Method Overriding**

Method Overriding (Runtime Polymorphism)

- A child class provides a specific implementation of a method already defined in its parent class.
- **Example**:

```
class Animal {
    void sound() {
        System.out.println("Animal makes 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 a1 = new Dog();
        Animal a2 = new Cat();
        a1.sound();
        a2.sound();
```

```
    }
}
```

Output:

Dog barks

Cat meows

Method Overloading (Compile-Time Polymorphism)

- Same method name but different parameters (type or number of arguments).
- **Example**:

```
class MathOperations {
    int add(int a, int 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) {
        MathOperations math = new
MathOperations();
        System.out.println(math.add(5, 10));
        System.out.println(math.add(5, 10, 15));
    }
}
```

Output:

15
30

6) Abstraction

- Abstraction means hiding the internal implementation details and showing only the essential features of the object.
- In Java, abstraction is achieved using:
  - Abstract classes
  - Interfaces
- It's like driving a car — you don't need to know how the engine works to drive it.
- **Example (using abstract class):**

```
abstract class Shape {
    abstract int area();  // abstract method
}

class Square extends Shape {
    int side;

    Square(int s) {
        side = s;
    }

    int area() {
        return side * side;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Shape s = new Square(5);
        System.out.println("Area of square: " +
s.area());
    }
}
```

Output:

Area of square: 25

In Short:

Concept	Definition	Java Example
Class	Blueprint/template defining variables and methods	class Car { ... }
Object	Instance of a class containing actual data	Car car1 = new Car();
Encapsulation	Binding variables and methods; hiding private data	private int age; + public methods
Inheritance	Child class inherits from parent class	class Dog extends Animal
Polymorphism	One method behaves differently depending on context	Overriding and overloading examples
Abstraction	Hides complex details, shows only	abstract class Shape

	relevant info to the user	

Types of Classes:

1) **Regular Class (Concrete Class)**
   - Can be used to create objects.
   - Contains complete method definitions.
   - Most commonly used class type.
   - **Example:**

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

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.id = 101;
        s1.name = "Ayaan";
        s1.display();
    }
}
```

Output:

ID: 101, Name: Ayaan

2) Abstract Class

- Declared using the abstract keyword.
- Cannot create objects directly from it.
- Can have:
- Abstract methods (no body)
- Concrete methods (with body)
- Used when you want to force subclasses to implement specific methods.
- **Example**:

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

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

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

public class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
```

```
        d1.sound();  // Calls overridden method
        d1.sleep();  // Calls concrete method from
abstract class
    }
}
```

Output:

```
Dog barks
Sleeping...
```

3) Final Class

- Declared using the final keyword.
- Cannot be extended or inherited.
- Useful when you want to prevent modification of the class.
- **Example**:

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

// ✕ This will cause an error:
// class Car extends Vehicle { }  // Error: Cannot
inherit from final class

public class Main {
    public static void main(String[] args) {
        Vehicle v1 = new Vehicle();
```

```
        v1.run();
    }
}
```

Output:

Vehicle is running

4) Static Class (Nested Static Class)

- Only allowed inside another class (i.e., nested).
- Declared using the static keyword.
- Cannot access non-static members of the outer class directly.
- Useful for utility or helper classes.
- **Example**:

```
class Outer {
    static int data = 100;

    static class Inner {
        void display() {
            System.out.println("Data: " + data);
        }
    }
}

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

}

Output:

Data: 100

In Short:

Class Type	Can Create Object?	Can Be Inherited?	Special Features
Regular Class	✅ Yes	✅ Yes	Fully implemented class
Abstract Class	❌ No	✅ Yes	Has abstract + concrete methods
Final Class	✅ Yes	❌ No	Cannot be extended
Static Class	✅ (nested)	✅ (nested)	Defined inside another class; limited access

Constructors:

What is a Constructor?

A constructor is a special method in Java that is automatically called when an object is created.

Key Points:

- Its name must be same as the class name.
- It doesn't have a return type (not even void).

- It is used to initialize objects (i.e., give values to variables).

Syntax:

```
class ClassName {
    ClassName() {
        // constructor body
    }
}
```

Types of Constructors in Java

1) Default Constructor

- A constructor with no parameters.
- Java automatically creates one if you don’t define any constructor.
- Used to initialize objects with default values.
- **Example**:

```
class Student {
    Student() {
        System.out.println("Default Constructor called");
    }
}

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

```
        Student s1 = new Student();  // constructor is
called automatically
    }
}
```

Output:

Default Constructor called

2) Parameterized Constructor

- A constructor that takes arguments.
- Used to initialize objects with custom values.
- **Example**:

```
class Student {
    String name;

    // Parameterized constructor
    Student(String n) {
        name = n;
    }

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

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

```
    }
}
```

Output:

Name: Ayaan

3) Copy Constructor

- Java doesn't have a built-in copy constructor like C++, but we can create our own.
- It copies values from one object to another.
- **Example**:

```
class Student {
    String name;

    // Parameterized constructor
    Student(String n) {
        name = n;
    }

    // Copy constructor
    Student(Student s) {
        this.name = s.name;
    }

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

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Ayaan");
        Student s2 = new Student(s1);  // Copy
constructor
        s2.display();
    }
}
```

Output:

Name: Ayaan

Instance Block:

What is an Instance Block?

An Instance Block, also called an Instance Initializer Block, is a block of code inside a class that runs automatically every time an object is created, before the constructor is executed.

Key Points:

- It is not a method, but a normal block of code placed directly in the class.
- It runs before every constructor call, no matter which constructor is used.
- Commonly used to put shared initialization logic.

Syntax:

```
class ClassName {
    {
```

```
        // Instance Block
        System.out.println("Instance Block");
    }

    ClassName() {
        System.out.println("Constructor");
    }
}
```

Example:

```
class Demo {
    {
        System.out.println("Instance Block");  // This runs before
the constructor
    }

    Demo() {
        System.out.println("Constructor");
    }
}

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

```
        Demo obj1 = new Demo();
        Demo obj2 = new Demo();
    }
}
```

Output:

Instance Block

Constructor

Instance Block

Constructor

Explanation:

- When obj1 is created, the instance block runs first, then the constructor.
- When obj2 is created, the same thing happens again.
- So for every object, instance block executes before the constructor.

Why Use Instance Block?

- To avoid repeating the same code in every constructor.
- If a class has multiple constructors, but you want to run some common logic before all of them — use an instance block.

Example with Multiple Constructors:

```
class Example {
    {
        System.out.println("Common setup in Instance Block");
    }

    Example() {
        System.out.println("No-arg Constructor");
    }

    Example(int x) {
        System.out.println("Parameterized Constructor: " + x);
    }
}

public class Main {
    public static void main(String[] args) {
        Example e1 = new Example();
        Example e2 = new Example(10);
    }
}
```

Output:

Common setup in Instance Block

No-arg Constructor

Common setup in Instance Block

Parameterized Constructor: 10