SIT311 ADVANCED OBJECT-ORIENTED PROGRAMMING

COURSE OUTLINE

WEEK	TOPIC
1	**Class Design and Core API:** Introduction to Classes and Objects, Records in Java, Instance vs. Static Fields and Methods, Constructors and Constructor Overloading, Instance and Static Initializers, Enumerations, Best Practices for Class Design
2	**Inheritance and Polymorphism:** Understanding Inheritance, Method Overriding, Use of super and this, Polymorphism and Dynamic Binding, Designing Class Hierarchies
3	**Abstract Classes, Interfaces, and Nested Classes:** Abstract Classes and Abstract Methods, Interfaces and Multiple Inheritance via Interfaces, Functional Interfaces and @FunctionalInterface Annotation, Nested and Inner Classes Immutable Classes in Java
4	**Lambdas and Functional Programming:** Introduction to Lambda Expressions, Functional Interfaces (Predicate, Function, Consumer, etc.), Method References and Constructor References, Streams API with Lambdas, Real-world Applications of Functional Programming
5	**Exception Handling and Internationalization:** Exception Hierarchy, try, catch, finally blocks, Multi-catch and Try-with-Resources, Creating Custom Exceptions, Internationalization and Localization, The Locale Class Formatting Numbers, Dates, and Text for Different Regions
	Collections and Generics: Java Arrays and Utility Methods, The Collections Framework (List, Set, Map, Deque), Adding, Removing, Updating, Retrieving, and Sorting Elements, Introduction to Generics, Generic Classes and Methods, Streams API and Parallel Streams
7	**Java Modules and Reflection:** Introduction to the Java Module System, Defining Modules and Dependencies, Reflection API Basics, Defining Services, Producers, and Consumers, Compiling Java Code into Modular and Non-Modular JARs, Runtime Images and Migration with Unnamed/Automatic Modules
8	**Concurrency in Java:** Threads and the Runnable Interface, Thread Lifecycle, The Callable Interface and Future Objects, Executor Services and the Concurrency API, Thread Safety and Synchronization, Parallel Streams and Fork/Join Framework
9	**Input and Output (I/O) in Java:** Console I/O, File I/O with Streams, Object Serialization and Deserialization, The java.nio.file API, Best Practices for Handling I/O
10	**JavaFX and GUI Development:** Introduction to JavaFX, Stages, Scenes, and Nodes, Layouts (Panes, Groups, etc.), UI Controls (Buttons, Labels, TextFields, etc.), Styling with Colors, Fonts, and Images, Event Handling and Handler

	Classes, SceneBuilder for Rapid UI Development, Connecting GUI with Business Logic
11	**Java Database Connectivity (JDBC):** JDBC Architecture, Creating Database Connections, Executing Statements (Statement, PreparedStatement, CallableStatement), Processing Query Results, Managing Transactions, Integrating JDBC with JavaFX Applications

CHAPTER 1: Class Design and Core API

1.1 Introduction

The foundation of Java programming lies in object-oriented programming (OOP). At the heart of OOP are classes and objects. A class defines **the structure and behavior** of objects, while an object is a concrete **instance of a class** that holds data and can perform actions.

Java provides a powerful Core API that defines how classes are designed, created, and extended. Mastering class design ensures that your programs are robust, reusable, maintainable, and efficient.

In this chapter, we explore:

- Classes and records
- Instance and static fields and methods
- Constructors and constructor overloading
- Instance and static initializers
- Enumerations
- Best practices in class design

By the end of this chapter, you should be able to confidently **design classes in Java**, use static and instance members effectively, and leverage **constructors and initializers** to create clean and predictable object behavior.

1.2. Classes in Java

A **class** in Java is like a **blueprint** or **template** for creating objects.
It defines what an object will look like (**fields**) and what it can do (**methods**).

Fields (Attributes / Properties)

- Fields are **variables declared inside a class**.
- They represent the **state** (data) of an object.
- Each object of the class will have **its own copy** of these fields.

Example:

String brand; // Represents the car brand like Toyota, BMW

int speed; // Represents the car's speed in km/h

Methods (Behaviors / Functions)

- Methods are **functions inside a class**.
- They represent **actions or behavior** that objects of the class can perform.
- They often work with the fields to perform tasks.

Example:

```
void drive() {
   System.out.println(brand + " is driving at " + speed + " km/h.");
}
```

Here the drive() method uses the values of the brand and speed fields to describe the car's action.

```
class Car {
   String brand;   // field - car brand
   int speed;     // field - current speed
   // method - simulates the car driving
   void drive() {
      System.out.println(brand + " is driving at " + speed + " km/h.");
   }
}
```

1.3. Objects and Instances

An **object** is an instance of a class. You create objects using the new keyword:
Once a class is defined, you can create **objects** (instances) from it.
Each object has **its own copy of fields** but **shares the same methods**.

Example:

```
public class Main {

   public static void main(String[] args) {

      // Creating first Car object

      Car car1 = new Car();

      car1.brand = "Toyota";

      car1.speed = 120;

      car1.drive();  // Output: Toyota is driving at 120 km/h.

      // Creating second Car object

      Car car2 = new Car();

      car2.brand = "BMW";

      car2.speed = 180;

      car2.drive();  // Output: BMW is driving at 180 km/h.

   }

}
```

Explanation:
Car car1 = new Car(); //Creates a **new object** of type Car called car1.

car1.brand = "Toyota"; // Assigns **Toyota** to car1's brand field.

car1.speed = 120; //Assigns **120** to car1's speed field.
car1.drive(); //Calls the drive() method, which prints: "Toyota is driving at 120 km/h.

1.4. Instance Fields and Methods

Instance Fields

- These are variables declared inside a class but outside any method.
- Each object (instance) of the class has its own copy of these fields.
- Changing the field of one object does not affect the field of another object.

Instance Methods

- These are methods that belong to objects (instances) of a class.
- They can access and modify the object's instance fields.
- You need an object to call them (not the class itself).

Example:

```
class Student {
   // Instance fields (each student has their own name and age)
   String name;
   int age;
   // Instance method (operates on instance fields)
   void introduce() {
      System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
   }
}
public class StudentDemo {
   public static void main(String[] args) {
      // Create first student object
      Student student1 = new Student();
      student1.name = "Alice";
      student1.age = 20;
      // Create second student object
      Student student2 = new Student();
      student2.name = "Bob";
```

```
        student2.age = 22;

        // Call instance methods
        student1.introduce();  // Output: Hi, I'm Alice and I'm 20 years old.
        student2.introduce();  // Output: Hi, I'm Bob and I'm 22 years old.
    }
}
```

When you create multiple students, each will have unique values for name and age.

1.5. Static Fields and Methods

- **Static Fields:** Shared across all objects of a class.
- **Static Methods:** Can be called without creating an object.

Common uses:

- Counters (tracking how many objects are created).
- Constants (public static final).
- Utility methods (Math.sqrt(), Collections.sort()).

Example:

```
class Counter {
    static int count = 0;   // shared variable
    Counter() {
        count++;
    }
}
```

1.6. Constructors

Constructors in Java

A constructor is a special method used to **initialize objects** when they are created.
Key points:

- The constructor name must be **the same as the class name**.
- Constructors **do not have a return type** (not even void).
- They are automatically invoked when you create an object using new.
- Constructors help ensure that objects are created in a **valid and initialized state**.

Types of Constructors

1. **Default Constructor**

Provided automatically by Java **if no constructor is defined**.

Initializes fields with **default values** (e.g., null for objects, 0 for numbers, false for booleans).

Example:

```
1. class Car {
2.     String brand;
3.     int speed;
4.     // Default constructor is provided automatically
5. }
```

```
1. public class Main {
2.     public static void main(String[] args) {
3.         Car c = new Car(); // uses default constructor
4.         System.out.println(c.brand); // null
5.         System.out.println(c.speed); // 0
6.     }
7. }
```

2. **Parameterized Constructor**

Accepts arguments so that fields can be initialized with specific values when creating objects.

Example (using your Book class, expanded):

```
1. class Book {
2.     String title;
```

```
    String author;
    // Parameterized constructor
    Book(String t, String a) {
        title = t;
        author = a;
    }
    void display() {
        System.out.println("Title: " + title + ", Author: " + author);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Book b1 = new Book("1984", "George Orwell");
        Book b2 = new Book("The Hobbit", "J.R.R. Tolkien");
        b1.display();
        b2.display();
    }
}
```

3. **Overloaded Constructors**

A class can have **multiple constructors** with different parameter lists (different number or type of arguments).

This provides **flexibility** when creating objects.

Example:

```
class Student {
String name;
int age;
// Default constructor
Student() {
        name = "Unknown";
```

```
        age = 0;
    }
    // Parameterized constructor (1 parameter)
    Student(String n) {
        name = n;
        age = 18; // default age
    }
    // Parameterized constructor (2 parameters)
    Student(String n, int a) {
        name = n;
        age = a;
    }
    void introduce() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
    }
}


public class MyMain {
public static void main(String[] args) {
        Student s1 = new Student();              // default
        Student s2 = new Student("Alice");       // 1 parameter
        Student s3 = new Student("Bob", 21);     // 2 parameters
        s1.introduce();
        s2.introduce();
        s3.introduce();
}
}
```

1.7. Initializers

Instance Initializer Block

Runs **before the constructor** every time an object is created.

Useful when you want **common initialization code** that all constructors should run, avoiding duplication.

Multiple initializer blocks can exist; they run **in the order they appear** in the class.

Runs **before the constructor body** but **after field initializers**.

```
1:  class Example {
2:  int value;
3:  // Instance initializer block
4:  {
5:  value = 100;
6:  System.out.println("Instance initializer executed");
7:  }
8:  // Constructor
9:  Example() {
10: System.out.println("Constructor executed, value = " + value);
11: }
12: }
```

```
1:  public class Main {
2:  public static void main(String[] args) {
3:  Example ex1 = new Example();
4:  Example ex2 = new Example();
5:  }
6:  }
```

OUTPUT:
Instance initializer executed

Constructor executed, value = 100

Instance initializer executed

Constructor executed, value = 100

1.8. Records (Java 14+)

A **record** is a special kind of class for immutable data-holding objects.
A **record** (introduced in Java 14 as a preview feature, stable since Java 16) is a **special kind of class** that is designed for **holding immutable data**.

Instead of writing boilerplate code (constructors, getters, toString, equals, hashCode), the **compiler generates them automatically**.

Key Properties of Records:

a) Immutable by design

   a. All fields are final and cannot be modified once created.

   b. Great for Data Transfer Objects (DTOs) or Value Objects.

b) Concise declaration

    a. A single line can replace dozens of lines of standard Java class code.

c) Compiler-generated methods

    a. Canonical constructor (matching all fields).

    b. toString() → Automatically prints values.

    c. equals() and hashCode() → Implemented based on fields.

d) Can contain methods: Records can still have additional methods for logic, validation, or formatting.

e) Cannot extend other classes: Records are implicitly final but can implement interfaces.

Example:

Instead of writing the class below:

```
class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getX() { return x; }
   public int getY() { return y; }
   @Override
   public String toString() {
      return "Point[x=" + x + ", y=" + y + "]";
   }
   @Override
   public boolean equals(Object o) { ... }
   @Override
   public int hashCode() { ... }
}
```

We write:

```
// File: Point.java
    1:  public record Point(int x, int y) {}
```

That's it — the compiler auto-generates everything above.

1.9. Enumerations (Enums)

Enums (short for Enumerations) in Java are special data types that represent a **fixed set of constants**.

They are **type-safe**, meaning you can't assign values outside the defined set.

Unlike integers or strings, they make code **more readable and less error-prone**.

```
1:  enum Day {
2:  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY
3:  }
```

```
1:  public class EnumDemo {
2:     public static void main(String[] args) {
3:      Day today = Day.MONDAY;
4:  if (today == Day.MONDAY) {
5:         System.out.println("Start of the week!");
6:  }
7:  }
8:  }
```

10. Best Practices in Class Design

1. **Encapsulation:** Make fields private, provide getters/setters.
2. **Prefer immutability:** Immutable objects are thread-safe and less error-prone.
3. **Use static for constants:** e.g., public static final double PI = 3.14159;.
4. **Keep constructors simple:** Use initializers for complex setup.
5. **Use records** when you only need data storage.
6. **Design enums** for fixed categories instead of strings or integers.

11. Code Examples

Example 1: Class with Instance and Static Members

```
class Student {
    String name;          // instance field
    int age;              // instance field
    static int total = 0;  // static field

    // Constructor
    Student(String n, int a) {
        name = n;
        age = a;
        total++;
    }

    void display() {
        System.out.println(name + " is " + age + " years old.");
    }

    static void displayTotal() {
        System.out.println("Total students: " + total);
    }
}

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

}

Explanation:

- total is shared among all students.
- Every new Student increments total.
- displayTotal() is a static method to show how many students exist.

Example 2: Constructor and Initializer Blocks

```
class Car {
   String brand;
   int speed;

   // Instance initializer
   { speed = 50; }

   // Constructor
   Car(String b) {
      brand = b;
   }

   void display() {
      System.out.println(brand + " starts at speed " + speed);
   }
}

public class TestCar {
   public static void main(String[] args) {
      Car c1 = new Car("Toyota");
      Car c2 = new Car("Honda");
      c1.display();
```

```
        c2.display();
    }
}
```

Explanation:

- speed is set to 50 in the initializer.
- Every car starts with default speed but brand is given in the constructor.

Example 3: Records and Enums

```
record Employee(String name, double salary) {}

enum Department {
    HR, IT, FINANCE
}

public class TestRecordEnum {
    public static void main(String[] args) {
        Employee e1 = new Employee("Alice", 50000);
        Employee e2 = new Employee("Bob", 60000);

        System.out.println(e1);
        System.out.println("Department: " + Department.IT);
    }
}
```

Explanation:

- Employee record is immutable and auto-generates toString().
- Department enum restricts values to HR, IT, or FINANCE only.

12. Practice Exercise

Task:
Create a Book class with the following requirements:

1. Fields: title (String), author (String), price (double).
2. A static field bookCount to track how many books are created.
3. A constructor that initializes the fields.
4. An instance method displayBook() that prints details of the book.
5. A static method getBookCount() that returns the total number of books created.
6. In main(), create at least 3 Book objects, display their details, and print the total count.

CHAPTER TWO: Inheritance and Polymorphism in Java

2.1. Introduction

Object-Oriented Programming (OOP) is built upon four main principles: **encapsulation, abstraction, inheritance, and polymorphism**. In this chapter, we will focus on **inheritance** and **polymorphism**, two powerful mechanisms in Java that allow developers to create reusable, extensible, and maintainable code.

- **Inheritance** allows one class (child or subclass) to acquire the fields and methods of another class (parent or superclass).
- **Polymorphism** enables objects to take many forms, meaning a single interface can be used with different underlying implementations.

Both concepts are core to OOP and are heavily used in Java's API and frameworks. By mastering them, you will understand how Java supports **code reuse, modularity, and flexibility**.

2.2. What is Inheritance?

Inheritance is the mechanism by which one class **inherits the properties and behavior** (fields and methods) of another class.

- The **class being inherited from** is called the **superclass (parent class)**.
- The **class that inherits** is called the **subclass (child class)**.

In Java, inheritance is declared using the keyword extends.

Syntax:

```
class Parent {
   // fields and methods
}
class Child extends Parent {
   // child-specific fields and methods
}
```

Key Points:

- Java supports **single inheritance only** (a class can have only one direct superclass).
- However, a class can **implement multiple interfaces**.
- Subclasses **inherit all accessible fields and methods** from the superclass (except constructors).
- Subclasses can also **add new fields and methods** or **override existing ones**.

2.3. Types of Inheritance in Java

1. Single Inheritance

In **single inheritance**, a class inherits directly from only one parent class.
This is the simplest and most common form.

```
class A {
   void displayA() {
      System.out.println("Class A method");
   }
}

class B extends A {
   void displayB() {
      System.out.println("Class B method");
   }
}

public class Main {
   public static void main(String[] args) {
      B obj = new B();
      obj.displayA(); // Inherited from A
      obj.displayB(); // Defined in B
   }
```

```
}
```

Here, B inherits from A.
B has access to both its own methods and those of A.

2. Multilevel Inheritance

In **multilevel inheritance**, a class is derived from another class, which itself is derived from another class.
This forms a **chain of inheritance**.

```
class A {
    void methodA() {
        System.out.println("Class A method");
    }
}
class B extends A {
    void methodB() {
        System.out.println("Class B method");
    }
}
class C extends B {
    void methodC() {
        System.out.println("Class C method");
    }
}
public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.methodA(); // From A
        obj.methodB(); // From B
        obj.methodC(); // From C
    }
```

```
}
```

C indirectly inherits from A through B.
This allows C to access methods from both A and B.

3. Hierarchical Inheritance

In **hierarchical inheritance**, multiple classes inherit from a **single parent class**.

```
class A {
   void commonMethod() {
      System.out.println("Common method in Class A");
   }
}
class B extends A {
   void methodB() {
      System.out.println("Class B method");
   }
}
class C extends A {
   void methodC() {
      System.out.println("Class C method");
   }
}

public class Main {
   public static void main(String[] args) {
      B objB = new B();
      objB.commonMethod();
      objB.methodB();

      C objC = new C();
```

```
        objC.commonMethod();
        objC.methodC();
    }
}
```

Both B and C share A as their parent.
Useful when you want different classes to reuse the same base functionality.

4. Multiple Inheritance (through Interfaces only)

Java **does not support multiple inheritance with classes** to avoid the **diamond problem** (ambiguity when two parent classes define the same method).
However, Java allows **multiple inheritance via interfaces**.

```
interface Interface1 {
    void method1();
}
interface Interface2 {
    void method2();
}
class MyClass implements Interface1, Interface2 {
    public void method1() {
        System.out.println("Method from Interface1");
    }
    public void method2() {
        System.out.println("Method from Interface2");
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.method1();
```

```
        obj.method2();
    }
}
```

A class can implement multiple interfaces.
This achieves multiple inheritance in a **safe and controlled way**.

2.4. The super Keyword

The super keyword in Java is a reference variable used to **refer to the immediate parent class object**.
It has **two main purposes**:

1. Accessing Parent Class Methods and Fields

When a subclass defines a method or field with the same name as its parent class (method overriding or field hiding), the super keyword is used to **differentiate** and explicitly access the parent's version.

Example: Accessing Parent Methods

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


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


    void printParentSound() {
        super.sound(); // Calls parent class method
    }
}
```

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

super.sound(); ensures we call the **parent version** of sound() even though it is overridden in Dog.

Example: Accessing Parent Fields

```
class Parent {
    int value = 100;
}

class Child extends Parent {
    int value = 200;

    void display() {
        System.out.println("Child value: " + value);
        System.out.println("Parent value: " + super.value);
    }
}

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

super.value accesses the parent's variable value when both parent and child define a field with the same name.

2. Calling Parent Class Constructors

The super() call is used inside a subclass constructor to **invoke the parent's constructor**.

- If **not explicitly written**, Java automatically inserts a call to the **default (no-argument) constructor** of the parent.
- If the parent class **does not have a default constructor**, the child class must explicitly call one of the parent's constructors using super(arguments).

Example: Default Constructor Call

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


class Child extends Parent {
   Child() {
      super(); // Implicitly added by Java if not written
      System.out.println("Child class constructor called");
   }
}


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

Output:

```
Parent class constructor called
```

Child class constructor called

Example: Parameterized Constructor Call

```
class Parent {
   Parent(String msg) {
      System.out.println("Parent constructor: " + msg);
   }
}

class Child extends Parent {
   Child(String msg) {
      super(msg); // Explicitly call parent's constructor
      System.out.println("Child constructor: " + msg);
   }
}

public class Main {
   public static void main(String[] args) {
      Child c = new Child("Hello World");
   }
}
```

Output:

Parent constructor: Hello World

Child constructor: Hello World

Without super(msg);, this code would cause a **compile-time error**, because the parent doesn't have a default constructor.

Key Points About super

1. super must be the **first statement** inside a constructor if used to call the parent's constructor.

2. If the parent class only has a **parameterized constructor**, the child must call it explicitly using super(arguments).
3. super can be used to:
    - Access parent fields
    - Call parent methods (even if overridden)
    - Invoke parent constructors

NB:

a. **super.methodName();** → Access parent class methods (especially when overridden).

b. **super.fieldName;** → Access parent class fields if hidden by child.

c. **super();** → Call parent constructor.

2.5. Method Overriding

Method overriding occurs when a **subclass provides a new implementation** for a method that is **already defined in its parent (superclass)**.
This allows a subclass to **customize or completely change** the behavior inherited from its parent.

It is one of the key concepts of **polymorphism** in Java (specifically, **runtime polymorphism**).

Requirements for Overriding

For a method to be considered **overridden** in Java:

1. **Same method name** – the child class method must have the exact name as the parent's method.
2. **Same parameter list** – the number and types of parameters must match exactly.
3. **Compatible return type** – the return type must be the same or a subtype (covariant return type).

Rules for Method Overriding

1. The method must be **inherited** from the superclass.
    - If the method is private, it is not inherited and **cannot** be overridden.

2. The **access modifier** in the child method cannot be more restrictive than in the parent.
    - Example: If the parent's method is public, the child method cannot be protected or private.
3. The overriding method **cannot throw broader checked exceptions** than the overridden method.
4. **Static methods** cannot be overridden. If you define a static method in both parent and child, it's called **method hiding**, not overriding.
5. The @Override annotation is **recommended** (but not required) because it makes the code clearer and helps the compiler catch mistakes.

Example: Basic Method Overriding

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


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


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

Output:

Dog barks

Even though the reference is of type Animal, the **actual object** is Dog.
At runtime, Java decides to call the overridden method (Dog's version).

This is **runtime polymorphism** in action.

Example: Using super with Overriding

Sometimes you want the child method to **extend** the parent method's behavior rather than completely replace it.

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

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();
    }
}
```

Output:

Animal makes a sound

Dog barks

Here, super.sound() allows the child to call the parent's version before adding its own behavior.

Access Modifier Rule Example

```
class Parent {
   protected void display() {
      System.out.println("Parent method");
   }
}

class Child extends Parent {
   @Override
   public void display() { // Can increase visibility
      System.out.println("Child method");
   }
}
```

Overriding vs Overloading

- **Overriding** → Same method signature, occurs across parent–child classes, enables polymorphism.
- **Overloading** → Same method name, but different parameter lists, within the same class.

NB:

- **Overriding** = redefining parent methods in child classes.
- Used for **runtime polymorphism**.
- Child method must have the **same signature** and a compatible return type.
- Cannot reduce **visibility** or throw broader checked exceptions.
- Use @Override for clarity and error checking.

2.6. What is Polymorphism?

The word **Polymorphism** comes from the Greek words:

- **Poly** = many
- **Morph** = forms

In Java, **polymorphism** means the ability of a single interface, method, or operator to represent **different forms or behaviors** depending on the context.

It is one of the **four pillars of Object-Oriented Programming (OOP)** along with **encapsulation, inheritance, and abstraction**.

Why is Polymorphism Important?

- It allows **code reusability**.
- It makes code **flexible and extensible**.
- It enables **runtime decisions** on which method to call (dynamic behavior).

Types of Polymorphism in Java

Java supports **two main types of polymorphism**:

1. Compile-time Polymorphism (Method Overloading)

This is also called **static polymorphism** because the method call is resolved **at compile time**.

How it works

- A class can have multiple methods with the **same name** but **different parameter lists** (number, order, or type of parameters).
- The compiler decides which method to call based on the method signature.

Example: Method Overloading

```
class Calculator {
  // Overloaded methods
  int add(int a, int b) {
    return a + b;
```

```
    }

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

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

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));        // Calls int version
        System.out.println(calc.add(5.5, 2.5));     // Calls double version
        System.out.println(calc.add(1, 2, 3));      // Calls three-parameter version
    }
}
```

Output:

15

8.0

6

Here, add() is polymorphic — it behaves differently based on the arguments provided.

2. Runtime Polymorphism (Method Overriding)

This is also called **dynamic polymorphism** because the method call is resolved **at runtime**, not at compile time.

How it works

- A subclass provides a **specific implementation** for a method already defined in its parent class.
- When we call the method on a **parent reference variable** pointing to a **child object**, Java decides at runtime which version to call (parent or child).

Example: Method Overriding

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

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

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

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Dog();  // Upcasting
        Animal a2 = new Cat();
```

```
        a1.sound(); // Runtime decision: Dog's sound()
        a2.sound(); // Runtime decision: Cat's sound()
    }
}
```

Output:

Dog barks

Cat meows

Even though the reference type is Animal, Java decides at **runtime** to call the child class methods.

This is a classic example of **runtime polymorphism**.

- **Overloading** = same method name, different parameters (**compile time**).
- **Overriding** = same method signature in parent and child (**runtime**).
- Polymorphism allows Java programs to be **more modular, scalable, and easier to maintain**.

NB:

Benefits of Inheritance and Polymorphism

- Code Reusability – Write once, reuse in multiple subclasses.
- Extensibility – Easily add new functionality by extending existing classes.
- Maintainability – Centralized code reduces redundancy.
- Flexibility – Polymorphism allows code to work with objects of different types seamlessly.

Limitations of Inheritance

- Java does not support multiple inheritance of classes (to avoid ambiguity, also called the "Diamond Problem").
- Overuse of inheritance can lead to **tight coupling**. Sometimes **composition** is a better choice.

2.7. Upcasting and Dynamic Method Dispatch

- **Upcasting**: Referring to a subclass object using a superclass reference.
- **Dynamic Method Dispatch**: The mechanism by which a call to an overridden method is resolved at runtime.

Example:

```
Animal a = new Dog(); // upcasting
a.sound(); // calls Dog's sound() due to dynamic method dispatch
```

2.8 Code Examples

Example 1: Basic Inheritance

```
class Vehicle {
    String brand = "Toyota";

    void honk() {
        System.out.println("Beep! Beep!");
    }
}

class Car extends Vehicle {
    int wheels = 4;

    void display() {
        System.out.println("Brand: " + brand + ", Wheels: " + wheels);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.honk();   // inherited method
        myCar.display(); // child method
    }
}
```

Explanation:

- The Car class inherits from Vehicle.
- honk() is inherited, while display() is defined in Car.

Example 2: Method Overriding and Polymorphism

```
class Animal {
    void sound() {
        System.out.println("Some generic animal sound");
    }
}

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

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

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Cat(); // upcasting
        Animal a2 = new Dog();
```

```
        a1.sound(); // Cat meows
        a2.sound(); // Dog barks
    }
}
```

Explanation:

- Both Cat and Dog override the sound() method.
- Due to **dynamic method dispatch**, the correct method is chosen at runtime.

Example 3: Using super Keyword

```
class Person {
    String name;

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

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

class Student extends Person {
    int grade;

    Student(String name, int grade) {
        super(name); // call parent constructor
        this.grade = grade;
    }
```

```
    @Override
    void displayInfo() {
        super.displayInfo(); // call parent method
        System.out.println("Grade: " + grade);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Alice", 10);
        s.displayInfo();
    }
}
```

Explanation:

- The Student class calls the **parent constructor** using super(name).
- It also overrides displayInfo() but reuses the parent's implementation.

2.9. Practice Exercise

Task:
Create a class Employee with fields name and salary and a method displayDetails().

- Then create two subclasses:

  Manager with an extra field department.

  Developer with an extra field programmingLanguage.
- Override displayDetails() in both subclasses.
- Write a Main class that demonstrates **polymorphism** by storing Employee references to Manager and Developer objects and calling displayDetails().