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	**CAT 1**
5	**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
6	**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
7	**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
8	**CAT 2**
9	**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
10	**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
11	**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

12	**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.

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:

```
class Car {
    String brand;
    int speed;
    // Default constructor is provided automatically
}
```

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

2. **Parameterized Constructor**

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

Example (using your Book class, expanded):

```
class Book {
    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.

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().

CHAPTER THREE: Abstract Classes and Interfaces in Java

3.1 Introduction

Java is an object-oriented programming language that provides powerful mechanisms to design and build reusable, modular, and extensible software components. Two such key mechanisms are **abstract classes** and **interfaces**. They serve as **contracts** or **blueprints** for other classes, ensuring that certain methods are implemented while promoting flexibility and reducing code duplication.

While **inheritance** allows us to reuse code from a parent class, and **polymorphism** provides runtime flexibility, **abstract classes and interfaces** take this further by allowing us to define **what a class must do**, without necessarily dictating **how it must do it**.

Understanding abstract classes and interfaces is crucial for mastering Java programming and is especially important when designing large-scale applications, frameworks, or APIs where code extensibility and maintainability are required.

3.2 Abstract Classes

3.2.1 Definition

An **abstract class** in Java is a class that cannot be instantiated on its own. It may contain both **abstract methods** (methods without a body, only a declaration) and **concrete methods** (methods with implementation).

- Abstract classes are declared using the abstract keyword.
- They act as **partially implemented blueprints** for other classes.

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


   void sleep() { // concrete method
      System.out.println("Sleeping...");
   }
}
```

In this example, the Animal class provides a **contract** (makeSound()) that must be implemented by any subclass, while also offering a ready-to-use method sleep().

3.2.2 Key Characteristics of Abstract Classes

1. **Cannot be instantiated directly.**
   You cannot create an object of an abstract class.
2. Animal a = new Animal(); // ❌ Error
3. **Can contain abstract methods.**
   These are declared but not implemented in the abstract class.
4. **Can contain concrete methods.**
   Subclasses inherit these directly.
5. **Can have constructors.**
   Abstract classes can define constructors to initialize fields, which are called when subclasses are instantiated.
6. **Can have instance variables and static members.**
   Unlike interfaces, abstract classes can maintain **state**.
7. **Support inheritance.**
   Subclasses extend abstract classes and must implement all abstract methods, unless the subclass is also abstract.

3.2.3 Example: Abstract Class with Implementation

```
abstract class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }
    // Abstract method
    abstract void drive();
    // Concrete method
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

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

    @Override
    void drive() {
        System.out.println(brand + " Car is driving...");
    }
}

public class AbstractDemo {
    public static void main(String[] args) {
        Vehicle v = new Car("Toyota");
        v.displayBrand();
        v.drive();
    }
}
```

Output:

Brand: Toyota

Toyota Car is driving...

Here, the abstract class Vehicle enforces a contract (drive()) while also providing a concrete implementation (displayBrand()).

3.3 Interfaces

3.3.1 Definition

An **interface** in Java is a completely abstract type that specifies a set of methods that a class must implement. Interfaces are declared using the interface keyword. Unlike abstract classes, **interfaces cannot contain instance fields or constructors**, but they can contain constants, abstract methods, default methods, and static methods.

Interfaces are used to define **capabilities** or **behaviors** that can be applied to multiple classes, even across unrelated inheritance hierarchies.

3.3.2 Characteristics of Interfaces

1. **No constructors.**
   Interfaces cannot be instantiated or have constructors.
2. **Multiple inheritance of type.**
   A class can implement multiple interfaces, providing a workaround to Java's single inheritance limitation.
3. **Methods.**
   - **Abstract methods**: Must be implemented by implementing classes.
   - **Default methods**: Contain implementation, introduced in Java 8.
   - **Static methods**: Belong to the interface itself.
4. **Fields.**
   All variables in an interface are implicitly public static final.

3.3.3 Example: Interface

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

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

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

```
        Animal a = new Dog();
        a.makeSound();
    }
}
```

Output:

Woof! Woof!

Here, Dog implements the Animal interface by providing the body of makeSound().

3.3.4 Interfaces with Default and Static Methods

```
interface Calculator {
    int add(int a, int b);

    // Default method
    default int subtract(int a, int b) {
        return a - b;
    }

    // Static method
    static int multiply(int a, int b) {
        return a * b;
    }
}

class SimpleCalculator implements Calculator {
    @Override
    public int add(int a, int b) {
        return a + b;
    }
}
```

```
public class InterfaceMethodsDemo {
    public static void main(String[] args) {
        SimpleCalculator calc = new SimpleCalculator();
        System.out.println("Add: " + calc.add(10, 5));
        System.out.println("Subtract: " + calc.subtract(10, 5));
        System.out.println("Multiply: " + Calculator.multiply(10, 5));
    }
}
```

Output:

Add: 15

Subtract: 5

Multiply: 50

3.4 Abstract Classes vs Interfaces

Feature	**Abstract Class**	**Interface**
Instantiation	Cannot be instantiated	Cannot be instantiated
Methods	Can have abstract and concrete methods	Abstract (default & static since Java 8)
Constructors	Yes	No
Variables	Instance & static allowed	Only public static final allowed
Multiple Inheritance	Single class inheritance only	Can implement multiple interfaces
When to use	Shared code among related classes	Common behavior across unrelated classes

3.5 Polymorphism with Abstract Classes and Interfaces

Both abstract classes and interfaces enable **polymorphism**. This means you can treat objects of different classes uniformly as long as they share a common abstract class or implement the same interface.

Example:

```
interface Shape {
    void draw();
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Circle...");
    }
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle...");
    }
}

public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Rectangle();

        s1.draw();
        s2.draw();
```

```
    }
}
```

Output:

Drawing Circle...

Drawing Rectangle...

Here, both Circle and Rectangle can be treated as Shape objects.

3.6 Practical Use Cases

1. **Abstract Classes:**
    - When classes share common attributes and methods.
    - Example: Vehicle → Car, Bike, Truck.
2. **Interfaces:**
    - When different classes need to exhibit similar behavior but are not related.
    - Example: Comparable, Serializable, or user-defined Payment interface for CreditCardPayment, PayPalPayment.

3.7 Code Examples

Example 1: Abstract Class

```
abstract class Appliance {
    abstract void turnOn();
    abstract void turnOff();
}


class Fan extends Appliance {
    @Override
    void turnOn() {
        System.out.println("Fan is spinning...");
    }
```

```
    @Override
    void turnOff() {
        System.out.println("Fan stopped.");
    }
}

public class ApplianceDemo {
    public static void main(String[] args) {
        Appliance fan = new Fan();
        fan.turnOn();
        fan.turnOff();
    }
}
```

Example 2: Interface with Multiple Implementations

```
interface Payment {
    void pay(double amount);
}

class CreditCardPayment implements Payment {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements Payment {
    @Override
    public void pay(double amount) {
```

```
        System.out.println("Paid " + amount + " using PayPal.");
    }
}

public class PaymentDemo {
    public static void main(String[] args) {
        Payment p1 = new CreditCardPayment();
        Payment p2 = new PayPalPayment();

        p1.pay(100.50);
        p2.pay(200.75);
    }
}
```

Example 3: Combining Abstract Class and Interface

```
interface Drivable {
    void accelerate();
}

abstract class Vehicle {
    abstract void brake();
}

class Car extends Vehicle implements Drivable {
    @Override
    public void accelerate() {
        System.out.println("Car accelerating...");
    }
```

```
    @Override
    void brake() {
        System.out.println("Car braking...");
    }
}

public class CombinedDemo {
    public static void main(String[] args) {
        Car c = new Car();
        c.accelerate();
        c.brake();
    }
}
```

3.8 Practice Exercise

Exercise:
Create a Java program with the following requirements:

1. Define an **abstract class** Employee with attributes name and salary.
    - Include an abstract method calculateBonus().
    - Include a concrete method displayDetails().
2. Define two subclasses:
    - Manager → calculates a 20% bonus.
    - Developer → calculates a 10% bonus.
3. Define an **interface** Reportable with a method generateReport().
    - Both Manager and Developer must implement this interface to display their role and bonus.
4. In the main method, create objects for Manager and Developer, calculate their bonuses, and generate reports.

CHAPTER 4: Immutable Classes and Nested Classes

4.1. Introduction

Java provides developers with multiple ways to model data and encapsulate behavior. Two important concepts that strengthen object-oriented programming are **immutable classes** and **nested classes**.

- **Immutable classes** provide thread-safe, stable, and predictable objects whose state cannot change after creation. They are vital in concurrent programming and functional programming.
- **Nested classes** allow grouping related classes logically inside another class, which increases code organization, readability, and encapsulation.

By mastering these concepts, students learn how to design objects that are reliable and safe to use in complex applications, especially when dealing with multi-threading and large-scale systems.

4.2. Immutable Classes in Java

An **immutable class** is a class whose objects, once created, cannot be modified. The internal state of an immutable object remains constant throughout its lifetime.

Examples:

- The most widely used immutable class in Java is **String**. Once you create a string, you cannot alter its value.
- Classes like **Integer**, **LocalDate**, and **BigDecimal** are also immutable.

Why Use Immutable Classes?

1. **Thread Safety**
   Since immutable objects cannot be modified, they are inherently thread-safe. Multiple threads can access the same immutable object without synchronization.
2. **Security**
   Immutable classes protect sensitive data from accidental or malicious modification.
3. **Cache and Reusability**
   Immutable objects can be safely cached and reused, improving memory management.
4. **Predictability**
   Since the state never changes, behavior is consistent, reducing bugs.

Rules for Creating Immutable Classes

To create an immutable class in Java:

1. Declare the class as **final** so it cannot be extended.
2. Make all fields **private** and **final**.
3. Do not provide setters.
4. Initialize all fields via the constructor.
5. If a field is mutable (e.g., a list or Date), return a **defensive copy** rather than the original reference.

Example of an Immutable Class

```
// Example 1: Immutable Student Class
public final class Student {
    private final String name;
    private final int age;

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

    // Getters only, no setters
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // No methods that modify the state
    @Override
```

```
    public String toString() {
        return "Student{name='" + name + "', age=" + age + "}";
    }
}

class TestImmutable {
    public static void main(String[] args) {
        Student s = new Student("Alice", 22);
        System.out.println(s); // Student{name='Alice', age=22}
        // s.setName("Bob"); // Not possible
    }
}
```

Here:

- Class is final
- Fields are private and final
- No setters provided

Defensive Copy in Immutable Class

If an immutable class has a mutable field (like Date or List), you must protect it with a defensive copy.

```
import java.util.Date;

public final class ImmutableEmployee {
    private final String name;
    private final Date joinDate;

    public ImmutableEmployee(String name, Date joinDate) {
        this.name = name;
        this.joinDate = new Date(joinDate.getTime()); // Defensive copy
    }
```

```
    public String getName() {
        return name;
    }

    public Date getJoinDate() {
        return new Date(joinDate.getTime()); // Return copy
    }
}

class TestEmployee {
    public static void main(String[] args) {
        Date date = new Date();
        ImmutableEmployee emp = new ImmutableEmployee("John", date);

        System.out.println(emp.getJoinDate());

        // Try to modify original date
        date.setTime(0);

        // Employee's joinDate remains unchanged
        System.out.println(emp.getJoinDate());
    }
}
```

This ensures immutability is preserved even if the original mutable object changes.

4.3. Nested Classes in Java

A **nested class** is a class defined within another class. Nested classes help logically group classes that are only used in one place, improve encapsulation, and make code more maintainable.

Types of Nested Classes

1. Static Nested Class

- A **static nested class** is declared using the static keyword inside another class.
- Since it is static, it behaves much like a top-level class but is logically grouped inside the outer class for better organization.
- It **does not require an instance** of the outer class to be created. Instead, it can be instantiated directly using the outer class name.
- However, it **cannot access non-static members** (instance variables or methods) of the outer class directly. It can only access the **static members** of the outer class.
- **Use case:** Useful when the nested class is closely related to the outer class but does not need a reference to an instance of the outer class.

Example:

```
1:  class Outer {
2:      static int outerValue = 10;
3:      static class StaticNested {
4:          void display() {
5:              System.out.println("Outer static value: " + outerValue);
6:          }
7:      }
8:  }


9:  public class Main {
10:     public static void main(String[] args) {
11:         Outer.StaticNested nested = new Outer.StaticNested();
12:         nested.display();
13:     }
14: }
```

2. Non-Static Inner Class

- A **non-static inner class** (simply called an inner class) is associated with an instance of the outer class.
- To create an object of an inner class, you must first create an object of the outer class.
- It can **access both static and non-static members** of the outer class, including private fields and methods, because it maintains a reference to the enclosing instance.
- **Use case:** Suitable when you need a class that logically belongs inside another class and frequently uses its instance members.

  Example:

```
1:  class Outer {
2:      private String message = "Hello from Outer";
3:
4:      class Inner {
5:          void showMessage() {
6:              System.out.println(message); // Accessing outer class instance variable
7:          }
8:      }
9:  }
10:
11: public class Main {
12:     public static void main(String[] args) {
13:         Outer outer = new Outer();
14:         Outer.Inner inner = outer.new Inner();
15:         inner.showMessage();
16:     }
17: }
```

3. Local Inner Class

- A **local inner class** is declared **inside a method, constructor, or block** of the outer class.
- Its scope is limited to the method or block where it is defined. It **cannot be accessed outside** that method.

- It can access all members of the enclosing class and **final or effectively final local variables** of the method.
- **Use case:** Best suited when you want a class for a specific, short-lived task inside a method and don't want it to be accessible elsewhere.

Example:

```
1:  class Outer {
2:      void outerMethod() {
3:          int number = 42; // effectively final variable
4:          class LocalInner {
5:              void printNumber() {
6:                  System.out.println("Number is: " + number);
7:              }
8:          }

9:          LocalInner local = new LocalInner();
10:         local.printNumber();
11:     }
12: }

13: public class Main {
14:     public static void main(String[] args) {
15:         Outer outer = new Outer();
16:         outer.outerMethod();
17:     }
18: }
```

4. Anonymous Inner Class

- An **anonymous inner class** is a class **without a name** that is declared and instantiated at the same time.

- It is typically used when you need to override methods of a class or implement an interface for one-time use.
- Because it doesn't have a name, it cannot be reused.
- Commonly used in **GUI event handling, threading, and callbacks**.
- **Use case:** When you need to implement a method quickly without writing a separate full class.

Example:

```
1:  abstract class Greeting {
2:      abstract void sayHello();
3:  }
4:
5:  public class Main {
6:      public static void main(String[] args) {
7:          Greeting g = new Greeting() { // Anonymous inner class
8:              void sayHello() {
9:                  System.out.println("Hello from anonymous inner class!");
10:             }
11:         };
12:         g.sayHello();
13:     }
14: }
```

NB:

- **Static Nested Class:** Independent, no outer instance needed.
- **Inner Class:** Dependent on outer instance, can access outer members.
- **Local Inner Class:** Exists within a method, limited scope.
- **Anonymous Inner Class:** Declared and used instantly, often for quick overrides or event handling.

4.4. Key Differences: Immutable Classes vs Nested Classes

Feature	Immutable Classes	Nested Classes
Purpose	Ensure object's state cannot change	Group related classes logically
Thread Safety	Always thread-safe	Depends on implementation
Encapsulation	Encapsulates data strongly	Encapsulates class definitions
Examples	String, Integer	Inner class, Static Nested Class

4.5. Practical Applications

Practical Applications of Immutable Classes

1. **Banking Systems**

   In financial applications, accuracy and security are crucial. Immutable classes (like String, Integer, BigDecimal) ensure that once data such as an account balance, transaction ID, or customer identifier is created, it cannot be altered.

   This reduces the chances of accidental modifications or malicious tampering.

   Example: Using an immutable Transaction object ensures that a transaction record remains unchanged after it is created.

2. **Caching Frameworks**

   Caching systems often store objects for reuse. If those objects are mutable, they could be altered unintentionally by different parts of the program, leading to incorrect results.

   Immutable classes guarantee that cached values stay consistent and safe for reuse.

   Example: In frameworks like **Ehcache** or when using in-memory caches (e.g., Map as a cache), immutable objects like String are often used as keys since their values cannot change.

3. **Concurrent Applications (Multithreading)**

   In multithreaded environments, immutable objects are inherently thread-safe because their state cannot change after creation.

This removes the need for synchronization, reducing complexity and improving performance.

Example: String and wrapper classes (Integer, Double, etc.) are frequently used in concurrent applications since multiple threads can safely share them.

Practical Applications of Nested Classes

1. **GUI Applications (JavaFX, Swing, AWT)**

   Nested classes, especially **anonymous inner classes**, are widely used to handle events like button clicks, mouse movements, or keyboard input.

   Instead of creating separate classes for each event, developers can use nested classes for cleaner, localized code.

   Example:

```
1: button.addActionListener(new ActionListener() {
2:     public void actionPerformed(ActionEvent e) {
3:         System.out.println("Button clicked!");
4:     }
5: });
```

2. **Collections Framework (e.g., Map.Entry)**

   The Java Collections Framework makes extensive use of nested classes.

   For example, the Map.Entry interface represents a key-value pair inside a Map.

   This structure keeps the Entry logically grouped with Map, making the design cleaner and easier to maintain.

   Example:

```
1: Map<String, Integer> scores = new HashMap<>();
2: scores.put("Alice", 90);
3: scores.put("Bob", 85);


4: for (Map.Entry<String, Integer> entry : scores.entrySet()) {
5:     System.out.println(entry.getKey() + " : " + entry.getValue());
6: }
```

3. **Event Handling**

   Local and anonymous inner classes are extremely useful for event-driven programming.

   They allow defining small handlers for specific events without cluttering the codebase with many extra classes.

   Example: In JavaFX, event listeners can be written as anonymous inner classes to handle user interactions like clicking, dragging, or typing.

4.6. Code Examples

Example 1: Immutable BankAccount

```
public final class BankAccount {
    private final String accountNumber;
    private final double balance;

    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public String getAccountNumber() { return accountNumber; }
    public double getBalance() { return balance; }

    @Override
    public String toString() {
        return "BankAccount{accountNumber='" + accountNumber + "', balance=" + balance + "}";
    }
}
```

Example 2: Static Nested Class

```
public class Computer {
    static class Processor {
        void details() {
            System.out.println("Processor: Intel i7, 3.2GHz");
        }
    }

    public static void main(String[] args) {
        Computer.Processor processor = new Computer.Processor();
        processor.details();
    }
}
```

Example 3: Inner Class in Event Handling

```
public class Button {
    class ClickListener {
        void onClick() {
            System.out.println("Button Clicked!");
        }
    }

    public static void main(String[] args) {
        Button btn = new Button();
        Button.ClickListener listener = btn.new ClickListener();
        listener.onClick();
    }
}
```

4.7. Practice Exercise

Task:
Create an **immutable Book class** with fields: title, author, and publicationDate.

- Ensure immutability (use defensive copies if needed).
- Inside the Book class, create a **nested static class Publisher** with fields name and address.
- Write a main method that:
    1. Creates a Book object.
    2. Creates a Publisher object.
    3. Prints their details.

CHAPTER 5: Enumerations and Lambda Expressions

5.1 Introduction

Enumerations (enums) and Lambda expressions are two powerful constructs in Java that make programming more structured, expressive, and concise.

- **Enumerations** provide a way to define a fixed set of constants with semantic meaning, enabling type-safety and more readable code. They are commonly used when a variable should hold one value from a predefined set (e.g., days of the week, traffic light signals, directions).
- **Lambda expressions**, introduced in Java 8, enable writing cleaner and more expressive code when dealing with functional programming patterns. They allow methods to be treated as data, which is particularly useful for passing behavior into methods like map, filter, or event handling.

5.2 Enumerations in Java

An enumeration (enum) is a special data type in Java used to define a collection of constants. Unlike primitive constants (final static int), enums are full-fledged classes that can have fields, methods, and constructors.

Example: Days of the week.

```
public enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Here, Day is an enum type, and the values are its constants.

Features of Enums

- Each enum constant is **implicitly public, static, and final**.
- Enums provide **type safety**—you cannot assign values outside the defined set.
- Enums can have fields, methods, and constructors.
- Enums are **inherently serializable and comparable**.
- Enums **extend java.lang.Enum**, so they cannot extend another class, but they can implement interfaces.

Defining Enums with Fields and Methods

Enums can have instance fields and methods. Each enum constant can have different values.

Example:

```
public enum TrafficLight {
   RED("Stop"),
   YELLOW("Caution"),
   GREEN("Go");

   private String action;

   // Constructor
   TrafficLight(String action) {
      this.action = action;
   }

   // Getter method
   public String getAction() {
      return action;
   }
}

class TestTraffic {
   public static void main(String[] args) {
      for (TrafficLight light : TrafficLight.values()) {
         System.out.println(light + ": " + light.getAction());
      }
   }
}
```

Output:

```
RED: Stop
YELLOW: Caution
```

GREEN: Go

Useful Enum methods in Java with details and examples:

1. values()

The values() method returns an array containing all the constants of the enum type, in the order they are declared.

It is automatically generated by the compiler for every enum.

Useful for iterating through all constants of an enum.

Example:

```
enum Day {
   MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}


public class Main {
   public static void main(String[] args) {
      for (Day d : Day.values()) {
         System.out.println(d);
      }
   }
}
```

Output:

```
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

SUNDAY

2. valueOf(String name)

The valueOf() method returns the enum constant whose name matches the specified string.

The string must exactly match the identifier used to declare the enum constant (case-sensitive).

Throws an IllegalArgumentException if no constant matches.

Example:

```
enum Color {
   RED, GREEN, BLUE
}


public class Main {
   public static void main(String[] args) {
      Color c1 = Color.valueOf("RED");
      System.out.println(c1);


      // Color c2 = Color.valueOf("Purple"); // This would throw IllegalArgumentException
   }
}
```

Output:

RED

3. ordinal()

The ordinal() method returns the position (index) of the enum constant in its declaration, starting from **0**.

It is often used when you need to know the relative order of constants.

However, relying on ordinal values is not recommended for business logic, as reordering enum constants in code will change their ordinals.

Example:

```
enum Direction {
   NORTH, EAST, SOUTH, WEST
}

public class Main {
   public static void main(String[] args) {
      System.out.println(Direction.NORTH.ordinal()); // 0
      System.out.println(Direction.SOUTH.ordinal()); // 2
   }
}
```

Output:

```
0
2
```

4. compareTo()

The compareTo() method compares two enum constants based on their **ordinal values**.

Returns:

- 0 if both are the same constant,
- a negative number if the first comes before the second,
- a positive number if the first comes after the second.

Implements the Comparable interface, which makes enums naturally ordered.

Example:

```
enum Size {
   SMALL, MEDIUM, LARGE, EXTRA_LARGE
}

public class Main {
   public static void main(String[] args) {
      Size s1 = Size.SMALL;
```

```
        Size s2 = Size.LARGE;

        System.out.println(s1.compareTo(s2)); // -2 (SMALL before LARGE)
        System.out.println(s2.compareTo(s1)); // 2  (LARGE after SMALL)
        System.out.println(s1.compareTo(Size.SMALL)); // 0 (equal)
    }
}
```

Output:

```
-2
2
0
```

Enum with Switch Statement

Enums are ideal in switch statements for better readability.

```
Day today = Day.MONDAY;

switch (today) {
    case MONDAY:
        System.out.println("Start of the work week!");
        break;
    case FRIDAY:
        System.out.println("Weekend is near!");
        break;
    default:
        System.out.println("Another day in the week.");
}
```

5.3 Lambda Expressions in Java

A **lambda expression** is an anonymous function that can be treated as a value. It enables writing short and concise implementations of functional interfaces.

General syntax:

(parameters) -> expression

(parameters) -> { statements }

Example:

// Lambda for addition

(int a, int b) -> a + b

Functional Interfaces

A **functional interface** is an interface with a single abstract method. Examples: Runnable, Callable, Comparator, Function, Consumer.

The annotation @FunctionalInterface ensures the interface has only one abstract method.

Example:

```
@FunctionalInterface
interface Greeting {
   void sayHello(String name);
}
```

You can implement this interface with a lambda:

```
Greeting greet = (name) -> System.out.println("Hello, " + name);
greet.sayHello("Alice");
```

Why Use Lambdas?

- Reduce boilerplate code.
- Enable functional programming in Java.
- Improve readability and maintainability.
- Useful for **streams**, **event handling**, and **multithreading**.

Examples of Lambda Usage

Example 1: Runnable with Lambda

```
public class LambdaRunnable {
    public static void main(String[] args) {
        Runnable task = () -> System.out.println("Task running using Lambda!");
        new Thread(task).start();
    }
}
```

Example 2: Sorting with Comparator

```
import java.util.*;

public class LambdaSort {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Alice", "Bob");

        // Sorting using lambda
        names.sort((a, b) -> a.compareTo(b));

        System.out.println(names);
    }
}
```

Example 3: Stream API with Lambda

```
import java.util.*;

public class StreamLambda {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        numbers.stream()
```

```
            .filter(n -> n % 2 == 0)
            .forEach(n -> System.out.println("Even: " + n));
    }
}
```

5.4 Combining Enums and Lambdas

You can use enums and lambdas together, especially when defining behaviors.

Example:

```
public enum Operation {
    ADD((a, b) -> a + b),
    SUBTRACT((a, b) -> a - b),
    MULTIPLY((a, b) -> a * b);

    private final Calculator calc;

    Operation(Calculator calc) {
        this.calc = calc;
    }

    public int apply(int a, int b) {
        return calc.operate(a, b);
    }

    @FunctionalInterface
    interface Calculator {
        int operate(int a, int b);
    }
}
```

```
class TestOperation {
    public static void main(String[] args) {
        System.out.println(Operation.ADD.apply(5, 3));      // 8
        System.out.println(Operation.SUBTRACT.apply(5, 3)); // 2
        System.out.println(Operation.MULTIPLY.apply(5, 3)); // 15
    }
}
```

NB:

- Use enums instead of constant values (final static) for readability and safety.
- Use enums when you have a fixed set of options.
- Use lambdas for short, one-time implementations of functional interfaces.
- Avoid overusing lambdas where traditional methods are clearer.
- Combine lambdas with Java Streams for concise data processing.

- **Enums** provide a type-safe way to define constants with optional fields, methods, and constructors.
- **Lambdas** allow concise implementation of functional interfaces, enabling functional-style programming in Java.
- Both constructs improve readability, maintainability, and expressiveness of code.
- Enums can be combined with lambdas for defining behaviors in a structured way.

5.7 Code Examples

Example 1: Enum with Methods

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL;

    public boolean isCold() {
        return this == WINTER;
```

```
    }
}

class TestSeason {
    public static void main(String[] args) {
        System.out.println(Season.WINTER.isCold()); // true
    }
}
```

Example 2: Lambda for Event Handling

```
import java.awt.*;
import javax.swing.*;

public class LambdaEvent {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Lambda Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(e -> System.out.println("Button Clicked!"));

        frame.add(button);
        frame.setSize(200, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Example 3: Stream Processing with Lambda

```
import java.util.*;
```

```
public class LambdaStream {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date");

        words.stream()
            .filter(w -> w.startsWith("b"))
            .forEach(System.out::println); // prints "banana"
    }
}
```

5.8 Practice Exercise

Problem:
Create an **enum** called MathOperation with constants ADD, SUBTRACT, MULTIPLY, and DIVIDE. Each constant should implement an abstract method apply(int a, int b) using **lambdas** inside the enum.

Tasks:

1. Implement the enum and define the behavior for each operation.
2. Write a main method to perform all four operations on a = 20 and b = 5.
3. Print the results clearly.

CHAPTER 6: Exception Handling and Internationalization

6.1. Introduction

In software development, errors are inevitable. Java provides a robust mechanism to handle such runtime errors through its *exception handling* framework. Exception handling ensures that a program can gracefully recover from unexpected situations without crashing.

In addition to handling exceptions, modern applications need to support users from different regions and cultures. This is where *Internationalization (i18n)* comes in. Java offers a set of APIs to build software that adapts to various languages, number formats, dates, and currencies.

This chapter covers both **Exception Handling** and **Internationalization**, two crucial aspects for building reliable, user-friendly, and globally adaptable Java applications.

6.2. Exception Handling in Java

An exception is an abnormal condition or event that occurs during program execution and disrupts the normal flow of instructions. Exceptions in Java are represented as objects derived from the Throwable class.

Hierarchy:

Throwable

Error: Serious problems that applications cannot recover from (e.g., OutOfMemoryError).

Exception: Conditions applications can handle.

- **Checked Exceptions**: Must be handled explicitly using try/catch or declared in method signatures (e.g., IOException, SQLException).
- **Unchecked Exceptions**: Runtime exceptions that do not require explicit handling (e.g., NullPointerException, ArrayIndexOutOfBoundsException).

Exception Handling Mechanism

Java uses five keywords to handle exceptions:

1. try – Defines a block of code where exceptions might occur.
2. catch – Defines a block of code to handle exceptions.
3. finally – Executes code after try/catch, regardless of exception occurrence.
4. throw – Used to explicitly throw an exception.

5. throws – Declares exceptions a method may throw.

Try-Catch Block

```
try {
   int result = 10 / 0; // ArithmeticException
} catch (ArithmeticException e) {
   System.out.println("Error: Division by zero is not allowed.");
}
```

Finally Block

The finally block always executes, whether an exception occurs or not. It is often used to release resources (files, database connections, etc.).

```
try {
   int[] arr = {1, 2, 3};
   System.out.println(arr[5]); // ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Error: Index out of bounds.");
} finally {
   System.out.println("Finally block executed.");
}
```

Try-with-Resources

Introduced in Java 7, this feature automatically closes resources like files and sockets when used with the AutoCloseable interface.

```
import java.io.*;

public class TryWithResourcesExample {
   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
```

```
            System.out.println(br.readLine());
        } catch (IOException e) {
            System.out.println("I/O Error: " + e.getMessage());
        }
    }
}
```

Multi-Catch Block

A single catch block can handle multiple exceptions.

```
try {
    String text = null;
    System.out.println(text.length()); // NullPointerException
} catch (ArithmeticException | NullPointerException e) {
    System.out.println("Caught: " + e);
}
```

Custom Exceptions

You can create user-defined exceptions by extending the Exception class.

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older.");
        } else {
```

```
            System.out.println("Valid age.");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

6.3. Internationalization (i18n)

Internationalization is the process of designing software so it can be adapted to different languages, regions, and cultures without requiring code changes.

Java provides the Locale class and ResourceBundle API to support i18n.

Locale Class

The Locale class represents a specific geographical, political, or cultural region.

```
import java.util.*;

public class LocaleExample {
    public static void main(String[] args) {
        Locale locale = new Locale("fr", "FR");
        System.out.println("Language: " + locale.getDisplayLanguage());
        System.out.println("Country: " + locale.getDisplayCountry());
    }
}
```

Number and Currency Formatting

NumberFormat and Currency classes help format numbers based on locale.

```
import java.text.*;
import java.util.*;

public class NumberFormatExample {
   public static void main(String[] args) {
      double number = 123456.789;
      Locale us = Locale.US;
      Locale france = Locale.FRANCE;

      NumberFormat nfUS = NumberFormat.getCurrencyInstance(us);
      NumberFormat nfFR = NumberFormat.getCurrencyInstance(france);

      System.out.println("US: " + nfUS.format(number));
      System.out.println("France: " + nfFR.format(number));
   }
}
```

Date Formatting

Java supports localized date/time formatting.

```
import java.text.*;
import java.util.*;

public class DateFormatExample {
   public static void main(String[] args) {
      Date today = new Date();
      Locale india = new Locale("hi", "IN");
```

```
        DateFormat dfIndia = DateFormat.getDateInstance(DateFormat.FULL, india);
        System.out.println("India Format: " + dfIndia.format(today));
    }
}
```

ResourceBundle

Used for managing localized messages.

File: MessagesBundle_en_US.properties

```
greeting=Hello
farewell=Goodbye
```

File: MessagesBundle_fr_FR.properties

```
greeting=Bonjour
farewell=Au revoir
```

Java Code:

```
import java.util.*;

public class ResourceBundleExample {
    public static void main(String[] args) {
        Locale fr = new Locale("fr", "FR");
        ResourceBundle bundle = ResourceBundle.getBundle("MessagesBundle", fr);

        System.out.println(bundle.getString("greeting"));
        System.out.println(bundle.getString("farewell"));
    }
}
```

NB:

- **Exception Handling** ensures programs handle runtime errors gracefully.

- **Key concepts**: try/catch/finally, try-with-resources, multi-catch, and custom exceptions.
- **Internationalization (i18n)** allows programs to adapt to multiple languages and cultures using Locale, NumberFormat, DateFormat, and ResourceBundle.

6.4. Code Examples Recap

Example 1: Try-Catch with Finally

```
try {
   int x = 10 / 0;
} catch (ArithmeticException e) {
   System.out.println("Division by zero.");
} finally {
   System.out.println("Cleanup code executed.");
}
```

Example 2: Try-with-Resources

```
try (BufferedReader br = new BufferedReader(new FileReader("data.txt"))) {
   System.out.println(br.readLine());
} catch (IOException e) {
   System.out.println("File error.");
}
```

Example 3: Internationalization with ResourceBundle

```
Locale fr = new Locale("fr", "FR");
ResourceBundle bundle = ResourceBundle.getBundle("MessagesBundle", fr);
System.out.println(bundle.getString("greeting"));
```

6.5. Practice Exercise

Task:

1. Create a program that:

- Reads a number from the user.
- Throws a custom exception if the number is negative.
- Uses a try-catch-finally block to handle exceptions.
- Prints the number formatted as currency in both the US and France locales.
- Prints today’s date in French format.

CHAPTER 7: Collections and Generics

7.1 Introduction

In Java, data structures are managed primarily through the **Collections Framework**, which provides classes and interfaces for storing and manipulating groups of objects. Collections eliminate the need to manually manage arrays with fixed sizes, giving flexibility, efficiency, and type safety when working with groups of data.

Alongside collections, **Generics** allow us to create type-safe code. With generics, we can specify the type of data a collection will hold, ensuring compile-time checks and reducing runtime errors. For example, a List<String> guarantees that only String objects can be added.

This chapter introduces the most common collection types (List, Set, Map, and Deque), operations (add, remove, update, retrieve, sort), as well as **Streams**, which provide a functional way to process collections.

7.2 The Collections Framework

The **Java Collections Framework (JCF)** provides interfaces, implementations, and algorithms for handling collections. It includes:

- **Interfaces** (blueprints, e.g., List, Set, Map)
- **Implementations** (concrete classes, e.g., ArrayList, HashSet, HashMap)
- **Algorithms** (utility methods, e.g., Collections.sort())

Main Collection Interfaces

1. **Collection** – root interface for lists, sets, and queues.
2. **List** – ordered collection allowing duplicates.
3. **Set** – unordered collection, no duplicates.
4. **Queue / Deque** – collections for processing elements in a queue-like structure.
5. **Map<K, V>** – key-value pairs, keys unique.

7.3 Java Arrays vs Collections

- **Arrays**: Fixed in size, less flexible, no built-in utility methods.
- **Collections**: Dynamically grow/shrink, come with rich APIs for searching, sorting, filtering, etc.

Example:

```
String[] arr = new String[3];
arr[0] = "Alice";
arr[1] = "Bob";
arr[2] = "Charlie";  // fixed size

List<String> list = new ArrayList<>();
list.add("Alice");
list.add("Bob");
list.add("Charlie");
list.add("Diana");  // dynamically grows
```

7.4 Working with Lists

Characteristics

- Ordered
- Allow duplicates
- Access by index

Implementations

- **ArrayList** – fast random access, slower insert/remove in middle.
- **LinkedList** – fast insert/remove, slower random access.

Example: ArrayList

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
```

```
        System.out.println("List: " + names);

        names.remove("Bob");  // Remove element
        names.set(1, "David"); // Update element
        System.out.println("Updated List: " + names);
        Collections.sort(names); // Sort alphabetically
        System.out.println("Sorted List: " + names);
    }
}
```

7.5 Working with Sets

Characteristics of Sets

1. Unordered

   A Set does not guarantee that elements will be stored in the order they were added.

   For example, if you insert [3, 1, 2] into a HashSet, you might get [1, 3, 2] when iterating.

   This makes sets different from lists, which preserve insertion order (e.g., ArrayList).

2. No Duplicates

   A Set ensures that duplicate elements are not allowed.

   If you try to insert the same element again, it will simply be ignored.

   This property makes sets ideal for situations where uniqueness matters (like storing user IDs, email addresses, or product codes).

3. Great for Membership Checking

   Sets provide very efficient contains() operations.

   Checking if an element exists in a HashSet is generally O(1) on average, compared to O(n) for lists.

   This makes them excellent for fast lookups, filtering, or implementing algorithms where quick membership tests are required.

Implementations

- **HashSet** – fast, unordered.

- **LinkedHashSet** – maintains insertion order.
- **TreeSet** – sorted order.

Example: HashSet

```
import java.util.*;


public class SetExample {
   public static void main(String[] args) {
      Set<String> set = new HashSet<>();
      set.add("Apple");
      set.add("Banana");
      set.add("Orange");
      set.add("Apple"); // Duplicate ignored


      System.out.println("Set: " + set);
   }
}
```

7.6 Working with Maps

Characteristics

- Key-value pairs
- Keys must be unique
- Values can duplicate

Implementations of Set in Java

Java provides different implementations of the Set interface depending on the required behavior:

1. HashSet – *Fast, Unordered*

- Backed by a **hash table**.
- Provides constant-time performance (O(1) average) for add, remove, and contains operations.
- Does not guarantee any particular order of elements.
- **Use case:** When you just need uniqueness and performance is critical, but order doesn't matter.

Example:

```
import java.util.HashSet;


public class Main {
   public static void main(String[] args) {
      HashSet<String> set = new HashSet<>();
      set.add("Apple");
      set.add("Banana");
      set.add("Orange");
      set.add("Apple"); // Duplicate ignored


      System.out.println(set); // Output could be [Orange, Banana, Apple]
   }
}
```

2. LinkedHashSet – *Maintains Insertion Order*

- Inherits from HashSet but also maintains a **linked list** running through its elements.
- Preserves the order in which elements were inserted.
- Slightly slower than HashSet because of the extra ordering overhead.
- **Use case:** When you need uniqueness **and** want elements to appear in insertion order.

Example:

```
import java.util.LinkedHashSet;


public class Main {
   public static void main(String[] args) {
      LinkedHashSet<String> set = new LinkedHashSet<>();
      set.add("Dog");
      set.add("Cat");
      set.add("Elephant");
```

```
        System.out.println(set); // Output: [Dog, Cat, Elephant]
    }
}
```

3. TreeSet – *Sorted Order*

- Backed by a **Red-Black Tree** (self-balancing binary search tree).
- Stores elements in **natural order** (e.g., numbers ascending, strings alphabetically) or using a **custom comparator**.
- Operations like add, remove, and contains take O(log n) time.
- **Use case:** When you need uniqueness with elements automatically sorted.

Example:

```
import java.util.TreeSet;


public class Main {
    public static void main(String[] args) {
        TreeSet<Integer> set = new TreeSet<>();
        set.add(30);
        set.add(10);
        set.add(20);


        System.out.println(set); // Output: [10, 20, 30]
    }
}
```

7.7 Working with Deques

Characteristics of Deques

1. **Double-Ended Queue**

- A **Deque** (pronounced “deck”) stands for **Double-Ended Queue**.
- Unlike a standard queue where elements are added at the **rear** and removed from the **front**, a deque allows insertion and removal from **both ends**.
- This makes it more flexible than regular queues.

2. **Add/Remove at Both Ends**
   - Deques provide methods to add elements at the **head (front)** or the **tail (end)**.
   - Similarly, elements can be removed from either side.
   - Example methods:
     - addFirst(), addLast()
     - removeFirst(), removeLast()
     - peekFirst(), peekLast()
3. **Supports Stack and Queue Operations**
   - A deque can act as both:
     - **Queue (FIFO – First In, First Out):** Insert at rear, remove from front.
     - **Stack (LIFO – Last In, First Out):** Insert at front, remove from front.
   - This dual functionality makes deques very versatile.

Common Implementations in Java

In Java, the Deque interface is part of the java.util package and is usually implemented by:

1. **ArrayDeque**
   - Resizable array implementation of a deque.
   - Faster than LinkedList for stack and queue operations.
   - Does not allow null elements.
   - Good for most use cases where a deque is needed.
2. **LinkedList**
   - Implements Deque using a doubly linked list.
   - Allows null elements (unlike ArrayDeque).
   - Better for scenarios where frequent insertions/removals occur in the middle of the list.

Example: Using Deque as a Queue (FIFO)

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<String> queue = new ArrayDeque<>();

        // Adding elements at the rear
        queue.addLast("Alice");
        queue.addLast("Bob");
        queue.addLast("Charlie");

        // Removing from the front (FIFO behavior)
        System.out.println(queue.removeFirst()); // Alice
        System.out.println(queue.removeFirst()); // Bob
    }
}
```

Example: Using Deque as a Stack (LIFO)

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<Integer> stack = new ArrayDeque<>();

        // Push elements (add at the front)
        stack.addFirst(10);
```

```
        stack.addFirst(20);
        stack.addFirst(30);

        // Pop elements (remove from the front)
        System.out.println(stack.removeFirst()); // 30
        System.out.println(stack.removeFirst()); // 20
    }
}
```

7.8 Generics

Generics in Java

- Generics allow **classes, interfaces, and methods** to operate on **types specified at compile-time** instead of being restricted to a single type (like Object).
- Introduced in **Java 5**, generics provide **compile-time type checking**, making code more reliable and reusable.
- Without generics, developers would use Object references, which required **type casting** and were prone to **runtime errors**.

Example (Without Generics):

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList list = new ArrayList(); // raw type
        list.add("Hello");
        list.add(123); // Allowed, but risky!

        String str = (String) list.get(0); // Casting required
        String num = (String) list.get(1); // Causes ClassCastException at runtime
    }
}
```

Example (With Generics):

```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
      ArrayList<String> list = new ArrayList<>(); // type-safe
      list.add("Hello");
      // list.add(123); // Compile-time error

      String str = list.get(0); // No casting needed
      System.out.println(str);
   }
}
```

7.9 Streams API

Introduced in Java 8, **Streams** process collections in a functional style. Streams allow operations like filter, map, and reduce.

Example: Stream Filtering

```
import java.util.*;
import java.util.stream.*;

public class StreamExample {
   public static void main(String[] args) {
      List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

      List<String> filtered = names.stream()
                              .filter(n -> n.startsWith("A"))
                              .collect(Collectors.toList());

      System.out.println("Names starting with A: " + filtered);
```

```
    }
}
```

7.9 Practice Exercise

Question:
Create a program that does the following:

1. Stores a list of student names and scores using a Map<String, Integer>.
2. Prints the map.
3. Updates one student's score.
4. Prints all students who scored above 80 using **Streams**.

CHAPTER 8: Concurrency in Java

8.1 Introduction to Concurrency

Concurrency refers to the ability of a program to execute multiple tasks seemingly at the same time. In Java, concurrency is achieved through **multithreading**, where different parts of a program (threads) execute independently but share the same resources such as memory and CPU.

Concurrency is not the same as parallelism:

- **Concurrency** is about dealing with multiple tasks at once. Tasks may not necessarily run simultaneously, but they make progress by time-sharing.
- **Parallelism** is when tasks are executed literally at the same time, often on multiple CPU cores.

Why concurrency?

- To perform **time-consuming tasks** (like I/O or computation) without freezing the program.
- To improve **performance and responsiveness** in applications (especially GUI applications).
- To utilize **multi-core processors** effectively.

8.2 Processes vs. Threads

- A **process** is an independent unit of execution with its own memory space. Starting a new process is expensive.
- A **thread** is a lightweight unit of execution within a process. Threads share the same memory space of the process, making them efficient for communication and data sharing.

Characteristics of Threads:

- Share resources (heap memory, files).
- Independent execution paths.
- Less overhead compared to processes.

8.3 Creating Threads in Java

There are two main ways to create threads in Java:

9.3.1 Extending the Thread Class

class MyThread extends Thread {

```
    @Override
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts the thread, calling run() internally
    }
}
```

run() contains the logic executed by the thread.

start() creates a new thread and calls run() indirectly. If you call run() directly, it executes in the main thread.

Implementing the Runnable Interface

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Runnable thread running: " + Thread.currentThread().getName());
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
```

```
    }
}
```

Using Runnable is preferred since Java allows multiple interfaces but only single inheritance.

Using Lambda Expressions

```
public class LambdaThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("Thread via Lambda: " + Thread.currentThread().getName());
        });
        t1.start();
    }
}
```

Lambdas provide a cleaner way to create threads.

8.4 Thread Lifecycle

A thread in Java goes through these states:

1. **New** – Created but not yet started (new Thread()).
2. **Runnable** – Ready to run after start(), waiting for CPU time.
3. **Running** – Currently executing.
4. **Blocked/Waiting** – Waiting for a resource or notification.
5. **Terminated** – Execution completed.

Java provides methods like:

- sleep(ms): Pause execution.
- join(): Wait for another thread to finish.
- yield(): Hint to scheduler to switch threads.

8.5 Executor Framework

Creating and managing many threads manually is error-prone. Java introduced the **Executor framework** in java.util.concurrent to simplify this.

Example with ExecutorService

```
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorExample {

   public static void main(String[] args) {

      ExecutorService executor = Executors.newFixedThreadPool(2);

      Runnable task1 = () -> System.out.println("Task 1 by " + Thread.currentThread().getName());

      Runnable task2 = () -> System.out.println("Task 2 by " + Thread.currentThread().getName());

      executor.submit(task1);

      executor.submit(task2);


      executor.shutdown(); // Graceful shutdown

   }

}
```

- Executors.newFixedThreadPool(n) creates a thread pool of n threads.
- Tasks are queued and executed by available threads.
- shutdown() ends executor service gracefully.

8.6 Callable and Future

Unlike Runnable, which does not return results, Callable allows a task to return a result or throw an exception.

```
import java.util.concurrent.*;


public class CallableExample {

   public static void main(String[] args) throws Exception {

      ExecutorService executor = Executors.newSingleThreadExecutor();
```

```
        Callable<Integer> task = () -> {
            return 10 * 2;
        };

        Future<Integer> result = executor.submit(task);
        System.out.println("Result: " + result.get()); // Blocks until result is available

        executor.shutdown();
    }
}
```

- Future is used to retrieve results of asynchronous computation.

8.7 Synchronization

Threads share data, which can cause race conditions if multiple threads modify shared variables simultaneously. Synchronization ensures only one thread accesses the resource at a time.

Synchronized Methods

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

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

    public int getCount() {
        return count;
    }
}
```

```
public class SyncExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });
        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

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

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Without synchronization, the result would be inconsistent.

8.8 Concurrent Collections

Java provides thread-safe collection classes such as:

- ConcurrentHashMap
- CopyOnWriteArrayList
- BlockingQueue

These are optimized for multi-threaded environments.

8.9 Parallel Streams

Java 8 introduced **parallel streams** to process data in parallel.

```
import java.util.Arrays;
public class ParallelStreamExample {
    public static void main(String[] args) {
        Arrays.asList(1, 2, 3, 4, 5, 6)
            .parallelStream()
            .forEach(num -> System.out.println(num + " processed by " + Thread.currentThread().getName()));
    }
}
```

Parallel streams split the data into chunks and process them simultaneously on multiple threads.

Code Examples Recap

Example 1: Creating Threads

```
public class SimpleThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> System.out.println("Hello from thread!"));
        t1.start();
    }
}
```

Example 2: ExecutorService

```
ExecutorService executor = Executors.newFixedThreadPool(2);
executor.submit(() -> System.out.println("Task running"));
executor.shutdown();
```

Example 3: Synchronization

```
class Counter {
    private int value = 0;
    public synchronized void increment() { value++; }
    public int getValue() { return value; }
}
```

Practice Exercise

Exercise:
Write a Java program that:

1. Creates three threads that simulate bank account transactions (deposit, withdraw, and balance inquiry).
2. Use synchronization to prevent race conditions on the shared account balance.
3. Print the final account balance after all threads complete.

CHAPTER 9: Input and Output in Java

9.1 Introduction

Input and Output (I/O) in Java allows programs to interact with the outside world. Input refers to receiving data from a source (e.g., keyboard, file, or network), while output refers to sending data to a destination (e.g., console, file, or network).
Java provides a comprehensive I/O API under the **java.io**, **java.nio**, and **java.util** packages, enabling reading, writing, serialization, and advanced file system handling.

This chapter explores I/O in detail, starting with basic streams, then moving to readers/writers, object serialization, the modern **NIO.2 API**, and practical applications.

9.2 Streams in Java

At the core of Java I/O is the **stream abstraction**. A stream represents a sequence of data elements (bytes or characters).

- **Input Stream**: Reads data from a source into a program.
- **Output Stream**: Writes data from a program to a destination.

Byte Streams

Byte streams handle **raw binary data** (e.g., images, audio, or any binary files). They are subclasses of:

- **InputStream** (abstract class for reading bytes)
- **OutputStream** (abstract class for writing bytes)

Examples:

- FileInputStream – Reads bytes from a file.
- FileOutputStream – Writes bytes to a file.

```
import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class ByteStreamExample {

   public static void main(String[] args) {

      try (FileInputStream fis = new FileInputStream("input.txt");

          FileOutputStream fos = new FileOutputStream("output.txt")) {
```

```
            int data;
            while ((data = fis.read()) != -1) {
                fos.write(data);  // Copy data from input to output
            }
            System.out.println("File copied successfully!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Character Streams

Character streams handle **textual data** (Unicode characters). They are subclasses of:

- **Reader** (abstract class for reading characters)
- **Writer** (abstract class for writing characters)

Examples:

- FileReader – Reads characters from a file.
- FileWriter – Writes characters to a file.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CharStreamExample {
    public static void main(String[] args) {
        try (FileReader reader = new FileReader("data.txt");
             FileWriter writer = new FileWriter("copy.txt")) {
            int c;
            while ((c = reader.read()) != -1) {
                writer.write(c);
```

```
        }
        System.out.println("Text file copied successfully!");
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

9.3 Buffered Streams

Buffered streams wrap around byte/character streams to improve efficiency by minimizing physical I/O operations.

- BufferedInputStream, BufferedOutputStream
- BufferedReader, BufferedWriter

Example with BufferedReader and BufferedWriter:

```
import java.io.*;
public class BufferedExample {
  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new FileReader("source.txt"));
         BufferedWriter bw = new BufferedWriter(new FileWriter("destination.txt"))) {
        String line;
        while ((line = br.readLine()) != null) {
            bw.write(line);
            bw.newLine();
        }
        System.out.println("Buffered file copy complete!");
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
    }
}
```

9.4 Console I/O

Java provides **System.in**, **System.out**, and **System.err** for console interaction.

- **System.in** – Standard input stream (keyboard).
- **System.out** – Standard output stream (console).
- **System.err** – Standard error stream (console for errors).

Example:

```
import java.util.Scanner;

public class ConsoleExample {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = sc.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

9.5 Object Serialization

Serialization allows saving an object's state to a file (or transferring it across a network) so it can be reconstructed later.

- **Serializable** interface must be implemented.
- Use ObjectOutputStream and ObjectInputStream.

```
import java.io.*;

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

```
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return name + " (" + age + ")";
    }
}

public class SerializationExample {
    public static void main(String[] args) {
        try {
            // Serialization
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("student.ser"));
            oos.writeObject(new Student("Alice", 21));
            oos.close();
            // Deserialization
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream("student.ser"));
            Student s = (Student) ois.readObject();
            ois.close();
            System.out.println("Deserialized Student: " + s);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

9.6 Java NIO.2 (New I/O)

Java NIO.2 introduced the **java.nio.file** package for modern file handling.

Key Classes

- **Path**: Represents file and directory paths.
- **Files**: Provides utility methods for file operations.
- **FileSystems**: Represents the file system.

Example: Reading and Writing with NIO.2

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;


public class NIOExample {
   public static void main(String[] args) {
      Path path = Paths.get("sample.txt");
      try {
         // Write to file
         Files.write(path, "Hello NIO.2!".getBytes());


         // Read from file
         List<String> lines = Files.readAllLines(path);
         for (String line : lines) {
            System.out.println(line);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

9.7 File Operations with NIO.2

NIO.2 makes it easy to perform advanced file operations:

- **Check existence**: Files.exists(path)
- **Copy files**: Files.copy(src, dest, StandardCopyOption.REPLACE_EXISTING)
- **Move files**: Files.move(src, dest)
- **Delete files**: Files.delete(path)
- **Get metadata**: Files.size(path), Files.getOwner(path)

9.8 Code Examples

Example 1: Copy a File Using NIO.2

```
import java.nio.file.*;


public class CopyFileNIO {
   public static void main(String[] args) {
      Path source = Paths.get("original.txt");
      Path target = Paths.get("backup.txt");


      try {
         Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
         System.out.println("File copied using NIO.2");
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Example 2: Count Words in a File

```
import java.io.*;
```

```
import java.util.*;
public class WordCount {
   public static void main(String[] args) {
      try (BufferedReader br = new BufferedReader(new FileReader("text.txt"))) {
         String line;
         int wordCount = 0;
         while ((line = br.readLine()) != null) {
            wordCount += line.split("\\s+").length;
         }
         System.out.println("Total words: " + wordCount);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Example 3: Write User Input to File

```
import java.io.*;
import java.util.Scanner;
public class WriteUserInput {
   public static void main(String[] args) {
      try (FileWriter fw = new FileWriter("userInput.txt");
          BufferedWriter bw = new BufferedWriter(fw)) {
         Scanner sc = new Scanner(System.in);
         System.out.print("Enter some text: ");
         String input = sc.nextLine();
         bw.write(input);
         System.out.println("Input saved to file.");
      } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

9.9 Practice Exercise

Exercise:
Write a Java program that:

1. Asks the user to input 5 lines of text.
2. Saves those lines into a file named notes.txt.
3. Reads the file and prints the content line by line to the console.

CHAPTER 10: JavaFX and SceneBuilder

10.1 Introduction

Graphical User Interfaces (GUIs) allow programs to interact with users visually instead of relying solely on text input and output. JavaFX is Java's modern toolkit for building GUIs. It provides a rich set of APIs for creating windows, buttons, forms, charts, animations, and more.

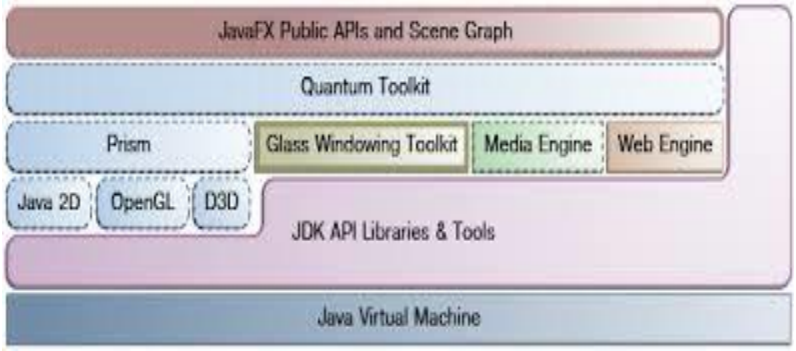
SceneBuilder, on the other hand, is a drag-and-drop design tool that allows developers to design JavaFX user interfaces visually without writing layout code manually. It generates **FXML** files (XML-based markup for JavaFX UIs), which can then be loaded and controlled from Java code.

10.2 Core Concepts of JavaFX

JavaFX Architecture

A JavaFX application consists of three main components:

1. **Stage** – The top-level container, similar to a window.
2. **Scene** – Represents the content inside the stage. It acts as a container for UI components (nodes).
3. **Nodes** – The individual UI components (e.g., buttons, text fields, labels, shapes).



The relationship looks like this:

Stage → Scene → Nodes

The Stage

The **Stage** is the main window of a JavaFX application. Every JavaFX application has at least one stage, provided by the runtime environment.

Example:

```
import javafx.application.Application;

import javafx.stage.Stage;

public class HelloStage extends Application {

   @Override

   public void start(Stage primaryStage) {

      primaryStage.setTitle("My First JavaFX App");

      primaryStage.show();

   }


   public static void main(String[] args) {

      launch(args);

   }

}
```

The Scene

The **Scene** contains all visual content inside a stage. You can think of it as the “canvas” for your GUI. A scene can contain multiple nodes arranged in layouts.

Example:

```
import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.stage.Stage;


public class HelloScene extends Application {

   @Override
```

```
    public void start(Stage primaryStage) {
        Button btn = new Button("Click Me");
        Scene scene = new Scene(btn, 300, 200);
        primaryStage.setTitle("Scene Example");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

Nodes

Nodes are UI elements such as buttons, labels, text fields, sliders, and shapes. Each node can be placed inside a scene.

Some commonly used nodes:

- **Label** – Displays text.
- **TextField** – Accepts user input.
- **Button** – Triggers actions.
- **CheckBox / RadioButton** – For selection options.
- **Slider / ProgressBar** – For ranges and progress tracking.

10.3 Layouts in JavaFX

Nodes must be arranged in a meaningful way. JavaFX provides **layout panes** for organizing UI elements.

Layout Types

1. **HBox** – Horizontal arrangement of nodes.
2. **VBox** – Vertical arrangement of nodes.
3. **BorderPane** – Divides the window into top, bottom, left, right, and center regions.
4. **GridPane** – Arranges elements in a grid of rows and columns.

5. **StackPane** – Stacks elements on top of each other.

Example (VBox Layout):

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class VBoxExample extends Application {
    @Override
    public void start(Stage stage) {
        Button b1 = new Button("Button 1");
        Button b2 = new Button("Button 2");
        VBox vbox = new VBox(10, b1, b2);

        Scene scene = new Scene(vbox, 200, 150);
        stage.setTitle("VBox Example");
        stage.setScene(scene);
        stage.show();
    }

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

10.4 Event Handling in JavaFX

JavaFX applications are event-driven. Events are generated when users interact with the UI (e.g., button clicks, mouse movements).

Handling Events

Events are handled by registering an event handler with a source node.

Example:

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;
public class EventExample extends Application {
   @Override
   public void start(Stage stage) {
      Button btn = new Button("Click Me");
      btn.setOnAction(e -> System.out.println("Button clicked!"));
      Scene scene = new Scene(btn, 200, 100);
      stage.setTitle("Event Handling");
      stage.setScene(scene);
      stage.show();
   }
   public static void main(String[] args) {
      launch(args);
   }
}
```

10.5 Styling with CSS

JavaFX allows styling of UIs using **Cascading Style Sheets (CSS)**, similar to web development.

Example (style.css):

```
.button {
   -fx-background-color: lightblue;
   -fx-font-size: 16px;
   -fx-text-fill: navy;
```

}

Example (Java):

Scene scene = new Scene(vbox, 300, 200);

scene.getStylesheets().add("style.css");

10.6 SceneBuilder and FXML

SceneBuilder is a **WYSIWYG (What You See Is What You Get)** tool for designing JavaFX GUIs visually. Instead of writing code for layouts manually, developers drag and drop UI elements. SceneBuilder generates an **FXML file**, which can be loaded into the JavaFX application.

FXML Basics

FXML is an XML format that describes JavaFX GUIs declaratively.

Example (sample.fxml):

<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>

<?import javafx.scene.layout.VBox?>

<VBox xmlns:fx="http://javafx.com/fxml" spacing="10">

  <Button text="Click Me" fx:id="myButton" />

</VBox>

Java code to load FXML:

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Scene;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

```
public class FXMLExample extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        Scene scene = new Scene(root, 300, 200);
        stage.setScene(scene);
        stage.setTitle("FXML Example");
        stage.show();
    }

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

10.7 JavaFX Controls and Events in SceneBuilder

- **Buttons** → Click actions.
- **Labels** → Display text.
- **TextFields** → Accept input.
- **TableView** → Display data in table format.

With SceneBuilder, you can:

1. Drag and drop controls.
2. Set properties (text, size, color).
3. Assign event handlers (methods in a controller class).

10.8 JavaFX and Databases

JavaFX can be combined with JDBC for database-driven GUIs. For example, a form built with SceneBuilder can collect user input and store it in a database.

Code Examples

Example 1: Basic JavaFX App with Layout

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloJavaFX extends Application {
    @Override
    public void start(Stage stage) {
        Label label = new Label("Hello, JavaFX!");
        StackPane root = new StackPane(label);
        Scene scene = new Scene(root, 300, 200);
        stage.setTitle("First JavaFX App");
        stage.setScene(scene);
        stage.show();
    }

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

Example 2: Event Handling

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.VBox;
```

```
import javafx.stage.Stage;

public class ButtonEvent extends Application {
   @Override
   public void start(Stage stage) {
      Button btn = new Button("Click Me");
      btn.setOnAction(e -> btn.setText("You clicked me!"));

      VBox vbox = new VBox(btn);
      Scene scene = new Scene(vbox, 200, 150);

      stage.setTitle("Event Example");
      stage.setScene(scene);
      stage.show();
   }

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

Example 3: FXML and Controller

FXML file (sample.fxml):

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.VBox?>

<VBox xmlns:fx="http://javafx.com/fxml" fx:controller="Controller" spacing="10">
```

```
    <Button text="Click Me" onAction="#handleClick"/>
</VBox>
```

Controller (Controller.java):

```
public class Controller {
    public void handleClick() {
        System.out.println("Button clicked from FXML!");
    }
}
```

Main Application:

```
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class FXMLApp extends Application {
    @Override
    public void start(Stage stage) throws Exception {
        VBox root = FXMLLoader.load(getClass().getResource("sample.fxml"));
        Scene scene = new Scene(root, 300, 200);
        stage.setScene(scene);
        stage.setTitle("FXML Demo");
        stage.show();
    }

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

10.9 Practice Exercise

Exercise:
Create a JavaFX application with the following features:

1. A **TextField** for the user to enter their name.
2. A **Button** labeled “Submit”.
3. When the button is clicked, display the message:
   "Hello, <name>!" in a **Label** below the button.

Hint: Use VBox for layout and event handling for the button click.

Chapter 11: Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is an API (Application Programming Interface) in Java that allows developers to connect to relational databases, execute queries, and manipulate data using SQL.

It acts as a bridge between Java applications and databases like MySQL, PostgreSQL, Oracle, SQL Server, and others.

11.1 Introduction

JDBC is a standard Java API that provides classes and interfaces for:

- Connecting to a database
- Sending SQL queries and updates
- Processing the results

It simplifies database programming by making database interaction platform-independent. Instead of writing vendor-specific code, developers use JDBC's standardized API.

JDBC Architecture

The JDBC architecture consists of two main layers:

1. **JDBC API** – Defines classes & interfaces for applications to interact with databases.
2. **JDBC Driver Manager & Drivers** – Translates JDBC calls into database-specific calls.

11.2 JDBC Drivers

To connect Java with a database, a **JDBC driver** is needed. The driver translates Java commands into database-specific operations.

Types of JDBC Drivers

1. **Type 1: JDBC-ODBC Bridge Driver**
    - Converts JDBC calls into ODBC calls.
    - Deprecated, not recommended.
2. **Type 2: Native-API Driver**
    - Converts JDBC calls into native database APIs (e.g., Oracle Call Interface).
    - Requires native client libraries.

3. **Type 3: Network Protocol Driver**
    - Uses middleware server to translate JDBC calls to database protocol.
4. **Type 4: Thin Driver (Pure Java)**
    - Directly converts JDBC calls to database protocol.
    - Most commonly used today (e.g., MySQL Connector/J).

11.3 Steps to Connect Java to a Database Using JDBC

Every JDBC program follows these steps:

1. **Load the JDBC Driver**
2. Class.forName("com.mysql.cj.jdbc.Driver");

(In modern Java, not always required since Service Provider Mechanism loads automatically).

3. **Establish Connection**
4. Connection con = DriverManager.getConnection(
5. "jdbc:mysql://localhost:3306/mydb", "username", "password");
6. **Create a Statement**
7. Statement stmt = con.createStatement();
8. **Execute Query**
9. ResultSet rs = stmt.executeQuery("SELECT * FROM students");
10. **Process Results**
11. while(rs.next()) {
12. System.out.println(rs.getInt(1) + " " + rs.getString(2));
13. }
14. **Close Connection**
15. con.close();

11.4 JDBC Classes and Interfaces

- **DriverManager** – Manages drivers and establishes connections.
- **Connection** – Represents a connection session with the database.

- **Statement** – Used for executing SQL queries.
- **PreparedStatement** – Precompiled SQL statements with parameters (prevents SQL injection).
- **CallableStatement** – Used to execute stored procedures.
- **ResultSet** – Represents query results.

11.5 Statement vs PreparedStatement vs CallableStatement

Feature	Statement	PreparedStatement	CallableStatement
SQL execution	General SQL	Precompiled SQL with parameters	Stored Procedures
Performance	Lower	Higher (compiled once)	Depends on DB
SQL Injection	Vulnerable	Secure	Secure

11.6 Transactions in JDBC

Transactions ensure that a set of SQL statements execute as a single unit.

- **Start Transaction** → Automatically starts with first SQL statement.
- **Commit** → Saves all changes.
- **Rollback** → Reverts changes in case of failure.

Example:

```
con.setAutoCommit(false); // start transaction
try {
   stmt.executeUpdate("INSERT INTO students VALUES (1, 'Alice')");
   stmt.executeUpdate("INSERT INTO students VALUES (2, 'Bob')");
   con.commit();  // commit transaction
} catch(Exception e) {
   con.rollback();  // rollback on error
}
```

11.7 Batch Processing

Batch processing allows execution of multiple queries in one go for better performance.

```
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO students VALUES (3, 'Chris')");
stmt.addBatch("INSERT INTO students VALUES (4, 'Diana')");
stmt.executeBatch();
```

11.8 ResultSet Types

- **TYPE_FORWARD_ONLY** – Default, moves forward only.
- **TYPE_SCROLL_INSENSITIVE** – Can scroll both directions, not sensitive to DB changes.
- **TYPE_SCROLL_SENSITIVE** – Can scroll both directions, sensitive to DB changes.

11.9 Handling Exceptions

Most JDBC operations throw **SQLException**.
You can catch details using:

```
catch (SQLException e) {
   System.out.println("Error Code: " + e.getErrorCode());
   System.out.println("SQL State: " + e.getSQLState());
   e.printStackTrace();
}
```

Code Examples

Example 1: Simple Database Connection

```
import java.sql.*;

public class JDBCExample {
   public static void main(String[] args) {
      try {
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb", "root", "password");
         System.out.println("Connected to database!");
```

```
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example 2: Using PreparedStatement

```
import java.sql.*;

public class PreparedStmtExample {
    public static void main(String[] args) {
        try {
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/mydb", "root", "password");

            String query = "INSERT INTO students (id, name) VALUES (?, ?)";
            PreparedStatement pstmt = con.prepareStatement(query);
            pstmt.setInt(1, 5);
            pstmt.setString(2, "Eve");
            pstmt.executeUpdate();

            System.out.println("Data inserted successfully!");
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example 3: Retrieving Data

```
import java.sql.*;


public class SelectExample {
   public static void main(String[] args) {
      try {
         Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/mydb", "root", "password");


         Statement stmt = con.createStatement();
         ResultSet rs = stmt.executeQuery("SELECT * FROM students");


         while(rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            System.out.println(id + " - " + name);
         }


         con.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Practice Exercise

Write a JDBC program that does the following:

1. Connects to a database school.

2. Creates a table teachers (id INT, name VARCHAR(50), subject VARCHAR(30)).
3. Inserts at least **three teachers** using PreparedStatement.
4. Retrieves and prints all teachers from the table.

APPLICATION

JavaFX GUI program created with SceneBuilder and connected to a database using JDBC.

Student Management System where you can:

- Add students (Name, Age, Course).
- View student records in a **TableView**.
- Delete a student.

Step 1: Database Setup

We'll use **SQLite** (lightweight, no server needed).

```
-- Create database file: students.db
CREATE TABLE students (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   name TEXT NOT NULL,
   age INTEGER NOT NULL,
   course TEXT NOT NULL
);
```

Save this as students.db in your project folder.

Step 2: Project Structure

```
StudentManagement/
 ├─ src/
 |   ├─ application/
```

```
|   |   ├─ Main.java
|   |   ├─ DBUtil.java
|   |   ├─ Student.java
|   |   ├─ StudentDAO.java
|   |   ├─ StudentController.java
|   └─ resources/
|       └─ student.fxml   (designed using SceneBuilder)
└─ students.db
```

Step 3: Student Model

```
package application;

public class Student {
    private int id;
    private String name;
    private int age;
    private String course;

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

    // Getters & Setters
    public int getId() { return id; }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String getCourse() { return course; }
```

```
}
```

Step 4: Database Utility

```
package application;

import java.sql.*;

public class DBUtil {
    private static final String URL = "jdbc:sqlite:students.db";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL);
    }
}
```

Step 5: Data Access Object (DAO)

```
package application;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import java.sql.*;

public class StudentDAO {
    public static void addStudent(String name, int age, String course) throws SQLException {
        String sql = "INSERT INTO students(name, age, course) VALUES(?, ?, ?)";
        try (Connection conn = DBUtil.getConnection(); PreparedStatement stmt =
conn.prepareStatement(sql)) {
            stmt.setString(1, name);
            stmt.setInt(2, age);
            stmt.setString(3, course);
            stmt.executeUpdate();
        }
    }
```

```
    public static ObservableList<Student> getStudents() throws SQLException {
        ObservableList<Student> list = FXCollections.observableArrayList();
        String sql = "SELECT * FROM students";
        try (Connection conn = DBUtil.getConnection(); Statement stmt = conn.createStatement(); ResultSet
rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                list.add(new Student(
                        rs.getInt("id"),
                        rs.getString("name"),
                        rs.getInt("age"),
                        rs.getString("course")
                ));
            }
        }
        return list;
    }

    public static void deleteStudent(int id) throws SQLException {
        String sql = "DELETE FROM students WHERE id=?";
        try (Connection conn = DBUtil.getConnection(); PreparedStatement stmt =
conn.prepareStatement(sql)) {
            stmt.setInt(1, id);
            stmt.executeUpdate();
        }
    }
}
```

Step 6: SceneBuilder FXML (student.fxml)

In **SceneBuilder**, design a GUI:

VBox → GridPane (for inputs)

- TextField: txtName
- TextField: txtAge
- TextField: txtCourse
- Button: btnAdd

TableView (fx:id=tableView)

- TableColumn: colId
- TableColumn: colName
- TableColumn: colAge
- TableColumn: colCourse

Button: btnDelete

Save as **student.fxml**.

Step 7: Controller

```
package application;

import javafx.collections.ObservableList;

import javafx.fxml.FXML;

import javafx.scene.control.*;


public class StudentController {

   @FXML private TextField txtName;

   @FXML private TextField txtAge;

   @FXML private TextField txtCourse;

   @FXML private TableView<Student> tableView;

   @FXML private TableColumn<Student, Integer> colId;

   @FXML private TableColumn<Student, String> colName;

   @FXML private TableColumn<Student, Integer> colAge;

   @FXML private TableColumn<Student, String> colCourse;
```

```
    @FXML
    public void initialize() {
        colId.setCellValueFactory(cell -> new
javafx.beans.property.SimpleIntegerProperty(cell.getValue().getId()).asObject());
        colName.setCellValueFactory(cell -> new
javafx.beans.property.SimpleStringProperty(cell.getValue().getName()));
        colAge.setCellValueFactory(cell -> new
javafx.beans.property.SimpleIntegerProperty(cell.getValue().getAge()).asObject());
        colCourse.setCellValueFactory(cell -> new
javafx.beans.property.SimpleStringProperty(cell.getValue().getCourse()));

        loadStudents();
    }

    private void loadStudents() {
        try {
            ObservableList<Student> list = StudentDAO.getStudents();
            tableView.setItems(list);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @FXML
    private void addStudent() {
        try {
            String name = txtName.getText();
            int age = Integer.parseInt(txtAge.getText());
            String course = txtCourse.getText();
            StudentDAO.addStudent(name, age, course);
```

```
            loadStudents();
            txtName.clear(); txtAge.clear(); txtCourse.clear();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    @FXML
    private void deleteStudent() {
        Student selected = tableView.getSelectionModel().getSelectedItem();
        if (selected != null) {
            try {
                StudentDAO.deleteStudent(selected.getId());
                loadStudents();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Step 8: Main Class

```
package application;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;
public class Main extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
```

```
        FXMLLoader loader = new FXMLLoader(getClass().getResource("/student.fxml"));
        Scene scene = new Scene(loader.load());
        primaryStage.setTitle("Student Management System");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Features in this program:

- GUI designed in **SceneBuilder**.
- Database interaction using **JDBC**.
- Add, display, and delete student records.
- Uses **TableView** to show live data.

ASSIGNMENT

Extend this application with **update functionality** (editing student details) so that the system becomes **fully CRUD**?