MODULE 6 CORE JAVA

1. Introduction to Java

📖 **Theory:**

- **History of Java:**
  Java was developed by James Gosling at Sun Microsystems and released in 1995. It was initially called *Oak*. Later, it was renamed to *Java*, inspired by coffee from Java Island. It was designed to overcome the limitations of C++ and aimed at developing platform-independent applications, especially for networked environments.

- **Features of Java:**
  - *Platform Independent*: Code runs on any machine with JVM.
  - *Object-Oriented*: Everything in Java is based on objects and classes.
  - *Simple & Secure*: Easier than C++ and has built-in security features.
  - *Robust*: Handles errors with exception handling and garbage collection.
  - *Portable*: Java programs can move easily from one system to another.
  - *Multithreaded*: Supports multiple threads of execution.
  - *High Performance*: Bytecode is optimized for fast execution.

- **JVM, JRE, and JDK:**
  - *JVM (Java Virtual Machine)*: Converts bytecode into machine code and executes it.
  - *JRE (Java Runtime Environment)*: Contains JVM and Java class libraries. Needed to run Java programs.
  - *JDK (Java Development Kit)*: Contains JRE plus tools like the compiler (javac), debugger, etc., needed for development.

- **Setting up Java Environment and IDE:**

- Download and install the latest JDK.
- Set environment variables (`JAVA_HOME`, update `Path`).
- Choose an IDE: Eclipse, IntelliJ IDEA, or NetBeans.
- Test setup using `java -version` and `javac -version` in terminal or CMD.

- **Java Program Structure:**
  Java programs are organized into:
  - *Packages*: Organize classes into namespaces.
  - *Classes*: Define data and behavior.
  - *Methods*: Contain the logic.
  - *Main method*: Entry point of Java application.

---

2. Data Types, Variables, and Operators

📖 **Theory:**

- **Primitive Data Types:**
  Java provides 8 built-in types: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.
- **Variable Declaration and Initialization:**
  Variables store data. Example: `int a = 10; float b = 2.5f;`
- **Operators in Java:**
  - *Arithmetic*: +, -, *, /, %
  - *Relational*: ==, !=, >, <, >=, <=
  - *Logical*: &&, ||, !
  - *Assignment*: =, +=, -=, *=, etc.

- *Unary*: +, -, ++, --
- *Bitwise*: &, |, ^, <<, >>

- **Type Conversion and Casting:**
  - *Implicit Casting*: Done automatically when converting a smaller type to a larger one. (`int to float`)
  - *Explicit Casting*: Manual casting using syntax. Example: `(int) 5.6`

---

3. Control Flow Statements

📖 **Theory:**

- **If-Else Statements**: Used for decision-making based on condition evaluation.
- **Switch-Case**: Efficient alternative to many if-else blocks, mostly used when comparing a variable against constant values.
- **Loops in Java**:
  - *For loop*: When the number of iterations is known.
  - *While loop*: When the condition is checked before execution.
  - *Do-while loop*: Executes at least once regardless of the condition.
- **Break and Continue**:
  - *Break*: Terminates the loop or switch.
  - *Continue*: Skips the current iteration and moves to the next.

---

4. Classes and Objects

📖 **Theory:**

- **Class**: A blueprint for creating objects. Contains fields (attributes) and methods (functions).
- **Object**: Instance of a class created using the `new` keyword.
- **Constructor**: Special method used to initialize objects. Has the same name as class.
- **Constructor Overloading**: Multiple constructors with different parameter lists.
- **this Keyword**: Refers to the current object's instance.

---

5. Methods in Java

📖 **Theory:**
Methods: Blocks of code that perform a specific task. Syntax:

java
CopyEdit
```
returnType methodName(parameters) { // code }
```

-
- **Parameters and Return Types**: Methods can accept parameters and return values, or be void (no return).
- **Method Overloading**: Same method name, different parameter types/number/order.
- **Static Methods and Variables**: Belong to the class, not the instance. Used with class name directly.

---

6. Object-Oriented Programming (OOPs) Concepts

Theory:

- **Basics of OOP**:

    - *Encapsulation*: Wrapping data and code into a single unit (class), using access modifiers to restrict access.
    - *Inheritance*: Mechanism for a class to acquire properties and methods of another class.
    - *Polymorphism*: Ability of one interface to be used for different underlying forms (method overloading/overriding).
    - *Abstraction*: Hiding complex implementation details and showing only the essential features.
- **Types of Inheritance**:
    - *Single Inheritance*: A class inherits from one superclass.
    - *Multilevel Inheritance*: A class is derived from a class that is also derived from another class.
    - *Hierarchical Inheritance*: Multiple classes inherit from one superclass.
- **Method Overriding and Dynamic Method Dispatch**:
    - Method overriding allows a subclass to provide a specific implementation of a method already defined in its parent class.
    - Dynamic method dispatch determines the method to invoke at runtime based on the object.

---

7. Constructors and Destructors

Theory:

- **Constructor Types**:
    - *Default Constructor*: No parameters.
    - *Parameterized Constructor*: Accepts parameters to initialize objects.
- **Copy Constructor**:
    - Java does not support copy constructors directly, but it can be emulated by copying values from one object to another manually.
- **Constructor Overloading**: Multiple constructors with different parameter lists in the same class.
- **Object Lifecycle and Garbage Collection**:
    - Java does not require destructors. The garbage collector automatically destroys unreferenced objects.

---

8. Arrays and Strings

Theory:

- **Arrays**:
  - *One-Dimensional Array*: Linear array.
  - *Multidimensional Array*: Arrays with more than one row/column, like matrix.
- **String Handling**:
  - *String Class*: Immutable, used for constant strings.
  - *StringBuffer*: Mutable and thread-safe.
  - *StringBuilder*: Mutable but not thread-safe.
- **Array of Objects**: Creating and managing multiple object instances in an array.
- **String Methods**:
  - `length()`, `charAt()`, `substring()`, `equals()`, `compareTo()`, etc.

---

9. Inheritance and Polymorphism

Theory:

- **Inheritance**:
  - Promotes code reuse and is fundamental for achieving polymorphism.
  - Helps in creating hierarchical classifications.
- **Method Overriding**: Redefining a superclass method in the subclass.
- **Dynamic Binding**: Runtime decision about method implementation (run-time polymorphism).
- **super Keyword**: Used to refer to the immediate parent class constructor, methods, or variables.
- **Method Hiding**: If a subclass defines a static method with the same signature as a static method in the superclass, the method is hidden—not overridden.

---

10. Interfaces and Abstract Classes

Theory:

- **Abstract Classes**:
  - Cannot be instantiated.
  - May contain abstract (no body) and concrete methods.
- **Interfaces**:
  - Define a contract of methods a class must implement.

    - Support multiple inheritance in Java (a class can implement multiple interfaces).
- **Implementing Multiple Interfaces**: A class can implement more than one interface, separating functionality across contracts.

11. Packages and Access Modifiers

Theory:

- **Java Packages**: Packages are used to group related classes and interfaces. Java provides built-in packages (e.g., java.util, java.io) and allows the creation of user-defined packages.
- **Access Modifiers**:
    - `private`: Accessible only within the class.
    - *default* (no modifier): Accessible within the same package.
    - `protected`: Accessible within the same package and subclasses.
    - `public`: Accessible from any other class.
- **Importing Packages and Classpath**:
    - Use `import` to include classes from packages.
    - `classpath` is the path where the Java compiler and JVM look for class files.

---

12. Exception Handling

Theory:

- **Types of Exceptions**:
    - *Checked Exceptions*: Known at compile time (e.g., IOException).
    - *Unchecked Exceptions*: Occur at runtime (e.g., ArithmeticException).
- **Exception Handling Keywords**:
    - `try`, `catch`, `finally`: Handle exceptions.
    - `throw`: Used to explicitly throw an exception.
    - `throws`: Declares exceptions that might be thrown by a method.
- **Custom Exceptions**: User-defined classes extending `Exception` class.

---

13. Multithreading

Theory:

- **Introduction to Threads**: Threads allow concurrent execution of two or more parts of a program.
- **Creating Threads**:
  - By extending `Thread` class.
  - By implementing `Runnable` interface.
- **Thread Life Cycle**: New, Runnable, Running, Blocked, Terminated.
- **Synchronization**: Controls access to shared resources using `synchronized` keyword.
- **Inter-thread Communication**: Using `wait()`, `notify()`, `notifyAll()` for coordination among threads.

---

14. File Handling

Theory:

- **File I/O in Java**: Managed via `java.io` package.
- **Classes**:
  - `FileReader` / `FileWriter`: Character-based streams.
  - `BufferedReader` / `BufferedWriter`: Provide buffering for efficient I/O.
- **Serialization and Deserialization**:
  - `ObjectOutputStream` and `ObjectInputStream` used to write/read objects.
  - Serializable objects must implement `Serializable` interface.

---

15. Collections Framework

Theory:

- **Overview**: Provides classes and interfaces for handling groups of objects.
- **Interfaces**:
  - `List`, `Set`, `Map`, `Queue`
- **Implementations**:
  - `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`
- **Iterators**: Used to iterate through collections (`Iterator`, `ListIterator`).

---

16. Java Input/Output (I/O)

Theory:

- **Streams**:
  - `InputStream`, `OutputStream`: Base classes for byte streams.
- **Reading/Writing Data**:
  - Use streams to read/write data from console, files, and other sources.
- **File I/O Operations**:
  - Reading from and writing to files using streams.

LAB EXERCISE ANSWERS

Lab Exercise 1

✅ Task 1: Install JDK and Set Up Environment Variables

1. **Download JDK** from the official Oracle website: https://www.oracle.com/java/technologies/javase-downloads.html

2. **Install JDK** and note the installation path (e.g., `C:\Program Files\Java\jdk-21`).

3. **Set Environment Variables**:

   - Go to System Properties → Environment Variables.

   - Add a new system variable:

     - `JAVA_HOME` = JDK install path

   - Edit the `Path` variable and add:

     - `%JAVA_HOME%\bin`

✅ Task 2: Write a "Hello World" Program

Use any basic text editor (Notepad, VS Code, etc.) and save the following code as `Hello.java`:

CODE :

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

✅ Task 3: Compile and Run the Program via Command Line

1. Open Command Prompt and navigate to the folder where `Hello.java` is saved.
2. Run the following commands:

```
javac Hello.java   // Compiles the Java program
java Hello         // Runs the compiled program
```

✅ Output:

CopyEdit

```
Hello, World!
```

Lab Exercise 2

✅ Task 1: Demonstrate the Use of Different Data Types

Create a Java program that shows how to declare and use various primitive data types:

```
public class DataTypesDemo {
    public static void main(String[] args) {
        int age = 25;
        float height = 5.9f;
        double weight = 70.5;
        char grade = 'A';
```

```
        boolean isStudent = true;

        System.out.println("Age: " + age);
        System.out.println("Height: " + height);
        System.out.println("Weight: " + weight);
        System.out.println("Grade: " + grade);
        System.out.println("Is student? " + isStudent);
    }
}
```

✅ Task 2: Basic Calculator Using Arithmetic and Relational Operators

A simple calculator program performing addition, subtraction, multiplication, division, and comparison:

```
public class Calculator {
    public static void main(String[] args) {
        int a = 10, b = 5;

        // Arithmetic Operations
        System.out.println("Addition: " + (a + b));
        System.out.println("Subtraction: " + (a - b));
        System.out.println("Multiplication: " + (a * b));
        System.out.println("Division: " + (a / b));

        // Relational Operation
        System.out.println("Are a and b equal? " + (a == b));
    }
}
```

✅ Task 3: Demonstrate Type Casting (Implicit and Explicit)

- **Implicit Casting** (widening conversion):

```
int i = 100;
```

```
double d = i;  // int automatically cast to double
System.out.println("Implicit Casting: " + d);
```

- **Explicit Casting** (narrowing conversion):

```
double x = 45.67;
int y = (int) x;  // double explicitly cast to int (decimal truncated)
System.out.println("Explicit Casting: " + y);
```

Lab Exercise 3

✅ Task 1: Check Even or Odd Using If-Else Statement

```
public class EvenOdd {
    public static void main(String[] args) {
        int num = 4;

        if (num % 2 == 0) {
            System.out.println(num + " is Even");
        } else {
            System.out.println(num + " is Odd");
        }
    }
}
```

✅ Task 2: Simple Menu-Driven Program Using Switch-Case

```
import java.util.Scanner;

public class MenuDriven {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Menu:\n1. Say Hello\n2. Say Goodbye\nEnter
choice:");
        int choice = sc.nextInt();
```

```
        switch (choice) {
            case 1:
                System.out.println("Hello!");
                break;
            case 2:
                System.out.println("Goodbye!");
                break;
            default:
                System.out.println("Invalid choice.");
        }
        sc.close();
    }
}
```

✅ **Task 3: Display Fibonacci Series Using a Loop**

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 10, a = 0, b = 1;

        System.out.print("Fibonacci Series: ");
        for (int i = 1; i <= n; i++) {
            System.out.print(a + " ");
            int sum = a + b;
            a = b;
            b = sum;
        }
    }
}
```

Lab Exercise 4

✅ **Task 1: Create a `Student` Class with Attributes and a Display Method**

```
public class Student {
    String name;
```

```
    int age;

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

    public static void main(String[] args) {
        Student s = new Student();
        s.name = "Ayush";
        s.age = 21;
        s.display();
    }
}
```

✅ Task 2: Constructor Overloading in the `Student` Class

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

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

    // Parameterized constructor
    Student(String name) {
        this.name = name;
        this.age = 0;
    }

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

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

    public static void main(String[] args) {
        Student s1 = new Student();
        Student s2 = new Student("Ayush");
        Student s3 = new Student("Rahul", 22);

        s1.display();
        s2.display();
        s3.display();
    }
}
```

✅ **Task 3: Encapsulation with Getters and Setters**

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

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }
```

```
    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }

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

    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Ayush");
        s.setAge(21);
        s.display();
    }
}
```

Lab Exercise 5

✅ Task 1: Find the Maximum of Three Numbers Using a Method

```
public class MaxOfThree {
    static int max(int a, int b, int c) {
        if (a >= b && a >= c)
            return a;
        else if (b >= a && b >= c)
            return b;
        else
            return c;
    }

    public static void main(String[] args) {
        System.out.println("Maximum is: " + max(10, 20, 15));
    }
}
```

✅ Task 2: Method Overloading for Different Data Types

```
public class MethodOverloading {
    int add(int a, int b) {
        return a + b;
    }

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

    public static void main(String[] args) {
        MethodOverloading obj = new MethodOverloading();
        System.out.println("Sum of ints: " + obj.add(5, 10));
        System.out.println("Sum of doubles: " + obj.add(5.5, 10.5));
    }
}
```

✅ Task 3: Static Variables and Methods

```
public class StaticDemo {
    static int count = 0;

    StaticDemo() {
        count++;  // Increment count each time constructor is called
    }

    static void displayCount() {
        System.out.println("Number of objects created: " + count);
    }

    public static void main(String[] args) {
        new StaticDemo();
        new StaticDemo();
        StaticDemo.displayCount();
    }
}
```

Lab Exercise 6

✅ Task 1: Demonstrate Single Inheritance

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

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

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

✅ Task 2: Demonstrate Multilevel Inheritance

```
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

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

```
    }
}

class BabyDog extends Dog {
    void weep() {
        System.out.println("Weeping...");
    }
}

public class MultiLevelInheritance {
    public static void main(String[] args) {
        BabyDog bd = new BabyDog();
        bd.eat();
        bd.bark();
        bd.weep();
    }
}
```

✅ **Task 3: Method Overriding and Polymorphism**

```
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 MethodOverridingDemo {
    public static void main(String[] args) {
        Animal a = new Dog();  // Reference of Animal, object of Dog
```

```
        a.sound();  // Calls overridden method in Dog class (runtime
polymorphism)
    }
}
```

Lab Exercise 7

✅ Task 1: Create and Initialize an Object Using Parameterized Constructor

```
class Car {
    String model;
    int year;

    // Parameterized constructor
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void display() {
        System.out.println("Model: " + model + ", Year: " + year);
    }

    public static void main(String[] args) {
        Car c = new Car("Honda", 2023);
        c.display();
    }
}
```

✅ Task 2: Demonstrate Constructor Overloading with Different Parameters

```
class Car {
    String model;
    int year;

    Car() {
```

```
        model = "Unknown";
        year = 0;
    }

    Car(String model) {
        this.model = model;
        year = 0;
    }

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void display() {
        System.out.println("Model: " + model + ", Year: " + year);
    }

    public static void main(String[] args) {
        Car c1 = new Car();
        Car c2 = new Car("Toyota");
        Car c3 = new Car("Ford", 2022);

        c1.display();
        c2.display();
        c3.display();
    }
}
```

Lab Exercise 8

✅ Task 1: Matrix Addition and Subtraction Using 2D Arrays

```
public class MatrixOperations {
    public static void main(String[] args) {
        int[][] matrixA = { {1, 2}, {3, 4} };
        int[][] matrixB = { {5, 6}, {7, 8} };

        int rows = matrixA.length;
```

```
        int cols = matrixA[0].length;

        int[][] sum = new int[rows][cols];
        int[][] diff = new int[rows][cols];

        // Matrix addition and subtraction
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                sum[i][j] = matrixA[i][j] + matrixB[i][j];
                diff[i][j] = matrixA[i][j] - matrixB[i][j];
            }
        }

        // Display results
        System.out.println("Sum:");
        printMatrix(sum);

        System.out.println("Difference:");
        printMatrix(diff);
    }

    static void printMatrix(int[][] matrix) {
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }
    }
}
```

✅ Task 2: Reverse a String and Check for Palindrome

```
public class StringReversePalindrome {
    public static void main(String[] args) {
        String str = "madam";
        String reversed = "";
```

```
        // Reverse string
        for (int i = str.length() - 1; i >= 0; i--) {
            reversed += str.charAt(i);
        }

        System.out.println("Original: " + str);
        System.out.println("Reversed: " + reversed);

        // Check palindrome
        if (str.equals(reversed)) {
            System.out.println("The string is a palindrome.");
        } else {
            System.out.println("The string is not a palindrome.");
        }
    }
}
```

✅ **Task 3: String Comparison Using equals() and compareTo()**

```
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "apple";
        String str2 = "banana";

        // equals() checks for exact content equality
        System.out.println("Using equals(): " + str1.equals(str2)); //
false

        // compareTo() returns:
        // 0 if both strings are equal,
        // a negative number if str1 < str2,
        // a positive number if str1 > str2
        System.out.println("Using compareTo(): " +
str1.compareTo(str2)); // negative value
    }
}
```

Lab Exercise 9

✅ Task 1: Demonstrate Inheritance Using `extends` Keyword

```
class Vehicle {
    void display() {
        System.out.println("This is a vehicle.");
    }
}

class Car extends Vehicle {
    void carDetails() {
        System.out.println("This is a car.");
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.display();
        myCar.carDetails();
    }
}
```

✅ Task 2: Runtime Polymorphism by Overriding Methods

```
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 RuntimePolymorphism {
    public static void main(String[] args) {
        Animal a = new Dog(); // Reference of Animal, object of Dog
        a.sound();            // Calls overridden method in Dog class
    }
}
```

✅ Task 3: Using `super` Keyword to Call Parent Constructor and Methods

```
class Person {
    String name;

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

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

class Student extends Person {
    int age;

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

    @Override
    void display() {
        super.display();  // Call parent method
        System.out.println("Age: " + age);
```

```
    }
}

public class SuperKeywordDemo {
    public static void main(String[] args) {
        Student s = new Student("Ayush", 21);
        s.display();
    }
}
```

Lab Exercise 10

✅ Task 1: Abstract Class and Implementing Its Methods in a Subclass

```
abstract class Shape {
    abstract void area();

    void display() {
        System.out.println("This is a shape.");
    }
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    void area() {
        System.out.println("Area of Circle: " + (3.14 * radius *
radius));
    }
}

public class AbstractClassDemo {
```

```
    public static void main(String[] args) {
        Circle c = new Circle(5);
        c.display();
        c.area();
    }
}
```

✅ **Task 2: Implement Multiple Interfaces in a Single Class**

```
interface Printable {
    void print();
}

interface Showable {
    void show();
}

class Demo implements Printable, Showable {
    public void print() {
        System.out.println("Printing...");
    }

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

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

✅ **Task 3: Interface for a Real-World Example — Payment Gateway**

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

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

class PaypalPayment implements PaymentGateway {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}

public class PaymentDemo {
    public static void main(String[] args) {
        PaymentGateway payment = new CreditCardPayment();
        payment.pay(1000);

        payment = new PaypalPayment();
        payment.pay(500);
    }
}
```

Lab Exercise 11

1. Create a User-Defined Package and Import It

Create package file: `mypackage/Hello.java`

```
package mypackage;

public class Hello {
    public void sayHello() {
```

```
        System.out.println("Hello from mypackage!");
    }
}
```

Import package in another program:

```
import mypackage.Hello;

public class TestPackage {
    public static void main(String[] args) {
        Hello h = new Hello();
        h.sayHello();
    }
}
```

---

2. Access Modifiers in Same and Different Packages

Class with different access modifiers:

```
package mypackage;

public class Demo {
    public int publicVar = 10;
    protected int protectedVar = 20;
    int defaultVar = 30;   // package-private
    private int privateVar = 40;

    public void display() {
        System.out.println("Public: " + publicVar);
        System.out.println("Protected: " + protectedVar);
        System.out.println("Default: " + defaultVar);
        System.out.println("Private: " + privateVar);
    }
}
```

Access from same package:

```
package mypackage;

public class SamePackageAccess {
    public static void main(String[] args) {
        Demo d = new Demo();
        System.out.println(d.publicVar);    // accessible
        System.out.println(d.protectedVar); // accessible
        System.out.println(d.defaultVar);   // accessible
        // System.out.println(d.privateVar); // NOT accessible
    }
}
```

Access from different package:

```
package otherpackage;

import mypackage.Demo;

public class DifferentPackageAccess {
    public static void main(String[] args) {
        Demo d = new Demo();
        System.out.println(d.publicVar);    // accessible
        // System.out.println(d.protectedVar); // NOT accessible
unless subclass
        // System.out.println(d.defaultVar);   // NOT accessible
        // System.out.println(d.privateVar);   // NOT accessible
    }
}
```

Lab Exercise 12

1. Exception Handling Using try-catch-finally

```
public class ExceptionHandlingDemo {
```

```
    public static void main(String[] args) {
        try {
            int division = 10 / 0; // causes ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception caught: " +
e.getMessage());
        } finally {
            System.out.println("This block always executes.");
        }
    }
}
```

2. Multiple Catch Blocks

```
public class MultipleCatchDemo {
    public static void main(String[] args) {
        try {
            int[] arr = new int[3];
            arr[5] = 10; // ArrayIndexOutOfBoundsException
            int division = 10 / 0; // ArithmeticException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index error: " +
e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " + e.getMessage());
        }
    }
}
```

3. Custom Exception Class

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

public class CustomExceptionDemo {
```

```
    static void checkAge(int age) throws CustomException {
        if (age < 18) {
            throw new CustomException("Age must be at least 18");
        } else {
            System.out.println("Age is valid");
        }
    }

    public static void main(String[] args) {
        try {
            checkAge(16);
        } catch (CustomException e) {
            System.out.println("Custom Exception caught: " +
e.getMessage());
        }
    }
}
```

Lab Exercise 13

1. Create and Run Multiple Threads Using Thread Class

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(getName() + " - Count: " + i);
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start();
        t2.start();
    }
```

```
}
```

2. Thread Synchronization Using `synchronized`

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

    public synchronized void increment() {
        count++;
        System.out.println(Thread.currentThread().getName() + " Count:
" + count);
    }
}

public class SyncDemo {
    public static void main(String[] args) {
        Counter counter = new Counter();

        Runnable task = () -> {
            for (int i = 0; i < 5; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);

        t1.start();
        t2.start();
    }
}
```

3. Inter-Thread Communication: `wait()`, `notify()`, `notifyAll()`

```
class Data {
    private int value;
    private boolean available = false;
```

```
    public synchronized void produce(int val) throws
InterruptedException {
        while (available) {
            wait();
        }
        value = val;
        available = true;
        System.out.println("Produced: " + value);
        notify();
    }

    public synchronized void consume() throws InterruptedException {
        while (!available) {
            wait();
        }
        System.out.println("Consumed: " + value);
        available = false;
        notify();
    }
}

public class InterThreadCommDemo {
    public static void main(String[] args) {
        Data data = new Data();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    data.produce(i);
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {}
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
```

```
                    data.consume();
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {}
        });

        producer.start();
        consumer.start();
    }
}
```

Lab Exercise 14

1. Read and Write Content Using FileReader and FileWriter

```
import java.io.*;

public class FileReadWriteDemo {
    public static void main(String[] args) {
        try (FileWriter writer = new FileWriter("output.txt")) {
            writer.write("Hello, this is file writing.\n");
        } catch (IOException e) {
            e.printStackTrace();
        }

        try (FileReader reader = new FileReader("output.txt")) {
            int ch;
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Read File Line by Line Using BufferedReader

```
import java.io.*;

public class BufferedReaderDemo {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
FileReader("output.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Object Serialization and Deserialization

```
import java.io.*;

class Student implements Serializable {
    String name;
    int age;

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

public class SerializationDemo {
    public static void main(String[] args) {
        Student s = new Student("Ayush", 21);

        // Serialization
        try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("student.ser"))) {
```

```
            out.writeObject(s);
            System.out.println("Serialization done.");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("student.ser"))) {
            Student s2 = (Student) in.readObject();
            System.out.println("Deserialized Student: " + s2.name + ",
Age: " + s2.age);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Lab Exercise 15

1. Use ArrayList and LinkedList

```
import java.util.*;

public class ListDemo {
    public static void main(String[] args) {
        ArrayList<String> arrayList = new ArrayList<>();
        arrayList.add("Apple");
        arrayList.add("Banana");
        System.out.println("ArrayList: " + arrayList);

        LinkedList<String> linkedList = new LinkedList<>();
        linkedList.add("Cat");
        linkedList.add("Dog");
        System.out.println("LinkedList: " + linkedList);
    }
}
```

2. Remove Duplicates Using HashSet

```
import java.util.*;

public class HashSetDemo {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
        HashSet<Integer> set = new HashSet<>(numbers);
        System.out.println("Without duplicates: " + set);
    }
}
```

3. Use HashMap to Store and Retrieve Key-Value Pairs

```
import java.util.*;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();
        map.put(101, "Alice");
        map.put(102, "Bob");

        System.out.println("Value for key 101: " + map.get(101));
        System.out.println("All entries: " + map);
    }
}
```

Lab Exercise 16

1. Read Input From Console Using Scanner

```
import java.util.Scanner;

public class ScannerDemo {
    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);
        sc.close();
    }
}
```

2. File Copy Using FileInputStream and FileOutputStream

```
import java.io.*;

public class FileCopyDemo {
    public static void main(String[] args) {
        try (FileInputStream in = new FileInputStream("source.txt");
             FileOutputStream out = new FileOutputStream("dest.txt"))
{

            int byteRead;
            while ((byteRead = in.read()) != -1) {
                out.write(byteRead);
            }
            System.out.println("File copied successfully.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Read From One File and Write to Another

```
import java.io.*;

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

```
             BufferedWriter bw = new BufferedWriter(new
FileWriter("dest.txt"))) {

            String line;
            while ((line = br.readLine()) != null) {
                bw.write(line);
                bw.newLine();
            }
            System.out.println("File content copied line by line.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```