Java Complete Study Notes (24CSA501)

Object-Oriented Programming Using Java - MCA I Semester

Table of Contents

MODULE 1: Object-Oriented Programming Concepts

1.1 Introduction to OOP

Object-Oriented Programming is a programming paradigm based on the concept of objects and classes. It focuses on modeling real-world entities and their interactions.

1.2 The Four Pillars of OOP

1.2.1 Abstraction

Definition: Abstraction is the process of hiding complex implementation details and showing only the essential features of an object.

Key Points:

- Reduces complexity by hiding unnecessary details
- Focuses on "what" rather than "how"
- Achieved through abstract classes and interfaces

Example:

```
// Abstraction Example
abstract class Vehicle {
    abstract void drive();
    abstract void stop();
}

class Car extends Vehicle {
    @Override
    void drive() {
        System.out.println("Car is driving on road");
    }

    @Override
    void stop() {
        System.out.println("Car stopped using brakes");
    }
}
```

1.2.2 Encapsulation

Definition: Encapsulation is the bundling of data (variables) and methods (functions) into a single unit called a class, and hiding the internal details from the outside world.

Key Points:

- Data hiding: Keep fields private
- Provide controlled access through getter and setter methods
- Maintains data integrity
- Improves maintainability

Example:

```
// Encapsulation Example
public class Student {
    // Private variables - data hiding
    private String name;
    private int age;
    private double cgpa;

    // Constructor
    public Student(String name, int age, double cgpa) {
        this.name = name;
        this.age = age;
        this.cgpa = cgpa;
    }

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

    public int getAge() {
        return age;
    }

    public double getCgpa() {
        return cgpa;
    }

    // Setter methods with validation
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public void setAge(int age) {
        if (age > 0 && age < 100) {
            this.age = age;
        }
    }

    public void setCgpa(double cgpa) {
        if (cgpa >= 0 && cgpa <= 10) {
            this.cgpa = cgpa;
        }
    }
}
```

```
// Usage
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Raj", 22, 8.5);
        s.setAge(23);
        System.out.println("Student: " + s.getName() + ", Age: " + s.getAge());
    }
}
```

1.2.3 Inheritance

Definition: Inheritance is a mechanism where a new class is derived from an existing class, inheriting its properties and behaviors.

Key Points:

- Promotes code reusability
- Establishes "IS-A" relationship
- Java supports single inheritance (extends keyword)
- Multiple inheritance supported through interfaces
- Superclass and Subclass relationship

Example:

```
// Inheritance Example

// Parent class (Superclass)
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    void eat() {
        System.out.println(name + " is eating");
    }

    void sleep() {
        System.out.println(name + " is sleeping");
    }
}

// Child class (Subclass)
class Dog extends Animal {
    String breed;

    public Dog(String name, String breed) {
        super(name); // Calls parent constructor
        this.breed = breed;
    }

    // Method overriding
    @Override
    void eat() {
        System.out.println(name + " (Dog) is eating dog food");
    }

    void bark() {
        System.out.println(name + " is barking: Woof Woof!");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Labrador");
        dog.eat();      // Output: Buddy (Dog) is eating dog food
        dog.bark();     // Output: Buddy is barking: Woof Woof!
        dog.sleep();    // Output: Buddy is sleeping
    }
}
```

Types of Inheritance:

1. **Single Inheritance**: A class inherits from one superclass

```
class A {}
class B extends A {}  // Single Inheritance
```

2. **Multilevel Inheritance**: A class inherits from a class which itself inherits from another class

```
class A {}
class B extends A {}
class C extends B {}  // Multilevel Inheritance
```

3. **Hierarchical Inheritance**: Multiple classes inherit from a single class

```
class A {}
class B extends A {}
class C extends A {}  // Hierarchical Inheritance
class D extends A {}
```

4. **Multiple Inheritance** (through interfaces): A class implements multiple interfaces

```
interface I1 {}
interface I2 {}
class A implements I1, I2 {}  // Multiple Inheritance
```

1.2.4 Polymorphism

Definition: Polymorphism means "many forms". It allows objects to take multiple forms and methods to behave differently based on context.

Key Points:

- Compile-time polymorphism: Method overloading
- Runtime polymorphism: Method overriding
- Enhances flexibility and reusability

Compile-time Polymorphism (Method Overloading):

```java
// Method Overloading Example
public class Calculator {
    // Method 1: Add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method 2: Add three integers (Overloading)
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method 3: Add two doubles (Overloading)
    public double add(double a, double b) {
        return a + b;
    }

    // Method 4: String concatenation (Overloading)
    public String add(String a, String b) {
        return a + b;
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));           // 15
        System.out.println(calc.add(5, 10, 15));       // 30
        System.out.println(calc.add(5.5, 10.5));       // 16.0
        System.out.println(calc.add("Hello", "World")); // HelloWorld
    }
}
```

Runtime Polymorphism (Method Overriding):

```java
// Method Overriding Example
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }

    double getArea() {
        return 0;
    }
}

class Circle extends Shape {
    double radius;

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

    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }

    @Override
    double getArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double length, width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    void draw() {
        System.out.println("Drawing a rectangle");
    }

    @Override
    double getArea() {
        return length * width;
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Shape shape1 = new Circle(5);
        Shape shape2 = new Rectangle(4, 6);

        shape1.draw();          // Output: Drawing a circle
        shape2.draw();          // Output: Drawing a rectangle

        System.out.println("Circle Area: " + shape1.getArea());     // Circle Area: 78.53...
        System.out.println("Rectangle Area: " + shape2.getArea());  // Rectangle Area: 24.0
    }
}
```

MODULE 2: Introduction to Java

2.1 Characteristics of Java

1. **Simple**: Easy to learn with familiar syntax
2. **Object-Oriented**: Everything is an object
3. **Platform-Independent**: "Write Once, Run Anywhere" (WORA)
4. **Secured**: Built-in security features
5. **Robust**: Strong memory management and exception handling
6. **Multi-threaded**: Supports concurrent programming
7. **Architecture-Neutral**: Bytecode is independent of platform
8. **Interpreted**: Bytecode is interpreted by JVM
9. **High Performance**: Uses JIT (Just-In-Time) compilation
10. **Distributed**: Supports network programming

2.2 Java Environment

JDK (Java Development Kit):

- Complete package for development
- Includes JRE, JVM, compiler, debugger, and other tools

JRE (Java Runtime Environment):

- Includes JVM and libraries
- Needed to run Java programs

JVM (Java Virtual Machine):

- Abstract computing machine

- Executes bytecode

2.3 Java Source File Structure

```
// 1. Package declaration (optional, must be first)
package com.example.app;

// 2. Import statements (optional)
import java.util.Scanner;
import java.io.*;

// 3. Class/Interface declaration
public class HelloWorld {

    // 4. Variable declarations (class-level)
    static int count = 0;

    // 5. Method definitions
    public static void main(String[] args) {
        // Method body
        System.out.println("Hello, World!");
    }

    public void displayMessage() {
        System.out.println("Welcome to Java");
    }
}
```

2.4 Compilation and Execution Process

Step 1: Write Source Code

```
HelloWorld.java (Text file)
```

Step 2: Compile

```
javac HelloWorld.java
Creates: HelloWorld.class (Bytecode)
```

Step 3: Execute

```
java HelloWorld
JVM interprets bytecode and runs the program
```

Detailed Compilation Process:

```
.java file → Lexical Analysis → Syntax Analysis → Semantic Analysis
→ Bytecode Generation → .class file
```

Runtime Execution:

```
.class file → ClassLoader → Bytecode Verifier → Execution Engine
→ JIT Compiler (if needed) → Machine Code → Output
```

MODULE 3: Fundamental Programming Constructs

3.1 Data Types

Java data types are divided into two categories:

3.1.1 Primitive Data Types

```java
public class DataTypeExample {
    public static void main(String[] args) {
        // Integer Types
        byte b = 127;              // 8-bit: -128 to 127
        short s = 32767;           // 16-bit: -32,768 to 32,767
        int i = 2147483647;        // 32-bit: -2^31 to 2^31-1
        long l = 9223372036854775807L;  // 64-bit, suffix 'L'

        // Floating-point Types
        float f = 3.14f;           // 32-bit, suffix 'f'
        double d = 3.14159265359;  // 64-bit (default)

        // Character Type
        char c = 'A';              // 16-bit, Unicode
        char c2 = 65;              // ASCII value

        // Boolean Type
        boolean isTrue = true;
        boolean isFalse = false;

        // Display
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("int: " + i);
        System.out.println("long: " + l);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
        System.out.println("char: " + c);
        System.out.println("boolean: " + isTrue);
    }
}
```

Data Type Reference Table:

Type	Size	Range	Default Value
byte	8 bits	-128 to 127	0
short	16 bits	-32,768 to 32,767	0
int	32 bits	-2^{31} to $2^{31}-1$	0
long	64 bits	-2^{63} to $2^{63}-1$	0L
float	32 bits	±3.40282347E+38	0.0f
double	64 bits	±1.79769313486231570E+308	0.0d
char	16 bits	0 to 65535	'\u0000'
boolean	1 bit	true/false	false

3.1.2 Non-Primitive Data Types (Reference Types)

```
// String Type
String str = "Hello Java";
String str2 = new String("Another way");

// Arrays
int[] arr = new int[5];
int[] arr2 = {1, 2, 3, 4, 5};

// Objects
class Person {}
Person p = new Person();
```

3.2 Variables

Variable Declaration and Initialization:

```
public class VariableExample {
    // Class variable (static)
    static int globalCount = 0;

    // Instance variable
    int instanceValue = 10;

    public static void main(String[] args) {
        // Local variable
        int localVariable = 20;

        // Variable scope example
        {
            int blockVariable = 30;
            System.out.println("Block variable: " + blockVariable); // OK
        }
        // System.out.println(blockVariable); // ERROR: out of scope

        System.out.println("Local variable: " + localVariable); // OK
    }
}
```

Variable Naming Conventions:

- Must start with letter, underscore (_), or dollar sign ($)
- Cannot start with digit
- Cannot contain spaces
- Case-sensitive
- Follow camelCase for variables

- Use UPPER_CASE for constants

3.3 Operators

3.3.1 Arithmetic Operators

```java
public class ArithmeticOperatorExample {
    public static void main(String[] args) {
        int a = 10, b = 3;

        System.out.println("Addition: " + (a + b));       // 13
        System.out.println("Subtraction: " + (a - b));    // 7
        System.out.println("Multiplication: " + (a * b)); // 30
        System.out.println("Division: " + (a / b));       // 3 (integer division)
        System.out.println("Modulus: " + (a % b));        // 1

        // Increment and Decrement
        int x = 5;
        System.out.println("x++: " + x++); // 5 (post-increment)
        System.out.println("x: " + x);     // 6

        System.out.println("++y: " + ++x); // 7 (pre-increment)
        System.out.println("x: " + x);     // 7
    }
}
```

3.3.2 Relational Operators

```java
public class RelationalOperatorExample {
    public static void main(String[] args) {
        int a = 10, b = 20;

        System.out.println("a == b: " + (a == b));   // false
        System.out.println("a != b: " + (a != b));   // true
        System.out.println("a < b: " + (a < b));     // true
        System.out.println("a > b: " + (a > b));     // false
        System.out.println("a <= b: " + (a <= b));   // true
        System.out.println("a >= b: " + (a >= b));   // false
    }
}
```

3.3.3 Logical Operators

```
public class LogicalOperatorExample {
    public static void main(String[] args) {
        boolean a = true, b = false;

        // AND (&&)
        System.out.println("a && b: " + (a && b));   // false

        // OR (||)
        System.out.println("a || b: " + (a || b));   // true

        // NOT (!)
        System.out.println("!a: " + (!a));           // false

        // Short-circuit evaluation
        int x = 5;
        if (x > 0 && ++x < 10) {  // x incremented only if first condition true
            System.out.println("x: " + x);  // x = 6
        }
    }
}
```

3.3.4 Bitwise Operators

```
public class BitwiseOperatorExample {
    public static void main(String[] args) {
        int a = 5;      // Binary: 0101
        int b = 3;      // Binary: 0011

        System.out.println("a & b: " + (a & b));     // 1 (0001)
        System.out.println("a | b: " + (a | b));     // 7 (0111)
        System.out.println("a ^ b: " + (a ^ b));     // 6 (0110)
        System.out.println("~a: " + (~a));           // -6
        System.out.println("a << 1: " + (a << 1));   // 10 (1010)
        System.out.println("a >> 1: " + (a >> 1));   // 2 (0010)
        System.out.println("a >>> 1: " + (a >>> 1)); // 2 (unsigned right shift)
    }
}
```

3.3.5 Assignment Operators

```
public class AssignmentOperatorExample {
    public static void main(String[] args) {
        int x = 10;

        x += 5;    // x = x + 5 = 15
        x -= 3;    // x = x - 3 = 12
        x *= 2;    // x = x * 2 = 24
        x /= 4;    // x = x / 4 = 6
        x %= 4;    // x = x % 4 = 2
        x &= 3;    // x = x & 3
        x |= 5;    // x = x | 5
        x ^= 2;    // x = x ^ 2

        System.out.println("Final x: " + x);
    }
}
```

3.3.6 Ternary/Conditional Operator

```
public class TernaryOperatorExample {
    public static void main(String[] args) {
        int a = 10, b = 20;

        // Syntax: condition ? value_if_true : value_if_false
        int max = (a > b) ? a : b;
        System.out.println("Maximum: " + max);  // 20

        // Nested ternary
        String result = (a > b) ? "a is greater" :
                        (a == b) ? "a equals b" : "b is greater";
        System.out.println(result);  // b is greater
    }
}
```

3.3.7 Operator Precedence

Precedence	**Operators**	**Associativity**
1	() [] .	Left to Right
2	++ -- + - ! ~	Right to Left
3	* / %	Left to Right
4	+ -	Left to Right
5	<< >> >>>	Left to Right
6	< > <= >= instanceof	Left to Right
7	== !=	Left to Right
8	&	Left to Right
9	^	Left to Right

Precedence	Operators	Associativity
10	|	Left to Right
11	&&	Left to Right
12	||	Left to Right
13	?:	Right to Left
14	= += -= *= /= %=	Right to Left

MODULE 4: Control Flow & Arrays & Strings

4.1 Conditional Statements

4.1.1 if Statement

```
public class IfExample {
    public static void main(String[] args) {
        int age = 18;

        // Simple if
        if (age >= 18) {
            System.out.println("You are an adult");
        }

        // if-else
        if (age >= 18) {
            System.out.println("You are an adult");
        } else {
            System.out.println("You are a minor");
        }

        // if-else if-else
        if (age < 13) {
            System.out.println("Child");
        } else if (age < 18) {
            System.out.println("Teenager");
        } else if (age < 60) {
            System.out.println("Adult");
        } else {
            System.out.println("Senior");
        }
    }
}
```

4.1.2 switch Statement

```java
public class SwitchExample {
    public static void main(String[] args) {
        int day = 3;
        String dayName;

        switch (day) {
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
            case 7:
                dayName = "Weekend";
                break;
            default:
                dayName = "Invalid day";
        }

        System.out.println("Day: " + dayName);  // Output: Day: Wednesday
    }
}

// Switch with String (Java 7+)
public class SwitchStringExample {
    public static void main(String[] args) {
        String fruit = "apple";

        switch (fruit) {
            case "apple":
                System.out.println("Apple is red");
                break;
            case "banana":
                System.out.println("Banana is yellow");
                break;
            case "orange":
                System.out.println("Orange is orange");
                break;
            default:
                System.out.println("Unknown fruit");
        }
    }
}
```

4.2 Looping Statements

4.2.1 for Loop

```
public class ForLoopExample {
    public static void main(String[] args) {
        // Simple for loop
        for (int i = 1; i <= 5; i++) {
            System.out.println("i = " + i);
        }

        // for loop with multiple expressions
        for (int i = 1, j = 10; i <= 5; i++, j--) {
            System.out.println("i = " + i + ", j = " + j);
        }

        // Nested for loop
        System.out.println("Multiplication Table:");
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                System.out.print((i * j) + " ");
            }
            System.out.println();
        }

        // Reverse for loop
        for (int i = 5; i >= 1; i--) {
            System.out.println("i = " + i);
        }
    }
}
```

4.2.2 while Loop

```java
public class WhileLoopExample {
    public static void main(String[] args) {
        // while loop
        int count = 1;
        while (count <= 5) {
            System.out.println("Count: " + count);
            count++;
        }

        // Infinite loop (careful!)
        // while (true) { ... }

        // Example: Sum of numbers
        int sum = 0, num = 1;
        while (num <= 10) {
            sum += num;
            num++;
        }
        System.out.println("Sum: " + sum);
    }
}
```

4.2.3 do-while Loop

```java
public class DoWhileLoopExample {
    public static void main(String[] args) {
        // do-while loop (executes at least once)
        int count = 1;
        do {
            System.out.println("Count: " + count);
            count++;
        } while (count <= 5);

        // Menu-driven program
        Scanner scanner = new Scanner(System.in);
        int choice;
        do {
            System.out.println("Menu: 1. Add  2. Subtract  3. Exit");
            System.out.print("Enter choice: ");
            choice = scanner.nextInt();

            if (choice == 1) {
                System.out.println("Addition selected");
            } else if (choice == 2) {
                System.out.println("Subtraction selected");
            }
        } while (choice != 3);
    }
}
```

4.2.4 Enhanced for Loop (for-each)

```
public class EnhancedForLoopExample {
    public static void main(String[] args) {
        // Array iteration
        int[] numbers = {1, 2, 3, 4, 5};
        for (int num : numbers) {
            System.out.println("Number: " + num);
        }

        // String array iteration
        String[] fruits = {"Apple", "Banana", "Orange"};
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // 2D Array iteration
        int[][] matrix = {{1, 2, 3}, {4, 5, 6}};
        for (int[] row : matrix) {
            for (int val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }
    }
}
```

4.2.5 break and continue

```java
public class BreakContinueExample {
    public static void main(String[] args) {
        // break statement
        System.out.println("Using break:");
        for (int i = 1; i <= 10; i++) {
            if (i == 5) {
                break;  // Exit loop when i equals 5
            }
            System.out.println(i);
        }

        // continue statement
        System.out.println("Using continue:");
        for (int i = 1; i <= 10; i++) {
            if (i % 2 == 0) {
                continue;  // Skip even numbers
            }
            System.out.println(i);  // Prints only odd numbers
        }

        // Labeled break
        System.out.println("Using labeled break:");
        outerLoop:
        for (int i = 1; i <= 3; i++) {
            for (int j = 1; j <= 3; j++) {
                if (i == 2 && j == 2) {
                    break outerLoop;  // Breaks outer loop
                }
                System.out.print("(" + i + "," + j + ") ");
            }
            System.out.println();
        }
    }
}
```

4.3 Arrays

4.3.1 Single-Dimensional Arrays

```java
public class SingleDArrayExample {
    public static void main(String[] args) {
        // Declaration and allocation
        int[] arr1 = new int[5];  // Default values: 0

        // Declaration with initialization
        int[] arr2 = {1, 2, 3, 4, 5};

        // Using array
        arr1[0] = 10;
        arr1[1] = 20;
        arr1[4] = 50;

        // Accessing elements
        System.out.println("Array elements:");
        for (int i = 0; i < arr1.length; i++) {
            System.out.println("arr1[" + i + "] = " + arr1[i]);
        }

        // Enhanced for loop
        System.out.println("Using enhanced for:");
        for (int num : arr2) {
            System.out.println(num);
        }

        // Array methods
        System.out.println("Length: " + arr2.length);
        System.out.println("Accessing last element: " + arr2[arr2.length - 1]);
    }
}
```

4.3.2 Multi-Dimensional Arrays

```java
public class MultiDArrayExample {
    public static void main(String[] args) {
        // 2D Array Declaration and Initialization
        int[][] matrix1 = new int[3][3];

        int[][] matrix2 = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // Initialization
        matrix1[0][0] = 1;
        matrix1[0][1] = 2;
        matrix1[1][1] = 5;

        // Accessing and printing
        System.out.println("2D Array:");
        for (int i = 0; i < matrix1.length; i++) {
            for (int j = 0; j < matrix1[i].length; j++) {
                System.out.print(matrix1[i][j] + " ");
            }
            System.out.println();
        }

        // 3D Array
        int[][][] cube = new int[2][3][4];
        cube[0][1][2] = 10;

        System.out.println("3D Array element: " + cube[0][1][2]);
    }
}
```

4.3.3 Jagged Arrays (IMPORTANT - Hidden Concept)

Definition: A jagged array is a multi-dimensional array where rows can have different lengths. It's an array of arrays where each row array can have a different size.

```java
public class JaggedArrayExample {
    public static void main(String[] args) {
        // Declaration of jagged array
        // Syntax: type[][] arrayName = new type[rows][];

        // 2x3 jagged array (2 rows, varying columns)
        int[][] jagged = new int[3][];

        // Initialize rows with different lengths
        jagged[0] = new int[2];     // First row has 2 elements
        jagged[1] = new int[4];     // Second row has 4 elements
        jagged[2] = new int[3];     // Third row has 3 elements

        // Assign values
        jagged[0][0] = 1;
        jagged[0][1] = 2;

        jagged[1][0] = 10;
        jagged[1][1] = 20;
        jagged[1][2] = 30;
        jagged[1][3] = 40;

        jagged[2][0] = 100;
        jagged[2][1] = 200;
        jagged[2][2] = 300;

        // Printing jagged array
        System.out.println("Jagged Array:");
        for (int i = 0; i < jagged.length; i++) {
            for (int j = 0; j < jagged[i].length; j++) {
                System.out.print(jagged[i][j] + " ");
            }
            System.out.println();
        }

        // Direct initialization
        int[][] jagged2 = {
            {1, 2},
            {10, 20, 30, 40},
            {100, 200, 300}
        };

        System.out.println("\nDirect Jagged Array Initialization:");
        for (int[] row : jagged2) {
            for (int val : row) {
                System.out.print(val + " ");
            }
            System.out.println();
        }

        // Practical Example: Student marks for different subjects
        System.out.println("\nStudent Marks (Different subjects per student):");
        int[][] studentMarks = {
            {85, 90, 78},           // Student 1: 3 subjects
            {92, 88, 79, 95},       // Student 2: 4 subjects
            {75, 82}                // Student 3: 2 subjects
        };

        for (int i = 0; i < studentMarks.length; i++) {
            System.out.print("Student " + (i + 1) + " marks: ");
            int sum = 0;
            for (int j = 0; j < studentMarks[i].length; j++) {
                System.out.print(studentMarks[i][j] + " ");
                sum += studentMarks[i][j];
            }
            System.out.println("| Average: " + (sum / (double)studentMarks[i].length));
        }
    }
}
```

Key Points about Jagged Arrays:

1. Each row can have a different number of columns
2. Memory is allocated row by row
3. Useful when data naturally has varying dimensions
4. Check row length before accessing: `jagged[i].length`
5. Different from rectangular 2D arrays

4.3.4 Array Utility Methods

```java
import java.util.Arrays;

public class ArrayUtilsExample {
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 7};

        // Sorting
        Arrays.sort(arr);
        System.out.println("Sorted: " + Arrays.toString(arr));

        // Searching (requires sorted array)
        int index = Arrays.binarySearch(arr, 7);
        System.out.println("Index of 7: " + index);

        // Filling array
        int[] arr2 = new int[5];
        Arrays.fill(arr2, 10);
        System.out.println("Filled array: " + Arrays.toString(arr2));

        // Copying array
        int[] arr3 = Arrays.copyOf(arr, arr.length);
        System.out.println("Copied array: " + Arrays.toString(arr3));

        // Partial copy
        int[] arr4 = Arrays.copyOfRange(arr, 1, 4);
        System.out.println("Partial copy: " + Arrays.toString(arr4));

        // Comparing arrays
        int[] arr5 = {1, 2, 7, 9, 5};
        boolean isEqual = Arrays.equals(arr, arr5);
        System.out.println("Arrays equal: " + isEqual);
    }
}
```

4.4 Strings

4.4.1 String Basics

```java
public class StringBasicsExample {
    public static void main(String[] args) {
        // String declaration
        String str1 = "Hello";          // String literal
        String str2 = new String("Hello");  // Using constructor

        // String concatenation
        String str3 = "Hello" + " " + "World";
        System.out.println(str3);  // Hello World

        // String methods
        String str = "Java Programming";

        System.out.println("Length: " + str.length());            // 17
        System.out.println("Uppercase: " + str.toUpperCase());    // JAVA PROGRAMMING
        System.out.println("Lowercase: " + str.toLowerCase());    // java programming
        System.out.println("Char at 5: " + str.charAt(5));        // P
        System.out.println("Substring: " + str.substring(5, 14)); // Program
        System.out.println("Index of 'Java': " + str.indexOf("Java")); // 0
        System.out.println("Starts with 'Java': " + str.startsWith("Java")); // true
        System.out.println("Ends with 'ing': " + str.endsWith("ing"));    // true
        System.out.println("Contains 'gram': " + str.contains("gram"));   // true

        // String comparison
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = new String("Hello");

        System.out.println("s1 == s2: " + (s1 == s2));           // true (same reference)
        System.out.println("s1 == s3: " + (s1 == s3));           // false (different objects)
        System.out.println("s1.equals(s3): " + s1.equals(s3));   // true (same value)
        System.out.println("s1.compareTo(s2): " + s1.compareTo(s2)); // 0 (equal)
    }
}
```

4.4.2 Important String Methods

```
public class StringMethodsExample {
    public static void main(String[] args) {
        String str = "  Java Programming  ";

        // trim - remove leading/trailing spaces
        System.out.println("Trimmed: '" + str.trim() + "'");

        // replace
        String replaced = str.replace("Java", "Python");
        System.out.println("Replaced: " + replaced);

        // split
        String csv = "apple,banana,orange";
        String[] fruits = csv.split(",");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // charAt and indexOf
        String str2 = "Hello World";
        System.out.println("Char at 6: " + str2.charAt(6));       // W
        System.out.println("Last index of 'o': " + str2.lastIndexOf('o')); // 7

        // substring variations
        System.out.println("From index 6: " + str2.substring(6));        // World
        System.out.println("From 0 to 5: " + str2.substring(0, 5));      // Hello

        // isEmpty and blank
        String empty = "";
        String blank = "   ";
        System.out.println("Empty: " + empty.isEmpty());          // true
        System.out.println("Blank: " + blank.isBlank());          // true (Java 11+)

        // repeat (Java 11+)
        String repeated = "Ha".repeat(3);
        System.out.println("Repeated: " + repeated);              // HaHaHa

        // valueOf - convert to String
        int num = 42;
        String numStr = String.valueOf(num);
        System.out.println("Number as string: " + numStr);

        // format
        String formatted = String.format("Pi is approximately %.2f", 3.14159);
        System.out.println(formatted);                            // Pi is approximately 3.14
    }
}
```

4.4.3 String Immutability

```
public class StringImmutabilityExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = str1;        // Both point to same object

        str1 = str1 + " World";    // Creates new string, str1 points to it

        System.out.println("str1: " + str1);  // "Hello World"
        System.out.println("str2: " + str2);  // "Hello" (unchanged)

        // String pool concept
        String s1 = "Java";
        String s2 = "Java";        // Reuses same object from string pool
        String s3 = new String("Java");  // Creates new object in heap

        System.out.println("s1 == s2: " + (s1 == s2));   // true
        System.out.println("s1 == s3: " + (s1 == s3));   // false
    }
}
```

4.4.4 StringBuilder and StringBuffer

```java
public class StringBuilderExample {
    public static void main(String[] args) {
        // StringBuilder (mutable, not thread-safe, faster)
        StringBuilder sb = new StringBuilder("Hello");
        sb.append(" World");
        sb.append(" Java");
        System.out.println("StringBuilder: " + sb);

        sb.insert(6, "Beautiful ");
        System.out.println("After insert: " + sb);

        sb.delete(6, 16);
        System.out.println("After delete: " + sb);

        sb.reverse();
        System.out.println("Reversed: " + sb);

        // StringBuffer (thread-safe, slower)
        StringBuffer sbf = new StringBuffer("Hello");
        sbf.append(" World");
        System.out.println("StringBuffer: " + sbf);

        // Performance comparison
        long startTime = System.currentTimeMillis();
        String result = "";
        for (int i = 0; i < 10000; i++) {
            result += "a";  // Creates new String each time
        }
        long stringTime = System.currentTimeMillis() - startTime;

        startTime = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder();
        for (int i = 0; i < 10000; i++) {
            sb2.append("a");
        }
        long builderTime = System.currentTimeMillis() - startTime;

        System.out.println("String concatenation time: " + stringTime + "ms");
        System.out.println("StringBuilder time: " + builderTime + "ms");
    }
}
```

MODULE 5: Classes and Objects

5.1 Class Definition

```java
public class Car {
    // 1. Data Members (Attributes)
    String color;
    String model;
    int year;
    double price;

    // 2. Constructor (Initializes object)
    public Car(String color, String model, int year, double price) {
        this.color = color;
        this.model = model;
        this.year = year;
        this.price = price;
    }

    // 3. Methods (Behaviors)
    void display() {
        System.out.println("Car: " + model + ", Color: " + color +
                         ", Year: " + year + ", Price: " + price);
    }

    void drive() {
        System.out.println(model + " is driving");
    }

    // Usage
    public static void main(String[] args) {
        Car car = new Car("Red", "Honda", 2023, 1500000);
        car.display();
        car.drive();
    }
}
```

5.2 Constructors

5.2.1 Default Constructor

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

    // Default constructor (no parameters)
    public Student() {
        name = "Unknown";
        roll = 0;
    }

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

// Usage
Student s = new Student();  // Calls default constructor
s.display();                // Output: Name: Unknown, Roll: 0
```

5.2.2 Parameterized Constructor

```
public class Student {
    String name;
    int roll;
    double cgpa;

    // Parameterized constructor
    public Student(String name, int roll, double cgpa) {
        this.name = name;
        this.roll = roll;
        this.cgpa = cgpa;
    }

    void display() {
        System.out.println("Name: " + name + ", Roll: " + roll +
                         ", CGPA: " + cgpa);
    }
}

// Usage
Student s = new Student("Raj", 101, 8.5);
s.display();
```

5.2.3 Constructor Overloading

```
public class Rectangle {
    double length, width;

    // Constructor 1: No parameters
    public Rectangle() {
        length = 1;
        width = 1;
    }

    // Constructor 2: One parameter (square)
    public Rectangle(double side) {
        length = side;
        width = side;
    }

    // Constructor 3: Two parameters (rectangle)
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    double getArea() {
        return length * width;
    }

    void display() {
        System.out.println("Length: " + length + ", Width: " + width +
                         ", Area: " + getArea());
    }
}

// Usage
Rectangle r1 = new Rectangle();           // 1x1
Rectangle r2 = new Rectangle(5);          // 5x5
Rectangle r3 = new Rectangle(4, 6);       // 4x6

r1.display();
r2.display();
r3.display();
```

5.2.4 Constructor Chaining (Using this() and super())

```java
public class ConstructorChainingExample {
    static class Parent {
        String name;

        public Parent(String name) {
            this.name = name;
        }
    }

    static class Child extends Parent {
        int age;

        // Constructor 1
        public Child(String name, int age) {
            super(name);  // Call parent constructor
            this.age = age;
        }

        // Constructor 2 - using this()
        public Child(String name) {
            this(name, 0);  // Call another constructor in same class
        }

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

    public static void main(String[] args) {
        Child c1 = new Child("Raj", 20);
        Child c2 = new Child("Priya");

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

5.3 Methods

5.3.1 Method Definition and Calling

```
public class MethodExample {
    // Method with no parameters and no return
    void greet() {
        System.out.println("Hello!");
    }

    // Method with parameters, no return
    void greet(String name) {
        System.out.println("Hello, " + name);
    }

    // Method with parameters and return value
    int add(int a, int b) {
        return a + b;
    }

    // Method with variable parameters
    void printNumbers(int... numbers) {
        for (int num : numbers) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        MethodExample obj = new MethodExample();

        obj.greet();           // Output: Hello!
        obj.greet("Raj");      // Output: Hello, Raj
        int sum = obj.add(5, 10);
        System.out.println("Sum: " + sum);  // Output: Sum: 15

        obj.printNumbers(1, 2, 3, 4, 5);    // Output: 1 2 3 4 5
    }
}
```

5.3.2 Method Overloading

```
public class MethodOverloadingExample {
    // Method overloading - same name, different parameters

    public void print(int value) {
        System.out.println("Integer: " + value);
    }

    public void print(double value) {
        System.out.println("Double: " + value);
    }

    public void print(String value) {
        System.out.println("String: " + value);
    }

    public void print(int a, int b) {
        System.out.println("Two integers: " + a + ", " + b);
    }

    public static void main(String[] args) {
        MethodOverloadingExample obj = new MethodOverloadingExample();

        obj.print(10);           // Integer: 10
        obj.print(3.14);         // Double: 3.14
        obj.print("Hello");      // String: Hello
        obj.print(5, 10);        // Two integers: 5, 10
    }
}
```

5.4 Access Specifiers

```
public class AccessSpecifiersExample {
    // public - accessible from anywhere
    public int publicVar = 1;

    // private - accessible only within the class
    private int privateVar = 2;

    // protected - accessible within package and subclasses
    protected int protectedVar = 3;

    // default (no modifier) - accessible within same package
    int defaultVar = 4;

    public void publicMethod() {
        System.out.println("This is public");
    }

    private void privateMethod() {
        System.out.println("This is private");
    }

    protected void protectedMethod() {
        System.out.println("This is protected");
    }

    void defaultMethod() {
        System.out.println("This is default");
    }

    public static void main(String[] args) {
        AccessSpecifiersExample obj = new AccessSpecifiersExample();

        // All accessible within the same class
        System.out.println(obj.publicVar);
        System.out.println(obj.privateVar);
        System.out.println(obj.protectedVar);
        System.out.println(obj.defaultVar);

        obj.publicMethod();
        obj.privateMethod();
        obj.protectedMethod();
        obj.defaultMethod();
    }
}
```

Access Modifiers Table:

Modifier	Same Class	Same Package	Subclass	Outside
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

5.5 this Keyword

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

    // Constructor
    public ThisKeywordExample(String name, int age) {
        this.name = name;  // 'this' refers to current object
        this.age = age;
    }

    // Method returning reference to current object
    public ThisKeywordExample getObject() {
        return this;
    }

    // Method calling another method
    public void display() {
        System.out.println("Name: " + this.name + ", Age: " + this.age);
        this.showDetails();  // Calling another method
    }

    private void showDetails() {
        System.out.println("Details accessed via this keyword");
    }

    public static void main(String[] args) {
        ThisKeywordExample obj = new ThisKeywordExample("Raj", 22);
        obj.display();

        // Method chaining
        ThisKeywordExample sameObj = obj.getObject();
        System.out.println(obj == sameObj);  // true
    }
}
```

MODULE 6: Static Members, Access Modifiers & Final Keywords

6.1 Static Variables

```
public class StaticVariableExample {
    // Static variable - shared among all instances
    static int count = 0;

    // Instance variable - unique for each instance
    String name;

    public StaticVariableExample(String name) {
        this.name = name;
        count++;  // Increment static counter
    }

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

    public static void main(String[] args) {
        StaticVariableExample obj1 = new StaticVariableExample("Raj");
        StaticVariableExample obj2 = new StaticVariableExample("Priya");
        StaticVariableExample obj3 = new StaticVariableExample("Amit");

        StaticVariableExample.displayCount();  // Total objects created: 3

        // Can also access through instance (not recommended)
        obj1.displayCount();  // Total objects created: 3
    }
}
```

6.2 Static Methods

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

    void instanceMethod() {
        System.out.println("This is an instance method");
    }

    public static void main(String[] args) {
        // Static method - called without creating object
        StaticMethodExample.staticMethod();

        // Instance method - needs object
        StaticMethodExample obj = new StaticMethodExample();
        obj.instanceMethod();
    }
}

// Practical example: Utility class
public class MathUtils {
    // Static methods for mathematical operations
    static int add(int a, int b) {
        return a + b;
    }

    static int subtract(int a, int b) {
        return a - b;
    }

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

    static double divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero");
        }
        return (double) a / b;
    }

    public static void main(String[] args) {
        System.out.println("5 + 3 = " + MathUtils.add(5, 3));
        System.out.println("5 - 3 = " + MathUtils.subtract(5, 3));
        System.out.println("5 * 3 = " + MathUtils.multiply(5, 3));
        System.out.println("5 / 3 = " + MathUtils.divide(5, 3));
    }
}
```

6.3 Static Block

```
public class StaticBlockExample {
    static int value;
    static String message;

    // Static block - executes once when class is loaded
    static {
        System.out.println("Static block executing...");
        value = 100;
        message = "Initialized in static block";
    }

    public static void main(String[] args) {
        System.out.println("Value: " + value);
        System.out.println("Message: " + message);
    }
}

// Output:
// Static block executing...
// Value: 100
// Message: Initialized in static block
```

6.4 Instance Block

```
public class InstanceBlockExample {
    int count = 0;

    // Instance block - executes every time object is created
    {
        System.out.println("Instance block executing...");
        count = 10;
    }

    public InstanceBlockExample() {
        System.out.println("Constructor executing...");
        count += 5;
    }

    void display() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        System.out.println("Creating first object...");
        InstanceBlockExample obj1 = new InstanceBlockExample();
        obj1.display();

        System.out.println("\nCreating second object...");
        InstanceBlockExample obj2 = new InstanceBlockExample();
        obj2.display();
    }
}
```

6.5 Final Keyword

6.5.1 Final Variables

```
public class FinalVariableExample {
    // Final instance variable
    final double PI = 3.14159;

    // Final static variable
    static final String COUNTRY = "India";

    public FinalVariableExample() {
        // Can initialize final variable in constructor
        // PI = 3.14;  // ERROR - already initialized in class
    }

    public static void main(String[] args) {
        FinalVariableExample obj = new FinalVariableExample();

        // System.out.println("PI = " + obj.PI);
        // obj.PI = 3.14;  // ERROR - final variable cannot be reassigned

        System.out.println("Country: " + FinalVariableExample.COUNTRY);
        // COUNTRY = "USA";  // ERROR - final static variable cannot be reassigned
    }
}
```

6.5.2 Final Methods

```
public class FinalMethodExample {
    // Final method - cannot be overridden
    final void displayInfo() {
        System.out.println("This method cannot be overridden");
    }

    void normalMethod() {
        System.out.println("This method can be overridden");
    }
}

class Child extends FinalMethodExample {
    @Override
    void normalMethod() {
        System.out.println("Overridden normal method");
    }

    // This would cause compilation error
    // @Override
    // void displayInfo() {
    //     System.out.println("Cannot override final method");
    // }
}
```

6.5.3 Final Classes

```
// Final class - cannot be extended/inherited
final class ImmutableClass {
    private final String value;

    public ImmutableClass(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}

// This would cause compilation error
// class ExtendedClass extends ImmutableClass {
//     // Cannot extend final class
// }
```

6.6 Protected Members

```
package com.example.package1;

public class ParentClass {
    protected int protectedVar = 10;

    protected void protectedMethod() {
        System.out.println("This is protected");
    }
}

package com.example.package2;

import com.example.package1.ParentClass;

public class ChildClass extends ParentClass {
    public void testProtected() {
        // Can access protected members from parent in subclass
        System.out.println("Protected var: " + protectedVar);
        protectedMethod();
    }
}
```

MODULE 7: Inheritance

7.1 Inheritance Basics

```
public class InheritanceExample {
    // Parent class
    static class Vehicle {
        String name;
        String color;

        public Vehicle(String name, String color) {
            this.name = name;
            this.color = color;
        }

        void display() {
            System.out.println("Vehicle: " + name + ", Color: " + color);
        }
    }

    // Child class inheriting from Vehicle
    static class Car extends Vehicle {
        int doors;

        public Car(String name, String color, int doors) {
            super(name, color);  // Call parent constructor
            this.doors = doors;
        }

        @Override
        void display() {
            super.display();  // Call parent method
            System.out.println("Doors: " + doors);
        }
    }

    public static void main(String[] args) {
        Car car = new Car("Toyota", "Red", 4);
        car.display();
    }
}
```

7.2 super Keyword

```
public class SuperKeywordExample {
    static class Animal {
        String name;

        public Animal(String name) {
            this.name = name;
        }

        void eat() {
            System.out.println(name + " is eating");
        }
    }

    static class Dog extends Animal {
        String breed;

        public Dog(String name, String breed) {
            super(name);  // Call parent constructor
            this.breed = breed;
        }

        @Override
        void eat() {
            super.eat();  // Call parent method
            System.out.println(name + " is eating dog food");
        }

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

    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", "Labrador");
        dog.eat();
        dog.display();
    }
}
```

7.3 Method Overriding

```java
public class MethodOverridingExample {
    static class Shape {
        void display() {
            System.out.println("This is a shape");
        }

        double getArea() {
            return 0;
        }
    }

    static class Circle extends Shape {
        double radius;

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

        @Override
        void display() {
            System.out.println("This is a circle");
        }

        @Override
        double getArea() {
            return Math.PI * radius * radius;
        }
    }

    static class Rectangle extends Shape {
        double length, width;

        public Rectangle(double length, double width) {
            this.length = length;
            this.width = width;
        }

        @Override
        void display() {
            System.out.println("This is a rectangle");
        }

        @Override
        double getArea() {
            return length * width;
        }
    }

    public static void main(String[] args) {
        Shape[] shapes = new Shape[2];
        shapes[0] = new Circle(5);
        shapes[1] = new Rectangle(4, 6);

        for (Shape shape : shapes) {
            shape.display();
            System.out.println("Area: " + shape.getArea());
        }
    }
}
```

7.4 Constructors in Inheritance

```
public class ConstructorInheritanceExample {
    static class Parent {
        int x;

        public Parent() {
            System.out.println("Parent default constructor");
            x = 0;
        }

        public Parent(int x) {
            System.out.println("Parent parameterized constructor");
            this.x = x;
        }
    }

    static class Child extends Parent {
        int y;

        public Child() {
            super();  // Calls parent default constructor
            System.out.println("Child default constructor");
            y = 0;
        }

        public Child(int x, int y) {
            super(x);  // Calls parent parameterized constructor
            System.out.println("Child parameterized constructor");
            this.y = y;
        }
    }

    public static void main(String[] args) {
        System.out.println("Creating Child object with parameters:");
        Child c = new Child(10, 20);
        System.out.println("x = " + c.x + ", y = " + c.y);
    }
}

// Output:
// Creating Child object with parameters:
// Parent parameterized constructor
// Child parameterized constructor
// x = 10, y = 20
```

7.5 The Object Class

```java
public class ObjectClassExample {
    static class Person {
        String name;
        int age;

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

        @Override
        public String toString() {
            return "Person{" +
                    "name='" + name + '\'' +
                    ", age=" + age +
                    '}';
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (this.getClass() != obj.getClass()) return false;

            Person other = (Person) obj;
            return this.name.equals(other.name) && this.age == other.age;
        }

        @Override
        public int hashCode() {
            return name.hashCode() + age;
        }
    }

    public static void main(String[] args) {
        Person p1 = new Person("Raj", 22);
        Person p2 = new Person("Raj", 22);
        Person p3 = new Person("Priya", 21);

        // toString()
        System.out.println(p1.toString());  // Person{name='Raj', age=22}
        System.out.println(p1);             // Person{name='Raj', age=22}

        // equals()
        System.out.println("p1.equals(p2): " + p1.equals(p2));  // true
        System.out.println("p1.equals(p3): " + p1.equals(p3));  // false
        // hashCode()
        System.out.println("p1.hashCode(): " + p1.hashCode());
        System.out.println("p2.hashCode(): " + p2.hashCode());
    }
}
```

MODULE 8: Abstract Classes and Interfaces

8.1 Abstract Classes

```
public class AbstractClassExample {
    // Abstract class - cannot be instantiated
    abstract static class Shape {
        String color;

        public Shape(String color) {
            this.color = color;
        }

        // Abstract method - must be implemented by subclass
        abstract double getArea();

        // Concrete method
        void display() {
            System.out.println("Color: " + color);
        }
    }

    // Concrete class implementing abstract methods
    static class Circle extends Shape {
        double radius;

        public Circle(String color, double radius) {
            super(color);
            this.radius = radius;
        }

        @Override
        double getArea() {
            return Math.PI * radius * radius;
        }
    }

    static class Rectangle extends Shape {
        double length, width;

        public Rectangle(String color, double length, double width) {
            super(color);
            this.length = length;
            this.width = width;
        }

        @Override
        double getArea() {
            return length * width;
        }
    }

    public static void main(String[] args) {
        // Shape s = new Shape("Red");  // ERROR - cannot instantiate abstract class

        Shape circle = new Circle("Red", 5);
        Shape rectangle = new Rectangle("Blue", 4, 6);

        circle.display();
        System.out.println("Circle Area: " + circle.getArea());

        rectangle.display();
        System.out.println("Rectangle Area: " + rectangle.getArea());
    }
}
```

8.2 Abstract Methods

```
abstract class Animal {
    // Abstract method - no body
    abstract void makeSound();

    // Abstract method with different return type
    abstract int getAge();

    // Concrete method in abstract class
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

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

    @Override
    int getAge() {
        return 5;
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Cat meows: Meow!");
    }

    @Override
    int getAge() {
        return 3;
    }
}

// Usage
Animal dog = new Dog();
dog.makeSound();  // Dog barks: Woof!
dog.sleep();      // Animal is sleeping

Animal cat = new Cat();
cat.makeSound();  // Cat meows: Meow!
```

8.3 Interfaces

```java
public class InterfaceExample {
    // Interface definition
    interface Animal {
        // Abstract methods (public by default)
        void eat();
        void sleep();
        void makeSound();

        // Default method (Java 8+)
        default void move() {
            System.out.println("Animal is moving");
        }

        // Static method (Java 8+)
        static void info() {
            System.out.println("This is animal interface");
        }
    }

    // Implementing interface
    static class Dog implements Animal {
        @Override
        public void eat() {
            System.out.println("Dog is eating");
        }

        @Override
        public void sleep() {
            System.out.println("Dog is sleeping");
        }

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

    static class Cat implements Animal {
        @Override
        public void eat() {
            System.out.println("Cat is eating");
        }

        @Override
        public void sleep() {
            System.out.println("Cat is sleeping");
        }

        @Override
        public void makeSound() {
            System.out.println("Cat meows: Meow!");
        }
    }

    public static void main(String[] args) {
        Animal dog = new Dog();
        dog.eat();
        dog.sleep();
        dog.makeSound();
        dog.move();

        System.out.println();

        Animal cat = new Cat();
        cat.eat();
        cat.sleep();
        cat.makeSound();
        cat.move();

        Animal.info();
    }
}
```

8.4 Multiple Inheritance Using Interfaces

```
public class MultipleInheritanceExample {
    interface Swimmer {
        void swim();
    }

    interface Flyer {
        void fly();
    }

    interface Runner {
        void run();
    }

    // Class implementing multiple interfaces
    static class Duck implements Swimmer, Flyer, Runner {
        @Override
        public void swim() {
            System.out.println("Duck is swimming");
        }

        @Override
        public void fly() {
            System.out.println("Duck is flying");
        }

        @Override
        public void run() {
            System.out.println("Duck is running");
        }
    }

    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.swim();
        duck.fly();
        duck.run();
    }
}
```

8.5 Difference between Abstract Classes and Interfaces

Feature	Abstract Class	Interface
Instantiation	Cannot instantiate	Cannot instantiate
Constructor	Yes, can have	No constructors
Access Modifiers	Any	Public only
Variables	Any type, any modifier	public static final only
Methods	Both abstract and concrete	Abstract methods (now default)
Inheritance	Single inheritance (extends)	Multiple inheritance
Keyword	extends	implements
Use Case	IS-A relationship	Behavior contract

MODULE 9: Exception Handling

9.1 Exception Hierarchy

```
/*
Throwable (root class)
├── Exception
│   ├── IOException
│   ├── SQLException
│   ├── RuntimeException
│   │   ├── NullPointerException
│   │   ├── ArithmeticException
│   │   ├── ArrayIndexOutOfBoundsException
│   │   └── ClassCastException
│   └── CheckedException
└── Error
    ├── OutOfMemoryError
    ├── StackOverflowError
    └── VirtualMachineError
*/

public class ExceptionHierarchyExample {
    public static void main(String[] args) {
        // Checked Exception - must be caught or declared
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Unchecked Exception - RuntimeException
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[10]);  // ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of bounds");
        }

        // ArithmeticException
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero");
        }

        // NullPointerException
        try {
            String str = null;
            System.out.println(str.length());
        } catch (NullPointerException e) {
            System.out.println("String is null");
        }
    }
}
```

9.2 try-catch Blocks

```
public class TryCatchExample {
    public static void main(String[] args) {
        // Single catch block
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Array index out of bounds");
            System.out.println("Message: " + e.getMessage());
            e.printStackTrace();
        }

        // Multiple catch blocks
        try {
            String str = "123a";
            int num = Integer.parseInt(str);
        } catch (NumberFormatException e) {
            System.out.println("Invalid number format");
        } catch (Exception e) {
            System.out.println("General exception occurred");
        }

        // Multi-catch (Java 7+)
        try {
            int x = 10 / 0;
        } catch (ArithmeticException | NullPointerException e) {
            System.out.println("Arithmetic or Null error occurred");
        }
    }
}
```

9.3 try-catch-finally

```
public class TryCatchFinallyExample {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println("Accessing array element: " + arr[2]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            // Always executes, whether exception occurs or not
            System.out.println("Finally block executed");
        }

        // Return in try-catch-finally
        System.out.println("Result: " + testFinally());
    }

    static int testFinally() {
        try {
            int x = 10 / 2;
            return x;
        } catch (Exception e) {
            return -1;
        } finally {
            System.out.println("Finally block is executing");
        }
    }
}

// Output:
// Accessing array element: 3
// Finally block executed
// Finally block is executing
// Result: 5
```

9.4 Throwing Exceptions

```
public class ThrowingExceptionExample {
    // Method throwing checked exception
    static void riskyMethod() throws IOException {
        throw new IOException("This is a checked exception");
    }

    // Method throwing custom exception
    static void checkAge(int age) throws AgeException {
        if (age < 0 || age > 120) {
            throw new AgeException("Invalid age: " + age);
        }
        System.out.println("Age is valid: " + age);
    }

    // Custom exception class
    static class AgeException extends Exception {
        public AgeException(String message) {
            super(message);
        }
    }

    public static void main(String[] args) {
        // Throwing unchecked exception
        try {
            int x = 10;
            if (x > 5) {
                throw new IllegalArgumentException("x should be <= 5");
            }
        } catch (IllegalArgumentException e) {
            System.out.println("Caught: " + e.getMessage());
        }

        // Throwing custom exception
        try {
            checkAge(150);
        } catch (AgeException e) {
            System.out.println("Caught: " + e.getMessage());
        }

        // Declaring throws
        try {
            riskyMethod();
        } catch (IOException e) {
            System.out.println("Caught: " + e.getMessage());
        }
        }
}
```

9.5 Built-in Exceptions

```java
public class BuiltInExceptionsExample {
    public static void main(String[] args) {
        // 1. NullPointerException
        try {
            String str = null;
            int len = str.length();
        } catch (NullPointerException e) {
            System.out.println("1. NullPointerException: Accessing null object");
        }

        // 2. ArrayIndexOutOfBoundsException
        try {
            int[] arr = {1, 2, 3};
            int val = arr[5];
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("2. ArrayIndexOutOfBoundsException: Invalid index");
        }

        // 3. ArithmeticException
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("3. ArithmeticException: Division by zero");
        }

        // 4. NumberFormatException
        try {
            String num = "abc";
            int value = Integer.parseInt(num);
        } catch (NumberFormatException e) {
            System.out.println("4. NumberFormatException: Invalid number format");
        }

        // 5. ClassCastException
        try {
            Object obj = "Hello";
            Integer num = (Integer) obj;
        } catch (ClassCastException e) {
            System.out.println("5. ClassCastException: Invalid type casting");
        }

        // 6. IllegalArgumentException
        try {
            if (true) throw new IllegalArgumentException("Invalid argument");
        } catch (IllegalArgumentException e) {
            System.out.println("6. IllegalArgumentException: Invalid argument passed");
        }

        // 7. StringIndexOutOfBoundsException
        try {
            String str = "Hello";
            char ch = str.charAt(10);
        } catch (StringIndexOutOfBoundsException e) {
            System.out.println("7. StringIndexOutOfBoundsException: Invalid string index");
        }

        // 8. IllegalStateException
        try {
            throw new IllegalStateException("Invalid state");
        } catch (IllegalStateException e) {
            System.out.println("8. IllegalStateException: Object in invalid state");
        }
    }
}
```

9.6 Custom Exceptions

```java
public class CustomExceptionExample {
    // Custom exception class extending Exception
    static class InvalidScoreException extends Exception {
        public InvalidScoreException(String message) {
            super(message);
        }

        public InvalidScoreException(String message, Throwable cause) {
            super(message, cause);
        }
    }

    // Another custom exception
    static class InsufficientBalanceException extends Exception {
        private double shortfall;

        public InsufficientBalanceException(String message, double shortfall) {
            super(message);
            this.shortfall = shortfall;
        }

        public double getShortfall() {
            return shortfall;
        }
    }

    static class Student {
        String name;
        int score;

        public Student(String name, int score) throws InvalidScoreException {
            if (score < 0 || score > 100) {
                throw new InvalidScoreException(
                    "Score must be between 0 and 100. Got: " + score
                );
            }
            this.name = name;
            this.score = score;
        }
    }

    static class BankAccount {
        double balance = 1000;

        void withdraw(double amount) throws InsufficientBalanceException {
            if (amount > balance) {
                double shortfall = amount - balance;
                throw new InsufficientBalanceException(
                    "Insufficient balance. Required: " + amount,
                    shortfall
                );
            }
            balance -= amount;
            System.out.println("Withdrawn: " + amount + ", Balance: " + balance);
        }
    }

    public static void main(String[] args) {
        // Using custom exception 1
        try {
            Student s = new Student("Raj", 105);
        } catch (InvalidScoreException e) {
            System.out.println("Exception: " + e.getMessage());
        }

        // Using custom exception 2
        try {
            BankAccount account = new BankAccount();
            account.withdraw(1500);
        } catch (InsufficientBalanceException e) {
            System.out.println("Exception: " + e.getMessage());
            System.out.println("Shortfall: " + e.getShortfall());
        }
    }
}
```

MODULE 10: Input/Output (I/O) Streams

10.1 Stream Basics

```
import java.io.*;

public class StreamBasicsExample {
    public static void main(String[] args) {
        /*
        Stream Hierarchy:

        InputStream (Byte)           Reader (Character)
        ├── FileInputStream          ├── FileReader
        ├── ByteArrayInputStream     ├── CharArrayReader
        └── BufferedInputStream      └── BufferedReader

        OutputStream (Byte)          Writer (Character)
        ├── FileOutputStream         ├── FileWriter
        ├── ByteArrayOutputStream    ├── CharArrayWriter
        └── BufferedOutputStream     └── BufferedWriter
        */
    }
}
```

10.2 Reading Input

```java
import java.io.*;
import java.util.Scanner;

public class ReadingInputExample {
    public static void main(String[] args) {
        // 1. Using Scanner
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter your name: ");
        String name = scanner.nextLine();

        System.out.print("Enter your age: ");
        int age = scanner.nextInt();

        System.out.println("Name: " + name + ", Age: " + age);

        // 2. Using BufferedReader
        try {
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(System.in)
            );

            System.out.print("Enter input: ");
            String input = reader.readLine();
            System.out.println("You entered: " + input);

        } catch (IOException e) {
            e.printStackTrace();
        }

        scanner.close();
    }
}
```

10.3 Reading from Files

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

public class ReadingFromFileExample {
    public static void main(String[] args) {
        // Method 1: Using FileReader and BufferedReader
        try (BufferedReader reader = new BufferedReader(
                new FileReader("input.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 2: Using FileInputStream
        try (FileInputStream fis = new FileInputStream("input.txt")) {
            int data;
            while ((data = fis.read()) != -1) {
                System.out.print((char) data);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 3: Using NIO (Modern approach)
        try {
            String content = new String(Files.readAllBytes(
                Paths.get("input.txt")
            ));
            System.out.println(content);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 4: Reading all lines
        try {
            java.util.List<String> lines = Files.readAllLines(
                Paths.get("input.txt")
            );
            for (String line : lines) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
                }
        }
}
```

10.4 Writing to Files

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

public class WritingToFileExample {
    public static void main(String[] args) {
        // Method 1: Using FileWriter and BufferedWriter
        try (BufferedWriter writer = new BufferedWriter(
                new FileWriter("output.txt"))) {
            writer.write("Hello, File!");
            writer.newLine();
            writer.write("This is a test file");
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 2: Using FileOutputStream
        try (FileOutputStream fos = new FileOutputStream("output2.txt")) {
            String text = "Writing bytes to file";
            fos.write(text.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 3: Using NIO (Append mode)
        try {
            Files.write(
                Paths.get("output3.txt"),
                "New content".getBytes(),
                StandardOpenOption.CREATE,
                StandardOpenOption.APPEND
            );
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Method 4: Using PrintWriter
        try (PrintWriter writer = new PrintWriter("output4.txt")) {
            writer.println("Line 1");
            writer.println("Line 2");
            writer.printf("Number: %d\n", 42);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

10.5 File Operations

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

public class FileOperationsExample {
    public static void main(String[] args) {
        File file = new File("test.txt");

        // Checking file properties
        System.out.println("File exists: " + file.exists());
        System.out.println("Is file: " + file.isFile());
        System.out.println("Is directory: " + file.isDirectory());
        System.out.println("Can read: " + file.canRead());
        System.out.println("Can write: " + file.canWrite());
        System.out.println("File size: " + file.length() + " bytes");
        System.out.println("Absolute path: " + file.getAbsolutePath());

        // Creating file
        try {
            if (file.createNewFile()) {
                System.out.println("File created successfully");
            } else {
                System.out.println("File already exists");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deleting file
        if (file.delete()) {
            System.out.println("File deleted successfully");
        }

        // Directory operations
        File dir = new File("myDirectory");
        if (dir.mkdir()) {
            System.out.println("Directory created");
        }

        // List files in directory
        File current = new File(".");
        File[] files = current.listFiles();
        if (files != null) {
            for (File f : files) {
                System.out.println(f.getName());
            }
        }
        }
}
```

10.6 Serialization

```java
import java.io.*;

public class SerializationExample {
    static class Student implements Serializable {
        private static final long serialVersionUID = 1L;

        String name;
        int roll;
        transient double cgpa;  // Not serialized

        public Student(String name, int roll, double cgpa) {
            this.name = name;
            this.roll = roll;
            this.cgpa = cgpa;
        }

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

    public static void main(String[] args) {
        // Serialization
        try (ObjectOutputStream oos = new ObjectOutputStream(
                new FileOutputStream("student.ser"))) {
            Student s = new Student("Raj", 101, 8.5);
            oos.writeObject(s);
            System.out.println("Object serialized: " + s);
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Deserialization
        try (ObjectInputStream ois = new ObjectInputStream(
                new FileInputStream("student.ser"))) {
            Student s = (Student) ois.readObject();
            System.out.println("Object deserialized: " + s);
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

MODULE 11: Multithreading and Concurrency

11.1 Thread Creation

11.1.1 Extending Thread Class

```
public class ThreadCreationExample {
    // Method 1: Extending Thread class
    static class MyThread extends Thread {
        String name;

        public MyThread(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println(name + " - " + i);
                try {
                    Thread.sleep(1000);  // Sleep for 1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread-1");
        MyThread t2 = new MyThread("Thread-2");

        t1.start();  // Start thread (calls run())
        t2.start();
    }
}
```

11.1.2 Implementing Runnable Interface

```java
public class RunnableExample {
    // Method 2: Implementing Runnable interface
    static class MyRunnable implements Runnable {
        String name;

        public MyRunnable(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println(name + " - " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        // Create threads with Runnable
        Thread t1 = new Thread(new MyRunnable("Thread-1"));
        Thread t2 = new Thread(new MyRunnable("Thread-2"));

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

// Lambda expression (Java 8+)
public class LambdaThreadExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread-1: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread-2: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

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

11.2 Thread Lifecycle

```java
public class ThreadLifecycleExample {
    public static void main(String[] args) throws InterruptedException {
        /*
        Thread States:
        1. NEW - Thread created but not started
        2. RUNNABLE - Thread running or ready to run
        3. BLOCKED - Thread waiting for monitor lock
        4. WAITING - Thread waiting indefinitely
        5. TIMED_WAITING - Thread waiting for specific time
        6. TERMINATED - Thread execution completed
        */

        Thread t = new Thread(() -> {
            System.out.println("Thread running");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Thread completed");
        });

        System.out.println("State before start: " + t.getState());  // NEW

        t.start();
        System.out.println("State after start: " + t.getState());   // RUNNABLE

        t.join();  // Wait for thread to complete
        System.out.println("State after completion: " + t.getState()); // TERMINATED
    }
}
```

11.3 Thread Priorities

```
public class ThreadPriorityExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Low Priority: " + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("High Priority: " + i);
            }
        });

        t1.setPriority(Thread.MIN_PRIORITY);    // Priority 1
        t2.setPriority(Thread.MAX_PRIORITY);    // Priority 10

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

11.4 Synchronization

```java
public class SynchronizationExample {
    static class Counter {
        int count = 0;

        // Synchronized method
        synchronized void increment() {
            count++;
        }

        synchronized int getCount() {
            return count;
        }
    }

    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());  // 2000
    }
}

// Synchronized block
public class SynchronizedBlockExample {
    static class Counter {
        int count = 0;

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

11.5 Inter-thread Communication

```java
public class InterThreadCommunicationExample {
    static class SharedResource {
        private int data = 0;
        private boolean hasData = false;

        synchronized void produce(int value) throws InterruptedException {
            while (hasData) {
                wait();  // Wait if data is not consumed
            }
            data = value;
            hasData = true;
            System.out.println("Produced: " + data);
            notify();  // Notify waiting thread
        }

        synchronized int consume() throws InterruptedException {
            while (!hasData) {
                wait();  // Wait if data is not produced
            }
            hasData = false;
            System.out.println("Consumed: " + data);
            notify();  // Notify waiting thread
            return data;
        }
    }

    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    resource.produce(i);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 0; i < 5; i++) {
                    resource.consume();
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

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

11.6 Thread Pools and Executors

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

public class ExecutorExample {
    public static void main(String[] args) {
        // Create thread pool with 3 threads
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks
        for (int i = 1; i <= 5; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println("Task " + taskId + " running on " +
                        Thread.currentThread().getName());
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Task " + taskId + " completed");
            });
        }

        // Shutdown executor
        executor.shutdown();
        try {
            if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
                executor.shutdownNow();
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }
    }
}

// Callable and Future
public class CallableFutureExample {
    static class Task implements Callable<Integer> {
        int num;

        public Task(int num) {
            this.num = num;
        }

        @Override
        public Integer call() throws Exception {
            Thread.sleep(1000);
            return num * num;
        }
    }

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Future<Integer> future1 = executor.submit(new Task(5));
        Future<Integer> future2 = executor.submit(new Task(10));

        try {
            int result1 = future1.get();  // Waits for result
            int result2 = future2.get();

            System.out.println("Result 1: " + result1);
            System.out.println("Result 2: " + result2);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }

        executor.shutdown();
    }
}
```

MODULE 12: Advanced Concepts

12.1 Enums (IMPORTANT - Hidden Concept)

Definition: An enum (enumeration) is a special data type in Java that allows you to define a set of named constants. It's used when you know all possible values for a variable at compile time.

```java
// Simple Enum
public class EnumExample {
    // Enum definition
    enum Color {
        RED, GREEN, BLUE, YELLOW
    }

    public static void main(String[] args) {
        // Using enum
        Color color = Color.RED;
        System.out.println("Selected color: " + color);

        // Switch with enum
        switch (color) {
            case RED:
                System.out.println("Color is red");
                break;
            case GREEN:
                System.out.println("Color is green");
                break;
            case BLUE:
                System.out.println("Color is blue");
                break;
            case YELLOW:
                System.out.println("Color is yellow");
                break;
        }

        // Iterating through enum
        System.out.println("\nAll colors:");
        for (Color c : Color.values()) {
            System.out.println(c);
        }

        // Getting enum by name
        Color colorFromString = Color.valueOf("GREEN");
        System.out.println("Color from string: " + colorFromString);

        // Enum ordinal
        System.out.println("RED ordinal: " + Color.RED.ordinal());  // 0
        System.out.println("BLUE ordinal: " + Color.BLUE.ordinal());  // 2
    }
}
```

12.1.2 Enum with Properties and Methods

```java
public class EnumWithPropertiesExample {
    // Enum with properties
    enum Size {
        SMALL(1, "$10"),
        MEDIUM(2, "$15"),
        LARGE(3, "$20"),
        XLARGE(4, "$25");

        private int code;
        private String price;

        // Constructor (always private)
        Size(int code, String price) {
            this.code = code;
            this.price = price;
        }

        // Getter methods
        public int getCode() {
            return code;
        }

        public String getPrice() {
            return price;
        }

        // Custom method
        public void displaySize() {
            System.out.println("Size: " + this.name() +
                             ", Code: " + code +
                             ", Price: " + price);
        }
    }

    public static void main(String[] args) {
        // Accessing enum properties
        Size size = Size.LARGE;
        System.out.println("Selected Size: " + size);
        System.out.println("Code: " + size.getCode());
        System.out.println("Price: " + size.getPrice());

        // Display all sizes
        System.out.println("\nAll Sizes:");
        for (Size s : Size.values()) {
            s.displaySize();
        }
    }
}

// Output:
// Selected Size: LARGE
// Code: 3
// Price: $20
//
// All Sizes:
// Size: SMALL, Code: 1, Price: $10
// Size: MEDIUM, Code: 2, Price: $15
// Size: LARGE, Code: 3, Price: $20
// Size: XLARGE, Code: 4, Price: $25
```

12.1.3 Enum Implementing Interface

```
public class EnumInterfaceExample {
    interface Operation {
        int perform(int a, int b);
    }

    enum Calculator implements Operation {
        ADD("+") {
            @Override
            public int perform(int a, int b) {
                return a + b;
            }
        },
        SUBTRACT("-") {
            @Override
            public int perform(int a, int b) {
                return a - b;
            }
        },
        MULTIPLY("*") {
            @Override
            public int perform(int a, int b) {
                return a * b;
            }
        },
        DIVIDE("/") {
            @Override
            public int perform(int a, int b) {
                return a / b;
            }
        };

        private String symbol;

        Calculator(String symbol) {
            this.symbol = symbol;
        }

        public String getSymbol() {
            return symbol;
        }
    }

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

        for (Calculator calc : Calculator.values()) {
            int result = calc.perform(a, b);
            System.out.println(a + " " + calc.getSymbol() + " " +
                              b + " = " + result);
        }
    }
}

// Output:
// 10 + 5 = 15
// 10 - 5 = 5
// 10 * 5 = 50
// 10 / 5 = 2
```

12.1.4 Enum Comparision and Usage Patterns

```
public class EnumComparisonExample {
    enum Status {
        PENDING("Pending"),
        PROCESSING("Processing"),
        COMPLETED("Completed"),
        FAILED("Failed");

        private String description;

        Status(String description) {
            this.description = description;
        }

        public String getDescription() {
            return description;
        }
    }

    public static void main(String[] args) {
        Status status = Status.PROCESSING;

        // Comparing enums (equality)
        if (status == Status.PROCESSING) {
            System.out.println("Status is processing");
        }

        // Using in conditional
        if (status.ordinal() > Status.PENDING.ordinal()) {
            System.out.println("Status is beyond pending");
        }

        // HashMap with enum
        java.util.Map<Status, String> statusMessages = new java.util.HashMap<>();
        statusMessages.put(Status.PENDING, "Please wait...");
        statusMessages.put(Status.COMPLETED, "Task completed!");
        statusMessages.put(Status.FAILED, "Task failed!");

        System.out.println("Message: " + statusMessages.get(status));
    }
}
```

Key Points about Enums:

1. Enums are a special type of class in Java
2. All enum constants are implicitly public static final
3. Enum constructor is always private

4. Can implement interfaces
5. Cannot extend classes (but can extend another enum)
6. Provides type safety
7. Can use in switch statements
8. Each enum constant is an instance of the enum

12.2 Varargs (Variable Arguments)

```
public class VarargsExample {
    // Varargs method
    static int sum(int... numbers) {
        int total = 0;
        for (int num : numbers) {
            total += num;
        }
        return total;
    }

    // Varargs with other parameters
    static void printDetails(String name, int... scores) {
        System.out.println("Student: " + name);
        System.out.println("Scores: " + java.util.Arrays.toString(scores));
    }

    public static void main(String[] args) {
        // Calling with different number of arguments
        System.out.println("Sum: " + sum(5, 10));
        System.out.println("Sum: " + sum(1, 2, 3, 4, 5));
        System.out.println("Sum: " + sum(100));

        // Array of elements
        System.out.println("Sum: " + sum(new int[]{10, 20, 30}));

        // With other parameters
        printDetails("Raj", 85, 90, 78);
        printDetails("Priya", 92, 88, 95, 89);
    }
}
```

12.3 Generics

```
// Generic class
public class GenericsExample {
    static class Box<T> {
        private T value;

        public void setValue(T value) {
            this.value = value;
        }

        public T getValue() {
            return value;
        }
    }

    // Generic method
    static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    // Generic with wildcard
    static void printList(java.util.List<?> list) {
        for (Object obj : list) {
            System.out.print(obj + " ");
        }
        System.out.println();
    }

    // Bounded generics
    static <T extends Comparable<T>> T findMax(T[] array) {
        if (array == null || array.length == 0) return null;
        T max = array[0];
        for (T element : array) {
            if (element.compareTo(max) > 0) {
                max = element;
            }
        }
        return max;
    }

    public static void main(String[] args) {
        // Type-safe
        Box<String> stringBox = new Box<>();
        stringBox.setValue("Hello");
        System.out.println("String Box: " + stringBox.getValue());

        Box<Integer> intBox = new Box<>();
        intBox.setValue(42);
        System.out.println("Integer Box: " + intBox.getValue());

        // Generic method
        Integer[] intArray = {5, 2, 9, 1, 7};
        printArray(intArray);

        String[] strArray = {"Hello", "World", "Java"};
        printArray(strArray);

        // Bounded generics
        Integer max = findMax(intArray);
        System.out.println("Maximum: " + max);
    }
}
```

12.4 Annotations

```java
public class AnnotationsExample {
    // Custom annotation
    @interface MyAnnotation {
        String author() default "Unknown";
        String date();
        int version() default 1;
    }

    // Using annotation
    @MyAnnotation(author = "Raj", date = "2025-12-20")
    public class MyClass {
        @Deprecated
        void oldMethod() {
            System.out.println("This method is deprecated");
        }

        @Override
        public String toString() {
            return "MyClass";
        }

        @SuppressWarnings("unchecked")
        void suppressWarning() {
            // Code with warnings
        }
    }
}

// Common Annotations
public class CommonAnnotationsExample {
    class Parent {
        void display() {}
    }

    class Child extends Parent {
        @Override  // Annotation to override parent method
        void display() {
            System.out.println("Child display");
        }

        @Deprecated  // Annotation indicating deprecated method
        void oldMethod() {}

        @SuppressWarnings("deprecation")  // Suppress warnings
        void useOldMethod() {
            oldMethod();
```

```
                }
        }
}
```

12.5 Packages

```
/*
Package structure:
project/
├── src/
|   ├── com/
|   |   └── example/
|   |       ├── util/
|   |       |   └── Utils.java
|   |       └── model/
|   |           └── Student.java
*/

// File: com/example/util/Utils.java
package com.example.util;

public class Utils {
    public static int add(int a, int b) {
        return a + b;
    }
}

// File: com/example/model/Student.java
package com.example.model;

public class Student {
    String name;
    int roll;
}

// File: Main.java
import com.example.util.Utils;
import com.example.model.Student;

public class Main {
    public static void main(String[] args) {
        int result = Utils.add(5, 10);
        System.out.println("Sum: " + result);

        Student student = new Student();
    }
}
```

12.6 Wrapper Classes

```
public class WrapperClassesExample {
    public static void main(String[] args) {
        // Boxing - primitive to wrapper
        int primitiveInt = 10;
        Integer boxedInt = Integer.valueOf(primitiveInt);  // Manual boxing
        Integer autoBoxedInt = primitiveInt;                // Auto-boxing

        // Unboxing - wrapper to primitive
        Integer wrappedInt = 20;
        int unboxedInt = wrappedInt.intValue();  // Manual unboxing
        int autoUnboxedInt = wrappedInt;         // Auto-unboxing

        // Wrapper methods
        String str = "42";
        int num = Integer.parseInt(str);
        System.out.println("Parsed integer: " + num);

        // To string
        String strFromInt = Integer.toString(100);
        System.out.println("String from int: " + strFromInt);

        // Other wrapper classes
        Double d = 3.14;
        Boolean b = true;
        Character c = 'A';

        System.out.println("Double: " + d);
        System.out.println("Boolean: " + b);
        System.out.println("Character: " + c);

        // Constants
        System.out.println("Integer MIN: " + Integer.MIN_VALUE);
        System.out.println("Integer MAX: " + Integer.MAX_VALUE);
        System.out.println("Double MAX: " + Double.MAX_VALUE);
    }
}
```

12.7 Type Casting

```java
public class TypeCastingExample {
    public static void main(String[] args) {
        // Implicit casting (widening)
        int intVal = 100;
        long longVal = intVal;      // int to long
        float floatVal = longVal;   // long to float
        double doubleVal = floatVal; // float to double

        // Explicit casting (narrowing)
        double d = 10.5;
        int i = (int) d;            // double to int (loses decimal)
        System.out.println("Double to int: " + i);  // 10

        // Casting objects
        Object obj = "Hello";
        String str = (String) obj;
        System.out.println("String cast: " + str);

        // Checking type before casting
        if (obj instanceof String) {
            String castedStr = (String) obj;
            System.out.println("Safe casting: " + castedStr);
        }
    }
}
```

Summary of Key Concepts

This comprehensive guide covers:

1. **OOP Principles**: Abstraction, Encapsulation, Inheritance, Polymorphism
2. **Java Basics**: Data types, variables, operators, control flow
3. **Arrays**: 1D arrays, 2D arrays, and **Jagged arrays**
4. **Classes & Objects**: Constructors, methods, access specifiers
5. **Advanced Features**: Static members, final keyword, inheritance
6. **Abstract Classes & Interfaces**: Multiple inheritance, contract definitions
7. **Exception Handling**: try-catch-finally, custom exceptions
8. **I/O Streams**: File reading/writing, serialization
9. **Multithreading**: Thread creation, synchronization, concurrency
10. **Advanced Concepts**: **Enums**, Generics, Annotations, Varargs

Important Hidden Concepts Explained:

- **Jagged Arrays**: Multi-dimensional arrays with varying row sizes
- **Enums**: Type-safe way to define a set of named constants

- **Static Blocks**: Code execution when class loads
- **Synchronized Methods**: Thread-safe operations
- **Generics**: Type-safe collections and methods

All examples are production-ready and suitable for lab exams!