Java Complete Guide: From Basic to Advanced

Table of Contents

1.1 Introduction to Java

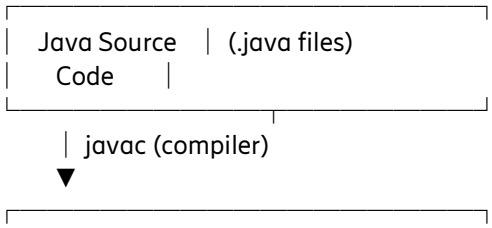
1.1.1 What is Java?

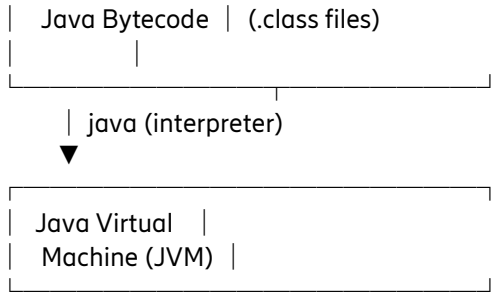
Java is a **high-level**, **object-oriented**, **platform-independent** programming language developed by Sun Microsystems (now Oracle) in 1995.

1.1.2 Key Features

Feature	Description
Platform Independent	"Write Once, Run Anywhere" (WORA)
Object-Oriented	Everything is an object
Secure	Built-in security features
Robust	Strong memory management
Multithreaded	Concurrent programming support
Interpreted	Bytecode execution via JVM

1.1.3 Java Ecosystem





1.2 Setting Up Development Environment

1.2.1 JDK Installation

1. **Download JDK** from Oracle or OpenJDK
2. **Set JAVA_HOME** environment variable
3. **Add ssPATH** for command-line access

1.2.2 IDE Options

IDE	Best For	Features
IntelliJ IDEA	Professional Development	Advanced debugging, refactoring
Eclipse	Enterprise Applications	Plugin ecosystem
VS Code	Lightweight Development	Extensions, Git integration

IDE	Best For	Features
NetBeans	Beginners	Simple interface

1.2.3 First Program

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

Compilation & Execution:

```
javacs HelloWorld.java  # Compile
java HelloWorld       # Execute
```

1.3 Basic Syntax and Structure

1.3.1 Java Program Structure

```
// Package declaration (optional)
package com.example;

// Import statements
import java.util.Scanner;

// Class declaration
```

```java
public class MyClass {
  // Class variables (fields)
  private int value;

  // Constructor
  public MyClass(int value) {
    this.value = value;
  }

  // Methods
  public void display() {
    System.out.println("Value: " + value);
  }

  // Main method (entry point)
  public static void main(String[] args) {
    MyClass obj = new MyClass(10);
    obj.display();
  }
}
```

1.3.2 Naming Conventions

Element	Convention	Example
Class	PascalCase	StudentRecord
Method	camelCase	calculateTotal()
Variable	camelCase	firstName
Constant	UPPER_SNAKE_CASE	MAX_SIZE

Element	Convention	Example
Package	lowercase	com.ericsson.project

1.3.3 Comments

```
// Single-line comment

/*
 * Multi-line comment
 * Used for detailed explanations
 */

/**
 * JavaDoc comment
 * @param name The user's name
 * @return Greeting message
 */
public String greet(String name) {
   return "Hello, " + name;
}
```

1.4 Variables and Data Types

1.4.1 Primitive Data Types

Type	Size	Range	Default	Example
byte	8 bits	-128 to 127	0	byte b = 100;
short	16 bits	-32,768 to 32,767	0	short s = 1000;
int	32 bits	-2^{31} to $2^{31}-1$	0	int i = 100000;
long	64 bits	-2^{63} to $2^{63}-1$	0L	long l = 100000L;
float	32 bits	IEEE 754	0.0f	float f = 3.14f;
double	64 bits	IEEE 754	0.0d	double d = 3.14159;
char	16 bits	0 to 65,535	'000'	char c = 'A';
boolean	1 bit	true/false	false	boolean flag = true;

1.4.2 Variable Declaration

```
// Declaration
int number;

// Initialization
number = 42;

// Declaration + Initialization
int count = 10;

// Multiple variables
```

```
int a = 1, b = 2, c = 3;

// Constants
final double PI = 3.14159;
```

1.4.3 Type Conversion

```
// Implicit (Widening)
int i = 100;
long l = i;      // int to long
double d = l;    // long to double

// Explicit (Narrowing)
double d = 9.78;
int i = (int) d;  // 9 (truncated)

// Wrapper Classes
Integer intObj = Integer.valueOf(42);
int primitive = intObj.intValue();

// Autoboxing/Unboxing
Integer autoBox = 42;      // Autoboxing
int autoUnbox = autoBox;   // Unboxing
```

1.5 Operators

1.5.1 Arithmetic Operators

```
int a = 10, b = 3;

int sum = a + b;       // 13
int diff = a - b;      // 7
int product = a * b;   // 30
int quotient = a / b;  // 3
int remainder = a % b; // 1

// Unary operators
int x = 5;
x++;   // Post-increment: x = 6
++x;   // Pre-increment: x = 7
x--;   // Post-decrement: x = 6
--x;   // Pre-decrement: x = 5
```

1.5.2 Comparison Operators

```
int a = 10, b = 20;

boolean equal = (a == b);      // false
boolean notEqual = (a != b);   // true
boolean greater = (a > b);     // false
boolean less = (a < b);        // true
boolean greaterEqual = (a >= b); // false
boolean lessEqual = (a <= b);  // true
```

1.5.3 Logical Operators

```java
boolean x = true, y = false;

boolean and = x && y;   // false (AND)
boolean or = x || y;    // true (OR)
boolean not = !x;       // false (NOT)

// Short-circuit evaluation
boolean result = (x != null) && (x.length() > 0);
```

1.5.4 Bitwise Operators

```java
int a = 5;  // 101 in binary
int b = 3;  // 011 in binary

int and = a & b;   // 1 (001)
int or = a | b;    // 7 (111)
int xor = a ^ b;   // 6 (110)
int complement = ~a; // -6 (inverted bits)
int leftShift = a << 1;  // 10 (1010)
int rightShift = a >> 1; // 2 (010)
```

1.5.5 Assignment Operators

```java
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
```

1.5.6 Ternary Operator

```
int a = 10, b = 20;
int max = (a > b) ? a : b;  // 20

String status = (age >= 18) ? "Adult" : "Minor";
```

1.6 Control Flow Statements

1.6.1 Conditional Statements

1.6.1.1 if-else Statement

```
int score = 85;

if (score >= 90) {
   System.out.println("Grade A");
} else if (score >= 80) {
   System.out.println("Grade B");
} else if (score >= 70) {
   System.out.println("Grade C");
} else {
   System.out.println("Grade F");
}
```

1.6.1.2 switch Statement

```java
// Traditional switch
int day = 3;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  default:
    System.out.println("Invalid day");
}

// Enhanced switch (Java 14+)
String dayName = switch (day) {
  case 1 -> "Monday";
  case 2 -> "Tuesday";
  case 3 -> "Wednesday";
  default -> "Invalid day";
};
```

1.6.2 Loop Statements

1.6.2.1 for Loop

```java
// Basic for loop
for (int i = 0; i < 5; i++) {
   System.out.println("Count: " + i);
}

// Enhanced for loop (for-each)
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
   System.out.println(num);
}
```

1.6.2.2 while Loop

```java
int count = 0;
while (count < 5) {
   System.out.println("Count: " + count);
   count++;
}
```

1.6.2.3 do-while Loop

```java
int num;
do {
   num = scanner.nextInt();
   System.out.println("You entered: " + num);
} while (num != 0);
```

1.6.3 Jump Statements

```java
// break statement
for (int i = 0; i < 10; i++) {
  if (i == 5) {
    break; // Exit loop when i equals 5
  }
  System.out.println(i);
}

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

// Labeled break/continue
outer: for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    if (i == 1 && j == 1) {
      break outer; // Break out of both loops
    }
    System.out.println(i + "," + j);
  }
}
```

1.7 Methods

1.7.1 Method Declaration

```
// Method syntax
[access_modifier] [static] return_type method_name(parameters) {
  // Method body
  return value; // if return_type is not void
}
```

1.7.2 Method Examples

```
public class Calculator {

  // Method with no parameters and no return value
  public void displayWelcome() {
    System.out.println("Welcome to Calculator!");
  }

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

  // Method with multiple parameters
  public double calculateArea(double length, double width) {
    return length * width;
  }

  // Static method
```

```
  public static int multiply(int x, int y) {
    return x * y;
  }

  // Method overloading
  public int add(int a, int b) {
    return a + b;
  }

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

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

1.7.3 Variable Arguments (Varargs)

```
public class VarargsExample {

  // Method with variable arguments
  public int sum(int... numbers) {
    int total = 0;
    for (int num : numbers) {
      total += num;
    }
    return total;
  }
```

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

    System.out.println(obj.sum(1, 2));       // 3
    System.out.println(obj.sum(1, 2, 3, 4));   // 10
    System.out.println(obj.sum());           // 0
  }
}
```

1.7.4 Method Call Stack

```
public class StackExample {

  public static void methodA() {
    System.out.println("Method A");
    methodB();
  }

  public static void methodB() {
    System.out.println("Method B");
    methodC();
  }

  public static void methodC() {
    System.out.println("Method C");
  }

  public static void main(String[] args) {
    methodA();  // Call stack: main -> methodA -> methodB -> methodC
  }
}
```

1.8 Arrays

1.8.1 Array Declaration and Initialization

```
// Declaration
int[] numbers;
int numbers[];  // Alternative syntax

// Initialization
numbers = new int[5];  // Array of size 5

// Declaration + Initialization
int[] scores = new int[10];

// Array literal
int[] values = {1, 2, 3, 4, 5};
String[] names = {"Alice", "Bob", "Charlie"};

// Using new keyword with values
int[] data = new int[]{10, 20, 30, 40};
```

1.8.2 Array Operations

```
public class ArrayOperations {

  public static void main(String[] args) {
    int[] numbers = {5, 2, 8, 1, 9, 3};
```

```
    // Accessing elements
    System.out.println("First element: " + numbers[0]);
    System.out.println("Array length: " + numbers.length);

    // Modifying elements
    numbers[0] = 10;

    // Iterating through array
    for (int i = 0; i < numbers.length; i++) {
        System.out.println("Index " + i + ": " + numbers[i]);
    }

    // Enhanced for loop
    for (int num : numbers) {
        System.out.println(num);
    }

    // Finding maximum
    int max = findMax(numbers);
    System.out.println("Maximum: " + max);
}

public static int findMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
```

```
  }
}
```

1.8.3 Multidimensional Arrays

```
public class MultiDimensionalArrays {

  public static void main(String[] args) {
    // 2D Array declaration
    int[][] matrix = new int[3][4];  // 3 rows, 4 columns

    // 2D Array initialization
    int[][] grid = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
    };

    // Accessing 2D array elements
    System.out.println(grid[1][2]);  // Output: 6

    // Iterating through 2D array
    for (int i = 0; i < grid.length; i++) {
      for (int j = 0; j < grid[i].length; j++) {
        System.out.print(grid[i][j] + " ");
      }
      System.out.println();
    }

    // Enhanced for loop for 2D array
    for (int[] row : grid) {
```

```
        for (int element : row) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    // Jagged arrays (arrays of different lengths)
    int[][] jaggedArray = {
        {1, 2},
        {3, 4, 5, 6},
        {7, 8, 9}
    };
  }
}
```

1.8.4 Array Utility Methods

```
import java.util.Arrays;

public class ArrayUtilities {

  public static void main(String[] args) {
    int[] numbers = {5, 2, 8, 1, 9, 3};

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

    // Binary search (array must be sorted)
    int index = Arrays.binarySearch(numbers, 5);
    System.out.println("Index of 5: " + index);
```

```
    // Copying arrays
    int[] copy = Arrays.copyOf(numbers, numbers.length);
    int[] partialCopy = Arrays.copyOfRange(numbers, 1, 4);

    // Filling array
    int[] filled = new int[5];
    Arrays.fill(filled, 42);

    // Comparing arrays
    boolean equal = Arrays.equals(numbers, copy);

    // Converting to string
    System.out.println(Arrays.toString(numbers));
  }
}
```

1.9 Classes and Objects

1.9.1 Class Definition

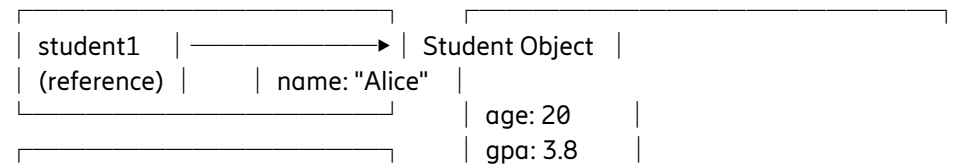
```
public class Student {
  // Instance variables (fields)
  private String name;
  private int age;
  private double gpa;

  // Class variable (static)
  private static int totalStudents = 0;
```

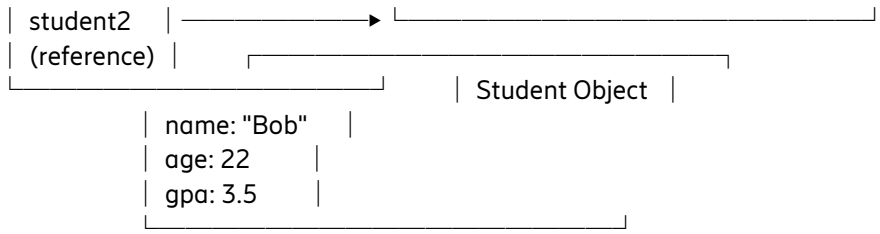
```
    // Constructor
    public Student(String name, int age, double gpa) {
        this.name = name;
        this.age = age;
        this.gpa = gpa;
        totalStudents++;
    }

    // Instance methods
    public void study() {
        System.out.println(name + " is studying.");
    }

    public void displayInfo() {
        System.out.println("Name: " + name + ", Age: " + age + ", GPA: " + gpa);
    }

    // Getters and Setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) {
        if (age > 0) this.age = age;
    }

    // Static method
    public static int getTotalStudents() {
        return totalStudents;
    }
}
```

1.9.2 Object Creation and Usage

```
public class StudentDemo {
    public static void main(String[] args) {
        // Creating objects
        Student student1 = new Student("Alice", 20, 3.8);
        Student student2 = new Student("Bob", 22, 3.5);

        // Using objects
        student1.study();
        student1.displayInfo();

        // Accessing static members
        System.out.println("Total students: " + Student.getTotalStudents());

        // Object reference
        Student student3 = student1;  // Both refer to same object
        student3.setName("Alice Smith");
        System.out.println(student1.getName());  // "Alice Smith"
    }
}
```

1.9.3 Memory Allocation





1.10 Constructors

1.10.1 Types of Constructors

```java
public class Car {
  private String brand;
  private String model;
  private int year;
  private double price;

  // Default constructor
  public Car() {
    this.brand = "Unknown";
    this.model = "Unknown";
    this.year = 2023;
    this.price = 0.0;
  }

  // Parameterized constructor
  public Car(String brand, String model, int year, double price) {
```

```
        this.brand = brand;
        this.model = model;
        this.year = year;
        this.price = price;
    }

    // Constructor overloading
    public Car(String brand, String model) {
        this(brand, model, 2023, 0.0);  // Constructor chaining
    }

    public Car(String brand, String model, int year) {
        this(brand, model, year, 0.0);
    }

    // Copy constructor
    public Car(Car other) {
        this.brand = other.brand;
        this.model = other.model;
        this.year = other.year;
        this.price = other.price;
    }

    public void displayInfo() {
        System.out.println(brand + " " + model + " (" + year + ") - $" + price);
    }
}
```

1.10.2 Constructor Usage

```
public class CarDemo {
  public static void main(String[] args) {
    // Using different constructors
    Car car1 = new Car(); // Default constructor
    Car car2 = new Car("Toyota", "Camry", 2022, 25000); // Parameterized
    Car car3 = new Car("Honda", "Civic"); // Partial parameters
    Car car4 = new Car(car2); // Copy constructor

    car1.displayInfo(); // Unknown Unknown (2023) - $0.0
    car2.displayInfo(); // Toyota Camry (2022) - $25000.0
    car3.displayInfo(); // Honda Civic (2023) - $0.0
    car4.displayInfo(); // Toyota Camry (2022) - $25000.0
  }
}
```

1.11 Inheritance

1.11.1 Basic Inheritance

```
// Base class (Parent/Superclass)
public class Animal {
  protected String name;
  protected int age;

  public Animal(String name, int age) {
    this.name = name;
```

```
        this.age = age;
    }

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

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

    public void makeSound() {
        System.out.println(name + " makes a sound.");
    }
}

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

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

    // Method overriding
    @Override
    public void makeSound() {
        System.out.println(name + " barks: Woof! Woof!");
    }
```

```
    // Additional method specific to Dog
    public void wagTail() {
        System.out.println(name + " is wagging tail.");
    }

    public void fetch() {
        System.out.println(name + " is fetching the ball.");
    }
}

public class Cat extends Animal {
    private boolean isIndoor;

    public Cat(String name, int age, boolean isIndoor) {
        super(name, age);
        this.isIndoor = isIndoor;
    }

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

    public void climb() {
        System.out.println(name + " is climbing.");
    }
}
```

1.11.2 Inheritance Demo

```java
public class InheritanceDemo {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy", 3, "Golden Retriever");
        Cat cat = new Cat("Whiskers", 2, true);

        // Inherited methods
        dog.eat();
        dog.sleep();

        // Overridden methods
        dog.makeSound();  // Barks
        cat.makeSound();  // Meows

        // Specific methods
        dog.wagTail();
        dog.fetch();
        cat.climb();

        // Polymorphic behavior
        Animal[] animals = {dog, cat};
        for (Animal animal : animals) {
            animal.makeSound();  // Calls overridden methods
        }
    }
}
```

1.11.3 Types of Inheritance

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

// Multilevel Inheritance
class Vehicle { }
class Car extends Vehicle { }
class SportsCar extends Car { }

// Hierarchical Inheritance
class Shape { }
class Circle extends Shape { }
class Rectangle extends Shape { }
class Triangle extends Shape { }

// Note: Java doesn't support multiple inheritance of classes
// But supports multiple inheritance through interfaces
```

1.12 Polymorphism

1.12.1 Method Overriding (Runtime Polymorphism)

```
public class Shape {
  protected double area;

  public void calculateArea() {
```

```
        System.out.println("Calculating area of generic shape");
    }

    public void display() {
        System.out.println("Area: " + area);
    }
}

public class Circle extends Shape {
    private double radius;

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

    @Override
    public void calculateArea() {
        area = Math.PI * radius * radius;
        System.out.println("Calculating area of circle");
    }
}

public class Rectangle extends Shape {
    private double length, width;

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

    @Override
```

```
    public void calculateArea() {
        area = length * width;
        System.out.println("Calculating area of rectangle");
    }
}
```

1.12.2 Method Overloading (Compile-time Polymorphism)

```
public class Calculator {

    // Method overloading - same name, different parameters
    public int add(int a, int b) {
        return a + b;
    }

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

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

    public String add(String a, String b) {
        return a + b;
    }
}
```

1.12.3 Polymorphism in Action

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        // Runtime polymorphism
        Shape[] shapes = {
            new Circle(5),
            new Rectangle(4, 6),
            new Circle(3)
        };

        for (Shape shape : shapes) {
            shape.calculateArea();  // Calls appropriate overridden method
            shape.display();
        }

        // Compile-time polymorphism
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));       // int version
        System.out.println(calc.add(5.5, 3.2));   // double version
        System.out.println(calc.add(1, 2, 3));    // three parameter version
        System.out.println(calc.add("Hello", " World"));  // String version
    }
}
```

1.12.4 Dynamic Method Dispatch

```
public class DynamicDispatchDemo {
    public static void main(String[] args) {
        Shape shape;  // Reference variable
```

```
        shape = new Circle(5);
        shape.calculateArea();  // Calls Circle's method

        shape = new Rectangle(4, 6);
        shape.calculateArea();  // Calls Rectangle's method

        // The actual method called is determined at runtime
        // based on the object type, not reference type
    }
}
```

1.13 Abstraction

1.13.1 Abstract Classes

```
// Abstract class
public abstract class Vehicle {
    protected String brand;
    protected int year;

    // Constructor in abstract class
    public Vehicle(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    // Concrete method
    public void displayInfo() {
```

```
        System.out.println("Brand: " + brand + ", Year: " + year);
    }

    // Abstract methods (must be implemented by subclasses)
    public abstract void start();
    public abstract void stop();
    public abstract double calculateFuelEfficiency();
}

// Concrete subclass
public class Car extends Vehicle {
    private double engineSize;

    public Car(String brand, int year, double engineSize) {
        super(brand, year);
        this.engineSize = engineSize;
    }

    @Override
    public void start() {
        System.out.println("Car engine started with key ignition");
    }

    @Override
    public void stop() {
        System.out.println("Car engine stopped");
    }

    @Override
    public double calculateFuelEfficiency() {
        return 25.0 - (engineSize * 2);  // Simple calculation
```

```
    }
}

public class Motorcycle extends Vehicle {
    private boolean hasElectricStart;

    public Motorcycle(String brand, int year, boolean hasElectricStart) {
        super(brand, year);
        this.hasElectricStart = hasElectricStart;
    }

    @Override
    public void start() {
        if (hasElectricStart) {
            System.out.println("Motorcycle started with electric start");
        } else {
            System.out.println("Motorcycle started with kick start");
        }
    }

    @Override
    public void stop() {
        System.out.println("Motorcycle engine stopped");
    }

    @Override
    public double calculateFuelEfficiency() {
        return 45.0;  // Generally better fuel efficiency
    }
}
```

1.13.2 Interface-based Abstraction

```java
// Interface
public interface Drawable {
    // All methods are implicitly public and abstract
    void draw();
    void resize(double factor);

    // Default method (Java 8+)
    default void display() {
        System.out.println("Displaying the drawable object");
    }

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

    // Constants (implicitly public, static, final)
    int MAX_SIZE = 1000;
}

// Multiple interfaces
public interface Colorable {
    void setColor(String color);
    String getColor();
}

// Implementation
public class Circle implements Drawable, Colorable {
    private double radius;
```

```
    private String color;

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

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

    @Override
    public void resize(double factor) {
        radius *= factor;
        System.out.println("Circle resized. New radius: " + radius);
    }

    @Override
    public void setColor(String color) {
        this.color = color;
    }

    @Override
    public String getColor() {
        return color;
    }
}
```

1.13.3 Abstraction Demo

```java
public class AbstractionDemo {
    public static void main(String[] args) {
        // Cannot instantiate abstract class
        // Vehicle vehicle = new Vehicle();  // Compilation error

        // Can use abstract class reference
        Vehicle car = new Car("Toyota", 2022, 2.0);
        Vehicle motorcycle = new Motorcycle("Honda", 2021, true);

        car.displayInfo();
        car.start();
        System.out.println("Fuel efficiency: " + car.calculateFuelEfficiency());

        motorcycle.displayInfo();
        motorcycle.start();
        System.out.println("Fuel efficiency: " + motorcycle.calculateFuelEfficiency());

        // Interface usage
        Drawable circle = new Circle(5);
        circle.draw();
        circle.resize(1.5);
        circle.display();  // Default method

        Drawable.printInfo();  // Static method

        // Multiple interface implementation
        if (circle instanceof Colorable) {
            Colorable colorableCircle = (Colorable) circle;
            colorableCircle.setColor("Red");
```

```
        System.out.println("Color: " + colorableCircle.getColor());
      }
    }
}
```

1.14 Encapsulation

1.14.1 Data Hiding and Access Control

```
public class BankAccount {
  // Private fields (data hiding)
  private String accountNumber;
  private String accountHolder;
  private double balance;
  private static double interestRate = 0.03;

  // Constructor
  public BankAccount(String accountNumber, String accountHolder, double initialBalance) {
    this.accountNumber = accountNumber;
    this.accountHolder = accountHolder;
    this.balance = (initialBalance >= 0) ? initialBalance : 0;
  }

  // Public methods (controlled access)
  public void deposit(double amount) {
    if (amount > 0) {
      balance += amount;
      System.out.println("Deposited: $" + amount);
```

```
    } else {
      System.out.println("Invalid deposit amount");
    }
  }

  public boolean withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
      balance -= amount;
      System.out.println("Withdrawn: $" + amount);
      return true;
    } else {
      System.out.println("Invalid withdrawal amount or insufficient funds");
      return false;
    }
  }

  // Getter methods (read access)
  public String getAccountNumber() {
    return accountNumber;
  }

  public String getAccountHolder() {
    return accountHolder;
  }

  public double getBalance() {
    return balance;
  }

  // Setter with validation
  public void setAccountHolder(String accountHolder) {
```

```
        if (accountHolder != null && !accountHolder.trim().isEmpty()) {
            this.accountHolder = accountHolder;
        }
    }

    // Static methods for class-level operations
    public static double getInterestRate() {
        return interestRate;
    }

    public static void setInterestRate(double rate) {
        if (rate >= 0 && rate <= 0.1) {  // Max 10% interest rate
            interestRate = rate;
        }
    }

    // Private helper method
    private void logTransaction(String type, double amount) {
        System.out.println("Transaction: " + type + " - $" + amount +
                           " | Balance: $" + balance);
    }

    public void calculateInterest() {
        double interest = balance * interestRate;
        balance += interest;
        logTransaction("Interest", interest);
    }
}
```

1.14.2 Access Modifiers

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

1.14.3 Encapsulation Example

```
public class EncapsulationDemo {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("12345", "John Doe", 1000.0);

        // Accessing through public methods only
        System.out.println("Account: " + account.getAccountNumber());
        System.out.println("Holder: " + account.getAccountHolder());
        System.out.println("Balance: $" + account.getBalance());

        // Controlled operations
        account.deposit(500);
        account.withdraw(200);
        account.calculateInterest();

        // Cannot access private fields directly
        // System.out.println(account.balance);  // Compilation error

        // Static method access
```

```
        System.out.println("Interest Rate: " + BankAccount.getInterestRate());
        BankAccount.setInterestRate(0.04);
    }
}
```

1.14.4 Benefits of Encapsulation

1. **Data Security**: Private fields prevent unauthorized access
2. **Data Validation**: Setters can validate input before assignment
3. **Flexibility**: Internal implementation can change without affecting clients
4. **Maintainability**: Easier to modify and debug code
5. **Code Reusability**: Well-encapsulated classes are more reusable

---

1.15 Interfaces

1.15.1 Interface Definition and Implementation

```
// Basic interface
public interface Playable {
    // Abstract methods (implicitly public abstract)
    void play();
    void pause();
    void stop();

    // Default method (Java 8+)
    default void restart() {
        stop();
```

```
    play();
  }

  // Static method (Java 8+)
  static void printVersion() {
    System.out.println("Playable Interface v2.0");
  }

  // Constants (implicitly public static final)
  int MAX_VOLUME = 100;
  int MIN_VOLUME = 0;
}

// Functional interface (Java 8+)
@FunctionalInterface
public interface Calculator {
  double calculate(double a, double b);

  // Can have default and static methods
  default void printResult(double result) {
    System.out.println("Result: " + result);
  }
}
```

1.15.2 Multiple Interface Implementation

```
public interface Drawable {
  void draw();
}

public interface Resizable {
```

```
    void resize(double factor);
}

public interface Rotatable {
    void rotate(double angle);
}

// Class implementing multiple interfaces
public class Shape implements Drawable, Resizable, Rotatable {
    protected double x, y;
    protected double size;
    protected double rotation;

    public Shape(double x, double y, double size) {
        this.x = x;
        this.y = y;
        this.size = size;
        this.rotation = 0;
    }

    @Override
    public void draw() {
        System.out.println("Drawing shape at (" + x + ", " + y +
                ") with size " + size + " and rotation " + rotation);
    }

    @Override
    public void resize(double factor) {
        size *= factor;
        System.out.println("Shape resized by factor " + factor);
    }
```

```
    @Override
    public void rotate(double angle) {
        rotation += angle;
        System.out.println("Shape rotated by " + angle + " degrees");
    }
}
```

1.15.3 Interface Inheritance

```
// Base interfaces
public interface Animal {
    void eat();
    void sleep();
}

public interface Mammal extends Animal {
    void giveBirth();
    void produceMilk();
}

public interface Carnivore extends Animal {
    void hunt();
}

// Multiple interface inheritance
public interface Predator extends Mammal, Carnivore {
    void stalk();

    // Can override default methods from parent interfaces
    @Override
```

```
    default void hunt() {
        stalk();
        System.out.println("Attacking prey");
    }
}

// Implementation
public class Lion implements Predator {
    @Override
    public void eat() {
        System.out.println("Lion is eating meat");
    }

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

    @Override
    public void giveBirth() {
        System.out.println("Lioness gives birth to cubs");
    }

    @Override
    public void produceMilk() {
        System.out.println("Lioness produces milk for cubs");
    }

    @Override
    public void stalk() {
        System.out.println("Lion is stalking prey");
```

```
  }
}
```

1.15.4 Functional Interfaces and Lambda Expressions

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

public class FunctionalInterfaceDemo {
  public static void main(String[] args) {
    // Custom functional interface
    Calculator add = (a, b) -> a + b;
    Calculator multiply = (a, b) -> a * b;

    System.out.println("Addition: " + add.calculate(5, 3));
    System.out.println("Multiplication: " + multiply.calculate(5, 3));

    // Built-in functional interfaces

    // Predicate<T> - takes T, returns boolean
    Predicate<Integer> isEven = n -> n % 2 == 0;
    System.out.println("Is 4 even? " + isEven.test(4));

    // Function<T, R> - takes T, returns R
    Function<String, Integer> stringLength = s -> s.length();
    System.out.println("Length of 'Hello': " + stringLength.apply("Hello"));

    // Consumer<T> - takes T, returns void
    Consumer<String> printer = s -> System.out.println("Printing: " + s);
    printer.accept("Hello World");

    // Supplier<T> - takes nothing, returns T
```

```
        Supplier<Double> randomValue = () -> Math.random();
        System.out.println("Random value: " + randomValue.get());

        // BiFunction<T, U, R> - takes T and U, returns R
        BiFunction<Integer, Integer, Integer> max = (a, b) -> a > b ? a : b;
        System.out.println("Max of 5 and 8: " + max.apply(5, 8));
    }
}
```

1.15.5 Interface vs Abstract Class

Feature	Interface	Abstract Class
Multiple Inheritance	✓ (implements multiple)	✗ (extends one only)
Constructor	✗	✓
Instance Variables	✗ (only constants)	✓
Access Modifiers	public (methods)	Any
Method Implementation	Default/Static only	Any method
When to Use	Contract definition	Partial implementation

1.16 Packages

1.16.1 Package Declaration and Structure

```
// File: com/company/project/model/Student.java
package com.company.project.model;
```

```java
import java.util.Date;
import java.util.List;
import java.util.ArrayList;

public class Student {
  private String name;
  private Date enrollmentDate;
  private List<String> courses;

  public Student(String name) {
    this.name = name;
    this.enrollmentDate = new Date();
    this.courses = new ArrayList<>();
  }

  // Methods...
}
```

1.16.2 Package Organization

Project Structure:

```
src/
 ├──── com/
 │    └──── company/
 │         └──── project/
 │              ├──── model/
 │              │    ├──── Student.java
 │              │    ├──── Course.java
 │              │    └──── Grade.java
```

```
│       ├──── service/
│       │   ├──── StudentService.java
│       │   └──── CourseService.java
│       ├──── util/
│       │   ├──── DateUtil.java
│       │   └──── ValidationUtil.java
│       └──── Main.java
```

1.16.3 Import Statements

```java
package com.company.project;

// Specific class import
import java.util.ArrayList;
import java.util.Date;

// Wildcard import (imports all classes from package)
import java.util.*;

// Static import
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;

// Import from same package (optional)
import com.company.project.model.Student;

public class ImportExample {
    public static void main(String[] args) {
        // Using imported classes
        ArrayList<String> list = new ArrayList<>();
```

```
        Date now = new Date();

        // Using static imports
        double area = PI * sqrt(25);  // No need for Math.PI or Math.sqrt

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

1.16.4 Access Control with Packages

```
// File: com/example/package1/ClassA.java
package com.example.package1;

public class ClassA {
    public int publicVar = 1;      // Accessible everywhere
    protected int protectedVar = 2;  // Accessible in package and subclasses
    int defaultVar = 3;           // Accessible only in same package
    private int privateVar = 4;    // Accessible only in same class

    public void testAccess() {
        ClassB b = new ClassB();
        System.out.println(b.publicVar);    // OK
        System.out.println(b.protectedVar); // OK (same package)
        System.out.println(b.defaultVar);   // OK (same package)
        // System.out.println(b.privateVar); // Error
    }
}

// File: com/example/package1/ClassB.java
package com.example.package1;
```

```java
class ClassB {  // Package-private class
    public int publicVar = 1;
    protected int protectedVar = 2;
    int defaultVar = 3;
    private int privateVar = 4;
}

// File: com/example/package2/ClassC.java
package com.example.package2;

import com.example.package1.ClassA;

public class ClassC extends ClassA {
    public void testAccess() {
        System.out.println(publicVar);     // OK (inherited)
        System.out.println(protectedVar);  // OK (inherited, subclass)
        // System.out.println(defaultVar);  // Error (different package)
        // System.out.println(privateVar);  // Error

        ClassA a = new ClassA();
        System.out.println(a.publicVar);   // OK
        // System.out.println(a.protectedVar); // Error (not inherited reference)
    }
}
```

1.16.5 Built-in Packages

```java
// java.lang (automatically imported)
String str = "Hello";        // java.lang.String
System.out.println(str);      // java.lang.System
```

```
Integer num = 42;           // java.lang.Integer

// java.util
import java.util.*;
List<String> list = new ArrayList<>();
Map<String, Integer> map = new HashMap<>();
Date date = new Date();

// java.io
import java.io.*;
FileReader reader = new FileReader("file.txt");
BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

// java.net
import java.net.*;
URL url = new URL("https://example.com");
Socket socket = new Socket("localhost", 8080);
```

1.16.6 Creating JAR Files

```
# Compile all Java files
javac -d build src/com/company/project/*.java
javac -d build src/com/company/project/model/*.java
javac -d build src/com/company/project/service/*.java

# Create JAR file
jar cvf myproject.jar -C build .

# Create executable JAR with manifest
echo "Main-Class: com.company.project.Main" > manifest.txt
jar cvfm myproject.jar manifest.txt -C build .
```

```
# Run JAR file
java -jar myproject.jar
```

1.17 Exception Handling

1.17.1 Exception Hierarchy

```
                Throwable
               /       \
          Error         Exception
         /   \        /       \
OutOfMemoryError ...  RuntimeException  IOException
StackOverflowError     /    |    \      |
   ...       NullPointerException ... FileNotFoundException
             ArrayIndexOutOfBoundsException
             IllegalArgumentException
```

1.17.2 Try-Catch-Finally

```
import java.io.*;

public class ExceptionHandlingDemo {

  public static void basicTryCatch() {
    try {
      int[] numbers = {1, 2, 3};
      System.out.println(numbers[5]);  // ArrayIndexOutOfBoundsException
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index out of bounds: " + e.getMessage());
```

```
    }
}

public static void multipleCatchBlocks() {
    try {
        String str = null;
        int length = str.length();  // NullPointerException
        int result = 10 / 0;      // ArithmeticException
    } catch (NullPointerException e) {
        System.out.println("Null pointer exception: " + e.getMessage());
    } catch (ArithmeticException e) {
        System.out.println("Arithmetic exception: " + e.getMessage());
    } catch (Exception e) {  // Generic catch block (should be last)
        System.out.println("General exception: " + e.getMessage());
    }
}

public static void tryWithFinally() {
    FileReader file = null;
    try {
        file = new FileReader("data.txt");
        // Read file operations
    } catch (FileNotFoundException e) {
        System.out.println("File not found: " + e.getMessage());
    } finally {
        // Always executes (cleanup code)
        if (file != null) {
            try {
                file.close();
            } catch (IOException e) {
                System.out.println("Error closing file: " + e.getMessage());
            }
```

```
        }
      }
    }

    // Try-with-resources (Java 7+)
    public static void tryWithResources() {
      try (FileReader file = new FileReader("data.txt");
           BufferedReader reader = new BufferedReader(file)) {

        String line = reader.readLine();
        System.out.println(line);

      } catch (IOException e) {
        System.out.println("IO Exception: " + e.getMessage());
      }
      // Resources automatically closed
    }
}
```

1.17.3 Custom Exceptions

```
// Custom checked exception
public class InsufficientFundsException extends Exception {
    private double amount;
    private double balance;

    public InsufficientFundsException(double amount, double balance) {
        super("Insufficient funds: Attempted to withdraw $" + amount +
              " but balance is only $" + balance);
        this.amount = amount;
        this.balance = balance;
```

```
    }

    public double getAmount() { return amount; }
    public double getBalance() { return balance; }
}

// Custom unchecked exception
public class InvalidAccountException extends RuntimeException {
    public InvalidAccountException(String message) {
        super(message);
    }

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

// Using custom exceptions
public class BankAccount {
    private double balance;
    private String accountNumber;

    public BankAccount(String accountNumber, double initialBalance) {
        if (accountNumber == null || accountNumber.trim().isEmpty()) {
            throw new InvalidAccountException("Account number cannot be null or empty");
        }
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    public void withdraw(double amount) throws InsufficientFundsException {
```

```
    if (amount > balance) {
      throw new InsufficientFundsException(amount, balance);
    }
    balance -= amount;
  }

  public double getBalance() { return balance; }
}
```

1.17.4 Exception Propagation

```
public class ExceptionPropagation {

  public static void method1() throws IOException {
    method2();
  }

  public static void method2() throws IOException {
    method3();
  }

  public static void method3() throws IOException {
    throw new IOException("Something went wrong in method3");
  }

  public static void main(String[] args) {
    try {
      method1();
    } catch (IOException e) {
      System.out.println("Caught exception: " + e.getMessage());
      e.printStackTrace();  // Print stack trace
```

```
    }
  }
}
```

1.17.5 Best Practices

```
public class ExceptionBestPractices {

  // 1. Be specific with exception types
  public void readFile(String filename) throws FileNotFoundException, IOException {
    // Don't just throw Exception
  }

  // 2. Don't ignore exceptions
  public void badPractice() {
    try {
      // Some risky operation
    } catch (Exception e) {
      // Don't do this - ignoring exception
    }
  }

  public void goodPractice() {
    try {
      // Some risky operation
    } catch (Exception e) {
      // Log the exception
      System.err.println("Error occurred: " + e.getMessage());
      // Or rethrow if appropriate
      throw new RuntimeException("Operation failed", e);
    }
```

```
    }

    // 3. Use try-with-resources for resource management
    public String readFirstLine(String filename) throws IOException {
        try (BufferedReader reader = Files.newBufferedReader(Paths.get(filename))) {
            return reader.readLine();
        }
    }

    // 4. Don't use exceptions for control flow
    public boolean isValidNumber(String str) {
        try {
            Integer.parseInt(str);
            return true;
        } catch (NumberFormatException e) {
            return false;  // Bad practice
        }
    }

    // Better approach
    public boolean isValidNumberBetter(String str) {
        if (str == null || str.trim().isEmpty()) {
            return false;
        }
        return str.matches("-?\\d+");
    }
}
```

1.18 String Handling

1.18.1 String Basics

```java
public class StringBasics {
  public static void main(String[] args) {
    // String creation
    String str1 = "Hello";        // String literal (in string pool)
    String str2 = new String("Hello"); // New object in heap
    String str3 = "Hello";        // References same object as str1

    // String comparison
    System.out.println(str1 == str2);      // false (different objects)
    System.out.println(str1 == str3);      // true (same object)
    System.out.println(str1.equals(str2)); // true (same content)

    // String properties
    String text = "Java Programming";
    System.out.println("Length: " + text.length());
    System.out.println("Character at index 5: " + text.charAt(5));
    System.out.println("Substring: " + text.substring(5, 11));
    System.out.println("Index of 'Pro': " + text.indexOf("Pro"));
    System.out.println("Starts with 'Java': " + text.startsWith("Java"));
    System.out.println("Ends with 'ing': " + text.endsWith("ing"));
  }
}
```

1.18.2 String Methods

```java
public class StringMethods {
    public static void demonstrateMethods() {
        String original = "  Java Programming Language  ";

        // Case conversion
        System.out.println("Upper: " + original.toUpperCase());
        System.out.println("Lower: " + original.toLowerCase());

        // Trimming
        System.out.println("Trimmed: '" + original.trim() + "'");

        // Replacement
        System.out.println("Replace 'Java' with 'Python': " +
                original.replace("Java", "Python"));
        System.out.println("Replace all 'a' with '@': " +
                original.replaceAll("a", "@"));

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

        // Joining (Java 8+)
        String joined = String.join(" | ", fruits);
        System.out.println("Joined: " + joined);

        // Checking content
```

```
    String text = "Hello World";
    System.out.println("Contains 'World': " + text.contains("World"));
    System.out.println("Is empty: " + text.isEmpty());
    System.out.println("Is blank: " + text.isBlank());  // Java 11+
  }
}
```

1.18.3 StringBuilder and StringBuffer

```
public class StringBuilderDemo {

  public static void stringConcatenationComparison() {
    // Inefficient - creates new String objects
    String result = "";
    for (int i = 0; i < 1000; i++) {
      result += "a";  // Creates new String object each time
    }

    // Efficient - uses mutable buffer
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 1000; i++) {
      sb.append("a");
    }
    String efficientResult = sb.toString();
  }

  public static void stringBuilderMethods() {
    StringBuilder sb = new StringBuilder("Hello");

    // Appending
    sb.append(" World");
```

```
        sb.append('!');
        sb.append(123);
        System.out.println("After appends: " + sb.toString());

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

        // Deleting
        sb.delete(6, 16);  // Delete "Beautiful "
        System.out.println("After delete: " + sb.toString());

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

        // Capacity and length
        System.out.println("Length: " + sb.length());
        System.out.println("Capacity: " + sb.capacity());

        // Setting length
        sb.setLength(5);  // Truncate to first 5 characters
        System.out.println("After setLength(5): " + sb.toString());
    }

    // StringBuffer vs StringBuilder
    public static void bufferVsBuilder() {
        // StringBuffer - thread-safe (synchronized)
        StringBuffer buffer = new StringBuffer("Thread-safe");

        // StringBuilder - not thread-safe but faster
```

```
        StringBuilder builder = new StringBuilder("Not thread-safe");

        // Use StringBuilder for single-threaded applications
        // Use StringBuffer for multi-threaded applications
    }
}
```

1.18.4 String Formatting

```
import java.text.DecimalFormat;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class StringFormatting {

    public static void printfFormatting() {
        String name = "John";
        int age = 25;
        double salary = 50000.75;

        // printf-style formatting
        System.out.printf("Name: %s, Age: %d, Salary: $%.2f%n", name, age, salary);

        // String.format()
        String formatted = String.format("Employee: %s (%d years old) earns $%.2f",
                                         name, age, salary);
        System.out.println(formatted);

        // Format specifiers
        System.out.printf("Integer: %d%n", 42);
        System.out.printf("Float: %.2f%n", 3.14159);
```

```
    System.out.printf("Scientific: %e%n", 1234.5);
    System.out.printf("Hexadecimal: %x%n", 255);
    System.out.printf("String: %s%n", "Hello");
    System.out.printf("Character: %c%n", 'A');
    System.out.printf("Boolean: %b%n", true);

    // Width and alignment
    System.out.printf("Right aligned: %10s%n", "Hello");
    System.out.printf("Left aligned: %-10s%n", "Hello");
    System.out.printf("Zero padded: %05d%n", 42);
}

public static void textBlocks() {  // Java 15+
    String html = """
            <html>
                <body>
                    <h1>Hello World</h1>
                    <p>This is a text block example.</p>
                </body>
            </html>
            """;
    System.out.println(html);

    String sql = """
            SELECT id, name, email
            FROM users
            WHERE age > 18
            ORDER BY name
            """;
    System.out.println(sql);
}
```

```
  public static void numberFormatting() {
    DecimalFormat df = new DecimalFormat("#,###.00");
    System.out.println("Formatted number: " + df.format(1234567.89));

    DecimalFormat currency = new DecimalFormat("$#,###.00");
    System.out.println("Currency: " + currency.format(1234.5));

    DecimalFormat percentage = new DecimalFormat("#.##%");
    System.out.println("Percentage: " + percentage.format(0.1234));
  }
}
```

1.18.5 Regular Expressions

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

public class RegexDemo {

  public static void basicRegex() {
    String text = "The year 2023 was great, but 2024 will be better!";

    // Pattern and Matcher
    Pattern pattern = Pattern.compile("\\d{4}");  // 4 digits
    Matcher matcher = pattern.matcher(text);

    while (matcher.find()) {
      System.out.println("Found year: " + matcher.group());
    }
```

```
    // String methods with regex
    System.out.println("Contains digits: " + text.matches(".*\\d+.*"));

    String[] words = text.split("\\s+");  // Split by whitespace
    System.out.println("Words: " + java.util.Arrays.toString(words));

    String replaced = text.replaceAll("\\d{4}", "YEAR");
    System.out.println("Replaced: " + replaced);
}

public static void commonPatterns() {
    // Email validation
    String emailPattern = "^[A-Za-z0-9+_.-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,}$";
    System.out.println("Valid email: " +
            "user@example.com".matches(emailPattern));

    // Phone number (US format)
    String phonePattern = "^\\(\\d{3}\\)\\s\\d{3}-\\d{4}$";
    System.out.println("Valid phone: " +
            "(555) 123-4567".matches(phonePattern));

    // Password (at least 8 chars, 1 uppercase, 1 lowercase, 1 digit)
    String passwordPattern = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)[a-zA-Z\\d@$!%*?&]{8,}$";
    System.out.println("Valid password: " +
            "MyPass123".matches(passwordPattern));

    // URL validation
    String urlPattern = "^https?://[\\w.-]+\\.[a-zA-Z]{2,}(/.*)?$";
    System.out.println("Valid URL: " +
            "https://www.example.com".matches(urlPattern));
}
```

```
  public static void groupsAndCapture() {
    String text = "John Doe (30), Jane Smith (25), Bob Johnson (35)";
    Pattern pattern = Pattern.compile("(\\w+)\\s+(\\w+)\\s+\\((\\d+)\\)");
    Matcher matcher = pattern.matcher(text);

    while (matcher.find()) {
      System.out.println("Full match: " + matcher.group(0));
      System.out.println("First name: " + matcher.group(1));
      System.out.println("Last name: " + matcher.group(2));
      System.out.println("Age: " + matcher.group(3));
      System.out.println("---");
    }
  }
}
```

1.19 Collections Framework

1.19.1 Collection Hierarchy

```
          Collection<E>
         /    |     \
      List<E> Set<E>  Queue<E>
     /  |    |  \    |
ArrayList Vector HashSet TreeSet PriorityQueue
LinkedList Stack LinkedHashSet  Deque<E>
                      |
                  ArrayDeque
```

1.19.2 List Interface

```
import java.util.*;

public class ListDemo {

  public static void arrayListDemo() {
    // ArrayList - resizable array implementation
    List<String> arrayList = new ArrayList<>();

    // Adding elements
    arrayList.add("Apple");
    arrayList.add("Banana");
    arrayList.add("Cherry");
    arrayList.add(1, "Blueberry");  // Insert at index

    // Accessing elements
    System.out.println("Element at index 0: " + arrayList.get(0));
    System.out.println("Size: " + arrayList.size());
    System.out.println("Contains 'Apple': " + arrayList.contains("Apple"));

    // Iterating
    for (String fruit : arrayList) {
      System.out.println(fruit);
    }

    // Using Iterator
    Iterator<String> iterator = arrayList.iterator();
    while (iterator.hasNext()) {
      String fruit = iterator.next();
      if (fruit.equals("Banana")) {
```

```
            iterator.remove();  // Safe removal during iteration
        }
    }

    // Removing elements
    arrayList.remove("Cherry");
    arrayList.remove(0);  // Remove by index

    System.out.println("Final list: " + arrayList);
}

public static void linkedListDemo() {
    // LinkedList - doubly-linked list implementation
    LinkedList<Integer> linkedList = new LinkedList<>();

    // Adding elements
    linkedList.add(10);
    linkedList.add(20);
    linkedList.addFirst(5);   // Add to beginning
    linkedList.addLast(30);   // Add to end

    // Queue operations
    linkedList.offer(40);    // Add to end (queue)
    Integer first = linkedList.poll();  // Remove from beginning
    Integer peek = linkedList.peek();   // Look at first without removing

    // Stack operations
    linkedList.push(100);    // Add to beginning (stack)
    Integer popped = linkedList.pop();  // Remove from beginning

    System.out.println("LinkedList: " + linkedList);
```

```
    }

    public static void listComparison() {
        // Performance comparison
        List<Integer> arrayList = new ArrayList<>();
        List<Integer> linkedList = new LinkedList<>();

        // ArrayList: Fast random access, slow insertion/deletion in middle
        // LinkedList: Slow random access, fast insertion/deletion anywhere

        // ArrayList is better for:
        // - Random access by index
        // - Iteration
        // - Memory efficiency

        // LinkedList is better for:
        // - Frequent insertion/deletion at beginning or middle
        // - Queue/Stack operations
        // - When you don't know the size in advance
    }
}
```

1.19.3 Set Interface

```
import java.util.*;

public class SetDemo {

    public static void hashSetDemo() {
        // HashSet - no duplicates, no ordering
        Set<String> hashSet = new HashSet<>();
```

```
        hashSet.add("Java");
        hashSet.add("Python");
        hashSet.add("JavaScript");
        hashSet.add("Java"); // Duplicate - won't be added

        System.out.println("HashSet: " + hashSet);
        System.out.println("Size: " + hashSet.size());

        // Checking membership
        System.out.println("Contains Java: " + hashSet.contains("Java"));

        // Set operations
        Set<String> otherSet = new HashSet<>(Arrays.asList("Java", "C++", "Go"));

        // Union
        Set<String> union = new HashSet<>(hashSet);
        union.addAll(otherSet);
        System.out.println("Union: " + union);

        // Intersection
        Set<String> intersection = new HashSet<>(hashSet);
        intersection.retainAll(otherSet);
        System.out.println("Intersection: " + intersection);

        // Difference
        Set<String> difference = new HashSet<>(hashSet);
        difference.removeAll(otherSet);
        System.out.println("Difference: " + difference);
    }
```

```java
public static void treeSetDemo() {
    // TreeSet - sorted set (implements NavigableSet)
    TreeSet<Integer> treeSet = new TreeSet<>();

    treeSet.add(50);
    treeSet.add(30);
    treeSet.add(70);
    treeSet.add(20);
    treeSet.add(40);

    System.out.println("TreeSet (sorted): " + treeSet);

    // NavigableSet methods
    System.out.println("First: " + treeSet.first());
    System.out.println("Last: " + treeSet.last());
    System.out.println("Lower than 40: " + treeSet.lower(40));
    System.out.println("Higher than 40: " + treeSet.higher(40));
    System.out.println("Floor of 35: " + treeSet.floor(35));
    System.out.println("Ceiling of 35: " + treeSet.ceiling(35));

    // Subset operations
    System.out.println("HeadSet (< 40): " + treeSet.headSet(40));
    System.out.println("TailSet (>= 40): " + treeSet.tailSet(40));
    System.out.println("SubSet [30, 60): " + treeSet.subSet(30, 60));
}

public static void linkedHashSetDemo() {
    // LinkedHashSet - maintains insertion order
    Set<String> linkedHashSet = new LinkedHashSet<>();

    linkedHashSet.add("First");
```

```
        linkedHashSet.add("Second");
        linkedHashSet.add("Third");
        linkedHashSet.add("First");  // Duplicate

        System.out.println("LinkedHashSet (insertion order): " + linkedHashSet);
    }
}
```

1.19.4 Map Interface

```
import java.util.*;

public class MapDemo {

    public static void hashMapDemo() {
        // HashMap - key-value pairs, no ordering
        Map<String, Integer> hashMap = new HashMap<>();

        // Adding key-value pairs
        hashMap.put("Alice", 25);
        hashMap.put("Bob", 30);
        hashMap.put("Charlie", 35);
        hashMap.put("Alice", 26);  // Updates existing value

        // Accessing values
        System.out.println("Alice's age: " + hashMap.get("Alice"));
        System.out.println("David's age: " + hashMap.get("David"));  // null
        System.out.println("David's age (default): " +
                           hashMap.getOrDefault("David", 0));

        // Checking keys and values
```

```
    System.out.println("Contains key 'Bob': " + hashMap.containsKey("Bob"));
    System.out.println("Contains value 30: " + hashMap.containsValue(30));

    // Iterating over map
    for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {
        System.out.println(entry.getKey() + " -> " + entry.getValue());
    }

    // Iterating over keys
    for (String key : hashMap.keySet()) {
        System.out.println("Key: " + key + ", Value: " + hashMap.get(key));
    }

    // Iterating over values
    for (Integer value : hashMap.values()) {
        System.out.println("Value: " + value);
    }
}

public static void treeMapDemo() {
    // TreeMap - sorted by keys
    TreeMap<String, String> treeMap = new TreeMap<>();

    treeMap.put("gamma", "γ");
    treeMap.put("alpha", "α");
    treeMap.put("beta", "β");
    treeMap.put("delta", "δ");

    System.out.println("TreeMap (sorted by keys): " + treeMap);

    // NavigableMap methods
```

```
    System.out.println("First key: " + treeMap.firstKey());
    System.out.println("Last key: " + treeMap.lastKey());
    System.out.println("Lower key than 'delta': " + treeMap.lowerKey("delta"));
    System.out.println("Higher key than 'beta': " + treeMap.higherKey("beta"));

    // Subset operations
    System.out.println("HeadMap (< 'delta'): " + treeMap.headMap("delta"));
    System.out.println("TailMap (>= 'beta'): " + treeMap.tailMap("beta"));
    System.out.println("SubMap ['beta', 'gamma'): " +
                       treeMap.subMap("beta", "gamma"));
  }

  public static void linkedHashMapDemo() {
    // LinkedHashMap - maintains insertion order
    Map<String, String> linkedHashMap = new LinkedHashMap<>();

    linkedHashMap.put("first", "1st");
    linkedHashMap.put("second", "2nd");
    linkedHashMap.put("third", "3rd");

    System.out.println("LinkedHashMap (insertion order): " + linkedHashMap);

    // LRU Cache implementation
    Map<String, String> lruCache = new LinkedHashMap<String, String>(16, 0.75f, true) {
      @Override
      protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {
        return size() > 3;  // Keep only 3 entries
      }
    };

    lruCache.put("A", "1");
```

```
    lruCache.put("B", "2");
    lruCache.put("C", "3");
    lruCache.get("A");  // Access A (moves to end)
    lruCache.put("D", "4");  // B should be removed

    System.out.println("LRU Cache: " + lruCache);
  }
}
```

1.19.5 Queue and Deque

```
import java.util.*;

public class QueueDemo {

  public static void queueOperations() {
    // PriorityQueue - heap-based priority queue
    Queue<Integer> priorityQueue = new PriorityQueue<>();

    priorityQueue.offer(30);
    priorityQueue.offer(10);
    priorityQueue.offer(50);
    priorityQueue.offer(20);

    System.out.println("PriorityQueue: " + priorityQueue);

    // Removing elements (in priority order)
    while (!priorityQueue.isEmpty()) {
      System.out.println("Poll: " + priorityQueue.poll());
    }
```

```
    // Custom comparator for priority queue
    Queue<String> stringQueue = new PriorityQueue<>(
        (a, b) -> Integer.compare(b.length(), a.length())  // Longer strings first
    );

    stringQueue.offer("Java");
    stringQueue.offer("Python");
    stringQueue.offer("C");
    stringQueue.offer("JavaScript");

    while (!stringQueue.isEmpty()) {
        System.out.println("Poll: " + stringQueue.poll());
    }
}

public static void dequeOperations() {
    // ArrayDeque - resizable array implementation of Deque
    Deque<String> deque = new ArrayDeque<>();

    // Adding elements
    deque.addFirst("First");
    deque.addLast("Last");
    deque.offerFirst("New First");
    deque.offerLast("New Last");

    System.out.println("Deque: " + deque);

    // Accessing elements
    System.out.println("First: " + deque.peekFirst());
    System.out.println("Last: " + deque.peekLast());
```

```
    // Removing elements
    System.out.println("Remove first: " + deque.removeFirst());
    System.out.println("Remove last: " + deque.removeLast());

    System.out.println("Final deque: " + deque);

    // Using as Stack
    Deque<Integer> stack = new ArrayDeque<>();
    stack.push(1);
    stack.push(2);
    stack.push(3);

    while (!stack.isEmpty()) {
      System.out.println("Pop: " + stack.pop());
    }
  }
}
```

1.19.6 Collections Utility Class

```
import java.util.*;

public class CollectionsUtility {

  public static void sortingAndSearching() {
    List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1, 9, 3));

    // Sorting
    Collections.sort(numbers);
    System.out.println("Sorted: " + numbers);
```

```
    Collections.sort(numbers, Collections.reverseOrder());
    System.out.println("Reverse sorted: " + numbers);

    // Binary search (list must be sorted)
    Collections.sort(numbers);
    int index = Collections.binarySearch(numbers, 5);
    System.out.println("Index of 5: " + index);

    // Custom sorting
    List<String> words = new ArrayList<>(Arrays.asList("apple", "Banana", "cherry"));
    Collections.sort(words, String.CASE_INSENSITIVE_ORDER);
    System.out.println("Case-insensitive sort: " + words);
}

public static void manipulationMethods() {
    List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C", "D", "E"));

    // Shuffling
    Collections.shuffle(list);
    System.out.println("Shuffled: " + list);

    // Reversing
    Collections.reverse(list);
    System.out.println("Reversed: " + list);

    // Rotating
    Collections.rotate(list, 2);
    System.out.println("Rotated by 2: " + list);

    // Swapping
    Collections.swap(list, 0, 4);
```

```
        System.out.println("Swapped 0 and 4: " + list);

        // Filling
        Collections.fill(list, "X");
        System.out.println("Filled with X: " + list);

        // Copying
        List<String> source = Arrays.asList("1", "2", "3");
        List<String> dest = new ArrayList<>(Arrays.asList("A", "B", "C"));
        Collections.copy(dest, source);
        System.out.println("After copy: " + dest);
    }

    public static void findingMinMax() {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 9, 3);

        System.out.println("Min: " + Collections.min(numbers));
        System.out.println("Max: " + Collections.max(numbers));

        // With custom comparator
        List<String> words = Arrays.asList("apple", "Banana", "cherry");
        System.out.println("Min (case-insensitive): " +
                Collections.min(words, String.CASE_INSENSITIVE_ORDER));
        System.out.println("Max (case-insensitive): " +
                Collections.max(words, String.CASE_INSENSITIVE_ORDER));

        // Frequency
        List<String> letters = Arrays.asList("a", "b", "a", "c", "a", "b");
        System.out.println("Frequency of 'a': " + Collections.frequency(letters, "a"));
    }
```

```
  public static void immutableCollections() {
    List<String> mutableList = new ArrayList<>(Arrays.asList("A", "B", "C"));

    // Unmodifiable views
    List<String> unmodifiableList = Collections.unmodifiableList(mutableList);
    // unmodifiableList.add("D");  // UnsupportedOperationException

    Set<String> unmodifiableSet = Collections.unmodifiableSet(
      new HashSet<>(Arrays.asList("X", "Y", "Z"))
    );

    Map<String, Integer> unmodifiableMap = Collections.unmodifiableMap(
      new HashMap<String, Integer>() {{
        put("one", 1);
        put("two", 2);
      }}
    );

    // Singleton collections
    List<String> singletonList = Collections.singletonList("Only");
    Set<String> singletonSet = Collections.singleton("Only");
    Map<String, String> singletonMap = Collections.singletonMap("key", "value");

    // Empty collections
    List<String> emptyList = Collections.emptyList();
    Set<String> emptySet = Collections.emptySet();
    Map<String, String> emptyMap = Collections.emptyMap();
  }
}
```

1.20 Generics

1.20.1 Generic Classes

```java
// Generic class with type parameter
public class Box<T> {
  private T content;

  public void set(T content) {
    this.content = content;
  }

  public T get() {
    return content;
  }

  public boolean isEmpty() {
    return content == null;
  }
}

// Multiple type parameters
public class Pair<T, U> {
  private T first;
  private U second;

  public Pair(T first, U second) {
    this.first = first;
    this.second = second;
  }
```

```
    public T getFirst() { return first; }
    public U getSecond() { return second; }
}

// Bounded type parameters
public class NumberBox<T extends Number> {
    private T number;

    public NumberBox(T number) {
        this.number = number;
    }

    public double getDoubleValue() {
        return number.doubleValue();  // Can call Number methods
    }
}
```

1.20.2 Generic Methods

```
public class GenericMethods {

    // Generic method
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // Multiple type parameters
    public static <T, U> boolean compare(Pair<T, U> p1, Pair<T, U> p2) {
```

```
        return p1.getFirst().equals(p2.getFirst()) &&
               p1.getSecond().equals(p2.getSecond());
    }

    // Bounded type parameter
    public static <T extends Comparable<T>> T findMax(T[] array) {
        T max = array[0];
        for (T element : array) {
            if (element.compareTo(max) > 0) {
                max = element;
            }
        }
        return max;
    }
}
```

1.20.3 Wildcards

```
import java.util.*;

public class WildcardDemo {

    // Upper bounded wildcard (? extends)
    public static double sumOfNumbers(List<? extends Number> numbers) {
        double sum = 0.0;
        for (Number num : numbers) {
            sum += num.doubleValue();
        }
        return sum;
    }
```

```java
    // Lower bounded wildcard (? super)
    public static void addNumbers(List<? super Integer> numbers) {
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
    }

    // Unbounded wildcard (?)
    public static void printList(List<?> list) {
        for (Object item : list) {
            System.out.println(item);
        }
    }

    public static void main(String[] args) {
        // Upper bounded - can read as Number
        List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
        List<Double> doubles = Arrays.asList(1.1, 2.2, 3.3);

        System.out.println("Sum of integers: " + sumOfNumbers(integers));
        System.out.println("Sum of doubles: " + sumOfNumbers(doubles));

        // Lower bounded - can add Integer or its subtypes
        List<Number> numbers = new ArrayList<>();
        addNumbers(numbers);

        // Unbounded - can read as Object
        printList(integers);
        printList(Arrays.asList("A", "B", "C"));
    }
}
```

1.21 Enums

1.21.1 Basic Enums

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

// Enum with fields and methods
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6),
    MARS(6.421e+23, 3.3972e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double getMass() { return mass; }
    public double getRadius() { return radius; }

    // Universal gravitational constant
```

```
  public static final double G = 6.67300E-11;

  public double surfaceGravity() {
    return G * mass / (radius * radius);
  }

  public double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
  }
}
```

1.21.2 Advanced Enum Features

```
public enum Operation {
  PLUS("+") {
    @Override
    public double apply(double x, double y) {
      return x + y;
    }
  },
  MINUS("-") {
    @Override
    public double apply(double x, double y) {
      return x - y;
    }
  },
  TIMES("*") {
    @Override
    public double apply(double x, double y) {
      return x * y;
    }
```

```
    },
    DIVIDE("/") {
        @Override
        public double apply(double x, double y) {
            return x / y;
        }
    };

    private final String symbol;

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

    public abstract double apply(double x, double y);

    @Override
    public String toString() {
        return symbol;
    }
}

// Enum implementing interface
public interface Describable {
    String getDescription();
}

public enum Status implements Describable {
    ACTIVE("Currently active"),
    INACTIVE("Currently inactive"),
    PENDING("Waiting for approval");
```

```
    private final String description;

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

    @Override
    public String getDescription() {
        return description;
    }
}
```

1.22 Annotations

1.22.1 Built-in Annotations

```
public class AnnotationDemo {

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

    @Deprecated
    public void oldMethod() {
        System.out.println("This method is deprecated");
    }
```

```
  @SuppressWarnings("unchecked")
  public void methodWithWarnings() {
    List rawList = new ArrayList();
    rawList.add("String");
  }

  @SafeVarargs
  public final <T> void safeVarargsMethod(T... args) {
    for (T arg : args) {
      System.out.println(arg);
    }
  }
}
```

1.22.2 Custom Annotations

```
import java.lang.annotation.*;

// Marker annotation
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}

// Annotation with elements
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Author {
  String name();
  String date();
```

```
    int version() default 1;
}

// Repeatable annotation (Java 8+)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Repeatable(Tags.class)
public @interface Tag {
    String value();
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Tags {
    Tag[] value();
}

// Using custom annotations
@Author(name = "John Doe", date = "2023-01-01", version = 2)
public class AnnotatedClass {

    @Test
    @Tag("unit")
    @Tag("fast")
    public void testMethod() {
        System.out.println("Test method");
    }

    @Author(name = "Jane Smith", date = "2023-02-01")
    public void anotherMethod() {
        System.out.println("Another method");
```

```
  }
}
```

1.23 File I/O

1.23.1 File Operations

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

public class FileIODemo {

  public static void basicFileOperations() throws IOException {
    // Creating files and directories
    File file = new File("example.txt");
    File directory = new File("testDir");

    directory.mkdir();
    file.createNewFile();

    // File information
    System.out.println("File exists: " + file.exists());
    System.out.println("Is file: " + file.isFile());
    System.out.println("Is directory: " + file.isDirectory());
    System.out.println("File size: " + file.length());
    System.out.println("Last modified: " + file.lastModified());
```

```
    // File permissions
    System.out.println("Can read: " + file.canRead());
    System.out.println("Can write: " + file.canWrite());
    System.out.println("Can execute: " + file.canExecute());
}

public static void readWriteWithStreams() throws IOException {
    // Writing to file
    try (FileWriter writer = new FileWriter("output.txt");
         BufferedWriter bufferedWriter = new BufferedWriter(writer)) {

        bufferedWriter.write("Hello, World!");
        bufferedWriter.newLine();
        bufferedWriter.write("This is a test file.");
    }

    // Reading from file
    try (FileReader reader = new FileReader("output.txt");
         BufferedReader bufferedReader = new BufferedReader(reader)) {

        String line;
        while ((line = bufferedReader.readLine()) != null) {
            System.out.println(line);
        }
    }
}

public static void binaryFileOperations() throws IOException {
    // Writing binary data
    try (FileOutputStream fos = new FileOutputStream("data.bin");
         DataOutputStream dos = new DataOutputStream(fos)) {
```

```
            dos.writeInt(42);
            dos.writeDouble(3.14159);
            dos.writeUTF("Hello");
            dos.writeBoolean(true);
        }

        // Reading binary data
        try (FileInputStream fis = new FileInputStream("data.bin");
             DataInputStream dis = new DataInputStream(fis)) {

            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            String stringValue = dis.readUTF();
            boolean booleanValue = dis.readBoolean();

            System.out.println("Int: " + intValue);
            System.out.println("Double: " + doubleValue);
            System.out.println("String: " + stringValue);
            System.out.println("Boolean: " + booleanValue);
        }
    }
}
```

1.23.2 NIO.2 (New I/O)

```
import java.nio.file.*;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.stream.Stream;
```

```java
public class NIODemo {

  public static void pathOperations() {
    Path path = Paths.get("documents", "files", "example.txt");

    System.out.println("Path: " + path);
    System.out.println("File name: " + path.getFileName());
    System.out.println("Parent: " + path.getParent());
    System.out.println("Root: " + path.getRoot());
    System.out.println("Absolute path: " + path.toAbsolutePath());

    // Path manipulation
    Path resolved = path.resolve("sibling.txt");
    Path relative = path.relativize(Paths.get("documents"));
    Path normalized = Paths.get("documents/../files/./example.txt").normalize();
  }

  public static void fileOperationsNIO() throws IOException {
    Path file = Paths.get("nio-example.txt");

    // Writing
    List<String> lines = List.of("Line 1", "Line 2", "Line 3");
    Files.write(file, lines, StandardCharsets.UTF_8);

    // Reading
    List<String> readLines = Files.readAllLines(file, StandardCharsets.UTF_8);
    readLines.forEach(System.out::println);

    // Reading as String
    String content = Files.readString(file, StandardCharsets.UTF_8);
    System.out.println("Content: " + content);
```

```
    // File attributes
    System.out.println("Size: " + Files.size(file));
    System.out.println("Last modified: " + Files.getLastModifiedTime(file));
    System.out.println("Is regular file: " + Files.isRegularFile(file));
    System.out.println("Is directory: " + Files.isDirectory(file));
}

public static void directoryOperations() throws IOException {
    Path dir = Paths.get("test-directory");

    // Create directory
    Files.createDirectories(dir);

    // List directory contents
    try (Stream<Path> paths = Files.list(dir)) {
        paths.forEach(System.out::println);
    }

    // Walk directory tree
    try (Stream<Path> paths = Files.walk(dir)) {
        paths.filter(Files::isRegularFile)
             .forEach(System.out::println);
    }

    // Find files
    try (Stream<Path> paths = Files.find(dir, 2,
            (path, attrs) -> path.toString().endsWith(".txt"))) {
        paths.forEach(System.out::println);
    }
```

```
  }
}
```

1.24 Multithreading

1.24.1 Thread Creation

```java
// Extending Thread class
class MyThread extends Thread {
  private String threadName;

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

  @Override
  public void run() {
    for (int i = 0; i < 5; i++) {
      System.out.println(threadName + " - Count: " + i);
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        System.out.println(threadName + " interrupted");
        return;
      }
    }
  }
}
```

```java
// Implementing Runnable interface
class MyRunnable implements Runnable {
    private String taskName;

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

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(taskName + " - Count: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                return;
            }
        }
    }
}

public class ThreadDemo {
    public static void main(String[] args) {
        // Using Thread class
        MyThread thread1 = new MyThread("Thread-1");
        MyThread thread2 = new MyThread("Thread-2");

        thread1.start();
        thread2.start();
```

```
        // Using Runnable interface
        Thread thread3 = new Thread(new MyRunnable("Task-1"));
        Thread thread4 = new Thread(new MyRunnable("Task-2"));

        thread3.start();
        thread4.start();

        // Using lambda expression
        Thread thread5 = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                System.out.println("Lambda thread - " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    return;
                }
            }
        });
        thread5.start();
    }
}
```

1.24.2 Synchronization

```
public class SynchronizationDemo {

    // Synchronized method
    public synchronized void synchronizedMethod() {
        System.out.println(Thread.currentThread().getName() + " entered synchronized method");
```

```
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println(Thread.currentThread().getName() + " exiting synchronized method");
    }

    // Synchronized block
    private final Object lock = new Object();

    public void synchronizedBlock() {
        synchronized (lock) {
            System.out.println(Thread.currentThread().getName() + " in synchronized block");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // Static synchronized method
    public static synchronized void staticSynchronizedMethod() {
        System.out.println(Thread.currentThread().getName() + " in static synchronized method");
    }
}

// Producer-Consumer example
class ProducerConsumer {
    private final Object lock = new Object();
```

```
private boolean available = false;
private String data;

public void produce(String value) {
  synchronized (lock) {
    while (available) {
      try {
        lock.wait();  // Wait until consumed
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return;
      }
    }
    data = value;
    available = true;
    System.out.println("Produced: " + value);
    lock.notifyAll();  // Notify consumers
  }
}

public String consume() {
  synchronized (lock) {
    while (!available) {
      try {
        lock.wait();  // Wait until produced
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        return null;
      }
    }
    available = false;
    System.out.println("Consumed: " + data);
```

```
            lock.notifyAll();  // Notify producers
            return data;
        }
    }
}
```

1.24.3 Executor Framework

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

public class ExecutorDemo {

    public static void executorTypes() {
        // Single thread executor
        ExecutorService singleExecutor = Executors.newSingleThreadExecutor();

        // Fixed thread pool
        ExecutorService fixedPool = Executors.newFixedThreadPool(4);

        // Cached thread pool
        ExecutorService cachedPool = Executors.newCachedThreadPool();

        // Scheduled executor
        ScheduledExecutorService scheduledExecutor = Executors.newScheduledThreadPool(2);

        // Submit tasks
        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            fixedPool.submit(() -> {
```

```
            System.out.println("Task " + taskId + " executed by " +
                    Thread.currentThread().getName());
            try {
              Thread.sleep(1000);
            } catch (InterruptedException e) {
              Thread.currentThread().interrupt();
            }
          });
        }

        // Shutdown executors
        fixedPool.shutdown();
        try {
          if (!fixedPool.awaitTermination(60, TimeUnit.SECONDS)) {
            fixedPool.shutdownNow();
          }
        } catch (InterruptedException e) {
          fixedPool.shutdownNow();
        }
      }

      public static void futureAndCallable() throws Exception {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Callable task that returns a result
        Callable<Integer> task = () -> {
          Thread.sleep(2000);
          return 42;
        };

        // Submit and get Future
```

```
        Future<Integer> future = executor.submit(task);

        System.out.println("Task submitted, doing other work...");

        // Get result (blocks until complete)
        Integer result = future.get(5, TimeUnit.SECONDS);
        System.out.println("Result: " + result);

        // Submit multiple tasks
        List<Callable<String>> tasks = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            final int taskId = i;
            tasks.add(() -> {
                Thread.sleep(1000);
                return "Task " + taskId + " result";
            });
        }

        // Execute all tasks
        List<Future<String>> futures = executor.invokeAll(tasks);
        for (Future<String> f : futures) {
            System.out.println(f.get());
        }

        executor.shutdown();
    }
}
```

1.24.4 Concurrent Collections

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

public class ConcurrentCollectionsDemo {

  public static void concurrentCollections() {
    // ConcurrentHashMap
    ConcurrentHashMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();
    concurrentMap.put("key1", 1);
    concurrentMap.put("key2", 2);

    // Atomic operations
    concurrentMap.putIfAbsent("key3", 3);
    concurrentMap.compute("key1", (key, val) -> val + 10);
    concurrentMap.merge("key2", 5, Integer::sum);

    // CopyOnWriteArrayList
    CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>();
    cowList.add("A");
    cowList.add("B");

    // BlockingQueue
    BlockingQueue<String> queue = new ArrayBlockingQueue<>(10);

    // Producer
    new Thread(() -> {
      try {
        for (int i = 0; i < 5; i++) {
          queue.put("Item " + i);
          System.out.println("Produced: Item " + i);
```

```
            Thread.sleep(1000);
          }
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
      }).start();

      // Consumer
      new Thread(() -> {
        try {
          while (true) {
            String item = queue.take();
            System.out.println("Consumed: " + item);
          }
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
      }).start();
    }
}
```

1.25 Lambda Expressions

1.25.1 Basic Lambda Syntax

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

public class LambdaBasics {
```

```java
public static void basicSyntax() {
    // Traditional anonymous class
    Runnable runnable1 = new Runnable() {
        @Override
        public void run() {
            System.out.println("Hello from anonymous class");
        }
    };

    // Lambda expression
    Runnable runnable2 = () -> System.out.println("Hello from lambda");

    // Lambda with parameters
    Comparator<String> comparator = (s1, s2) -> s1.compareTo(s2);

    // Lambda with block body
    Consumer<String> printer = (s) -> {
        System.out.println("Processing: " + s);
        System.out.println("Length: " + s.length());
    };

    // Method reference
    Consumer<String> methodRef = System.out::println;
}

public static void functionalInterfaces() {
    // Predicate<T> - takes T, returns boolean
    Predicate<String> isEmpty = String::isEmpty;
    Predicate<Integer> isEven = n -> n % 2 == 0;
```

```
        System.out.println("Is empty: " + isEmpty.test(""));
        System.out.println("Is 4 even: " + isEven.test(4));

        // Function<T, R> - takes T, returns R
        Function<String, Integer> stringLength = String::length;
        Function<Integer, String> intToString = Object::toString;

        System.out.println("Length of 'Hello': " + stringLength.apply("Hello"));

        // Consumer<T> - takes T, returns void
        Consumer<String> printer = System.out::println;
        printer.accept("Hello World");

        // Supplier<T> - takes nothing, returns T
        Supplier<Double> randomValue = Math::random;
        System.out.println("Random: " + randomValue.get());

        // BiFunction<T, U, R> - takes T and U, returns R
        BiFunction<String, String, String> concat = String::concat;
        System.out.println("Concatenated: " + concat.apply("Hello", " World"));
    }
}
```

1.25.2 Method References

```
import java.util.*;
import java.util.function.Function;

public class MethodReferences {

    public static void methodReferenceTypes() {
```

```
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

    // Static method reference
    names.sort(String::compareToIgnoreCase);

    // Instance method reference of particular object
    String prefix = "Name: ";
    names.forEach(prefix::concat);

    // Instance method reference of arbitrary object
    names.stream()
        .map(String::toUpperCase)
        .forEach(System.out::println);

    // Constructor reference
    Function<String, StringBuilder> sbCreator = StringBuilder::new;
    StringBuilder sb = sbCreator.apply("Hello");

    // Array constructor reference
    Function<Integer, String[]> arrayCreator = String[]::new;
    String[] array = arrayCreator.apply(5);
}

public static class Person {
    private String name;
    private int age;

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

```
  public String getName() { return name; }
  public int getAge() { return age; }

  public static int compareByAge(Person p1, Person p2) {
    return Integer.compare(p1.age, p2.age);
  }

  @Override
  public String toString() {
    return name + " (" + age + ")";
  }
}

public static void personExample() {
  List<Person> people = Arrays.asList(
    new Person("Alice", 30),
    new Person("Bob", 25),
    new Person("Charlie", 35)
  );

  // Method reference to static method
  people.sort(Person::compareByAge);

  // Method reference to instance method
  people.stream()
      .map(Person::getName)
      .forEach(System.out::println);

  // Constructor reference
  Function<String, Person> personCreator = name -> new Person(name, 0);
```

```
    // Or using constructor reference (if constructor exists)
    // Function<String, Person> personCreator = Person::new;
  }
}
```

1.26 Stream API

1.26.1 Stream Creation and Basic Operations

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

public class StreamBasics {

  public static void streamCreation() {
    // From collection
    List<String> list = Arrays.asList("a", "b", "c");
    Stream<String> stream1 = list.stream();

    // From array
    String[] array = {"x", "y", "z"};
    Stream<String> stream2 = Arrays.stream(array);

    // Using Stream.of()
    Stream<String> stream3 = Stream.of("1", "2", "3");

    // Empty stream
    Stream<String> empty = Stream.empty();
```

```
    // Infinite streams
    Stream<Integer> infinite = Stream.iterate(0, n -> n + 2);
    Stream<Double> random = Stream.generate(Math::random);

    // Range streams
    IntStream range = IntStream.range(1, 10);
    IntStream rangeClosed = IntStream.rangeClosed(1, 10);
}

public static void basicOperations() {
    List<String> words = Arrays.asList("apple", "banana", "cherry", "date", "elderberry");

    // Filter
    words.stream()
        .filter(word -> word.length() > 5)
        .forEach(System.out::println);

    // Map
    words.stream()
        .map(String::toUpperCase)
        .forEach(System.out::println);

    // Sorted
    words.stream()
        .sorted()
        .forEach(System.out::println);

    // Distinct
    Arrays.asList("a", "b", "a", "c", "b")
        .stream()
```

```
            .distinct()
            .forEach(System.out::println);

        // Limit and Skip
        words.stream()
            .skip(2)
            .limit(2)
            .forEach(System.out::println);
    }
}
```

1.26.2 Advanced Stream Operations

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

public class AdvancedStreams {

    public static class Employee {
        private String name;
        private String department;
        private double salary;

        public Employee(String name, String department, double salary) {
            this.name = name;
            this.department = department;
            this.salary = salary;
        }

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

```
  public String getDepartment() { return department; }
  public double getSalary() { return salary; }

  @Override
  public String toString() {
    return name + " (" + department + ") - $" + salary;
  }
}

public static void collectOperations() {
  List<Employee> employees = Arrays.asList(
    new Employee("Alice", "IT", 70000),
    new Employee("Bob", "HR", 60000),
    new Employee("Charlie", "IT", 80000),
    new Employee("Diana", "Finance", 75000),
    new Employee("Eve", "IT", 65000)
  );

  // Collect to List
  List<String> names = employees.stream()
                    .map(Employee::getName)
                    .collect(Collectors.toList());

  // Collect to Set
  Set<String> departments = employees.stream()
                       .map(Employee::getDepartment)
                       .collect(Collectors.toSet());

  // Group by department
  Map<String, List<Employee>> byDepartment = employees.stream()
       .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
    // Partition by salary
    Map<Boolean, List<Employee>> partitioned = employees.stream()
            .collect(Collectors.partitioningBy(emp -> emp.getSalary() > 70000));

    // Collect statistics
    DoubleSummaryStatistics salaryStats = employees.stream()
            .collect(Collectors.summarizingDouble(Employee::getSalary));

    System.out.println("Average salary: " + salaryStats.getAverage());
    System.out.println("Max salary: " + salaryStats.getMax());

    // Joining strings
    String allNames = employees.stream()
                    .map(Employee::getName)
                    .collect(Collectors.joining(", "));
}

public static void reductionOperations() {
    List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

    // Sum
    int sum = numbers.stream()
                .reduce(0, Integer::sum);

    // Product
    int product = numbers.stream()
                  .reduce(1, (a, b) -> a * b);

    // Max
    Optional<Integer> max = numbers.stream()
```

```
                .reduce(Integer::max);

        // Count
        long count = numbers.stream()
                .filter(n -> n % 2 == 0)
                .count();

        // Any match
        boolean hasEven = numbers.stream()
                .anyMatch(n -> n % 2 == 0);

        // All match
        boolean allPositive = numbers.stream()
                .allMatch(n -> n > 0);

        // None match
        boolean noNegative = numbers.stream()
                .noneMatch(n -> n < 0);

        // Find first
        Optional<Integer> first = numbers.stream()
                .filter(n -> n > 5)
                .findFirst();

        // Find any
        Optional<Integer> any = numbers.stream()
                .filter(n -> n > 5)
                .findAny();
    }

    public static void parallelStreams() {
```

```
        List<Integer> largeList = IntStream.rangeClosed(1, 1000000)
                                .boxed()
                                .collect(Collectors.toList());

        // Sequential processing
        long start = System.currentTimeMillis();
        long sequentialSum = largeList.stream()
                           .mapToLong(Integer::longValue)
                           .sum();
        long sequentialTime = System.currentTimeMillis() - start;

        // Parallel processing
        start = System.currentTimeMillis();
        long parallelSum = largeList.parallelStream()
                         .mapToLong(Integer::longValue)
                         .sum();
        long parallelTime = System.currentTimeMillis() - start;

        System.out.println("Sequential: " + sequentialTime + "ms");
        System.out.println("Parallel: " + parallelTime + "ms");
    }
}
```

1.27 Reflection

1.27.1 Basic Reflection

```
import java.lang.reflect.*;

public class ReflectionDemo {
```

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

  public Person() {}

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

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

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  private void privateMethod() {
    System.out.println("Private method called");
  }

  public void greet(String message) {
    System.out.println(name + " says: " + message);
  }
}
```

```java
public static void classInformation() throws Exception {
    Class<?> clazz = Person.class;

    // Basic class information
    System.out.println("Class name: " + clazz.getName());
    System.out.println("Simple name: " + clazz.getSimpleName());
    System.out.println("Package: " + clazz.getPackage().getName());
    System.out.println("Superclass: " + clazz.getSuperclass().getName());

    // Modifiers
    int modifiers = clazz.getModifiers();
    System.out.println("Is public: " + Modifier.isPublic(modifiers));
    System.out.println("Is final: " + Modifier.isFinal(modifiers));
    System.out.println("Is abstract: " + Modifier.isAbstract(modifiers));

    // Interfaces
    Class<?>[] interfaces = clazz.getInterfaces();
    System.out.println("Implements " + interfaces.length + " interfaces");
}

public static void fieldReflection() throws Exception {
    Class<?> clazz = Person.class;
    Person person = new Person("John", 30);

    // Get all fields
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
        System.out.println("Field: " + field.getName() +
                           " Type: " + field.getType().getSimpleName());
    }
```

```
    // Access private field
    Field nameField = clazz.getDeclaredField("name");
    nameField.setAccessible(true);  // Bypass access control

    String name = (String) nameField.get(person);
    System.out.println("Name via reflection: " + name);

    nameField.set(person, "Jane");
    System.out.println("Updated name: " + person.getName());

    // Public field access
    Field emailField = clazz.getField("email");
    emailField.set(person, "jane@example.com");
    System.out.println("Email: " + emailField.get(person));
}

public static void methodReflection() throws Exception {
    Class<?> clazz = Person.class;
    Person person = new Person("Alice", 25);

    // Get all methods
    Method[] methods = clazz.getDeclaredMethods();
    for (Method method : methods) {
        System.out.println("Method: " + method.getName() +
                " Parameters: " + method.getParameterCount());
    }

    // Invoke public method
    Method greetMethod = clazz.getMethod("greet", String.class);
    greetMethod.invoke(person, "Hello World!");
```

```
    // Invoke private method
    Method privateMethod = clazz.getDeclaredMethod("privateMethod");
    privateMethod.setAccessible(true);
    privateMethod.invoke(person);

    // Method with return value
    Method getNameMethod = clazz.getMethod("getName");
    String result = (String) getNameMethod.invoke(person);
    System.out.println("Method result: " + result);
}

public static void constructorReflection() throws Exception {
    Class<?> clazz = Person.class;

    // Get all constructors
    Constructor<?>[] constructors = clazz.getDeclaredConstructors();
    for (Constructor<?> constructor : constructors) {
        System.out.println("Constructor parameters: " +
                constructor.getParameterCount());
    }

    // Create instance using default constructor
    Constructor<?> defaultConstructor = clazz.getConstructor();
    Object person1 = defaultConstructor.newInstance();

    // Create instance using parameterized constructor
    Constructor<?> paramConstructor = clazz.getConstructor(String.class, int.class);
    Object person2 = paramConstructor.newInstance("Bob", 35);

    // Access private constructor
    Constructor<?> privateConstructor = clazz.getDeclaredConstructor(
```

```
            String.class, int.class, String.class);
        privateConstructor.setAccessible(true);
        Object person3 = privateConstructor.newInstance("Charlie", 40, "charlie@example.com");
    }
}
```

1.27.2 Annotations and Reflection

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@interface MyAnnotation {
    String value() default "";
    int priority() default 0;
}

@MyAnnotation(value = "Entity", priority = 1)
public class AnnotatedClass {

    @MyAnnotation("ID Field")
    private Long id;

    @MyAnnotation(value = "Name Field", priority = 2)
    private String name;

    @MyAnnotation("Service Method")
    public void process() {
        System.out.println("Processing...");
    }
```

```java
public static void processAnnotations() throws Exception {
    Class<?> clazz = AnnotatedClass.class;

    // Class annotations
    if (clazz.isAnnotationPresent(MyAnnotation.class)) {
        MyAnnotation annotation = clazz.getAnnotation(MyAnnotation.class);
        System.out.println("Class annotation: " + annotation.value() +
                ", Priority: " + annotation.priority());
    }

    // Field annotations
    Field[] fields = clazz.getDeclaredFields();
    for (Field field : fields) {
        if (field.isAnnotationPresent(MyAnnotation.class)) {
            MyAnnotation annotation = field.getAnnotation(MyAnnotation.class);
            System.out.println("Field " + field.getName() +
                    " annotation: " + annotation.value());
        }
    }

    // Method annotations
    Method[] methods = clazz.getDeclaredMethods();
    for (Method method : methods) {
        if (method.isAnnotationPresent(MyAnnotation.class)) {
            MyAnnotation annotation = method.getAnnotation(MyAnnotation.class);
            System.out.println("Method " + method.getName() +
                    " annotation: " + annotation.value());
        }
    }
```

```
  }
}
```

1.28 Design Patterns

1.28.1 Creational Patterns

```
// Singleton Pattern
public class Singleton {
  private static volatile Singleton instance;

  private Singleton() {}

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}

// Factory Pattern
interface Shape {
  void draw();
```

```
}

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

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

class ShapeFactory {
   public static Shape createShape(String type) {
      switch (type.toLowerCase()) {
         case "circle": return new Circle();
         case "rectangle": return new Rectangle();
         default: throw new IllegalArgumentException("Unknown shape: " + type);
      }
   }
}

// Builder Pattern
public class Product {
   private String name;
   private double price;
   private String category;
```

```
    private boolean available;

    private Product(Builder builder) {
        this.name = builder.name;
        this.price = builder.price;
        this.category = builder.category;
        this.available = builder.available;
    }

    public static class Builder {
        private String name;
        private double price;
        private String category = "General";
        private boolean available = true;

        public Builder(String name, double price) {
            this.name = name;
            this.price = price;
        }

        public Builder category(String category) {
            this.category = category;
            return this;
        }

        public Builder available(boolean available) {
            this.available = available;
            return this;
        }

        public Product build() {
```

```
            return new Product(this);
        }
    }

    // Usage
    public static void main(String[] args) {
        Product product = new Product.Builder("Laptop", 999.99)
                .category("Electronics")
                .available(true)
                .build();
    }
}
```

1.28.2 Structural Patterns

```
// Adapter Pattern
interface MediaPlayer {
    void play(String audioType, String fileName);
}

interface AdvancedMediaPlayer {
    void playVlc(String fileName);
    void playMp4(String fileName);
}

class VlcPlayer implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file: " + fileName);
    }
```

```
    @Override
    public void playMp4(String fileName) {
        // Do nothing
    }
}

class Mp4Player implements AdvancedMediaPlayer {
    @Override
    public void playVlc(String fileName) {
        // Do nothing
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file: " + fileName);
    }
}

class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer advancedPlayer;

    public MediaAdapter(String audioType) {
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer = new VlcPlayer();
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer = new Mp4Player();
        }
    }

    @Override
    public void play(String audioType, String fileName) {
```

```
        if (audioType.equalsIgnoreCase("vlc")) {
            advancedPlayer.playVlc(fileName);
        } else if (audioType.equalsIgnoreCase("mp4")) {
            advancedPlayer.playMp4(fileName);
        }
    }
}

// Decorator Pattern
interface Coffee {
    double getCost();
    String getDescription();
}

class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 2.0;
    }

    @Override
    public String getDescription() {
        return "Simple coffee";
    }
}

abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;

    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
```

```
    }
}

class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", milk";
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.2;
    }

    @Override
    public String getDescription() {
```

```
        return coffee.getDescription() + ", sugar";
    }
}
```

1.28.3 Behavioral Patterns

```
// Observer Pattern
import java.util.*;

interface Observer {
    void update(String message);
}

interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}

class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String news;

    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
```

```
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void setNews(String news) {
        this.news = news;
        notifyObservers(news);
    }
}

class NewsChannel implements Observer {
    private String name;

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

    @Override
    public void update(String message) {
        System.out.println(name + " received news: " + message);
    }
}

// Strategy Pattern
interface PaymentStrategy {
    void pay(double amount);
```

```
}

class CreditCardPayment implements PaymentStrategy {
  private String cardNumber;

  public CreditCardPayment(String cardNumber) {
    this.cardNumber = cardNumber;
  }

  @Override
  public void pay(double amount) {
    System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);
  }
}

class PayPalPayment implements PaymentStrategy {
  private String email;

  public PayPalPayment(String email) {
    this.email = email;
  }

  @Override
  public void pay(double amount) {
    System.out.println("Paid $" + amount + " using PayPal: " + email);
  }
}

class ShoppingCart {
  private PaymentStrategy paymentStrategy;
```

```
    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(double amount) {
        paymentStrategy.pay(amount);
    }
}
```

1.29 Memory Management

1.29.1 Heap and Stack Memory

```
public class MemoryDemo {

    // Static variables - stored in Method Area
    private static int staticCounter = 0;

    // Instance variables - stored in Heap
    private String name;
    private int value;

    public void demonstrateMemory() {
        // Local variables - stored in Stack
        int localVar = 10;
        String localString = "Hello";

        // Object creation - stored in Heap
```

```
    StringBuilder sb = new StringBuilder("World");

    // Method call - new stack frame created
    helperMethod(localVar);

    // Array creation - stored in Heap
    int[] array = new int[100];

    // String literal - stored in String Pool (part of Heap)
    String literal = "Literal";

    // String object - stored in Heap
    String object = new String("Object");
}

private void helperMethod(int param) {
    // New stack frame for this method
    int localHelper = param * 2;

    // Recursive call demonstration
    if (param > 0) {
        helperMethod(param - 1);  // Each call creates new stack frame
    }
}

public static void memoryLeakExample() {
    // Memory leak example - static collection growing indefinitely
    List<String> staticList = new ArrayList<>();

    for (int i = 0; i < 1000000; i++) {
        staticList.add("String " + i);  // Memory leak if not cleared
```

```
    }

    // To prevent leak, clear when done
    // staticList.clear();
  }
}
```

1.29.2 Garbage Collection

```
public class GarbageCollectionDemo {

  private String data;

  public GarbageCollectionDemo(String data) {
    this.data = data;
  }

  // finalize() method - called before GC (deprecated in Java 9+)
  @Override
  protected void finalize() throws Throwable {
    System.out.println("Object with data '" + data + "' is being garbage collected");
    super.finalize();
  }

  public static void demonstrateGC() {
    // Create objects
    GarbageCollectionDemo obj1 = new GarbageCollectionDemo("Object 1");
    GarbageCollectionDemo obj2 = new GarbageCollectionDemo("Object 2");

    // Remove references
    obj1 = null;
```

```
    obj2 = null;

    // Suggest garbage collection (not guaranteed)
    System.gc();

    // Force finalization (deprecated)
    System.runFinalization();

    try {
        Thread.sleep(1000);  // Give GC time to run
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}

public static void memoryUsage() {
    Runtime runtime = Runtime.getRuntime();

    System.out.println("Total memory: " + runtime.totalMemory() / (1024 * 1024) + " MB");
    System.out.println("Free memory: " + runtime.freeMemory() / (1024 * 1024) + " MB");
    System.out.println("Used memory: " +
            (runtime.totalMemory() - runtime.freeMemory()) / (1024 * 1024) + " MB");
    System.out.println("Max memory: " + runtime.maxMemory() / (1024 * 1024) + " MB");

    // Create large array to see memory usage change
    int[] largeArray = new int[1000000];

    System.out.println("\nAfter creating large array:");
    System.out.println("Used memory: " +
            (runtime.totalMemory() - runtime.freeMemory()) / (1024 * 1024) + " MB");
```

```
        // Remove reference and suggest GC
        largeArray = null;
        System.gc();

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        System.out.println("\nAfter garbage collection:");
        System.out.println("Used memory: " +
                (runtime.totalMemory() - runtime.freeMemory()) / (1024 * 1024) + " MB");
    }
}
```

1.29.3 Memory Optimization

```
import java.lang.ref.*;
import java.util.*;

public class MemoryOptimization {

    // Use StringBuilder for string concatenation
    public static String efficientStringConcatenation(String[] strings) {
        StringBuilder sb = new StringBuilder();
        for (String str : strings) {
            sb.append(str);
        }
        return sb.toString();
    }
```

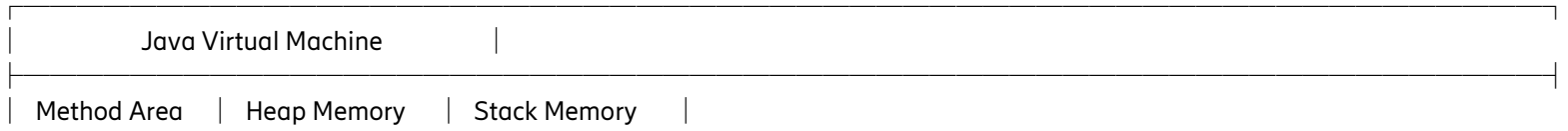
```java
// Avoid creating unnecessary objects
public static void objectCreationOptimization() {
  // Bad - creates new Integer objects
  List<Integer> badList = new ArrayList<>();
  for (int i = 0; i < 1000; i++) {
    badList.add(new Integer(i));  // Deprecated, creates new objects
  }

  // Good - uses autoboxing and Integer cache
  List<Integer> goodList = new ArrayList<>();
  for (int i = 0; i < 1000; i++) {
    goodList.add(i);  // Uses Integer.valueOf() internally
  }

  // Even better - use primitive collections if possible
  // Or avoid boxing altogether with arrays or specialized collections
}

// Weak references for caches
public static class WeakCache<K, V> {
  private Map<K, WeakReference<V>> cache = new HashMap<>();

  public void put(K key, V value) {
    cache.put(key, new WeakReference<>(value));
  }

  public V get(K key) {
    WeakReference<V> ref = cache.get(key);
    if (ref != null) {
      V value = ref.get();
```

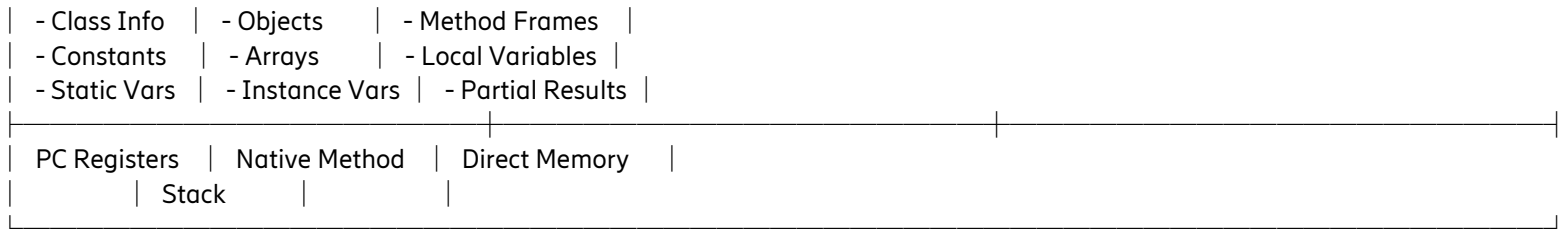
```
        if (value == null) {
          cache.remove(key);  // Clean up dead reference
        }
        return value;
      }
      return null;
    }
}

// Soft references for memory-sensitive caches
public static class SoftCache<K, V> {
  private Map<K, SoftReference<V>> cache = new HashMap<>();

  public void put(K key, V value) {
    cache.put(key, new SoftReference<>(value));
  }

  public V get(K key) {
    SoftReference<V> ref = cache.get(key);
    if (ref != null) {
      V value = ref.get();
      if (value == null) {
        cache.remove(key);
      }
      return value;
    }
    return null;
  }
}

// Memory-efficient data structures
```

```
  public static void efficientDataStructures() {
    // Use ArrayList instead of LinkedList for most cases
    List<String> arrayList = new ArrayList<>(); // Better memory locality

    // Use HashMap instead of TreeMap if ordering not needed
    Map<String, String> hashMap = new HashMap<>(); // O(1) vs O(log n)

    // Use EnumSet for enum collections
    EnumSet<Day> workDays = EnumSet.of(Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY,
                                       Day.THURSDAY, Day.FRIDAY);

    // Use primitive collections when possible (external libraries)
    // TIntList intList = new TIntArrayList(); // Trove4j example
  }

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

1.30 JVM Internals

1.30.1 JVM Architecture





1.30.2 Class Loading

```
public class ClassLoadingDemo {

  static {
    System.out.println("Static block executed during class loading");
  }

  public static void demonstrateClassLoading() {
    // Class loading happens when:
    // 1. First time a class is referenced
    // 2. Creating an instance
    // 3. Accessing static members
    // 4. Using reflection

    System.out.println("Before class loading");

    // This triggers class loading
    MyClass obj = new MyClass();

    // Class is already loaded, no static block execution
    MyClass obj2 = new MyClass();
```

```
    }

    static class MyClass {
        static {
            System.out.println("MyClass static block executed");
        }

        public MyClass() {
            System.out.println("MyClass constructor called");
        }
    }

    public static void classLoaderHierarchy() {
        ClassLoader currentClassLoader = ClassLoadingDemo.class.getClassLoader();

        System.out.println("Current class loader: " + currentClassLoader);
        System.out.println("Parent class loader: " + currentClassLoader.getParent());
        System.out.println("Bootstrap class loader: " + currentClassLoader.getParent().getParent());

        // Class loader hierarchy:
        // Bootstrap ClassLoader (null) - loads core Java classes
        // Extension ClassLoader - loads extension classes
        // Application ClassLoader - loads application classes
    }
}
```

1.30.3 JVM Parameters and Tuning

```
public class JVMTuning {

    public static void printJVMInfo() {
```

```
    Runtime runtime = Runtime.getRuntime();

    // Memory information
    long maxMemory = runtime.maxMemory();
    long totalMemory = runtime.totalMemory();
    long freeMemory = runtime.freeMemory();
    long usedMemory = totalMemory - freeMemory;

    System.out.println("=== JVM Memory Information ===");
    System.out.println("Max memory (Xmx): " + formatBytes(maxMemory));
    System.out.println("Total memory: " + formatBytes(totalMemory));
    System.out.println("Used memory: " + formatBytes(usedMemory));
    System.out.println("Free memory: " + formatBytes(freeMemory));

    // System properties
    System.out.println("\n=== JVM System Properties ===");
    System.out.println("Java version: " + System.getProperty("java.version"));
    System.out.println("Java vendor: " + System.getProperty("java.vendor"));
    System.out.println("JVM name: " + System.getProperty("java.vm.name"));
    System.out.println("JVM version: " + System.getProperty("java.vm.version"));
    System.out.println("OS name: " + System.getProperty("os.name"));
    System.out.println("OS arch: " + System.getProperty("os.arch"));

    // Processors
    System.out.println("Available processors: " + runtime.availableProcessors());
}

private static String formatBytes(long bytes) {
    if (bytes < 1024) return bytes + " B";
    if (bytes < 1024 * 1024) return (bytes / 1024) + " KB";
    if (bytes < 1024 * 1024 * 1024) return (bytes / (1024 * 1024)) + " MB";
```

```
    return (bytes / (1024 * 1024 * 1024)) + " GB";
}

/*
Common JVM Parameters:

Memory Settings:
-Xms<size>    Initial heap size
-Xmx<size>    Maximum heap size
-Xss<size>    Stack size per thread
-XX:NewRatio=<ratio>  Ratio of old/young generation
-XX:MaxMetaspaceSize=<size>  Maximum metaspace size

Garbage Collection:
-XX:+UseG1GC           Use G1 garbage collector
-XX:+UseParallelGC       Use parallel garbage collector
-XX:+UseConcMarkSweepGC   Use CMS garbage collector
-XX:+UseZGC            Use ZGC (Java 11+)

Debugging and Monitoring:
-XX:+PrintGC           Print GC information
-XX:+PrintGCDetails      Print detailed GC information
-XX:+HeapDumpOnOutOfMemoryError  Create heap dump on OOM
-Xloggc:<file>          Log GC to file

Performance:
-server              Use server JVM
-XX:+AggressiveOpts      Enable aggressive optimizations
-XX:+UseFastAccessorMethods  Use fast accessor methods

Example command:
```

```
   java -Xms512m -Xmx2g -XX:+UseG1GC -XX:+PrintGCDetails MyApplication
    */
}
```

1.31 Java 8+ Features

1.31.1 Optional Class

```
import java.util.Optional;

public class OptionalDemo {

   public static void basicOptional() {
      // Creating Optional
      Optional<String> empty = Optional.empty();
      Optional<String> nonEmpty = Optional.of("Hello");
      Optional<String> nullable = Optional.ofNullable(null);

      // Checking presence
      System.out.println("Empty present: " + empty.isPresent());
      System.out.println("Non-empty present: " + nonEmpty.isPresent());

      // Getting values
      if (nonEmpty.isPresent()) {
         System.out.println("Value: " + nonEmpty.get());
      }

      // Using orElse
```

```
    String value1 = empty.orElse("Default");
    String value2 = nonEmpty.orElse("Default");

    System.out.println("Empty or else: " + value1);
    System.out.println("Non-empty or else: " + value2);

    // Using orElseGet (lazy evaluation)
    String value3 = empty.orElseGet(() -> "Computed Default");

    // Using orElseThrow
    try {
        String value4 = empty.orElseThrow(() -> new RuntimeException("No value"));
    } catch (RuntimeException e) {
        System.out.println("Exception: " + e.getMessage());
    }
}

public static void optionalOperations() {
    Optional<String> optional = Optional.of("Hello World");

    // map operation
    Optional<Integer> length = optional.map(String::length);
    System.out.println("Length: " + length.orElse(0));

    // filter operation
    Optional<String> filtered = optional.filter(s -> s.contains("World"));
    System.out.println("Filtered: " + filtered.orElse("Not found"));

    // flatMap operation
    Optional<String> upperCase = optional.flatMap(s ->
        s.isEmpty() ? Optional.empty() : Optional.of(s.toUpperCase()));
```

```
    // ifPresent operation
    optional.ifPresent(System.out::println);

    // ifPresentOrElse (Java 9+)
    optional.ifPresentOrElse(
        System.out::println,
        () -> System.out.println("Value not present")
    );
}

public static Optional<String> findUserById(int id) {
    // Simulate database lookup
    if (id == 1) {
        return Optional.of("John Doe");
    }
    return Optional.empty();
}

public static void practicalExample() {
    int userId = 1;

    // Traditional approach
    String user = findUserById(userId).orElse("Unknown User");
    System.out.println("User: " + user);

    // Chaining operations
    String result = findUserById(userId)
            .map(String::toUpperCase)
            .filter(name -> name.length() > 5)
            .orElse("Default User");
```

```
        System.out.println("Processed user: " + result);
    }
}
```

1.31.2 Date and Time API

```
import java.time.*;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;

public class DateTimeDemo {

    public static void basicDateTimeOperations() {
        // Current date and time
        LocalDate today = LocalDate.now();
        LocalTime now = LocalTime.now();
        LocalDateTime dateTime = LocalDateTime.now();
        ZonedDateTime zonedDateTime = ZonedDateTime.now();

        System.out.println("Today: " + today);
        System.out.println("Now: " + now);
        System.out.println("Date Time: " + dateTime);
        System.out.println("Zoned Date Time: " + zonedDateTime);

        // Creating specific dates
        LocalDate specificDate = LocalDate.of(2023, Month.DECEMBER, 25);
        LocalTime specificTime = LocalTime.of(14, 30, 0);
        LocalDateTime specificDateTime = LocalDateTime.of(specificDate, specificTime);

        System.out.println("Christmas 2023: " + specificDate);
```

```
    System.out.println("Specific time: " + specificTime);
    System.out.println("Specific date time: " + specificDateTime);
}

public static void dateTimeOperations() {
    LocalDate date = LocalDate.of(2023, 6, 15);

    // Adding and subtracting
    LocalDate nextWeek = date.plusWeeks(1);
    LocalDate lastMonth = date.minusMonths(1);
    LocalDate nextYear = date.plusYears(1);

    System.out.println("Original: " + date);
    System.out.println("Next week: " + nextWeek);
    System.out.println("Last month: " + lastMonth);
    System.out.println("Next year: " + nextYear);

    // Getting components
    System.out.println("Year: " + date.getYear());
    System.out.println("Month: " + date.getMonth());
    System.out.println("Day of month: " + date.getDayOfMonth());
    System.out.println("Day of week: " + date.getDayOfWeek());
    System.out.println("Day of year: " + date.getDayOfYear());

    // Comparisons
    LocalDate other = LocalDate.of(2023, 6, 20);
    System.out.println("Is before: " + date.isBefore(other));
    System.out.println("Is after: " + date.isAfter(other));
    System.out.println("Is equal: " + date.isEqual(other));
}
```

```java
public static void formatting() {
    LocalDateTime dateTime = LocalDateTime.of(2023, 6, 15, 14, 30, 45);

    // Predefined formatters
    System.out.println("ISO Local Date: " + dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE));
    System.out.println("ISO Local Time: " + dateTime.format(DateTimeFormatter.ISO_LOCAL_TIME));
    System.out.println("ISO Local DateTime: " + dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    // Custom formatters
    DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");
    System.out.println("Custom format: " + dateTime.format(customFormatter));

    DateTimeFormatter shortFormat = DateTimeFormatter.ofPattern("dd-MMM-yy");
    System.out.println("Short format: " + dateTime.format(shortFormat));

    // Parsing
    String dateString = "25/12/2023 09:30:00";
    LocalDateTime parsed = LocalDateTime.parse(dateString, customFormatter);
    System.out.println("Parsed: " + parsed);
}

public static void timeZones() {
    // Different time zones
    ZonedDateTime utc = ZonedDateTime.now(ZoneId.of("UTC"));
    ZonedDateTime newYork = ZonedDateTime.now(ZoneId.of("America/New_York"));
    ZonedDateTime tokyo = ZonedDateTime.now(ZoneId.of("Asia/Tokyo"));

    System.out.println("UTC: " + utc);
    System.out.println("New York: " + newYork);
    System.out.println("Tokyo: " + tokyo);
```

```
    // Converting between time zones
    ZonedDateTime utcTime = ZonedDateTime.of(2023, 6, 15, 12, 0, 0, 0, ZoneId.of("UTC"));
    ZonedDateTime localTime = utcTime.withZoneSameInstant(ZoneId.systemDefault());

    System.out.println("UTC time: " + utcTime);
    System.out.println("Local time: " + localTime);
}

public static void periods() {
    LocalDate start = LocalDate.of(2023, 1, 1);
    LocalDate end = LocalDate.of(2023, 12, 31);

    // Period between dates
    Period period = Period.between(start, end);
    System.out.println("Period: " + period);
    System.out.println("Months: " + period.getMonths());
    System.out.println("Days: " + period.getDays());

    // Duration between times
    LocalTime startTime = LocalTime.of(9, 0);
    LocalTime endTime = LocalTime.of(17, 30);
    Duration duration = Duration.between(startTime, endTime);

    System.out.println("Duration: " + duration);
    System.out.println("Hours: " + duration.toHours());
    System.out.println("Minutes: " + duration.toMinutes());

    // ChronoUnit for calculations
    long daysBetween = ChronoUnit.DAYS.between(start, end);
    long hoursBetween = ChronoUnit.HOURS.between(startTime, endTime);
```

```
        System.out.println("Days between: " + daysBetween);
        System.out.println("Hours between: " + hoursBetween);
    }
}
```

1.32 Modules (Java 9+)

1.32.1 Module System Basics

```
// module-info.java
module com.example.myapp {
    // Exports packages to other modules
    exports com.example.myapp.api;
    exports com.example.myapp.util to com.example.client;

    // Requires other modules
    requires java.base;  // Implicit, always present
    requires java.logging;
    requires transitive java.sql;  // Transitive dependency

    // Uses services
    uses com.example.myapp.spi.DatabaseProvider;

    // Provides service implementations
    provides com.example.myapp.spi.DatabaseProvider
        with com.example.myapp.impl.MySQLProvider;

    // Opens packages for reflection
```

```
    opens com.example.myapp.model to com.fasterxml.jackson.databind;
}
```

1.32.2 Module Structure

```
myapp/
├── src/
│   ├── module-info.java
│   └── com/
│       └── example/
│           └── myapp/
│               ├── api/
│               │   └── UserService.java
│               ├── impl/
│               │   └── UserServiceImpl.java
│               ├── model/
│               │   └── User.java
│               └── util/
│                   └── StringUtils.java
└── target/
    └── classes/
```

1.32.3 Service Provider Interface

```
// Service interface
package com.example.myapp.spi;

public interface DatabaseProvider {
    String getConnectionString();
```

```
    void connect();
    void disconnect();
}

// Service implementation
package com.example.myapp.impl;

import com.example.myapp.spi.DatabaseProvider;

public class MySQLProvider implements DatabaseProvider {
    @Override
    public String getConnectionString() {
        return "jdbc:mysql://localhost:3306/mydb";
    }

    @Override
    public void connect() {
        System.out.println("Connecting to MySQL database");
    }

    @Override
    public void disconnect() {
        System.out.println("Disconnecting from MySQL database");
    }
}

// Service consumer
package com.example.myapp.service;

import com.example.myapp.spi.DatabaseProvider;
import java.util.ServiceLoader;
```

```
public class DatabaseService {

  public void initializeDatabase() {
    ServiceLoader<DatabaseProvider> loader = ServiceLoader.load(DatabaseProvider.class);

    for (DatabaseProvider provider : loader) {
      System.out.println("Found provider: " + provider.getClass().getName());
      provider.connect();
      // Use the provider...
      provider.disconnect();
    }
  }
}
```

1.33 Records (Java 14+)

1.33.1 Basic Records

```
// Simple record
public record Person(String name, int age) {
  // Automatically generates:
  // - Constructor: Person(String name, int age)
  // - Getters: name(), age()
  // - equals(), hashCode(), toString()
}

// Record with validation
```

```java
public record BankAccount(String accountNumber, double balance) {

  // Compact constructor for validation
  public BankAccount {
    if (accountNumber == null || accountNumber.isBlank()) {
      throw new IllegalArgumentException("Account number cannot be null or blank");
    }
    if (balance < 0) {
      throw new IllegalArgumentException("Balance cannot be negative");
    }
  }

  // Additional methods
  public boolean isOverdrawn() {
    return balance < 0;
  }

  public BankAccount deposit(double amount) {
    if (amount <= 0) {
      throw new IllegalArgumentException("Deposit amount must be positive");
    }
    return new BankAccount(accountNumber, balance + amount);
  }

  public BankAccount withdraw(double amount) {
    if (amount <= 0) {
      throw new IllegalArgumentException("Withdrawal amount must be positive");
    }
    if (amount > balance) {
      throw new IllegalArgumentException("Insufficient funds");
    }
```

```
        return new BankAccount(accountNumber, balance - amount);
    }
}

// Record with static methods
public record Point(double x, double y) {

    public static Point origin() {
        return new Point(0, 0);
    }

    public static Point of(double x, double y) {
        return new Point(x, y);
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public Point translate(double dx, double dy) {
        return new Point(x + dx, y + dy);
    }
}
```

1.33.2 Advanced Record Features

```
// Record implementing interfaces
public record Employee(String name, String department, double salary)
    implements Comparable<Employee> {

    @Override
```

```
    public int compareTo(Employee other) {
        return Double.compare(this.salary, other.salary);
    }

    // Custom constructor
    public Employee(String name, String department) {
        this(name, department, 0.0);
    }

    // Static factory method
    public static Employee intern(String name, String department) {
        return new Employee(name, department, 25000.0);
    }
}

// Generic record
public record Pair<T, U>(T first, U second) {

    public static <T, U> Pair<T, U> of(T first, U second) {
        return new Pair<>(first, second);
    }

    public Pair<U, T> swap() {
        return new Pair<>(second, first);
    }
}

// Nested records
public class OrderService {

    public record Order(String id, Customer customer, List<OrderItem> items) {
```

```
    public record Customer(String name, String email) {}

    public record OrderItem(String productId, int quantity, double price) {
      public double total() {
        return quantity * price;
      }
    }

    public double totalAmount() {
      return items.stream()
            .mapToDouble(OrderItem::total)
            .sum();
    }
  }
}

// Record usage examples
public class RecordDemo {

  public static void main(String[] args) {
    // Creating records
    Person person = new Person("John Doe", 30);
    System.out.println(person.name()); // Getter method
    System.out.println(person.age());
    System.out.println(person); // toString()

    // Records are immutable
    BankAccount account = new BankAccount("12345", 1000.0);
    BankAccount newAccount = account.deposit(500.0); // Returns new instance
```

```
        System.out.println("Original: " + account.balance());
        System.out.println("After deposit: " + newAccount.balance());

        // Using generic records
        Pair<String, Integer> nameAge = Pair.of("Alice", 25);
        Pair<Integer, String> swapped = nameAge.swap();

        System.out.println("Original: " + nameAge);
        System.out.println("Swapped: " + swapped);

        // Pattern matching with records (Java 17+)
        processPoint(new Point(3, 4));
    }

    // Pattern matching with records
    public static void processPoint(Point point) {
        switch (point) {
            case Point(0, 0) -> System.out.println("Origin point");
            case Point(double x, 0) -> System.out.println("Point on X-axis: " + x);
            case Point(0, double y) -> System.out.println("Point on Y-axis: " + y);
            case Point(double x, double y) -> System.out.println("Point at (" + x + ", " + y + ")");
        }
    }
}
```

1.34 Pattern Matching

1.34.1 Pattern Matching with instanceof (Java 16+)

```
public class PatternMatchingDemo {

  // Traditional instanceof
  public static String processObjectOld(Object obj) {
    if (obj instanceof String) {
      String str = (String) obj;  // Explicit cast
      return "String of length " + str.length();
    } else if (obj instanceof Integer) {
      Integer num = (Integer) obj;  // Explicit cast
      return "Integer with value " + num;
    } else if (obj instanceof Double) {
      Double d = (Double) obj;  // Explicit cast
      return "Double with value " + d;
    }
    return "Unknown type";
  }

  // Pattern matching with instanceof
  public static String processObject(Object obj) {
    if (obj instanceof String str) {  // Pattern variable
      return "String of length " + str.length();
    } else if (obj instanceof Integer num) {
      return "Integer with value " + num;
    } else if (obj instanceof Double d) {
      return "Double with value " + d;
    }
    return "Unknown type";
```

```
  }

  // Pattern matching with guards
  public static String processObjectWithGuards(Object obj) {
    if (obj instanceof String str && str.length() > 5) {
      return "Long string: " + str;
    } else if (obj instanceof String str) {
      return "Short string: " + str;
    } else if (obj instanceof Integer num && num > 0) {
      return "Positive integer: " + num;
    } else if (obj instanceof Integer num) {
      return "Non-positive integer: " + num;
    }
    return "Unknown or null";
  }
}
```

1.34.2 Switch Expressions (Java 14+)

```
public class SwitchExpressions {

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

  // Traditional switch statement
  public static String getDayTypeOld(Day day) {
    String dayType;
    switch (day) {
      case MONDAY:
      case TUESDAY:
```

```
      case WEDNESDAY:
      case THURSDAY:
      case FRIDAY:
        dayType = "Weekday";
        break;
      case SATURDAY:
      case SUNDAY:
        dayType = "Weekend";
        break;
      default:
        dayType = "Unknown";
    }
    return dayType;
}

// Switch expression
public static String getDayType(Day day) {
    return switch (day) {
      case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> "Weekday";
      case SATURDAY, SUNDAY -> "Weekend";
    };
}

// Switch expression with yield
public static String getDayDescription(Day day) {
    return switch (day) {
      case MONDAY -> {
        System.out.println("Start of work week");
        yield "Monday Blues";
      }
      case FRIDAY -> {
        System.out.println("End of work week");
```

```
            yield "TGIF";
        }
        case SATURDAY, SUNDAY -> "Weekend Fun";
        default -> "Regular Day";
    };
  }

  // Pattern matching in switch (Java 17+)
  public static String formatValue(Object obj) {
    return switch (obj) {
        case null -> "null value";
        case String s -> "String: " + s;
        case Integer i -> "Integer: " + i;
        case Double d -> "Double: " + d;
        case Boolean b -> "Boolean: " + b;
        default -> "Unknown type: " + obj.getClass().getSimpleName();
    };
  }

  // Guarded patterns (Java 17+)
  public static String categorizeNumber(Object obj) {
    return switch (obj) {
        case Integer i when i > 0 -> "Positive integer: " + i;
        case Integer i when i < 0 -> "Negative integer: " + i;
        case Integer i -> "Zero";
        case Double d when d > 0.0 -> "Positive double: " + d;
        case Double d when d < 0.0 -> "Negative double: " + d;
        case Double d -> "Zero double";
        default -> "Not a number";
    };
    }
}
```

1.35 Virtual Threads (Java 19+)

1.35.1 Virtual Threads Basics

```
import java.time.Duration;
import java.util.concurrent.Executors;

public class VirtualThreadsDemo {

  public static void traditionalThreads() {
    System.out.println("=== Traditional Threads ===");

    // Creating traditional threads
    for (int i = 0; i < 5; i++) {
      final int taskId = i;
      Thread thread = new Thread(() -> {
        System.out.println("Traditional thread " + taskId +
                  " running on " + Thread.currentThread());
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
        System.out.println("Traditional thread " + taskId + " completed");
      });
      thread.start();
    }
  }
```

```
public static void virtualThreads() {
  System.out.println("=== Virtual Threads ===");

  // Creating virtual threads
  for (int i = 0; i < 5; i++) {
    final int taskId = i;
    Thread virtualThread = Thread.ofVirtual().start(() -> {
      System.out.println("Virtual thread " + taskId +
                  " running on " + Thread.currentThread());
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
      System.out.println("Virtual thread " + taskId + " completed");
    });
  }
}

public static void virtualThreadExecutor() {
  System.out.println("=== Virtual Thread Executor ===");

  // Using virtual thread executor
  try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
    for (int i = 0; i < 10; i++) {
      final int taskId = i;
      executor.submit(() -> {
        System.out.println("Task " + taskId +
                    " on " + Thread.currentThread());
        try {
```

```
            Thread.sleep(Duration.ofSeconds(1));
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
          }
          return "Task " + taskId + " result";
        });
      }
    }
  }

  public static void massiveVirtualThreads() {
    System.out.println("=== Massive Virtual Threads ===");

    long startTime = System.currentTimeMillis();

    // Create many virtual threads (would be impossible with platform threads)
    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
      for (int i = 0; i < 100_000; i++) {
        final int taskId = i;
        executor.submit(() -> {
          try {
            Thread.sleep(Duration.ofMillis(100));
          } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
          }
          if (taskId % 10000 == 0) {
            System.out.println("Completed task " + taskId);
          }
        });
      }
    }
```

```
        long endTime = System.currentTimeMillis();
        System.out.println("Completed 100,000 virtual threads in " +
                (endTime - startTime) + "ms");
    }

    public static void virtualThreadProperties() {
        Thread virtualThread = Thread.ofVirtual()
                .name("my-virtual-thread")
                .start(() -> {
                    Thread current = Thread.currentThread();
                    System.out.println("Thread name: " + current.getName());
                    System.out.println("Is virtual: " + current.isVirtual());
                    System.out.println("Thread ID: " + current.threadId());
                });

        try {
            virtualThread.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

1.36 JDBC

1.36.1 Basic JDBC Operations

```java
import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class JDBCDemo {

    private static final String URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String USERNAME = "user";
    private static final String PASSWORD = "password";

    public static class User {
        private int id;
        private String name;
        private String email;

        public User(int id, String name, String email) {
            this.id = id;
            this.name = name;
            this.email = email;
        }

        // Getters and setters
        public int getId() { return id; }
        public String getName() { return name; }
        public String getEmail() { return email; }
```

```
    @Override
    public String toString() {
        return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";
    }
}

public static Connection getConnection() throws SQLException {
    return DriverManager.getConnection(URL, USERNAME, PASSWORD);
}

public static void createTable() {
    String sql = """
            CREATE TABLE IF NOT EXISTS users (
                id INT PRIMARY KEY AUTO_INCREMENT,
                name VARCHAR(100) NOT NULL,
                email VARCHAR(100) UNIQUE NOT NULL
            )
            """;

    try (Connection conn = getConnection();
         Statement stmt = conn.createStatement()) {

        stmt.executeUpdate(sql);
        System.out.println("Table created successfully");

    } catch (SQLException e) {
        System.err.println("Error creating table: " + e.getMessage());
    }
}

public static void insertUser(String name, String email) {
```

```
    String sql = "INSERT INTO users (name, email) VALUES (?, ?)";

    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setString(1, name);
        pstmt.setString(2, email);

        int rowsAffected = pstmt.executeUpdate();
        System.out.println("Inserted " + rowsAffected + " row(s)");

    } catch (SQLException e) {
        System.err.println("Error inserting user: " + e.getMessage());
    }
}

public static List<User> getAllUsers() {
    List<User> users = new ArrayList<>();
    String sql = "SELECT id, name, email FROM users";

    try (Connection conn = getConnection();
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            int id = rs.getInt("id");
            String name = rs.getString("name");
            String email = rs.getString("email");
            users.add(new User(id, name, email));
        }
```

```
    } catch (SQLException e) {
        System.err.println("Error retrieving users: " + e.getMessage());
    }

    return users;
}

public static User getUserById(int id) {
    String sql = "SELECT id, name, email FROM users WHERE id = ?";

    try (Connection conn = getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, id);

        try (ResultSet rs = pstmt.executeQuery()) {
            if (rs.next()) {
                return new User(
                    rs.getInt("id"),
                    rs.getString("name"),
                    rs.getString("email")
                );
            }
        }

    } catch (SQLException e) {
        System.err.println("Error retrieving user: " + e.getMessage());
    }

    return null;
}
```

```
public static void updateUser(int id, String name, String email) {
  String sql = "UPDATE users SET name = ?, email = ? WHERE id = ?";

  try (Connection conn = getConnection();
      PreparedStatement pstmt = conn.prepareStatement(sql)) {

    pstmt.setString(1, name);
    pstmt.setString(2, email);
    pstmt.setInt(3, id);

    int rowsAffected = pstmt.executeUpdate();
    System.out.println("Updated " + rowsAffected + " row(s)");

  } catch (SQLException e) {
    System.err.println("Error updating user: " + e.getMessage());
  }
}

public static void deleteUser(int id) {
  String sql = "DELETE FROM users WHERE id = ?";

  try (Connection conn = getConnection();
      PreparedStatement pstmt = conn.prepareStatement(sql)) {

    pstmt.setInt(1, id);

    int rowsAffected = pstmt.executeUpdate();
    System.out.println("Deleted " + rowsAffected + " row(s)");

  } catch (SQLException e) {
```

```
            System.err.println("Error deleting user: " + e.getMessage());
        }
    }
}
```

1.36.2 Advanced JDBC Features

```
import java.sql.*;

public class AdvancedJDBC {

    public static void transactionExample() {
        String insertUser = "INSERT INTO users (name, email) VALUES (?, ?)";
        String insertProfile = "INSERT INTO user_profiles (user_id, bio) VALUES (?, ?)";

        try (Connection conn = DriverManager.getConnection(URL, USERNAME, PASSWORD)) {

            // Disable auto-commit for transaction
            conn.setAutoCommit(false);

            try (PreparedStatement userStmt = conn.prepareStatement(insertUser,
                                 Statement.RETURN_GENERATED_KEYS);
                 PreparedStatement profileStmt = conn.prepareStatement(insertProfile)) {

                // Insert user
                userStmt.setString(1, "John Doe");
                userStmt.setString(2, "john@example.com");
                userStmt.executeUpdate();

                // Get generated user ID
                ResultSet generatedKeys = userStmt.getGeneratedKeys();
```

```
            int userId = 0;
            if (generatedKeys.next()) {
                userId = generatedKeys.getInt(1);
            }

            // Insert profile
            profileStmt.setInt(1, userId);
            profileStmt.setString(2, "Software Developer");
            profileStmt.executeUpdate();

            // Commit transaction
            conn.commit();
            System.out.println("Transaction completed successfully");

        } catch (SQLException e) {
            // Rollback on error
            conn.rollback();
            System.err.println("Transaction rolled back: " + e.getMessage());
        } finally {
            // Restore auto-commit
            conn.setAutoCommit(true);
        }

    } catch (SQLException e) {
        System.err.println("Connection error: " + e.getMessage());
    }
}

public static void batchProcessing() {
    String sql = "INSERT INTO users (name, email) VALUES (?, ?)";
```

```
        try (Connection conn = DriverManager.getConnection(URL, USERNAME, PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            // Disable auto-commit for better performance
            conn.setAutoCommit(false);

            // Add multiple statements to batch
            String[][] users = {
                {"Alice", "alice@example.com"},
                {"Bob", "bob@example.com"},
                {"Charlie", "charlie@example.com"}
            };

            for (String[] user : users) {
                pstmt.setString(1, user[0]);
                pstmt.setString(2, user[1]);
                pstmt.addBatch();
            }

            // Execute batch
            int[] results = pstmt.executeBatch();
            conn.commit();

            System.out.println("Batch executed. Rows affected: " + results.length);

        } catch (SQLException e) {
            System.err.println("Batch processing error: " + e.getMessage());
        }
    }

    public static void callableStatementExample() {
```

```
        // Assuming a stored procedure exists
        String sql = "{CALL getUsersByDepartment(?, ?)}";

        try (Connection conn = DriverManager.getConnection(URL, USERNAME, PASSWORD);
             CallableStatement cstmt = conn.prepareCall(sql)) {

            // Set input parameter
            cstmt.setString(1, "Engineering");

            // Register output parameter
            cstmt.registerOutParameter(2, Types.INTEGER);

            // Execute stored procedure
            ResultSet rs = cstmt.executeQuery();

            while (rs.next()) {
                System.out.println("User: " + rs.getString("name"));
            }

            // Get output parameter
            int totalCount = cstmt.getInt(2);
            System.out.println("Total users: " + totalCount);

        } catch (SQLException e) {
            System.err.println("Callable statement error: " + e.getMessage());
        }
    }
}
```

1.37 Networking

1.37.1 Socket Programming

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

// Simple TCP Server
public class SimpleServer {
  private static final int PORT = 8080;

  public static void main(String[] args) {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
      System.out.println("Server started on port " + PORT);

      while (true) {
        Socket clientSocket = serverSocket.accept();
        System.out.println("Client connected: " + clientSocket.getInetAddress());

        // Handle client in separate thread
        new Thread(() -> handleClient(clientSocket)).start();
      }

    } catch (IOException e) {
      System.err.println("Server error: " + e.getMessage());
    }
  }

  private static void handleClient(Socket clientSocket) {
    try (BufferedReader in = new BufferedReader(
```

```
            new InputStreamReader(clientSocket.getInputStream()));
           PrintWriter out = new PrintWriter(
            clientSocket.getOutputStream(), true)) {

          String inputLine;
          while ((inputLine = in.readLine()) != null) {
            System.out.println("Received: " + inputLine);

            if ("bye".equalsIgnoreCase(inputLine)) {
              out.println("Goodbye!");
              break;
            }

            // Echo back to client
            out.println("Echo: " + inputLine);
          }

        } catch (IOException e) {
          System.err.println("Client handling error: " + e.getMessage());
        } finally {
          try {
            clientSocket.close();
          } catch (IOException e) {
            System.err.println("Error closing client socket: " + e.getMessage());
          }
        }
      }
    }

    // Simple TCP Client
    public class SimpleClient {
```

```
    private static final String HOST = "localhost";
    private static final int PORT = 8080;

    public static void main(String[] args) {
      try (Socket socket = new Socket(HOST, PORT);
          PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
          BufferedReader in = new BufferedReader(
            new InputStreamReader(socket.getInputStream()));
          BufferedReader stdIn = new BufferedReader(
            new InputStreamReader(System.in))) {

        System.out.println("Connected to server. Type 'bye' to exit.");

        String userInput;
        while ((userInput = stdIn.readLine()) != null) {
          out.println(userInput);

          String response = in.readLine();
          System.out.println("Server response: " + response);

          if ("bye".equalsIgnoreCase(userInput)) {
            break;
          }
        }

      } catch (IOException e) {
        System.err.println("Client error: " + e.getMessage());
      }
    }
}
```

1.37.2 HTTP Client (Java 11+)

```
import java.net.http.*;
import java.net.URI;
import java.time.Duration;
import java.util.concurrent.CompletableFuture;

public class HTTPClientDemo {

   public static void synchronousRequests() throws Exception {
      HttpClient client = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(10))
            .build();

      // GET request
      HttpRequest getRequest = HttpRequest.newBuilder()
            .uri(URI.create("https://jsonplaceholder.typicode.com/posts/1"))
            .timeout(Duration.ofSeconds(30))
            .build();

      HttpResponse<String> getResponse = client.send(getRequest,
            HttpResponse.BodyHandlers.ofString());

      System.out.println("GET Response Code: " + getResponse.statusCode());
      System.out.println("GET Response Body: " + getResponse.body());

      // POST request
      String jsonBody = """
            {
              "title": "My Post",
              "body": "This is my post content",
```

```
            "userId": 1
        }
        """;

    HttpRequest postRequest = HttpRequest.newBuilder()
        .uri(URI.create("https://jsonplaceholder.typicode.com/posts"))
        .header("Content-Type", "application/json")
        .POST(HttpRequest.BodyPublishers.ofString(jsonBody))
        .build();

    HttpResponse<String> postResponse = client.send(postRequest,
        HttpResponse.BodyHandlers.ofString());

    System.out.println("POST Response Code: " + postResponse.statusCode());
    System.out.println("POST Response Body: " + postResponse.body());
}

public static void asynchronousRequests() {
    HttpClient client = HttpClient.newHttpClient();

    HttpRequest request = HttpRequest.newBuilder()
        .uri(URI.create("https://jsonplaceholder.typicode.com/posts"))
        .build();

    // Asynchronous request
    CompletableFuture<HttpResponse<String>> future = client.sendAsync(request,
        HttpResponse.BodyHandlers.ofString());

    future.thenApply(HttpResponse::body)
        .thenAccept(System.out::println)
        .join();  // Wait for completion
```

```
    // Multiple asynchronous requests
    CompletableFuture<HttpResponse<String>>[] futures = new CompletableFuture[5];

    for (int i = 1; i <= 5; i++) {
      HttpRequest req = HttpRequest.newBuilder()
          .uri(URI.create("https://jsonplaceholder.typicode.com/posts/" + i))
          .build();

      futures[i-1] = client.sendAsync(req, HttpResponse.BodyHandlers.ofString());
    }

    // Wait for all requests to complete
    CompletableFuture.allOf(futures)
        .thenRun(() -> {
          for (CompletableFuture<HttpResponse<String>> future1 : futures) {
            try {
              HttpResponse<String> response = future1.get();
              System.out.println("Status: " + response.statusCode());
            } catch (Exception e) {
              System.err.println("Error: " + e.getMessage());
            }
          }
        })
        .join();
  }
}
```

1.38 Serialization

1.38.1 Basic Serialization

```
import java.io.*;

public class SerializationDemo {

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

    private String name;
    private int age;
    private transient String password;  // Won't be serialized
    private static String company = "TechCorp";  // Won't be serialized

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

    // Custom serialization
    private void writeObject(ObjectOutputStream out) throws IOException {
      out.defaultWriteObject();
      // Custom serialization logic
      out.writeObject(encrypt(password));
    }

    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
```

```
        in.defaultReadObject();
        // Custom deserialization logic
        String encryptedPassword = (String) in.readObject();
        this.password = decrypt(encryptedPassword);
    }

    private String encrypt(String data) {
        // Simple encryption (not secure)
        return data != null ? new StringBuilder(data).reverse().toString() : null;
    }

    private String decrypt(String data) {
        // Simple decryption
        return data != null ? new StringBuilder(data).reverse().toString() : null;
    }

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

public static void serializeObject() {
    Person person = new Person("John Doe", 30, "secret123");

    try (FileOutputStream fos = new FileOutputStream("person.ser");
         ObjectOutputStream oos = new ObjectOutputStream(fos)) {

        oos.writeObject(person);
        System.out.println("Object serialized successfully");
```

```
    } catch (IOException e) {
      System.err.println("Serialization error: " + e.getMessage());
    }
  }

  public static void deserializeObject() {
    try (FileInputStream fis = new FileInputStream("person.ser");
         ObjectInputStream ois = new ObjectInputStream(fis)) {

      Person person = (Person) ois.readObject();
      System.out.println("Deserialized object: " + person);

    } catch (IOException | ClassNotFoundException e) {
      System.err.println("Deserialization error: " + e.getMessage());
    }
  }
}
```

1.38.2 Externalization

```
import java.io.*;

public class ExternalizationDemo {

  public static class Employee implements Externalizable {
    private String name;
    private int id;
    private double salary;
    private String department;
```

```java
// Required no-arg constructor for Externalizable
public Employee() {}

public Employee(String name, int id, double salary, String department) {
  this.name = name;
  this.id = id;
  this.salary = salary;
  this.department = department;
}

@Override
public void writeExternal(ObjectOutput out) throws IOException {
  // Custom serialization - full control
  out.writeUTF(name);
  out.writeInt(id);
  out.writeDouble(salary);
  out.writeUTF(department);

  // Can add compression, encryption, etc.
  System.out.println("Custom serialization performed");
}

@Override
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
  // Custom deserialization - must match writeExternal order
  this.name = in.readUTF();
  this.id = in.readInt();
  this.salary = in.readDouble();
  this.department = in.readUTF();

  System.out.println("Custom deserialization performed");
```

```
    }

    @Override
    public String toString() {
      return "Employee{name='" + name + "', id=" + id +
          ", salary=" + salary + ", department='" + department + "'}";
    }
  }

  public static void main(String[] args) {
    Employee emp = new Employee("Alice Johnson", 123, 75000.0, "Engineering");

    // Serialize
    try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("employee.ser"))) {
      oos.writeObject(emp);
    } catch (IOException e) {
      System.err.println("Serialization error: " + e.getMessage());
    }

    // Deserialize
    try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("employee.ser"))) {
      Employee deserializedEmp = (Employee) ois.readObject();
      System.out.println("Deserialized: " + deserializedEmp);
    } catch (IOException | ClassNotFoundException e) {
      System.err.println("Deserialization error: " + e.getMessage());
    }
  }
}
```

1.39 Testing with JUnit

1.39.1 JUnit 5 Basics

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

  private Calculator calculator;

  @BeforeAll
  static void setUpClass() {
    System.out.println("Setting up test class");
  }

  @BeforeEach
  void setUp() {
    calculator = new Calculator();
    System.out.println("Setting up test method");
  }

  @AfterEach
  void tearDown() {
    System.out.println("Cleaning up after test method");
  }
```

```
@AfterAll
static void tearDownClass() {
   System.out.println("Cleaning up test class");
}

@Test
@DisplayName("Addition should work correctly")
void testAddition() {
   assertEquals(5, calculator.add(2, 3));
   assertEquals(0, calculator.add(-1, 1));
   assertEquals(-5, calculator.add(-2, -3));
}

@Test
void testDivision() {
   assertEquals(2.0, calculator.divide(10, 5), 0.001);

   // Test exception
   assertThrows(ArithmeticException.class, () -> {
      calculator.divide(10, 0);
   });
}

@ParameterizedTest
@ValueSource(ints = {1, 2, 3, 4, 5})
void testSquare(int number) {
   int result = calculator.square(number);
   assertEquals(number * number, result);
}

@ParameterizedTest
```

```java
@CsvSource({
  "1, 1, 2",
  "2, 3, 5",
  "5, 7, 12"
})
void testAdditionWithCsv(int a, int b, int expected) {
  assertEquals(expected, calculator.add(a, b));
}

@Test
@Timeout(value = 2, unit = TimeUnit.SECONDS)
void testPerformance() {
  // Test that should complete within 2 seconds
  calculator.complexCalculation();
}

@Test
@Disabled("Not implemented yet")
void testFutureFeature() {
  // This test will be skipped
}

@Nested
@DisplayName("Tests for negative numbers")
class NegativeNumberTests {

  @Test
  void testNegativeAddition() {
    assertEquals(-5, calculator.add(-2, -3));
  }
```

```
    @Test
    void testNegativeMultiplication() {
      assertEquals(6, calculator.multiply(-2, -3));
    }
  }
}

// Calculator class for testing
class Calculator {

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

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

  public int square(int number) {
    return number * number;
  }

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

  public void complexCalculation() {
    // Simulate complex calculation
```

```
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

1.40 Build Tools

1.40.1 Maven

```
<!-- pom.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-java-app</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <name>My Java Application</name>
  <description>A sample Java application</description>
```

```
<properties>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <junit.version>5.9.2</junit.version>
</properties>

<dependencies>
  <!-- JUnit 5 -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>

  <!-- Jackson for JSON processing -->
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.2</version>
  </dependency>

  <!-- Apache Commons Lang -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
  </dependency>
</dependencies>
```

```
<build>
  <plugins>
    <!-- Compiler Plugin -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <source>17</source>
        <target>17</target>
      </configuration>
    </plugin>

    <!-- Surefire Plugin for running tests -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0</version>
    </plugin>

    <!-- JAR Plugin -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.3.0</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>com.example.Main</mainClass>
          </manifest>
        </archive>
```

```
      </configuration>
    </plugin>

    <!-- JaCoCo for code coverage -->
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.8</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
        </execution>
        <execution>
          <id>report</id>
          <phase>test</phase>
          <goals>
            <goal>report</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
 </build>
</project>
```

1.40.2 Gradle

```
// build.gradle
plugins {
```

```
    id 'java'
    id 'application'
    id 'jacoco'
}

group = 'com.example'
version = '1.0.0'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    // JUnit 5
    testImplementation 'org.junit.jupiter:junit-jupiter:5.9.2'

    // Jackson for JSON processing
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.15.2'

    // Apache Commons Lang
    implementation 'org.apache.commons:commons-lang3:3.12.0'

    // Logging
    implementation 'org.slf4j:slf4j-api:2.0.7'
    implementation 'ch.qos.logback:logback-classic:1.4.7'
}

application {
    mainClass = 'com.example.Main'
}
```

```
test {
    useJUnitPlatform()

    // JVM arguments for tests
    jvmArgs '-Xmx1g'

    // Test logging
    testLogging {
        events "passed", "skipped", "failed"
        exceptionFormat "full"
    }
}

jacoco {
    toolVersion = "0.8.8"
}

jacocoTestReport {
    reports {
        xml.required = true
        html.required = true
    }
}

// Custom task
task printClasspath {
    doLast {
        configurations.runtimeClasspath.each { println it }
    }
}
```

```
// Fat JAR task
task fatJar(type: Jar) {
  manifest {
    attributes 'Main-Class': 'com.example.Main'
  }
  archiveClassifier = 'all'
  from {
    configurations.runtimeClasspath.collect { it.isDirectory() ? it : zipTree(it) }
  }
  with jar
}
```