Problem Solver (DI03032021) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Introduction to Java Programming

1.1 Platform Independence and Execution Architecture

Methodology:

Platform Independence Execution Flow The Java execution model guarantees platform independence through a systematic two-step process. This mechanism eliminates the need to rewrite code for different operating systems:

1. **Source to Bytecode Compilation:** The Java compiler translates the original `.java` source files into an intermediate standardized representation known as bytecode (stored in `.class` files). This bytecode is not specific to any hardware architecture.
2. **Bytecode Translation:** The Java Virtual Machine (JVM) acts as a software-based interpreter and processor. It reads the platform-agnostic bytecode and dynamically translates it into the native machine code of the host operating system.
3. **Execution Paradigm:** Because the JVM handles hardware-specific translation the exact same compiled bytecode can be distributed and run on any machine equipped with a JVM resulting in the "Write Once Run Anywhere" capability.

1.2 Java Program Structure and Execution

Methodology:

Compiling and Executing Java Programs To transform Java source code into a functional executing application follow these procedural steps:

1. **Write Source Code:** Write the Java program and save the file with a `.java` extension. The filename must exactly match the name of the public class defined within it.
2. **Open Terminal:** Launch the command-line interface and navigate to the directory where the source file is stored.
3. **Compile Code:** Translate the human-readable Java code into machine-independent bytecode using the Java compiler command: `javac ClassName.java`. If successful this generates a `ClassName.class` file.
4. **Execute Code:** Run the generated bytecode on the JVM using the Java execution command: `java ClassName`. Do not append the `.class` extension during execution.

Worked Example:

Compile and Execute a Welcome Program Write a simple Java program that prints "Welcome to Java Programming!" to the console. Detail the exact commands to compile and run it.

Solution:
Step 1: Write the Java code and save it as `Welcome.java`.

```
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java Programming!");
    }
}
```

Step 2: Open the terminal and navigate to the file's directory.

```
cd /path/to/directory
```

Step 3: Compile the source code.

```
javac Welcome.java
```

Result: A new file named `Welcome.class` is generated containing the bytecode.
Step 4: Execute the compiled class.

```
java Welcome
```

Output:

```
Welcome to Java Programming!
```

1.3 Data Types and Type Casting

Methodology:

Type Conversion and Casting When assigning a value of one primitive data type to another type casting is frequently required. Apply the following algorithmic rules:

1. **Identify Types:** Determine the source data type and the destination data type.
2. **Implicit Casting (Widening):** If the destination type has a larger memory footprint than the source type (e.g. `int` to `double`) simply assign the value. The JVM handles this automatically without data loss.
3. **Explicit Casting (Narrowing):** If the destination type is smaller than the source type (e.g. `double` to `int`) manually cast it by placing the target type in parentheses before the value: `(target_type) value`.
4. **Account for Data Loss:** During explicit casting fractional parts will be truncated (not rounded) and values exceeding the target type's maximum limit will wrap around causing unexpected numerical results.

Worked Example:

Implement Implicit and Explicit Casting Write a Java program to convert an integer to a double and a double to a byte. Observe the output to verify data loss.
Solution:
Step 1: Write the Java code implementing both casting methods.

```
public class TypeCasting {
    public static void main(String[] args) {
        // 1. Implicit Casting (int to double)
        int originalInt = 150;
        double widenedDouble = originalInt;
```

```
        // 2. Explicit Casting (double to byte)
        double originalDouble = 130.75;
        byte narrowedByte = (byte) originalDouble;

        System.out.println("Original Integer: " + originalInt);
        System.out.println("Widened Double: " + widenedDouble);
        System.out.println("Original Double: " + originalDouble);
        System.out.println("Narrowed Byte: " + narrowedByte);
    }
}
```

Step 2: Analyze the transformations.

- The integer 150 is safely converted to 150.0 because a `double` has more precision.
- The double 130.75 first loses its fractional part (0.75) becoming 130.
- A `byte` can only hold values from -128 to 127. Since 130 exceeds this range it wraps around the byte limit: $130 - 256 = -126$.

Output:

```
Original Integer: 150
Widened Double: 150.0
Original Double: 130.75
Narrowed Byte: -126
```

1.4 Operators and Expression Evaluation

Methodology:

Evaluating Expressions with Operators To correctly compute mathematical and logical expressions in Java apply these procedural rules:

1. **Precedence Rules:** Evaluate operators with higher precedence first. Multiplication (`*`), Division (`/`), and Modulo (`%`) are evaluated before Addition (`+`) and Subtraction (`-`).
2. **Associativity:** When operators have the same precedence evaluate them from left to right for arithmetic operations.
3. **Parentheses:** Use parentheses `()` to override default precedence rules. Innermost parentheses are resolved first.
4. **Bitwise Operations:** For bitwise operators (`& | ^ « »`) first convert the decimal operands into their binary equivalents. Perform the bitwise logic on corresponding bits and convert the resulting binary sequence back to a decimal number.

Worked Example:

Arithmetic Operator Precedence Evaluate the expression `result = a + b * c / d - e % f` where $a = 10$, $b = 5$, $c = 4$, $d = 2$, $e = 9$, and $f = 4$.

Solution:

Step 1: Substitute the values into the expression.

```
result = 10 + 5 * 4 / 2 - 9 % 4
```

Step 2: Apply Precedence (Multiplication Division Modulo). Evaluate `5 * 4`, `20 / 2`, and `9 % 4` from left to right:

- `5 * 4 = 20`
- `20 / 2 = 10`
- `9 % 4 = 1` (Remainder of $9 \div 4$)

The expression simplifies to: `result = 10 + 10 - 1`
Step 3: Apply Associativity (Addition Subtraction). Evaluate from left to right:

- `10 + 10 = 20`
- `20 - 1 = 19`

Step 4: Write the Java verification code.

```
public class Precedence {
    public static void main(String[] args) {
        int result = 10 + 5 * 4 / 2 - 9 % 4;
        System.out.println("Result = " + result);
    }
}
```

Output:

```
Result = 19
```

Worked Example:

Swap Variables using Bitwise XOR Write a Java program to swap the values of two integer variables without using a temporary third variable by utilizing the bitwise XOR (`^`) operator.
Solution:
Step 1: Understand the XOR properties.

- $A \oplus A = 0$
- $A \oplus 0 = A$
- $A \oplus B \oplus A = B$

Step 2: Trace the algorithm with $x = 5$ and $y = 9$. Binary representation: $x = 0101_2$, $y = 1001_2$

1. `x = x ^ y;`
   $0101 \oplus 1001 = 1100_2$ (Decimal 12). Now $x = 12$.
2. `y = x ^ y;`
   $1100 \oplus 1001 = 0101_2$ (Decimal 5). Now $y = 5$.
3. `x = x ^ y;`
   $1100 \oplus 0101 = 1001_2$ (Decimal 9). Now $x = 9$.

Step 3: Write the Java code.

```
public class SwapXor {
    public static void main(String[] args) {
        int x = 5;
        int y = 9;

        System.out.println("Before Swap: x = " + x + " y = " + y);

        x = x ^ y;
        y = x ^ y;
```

```
        x = x ^ y;

        System.out.println("After Swap: x = " + x + " y = " + y);
    }
}
```

Output:

```
Before Swap: x = 5 y = 9
After Swap: x = 9 y = 5
```

CHAPTER 2

Object-Oriented Programming Concepts

2.1 Object-Oriented Programming Fundamentals

Methodology:

Transitioning to Object-Oriented Design To model a real-world problem using Object-Oriented Programming (OOP) in Java:

1. **Identify Entities:** Determine the objects involved in the problem (e.g. Student or Account).
2. **Define Attributes:** List the data or state associated with each object (e.g. name or balance).
3. **Define Behaviors:** Identify actions that the object can perform (e.g. study or withdraw).
4. **Encapsulate:** Combine attributes and behaviors into a single unit called a `class`.

Worked Example:

Model a basic real-world entity **Problem:** Model a simple `Rectangle` entity that calculates its area and perimeter.
Step 1: Identify Entity and Attributes Entity: Rectangle Attributes: `length` and `width` (Data type: `double`)
Step 2: Identify Behaviors Behaviors: `calculateArea()` and `calculatePerimeter()`
Step 3: Java Implementation

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

    double calculateArea() {
        return length * width;
    }

    double calculatePerimeter() {
        return 2 * (length + width);
    }
}
```

2.2 Classes and Objects

Methodology:

Creating Classes and Instantiating Objects

1. **Class Definition:** Use the `class` keyword to define a blueprint.
2. **Object Declaration:** Declare a reference variable of the class type: `ClassName objName;`
3. **Memory Allocation:** Use the `new` keyword to dynamically allocate memory on the heap: `objName = new ClassName();`
4. **Garbage Collection:** Objects that are no longer referenced are automatically destroyed by the Java Garbage Collector to free memory.
5. **Accessing Members:** Use the dot operator (`.`) to access attributes and methods: `objName.attributeName` or `objName.methodName()`.

Worked Example:

Define a class and create objects **Problem:** Write a Java program to create a `Book` class with attributes for title and price. Instantiate two book objects, assign values, and display their details.

Step 1: Define the `Book` class

```
class Book {
    String title;
    double price;

    void displayDetails() {
        System.out.println("Title: " + title);
        System.out.println("Price: $" + price);
    }
}
```

Step 2: Create the main class to instantiate objects

```
public class Library {
    public static void main(String[] args) {
        // Step 3: Allocate memory using 'new'
        Book book1 = new Book();
        Book book2 = new Book();

        // Step 4: Access members using dot operator
        book1.title = "Java Programming";
        book1.price = 45.50;

        book2.title = "Data Structures";
        book2.price = 55.00;

        // Display details
        book1.displayDetails();
        System.out.println("---");
        book2.displayDetails();

        // Step 5: Removing references triggers garbage collection
        book1 = null;
    }
}
```

2.3 Encapsulation and Access Modifiers

Methodology:

Applying Encapsulation and Data Hiding

1. **Private Fields:** Declare instance variables with the `private` access modifier to prevent direct external access.
2. **Getter Methods:** Create `public` methods returning the field's type to read the value of a private variable.
3. **Setter Methods:** Create `public` methods with a parameter of the field's type to modify the value, integrating any necessary validation rules.

Worked Example:

Implement secure data access using Encapsulation **Problem:** Create a `BankAccount` class where the balance cannot be modified directly or set to a negative value. Provide methods to deposit and withdraw money safely.

Step 1: Declare private fields

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

Step 2: Provide Getter and Setter for Account Number

```
    public String getAccountNumber() {
        return accountNumber;
    }

    public void setAccountNumber(String accNum) {
        this.accountNumber = accNum;
    }
```

Step 3: Implement controlled modifications for Balance

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

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: $" + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: $" + amount);
        } else {
            System.out.println("Insufficient funds or invalid amount.");
        }
```

```
    }
}
```

Step 4: Test the encapsulation in main method

```
public class BankSystem {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        acc.setAccountNumber("1001A");

        // acc.balance = 5000; // Compilation Error: private access

        acc.deposit(1000);
        acc.withdraw(300);
        acc.withdraw(2000); // Handled by validation logic

        System.out.println("Final Balance: $" + acc.getBalance());
    }
}
```

2.4 Constructors and the `this` Keyword

Methodology:

Implementing Constructors and Keyword `this`

1. **Default Constructor:** Define a parameterless method with the same name as the class to initialize default state.
2. **Parameterized Constructor:** Define a constructor with parameters to accept initialization values at creation time.
3. **Resolving Shadowing:** Use `this.variableName` to refer to instance variables when they are shadowed by constructor or method parameters with identical names.
4. **Constructor Chaining:** Use `this(args)` as the first statement in a constructor to call another overloaded constructor within the same class.

Worked Example:

Use Constructors and `this` keyword **Problem:** Write a program to manage an `Employee` class with ID and name. Use default and parameterized constructors. Chain constructors to reuse code.
Step 1: Define the class and constructors

```
class Employee {
    int id;
    String name;

    // Default constructor
    Employee() {
        this(0, "Unknown"); // Constructor chaining
        System.out.println("Default constructor called.");
    }

    // Parameterized constructor
    Employee(int id, String name) {
```

```
        // Resolving shadowing using 'this'
        this.id = id;
        this.name = name;
    }

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

Step 2: Instantiate objects using different constructors

```
public class Company {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        Employee emp2 = new Employee(101, "Alice");

        emp1.display();
        emp2.display();
    }
}
```

2.5 Method Overloading and Static Components

Methodology:

Method Overloading and Static Application

1. **Method Overloading:** Define multiple methods named identically but with different parameter signatures (type or count). The compiler determines which to invoke (Compile-time Polymorphism).
2. **Static Variables:** Use the `static` keyword to allocate memory only once per class. It is shared across all instances.
3. **Static Methods:** Use `static` methods to access static variables directly without object instantiation via `ClassName.methodName()`.
4. **Static Blocks:** Use `static { }` to initialize static variables; this block executes once when the class is loaded.
5. **Final Keyword:** Apply `final` to a variable to define a constant whose value cannot change post-initialization.

Worked Example:

Implement Method Overloading and Static Components **Problem:** Create a `MathUtility` class that overloads a method to calculate the area of different shapes. Track the total number of utility calls using a static counter.

Step 1: Define Static Members and Overloaded Methods

```
class MathUtility {
    // Static variable to track method calls
    static int callCount = 0;

    // Final variable as a constant
    final static double PI = 3.14159;
```

```
    // Static block
    static {
        System.out.println("MathUtility Class Loaded.");
    }

    // Overloaded method 1: Area of Square (int parameter)
    static double calculateArea(int side) {
        callCount++;
        return side * side;
    }

    // Overloaded method 2: Area of Rectangle (two double parameters)
    static double calculateArea(double length, double width) {
        callCount++;
        return length * width;
    }

    // Overloaded method 3: Area of Circle (double parameter)
    static double calculateArea(double radius) {
        callCount++;
        return PI * radius * radius;
    }
}
```

Step 2: Execute and Access Static Members directly

```
public class GeometryApp {
    public static void main(String[] args) {
        // Accessing methods via Class Name directly
        double squareArea = MathUtility.calculateArea(5);
        System.out.println("Area of Square: " + squareArea);

        double rectArea = MathUtility.calculateArea(4.0, 6.0);
        System.out.println("Area of Rectangle: " + rectArea);

        double circleArea = MathUtility.calculateArea(3.0);
        System.out.println("Area of Circle: " + circleArea);

        // Accessing static variable
        System.out.println("Total utility calls: " + MathUtility.callCount);
    }
}
```

CHAPTER 3

Inheritance Interfaces and Packages

Inheritance, interfaces, and packages are fundamental pillars of Object-Oriented Programming in Java. They enable code reusability, polymorphic behavior, and logical organization of classes.

3.1 Implementing Inheritance

Inheritance allows a new class (subclass) to acquire the properties and behaviors of an existing class (superclass). Java supports single, multilevel, and hierarchical inheritance for classes.

Methodology:

Implementing Class Inheritance Follow these steps to establish an inheritance relationship between classes:

1. **Define the Superclass:** Identify common attributes and behaviors and define the base class.
2. **Create the Subclass:** Define the derived class using the `extends` keyword followed by the superclass name.
3. **Invoke Superclass Constructor:** Use the `super()` keyword inside the subclass constructor to invoke the constructor of the superclass and initialize inherited fields.
4. **Extend Functionality:** Add specific fields and methods in the subclass that are not present in the superclass.

Worked Example:

Employee and Manager Inheritance Write a Java program to demonstrate single inheritance. Create an `Employee` class with `name` and `salary`. Create a subclass `Manager` that adds a `bonus` attribute. Calculate the total compensation for the manager.

Step 1: Define the Superclass

```
class Employee {
    String name;
    double salary;

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

    public void displayDetails() {
        System.out.println("Name: " + name);
        System.out.println("Base Salary: $" + salary);
    }
}
```

Step 2: Define the Subclass using extends

```
class Manager extends Employee {
    double bonus;

    // Step 3: Invoke Superclass Constructor
    public Manager(String name, double salary, double bonus) {
        super(name, salary);
        this.bonus = bonus;
    }

    // Step 4: Extend Functionality
    public double getTotalCompensation() {
        return salary + bonus;
    }

    public void displayManagerDetails() {
        displayDetails(); // Inherited method
        System.out.println("Bonus: $" + bonus);
        System.out.println("Total Compensation: $" + getTotalCompensation());
    }
}
```

Step 5: Test the Implementation

```
public class InheritanceExample {
    public static void main(String[] args) {
        Manager mgr = new Manager("Alice Smith", 80000, 15000);
        mgr.displayManagerDetails();
    }
}
```

Output:

```
Name: Alice Smith
Base Salary: $80000.0
Bonus: $15000.0
Total Compensation: $95000.0
```

3.2 Method Overriding and Polymorphism

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. It forms the basis for runtime polymorphism through dynamic method dispatch.

Methodology:

Dynamic Method Dispatch To implement runtime polymorphism in Java:

1. **Define Overridable Method:** Create a superclass with a method intended to be overridden.
2. **Override the Method:** In the subclass, redefine the method with the exact same signature (name, return type, and parameters). Use the `@Override` annotation.
3. **Upcasting:** In the driver code, declare a reference variable of the superclass type but instantiate it with a subclass object.
4. **Method Invocation:** Call the overridden method using the superclass reference. The JVM dynamically determines which implementation to execute based on the actual object type at runtime.

Worked Example:

Shape Area Calculation with Polymorphism Create a base class `Shape` with a method `calculateArea()`. Create subclasses `Circle` and `Rectangle` that override this method. Demonstrate dynamic method dispatch to calculate their areas.

Step 1: Define Overridable Method

```
class Shape {
    public void calculateArea() {
        System.out.println("Area of a generic shape is undefined.");
    }
}
```

Step 2: Override the Method in Subclasses

```
class Circle extends Shape {
    double radius;

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

    @Override
    public void calculateArea() {
        double area = Math.PI * radius * radius;
        System.out.println("Area of Circle: " + area);
    }
}

class Rectangle extends Shape {
    double length, width;

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

    @Override
    public void calculateArea() {
        double area = length * width;
        System.out.println("Area of Rectangle: " + area);
    }
}
```

Step 3 and 4: Upcasting and Method Invocation

```
public class PolymorphismExample {
    public static void main(String[] args) {
        // Superclass reference holding subclass object
        Shape shapeRef;

        shapeRef = new Circle(5.0);
        shapeRef.calculateArea(); // Executes Circle's version dynamically

        shapeRef = new Rectangle(4.0, 6.0);
        shapeRef.calculateArea(); // Executes Rectangle's version dynamically
    }
}
```

Output:

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24.0
```

3.3 Interfaces and Multiple Inheritance

Java does not support multiple inheritance with classes to avoid the diamond problem. However, multiple inheritance can be achieved using interfaces.

Methodology:

Implementing Multiple Inheritance via Interfaces

1. **Define Interfaces:** Create two or more interfaces using the `interface` keyword. Declare abstract methods inside them.
2. **Implement Interfaces:** Create a class and use the `implements` keyword followed by a comma-separated list of interfaces.
3. **Provide Concrete Implementations:** The implementing class must override and provide a body for all abstract methods declared in the interfaces.
4. **Instantiation:** Create an object of the implementing class to utilize the methods.

Worked Example:

Printable and Showable Interfaces Create two interfaces `Printable` and `Showable`. Implement both in a `Document` class to demonstrate multiple inheritance of types.

Step 1: Define Interfaces

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

interface Showable {
    void showPreview();
}
```

Step 2 and 3: Implement Interfaces and Provide Bodies

```
class Document implements Printable, Showable {
    String content;

    public Document(String content) {
        this.content = content;
    }

    @Override
    public void printDocument() {
        System.out.println("Printing to printer...");
        System.out.println("Content: " + content);
    }

    @Override
    public void showPreview() {
        System.out.println("Displaying on screen preview:");
```

```
        System.out.println("[ " + content + " ]");
    }
}
```

Step 4: Instantiation

```
public class InterfaceExample {
    public static void main(String[] args) {
        Document doc = new Document("Annual Financial Report");

        // Calling methods from both interfaces
        doc.showPreview();
        doc.printDocument();
    }
}
```

Output:

```
Displaying on screen preview:
[ Annual Financial Report ]
Printing to printer...
Content: Annual Financial Report
```

3.4 Abstract Classes

Abstract classes act as a blueprint for other classes. They can contain both abstract methods (without bodies) and concrete methods (with bodies).

Methodology:

Developing and Extending Abstract Classes

1. **Declare Abstract Class:** Use the `abstract` keyword before the `class` declaration.
2. **Define Abstract Methods:** Declare methods that subclasses must implement using the `abstract` keyword, terminating with a semicolon.
3. **Provide Shared Behavior:** Write concrete methods for logic that is common to all subclasses.
4. **Extend and Implement:** Create a subclass using `extends` and provide bodies for all inherited abstract methods.

Worked Example:

Vehicle Abstraction Create an abstract class `Vehicle` with an abstract method `startEngine()` and a concrete method `stopEngine()`. Implement subclasses `Car` and `Motorcycle`.
Step 1 2 and 3: Abstract Class Definition

```
abstract class Vehicle {
    String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method
    public abstract void startEngine();
```

```
    // Concrete method
    public void stopEngine() {
        System.out.println(brand + " engine is now stopped.");
    }
}
```

Step 4: Extend and Implement

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

    @Override
    public void startEngine() {
        System.out.println(brand + " car engine starts with a push button.");
    }
}

class Motorcycle extends Vehicle {
    public Motorcycle(String brand) {
        super(brand);
    }

    @Override
    public void startEngine() {
        System.out.println(brand + " motorcycle engine starts with a kick.");
    }
}
```

Testing the Abstract Class:

```
public class AbstractClassExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car("Toyota");
        myCar.startEngine();
        myCar.stopEngine();

        System.out.println("-----------------");

        Vehicle myBike = new Motorcycle("Yamaha");
        myBike.startEngine();
        myBike.stopEngine();
    }
}
```

Output:

```
Toyota car engine starts with a push button.
Toyota engine is now stopped.
-----------------
Yamaha motorcycle engine starts with a kick.
Yamaha engine is now stopped.
```

3.5 Packages and Access Control

Packages in Java are used to group related classes, interfaces, and sub-packages. They resolve naming conflicts and govern access protection.

Methodology:

Creating and Importing Packages

1. **Declare Package:** Place `package packagename;` as the first statement in the Java source file.
2. **Set Access Modifiers:** Ensure the classes and methods intended for external use are marked as `public`.
3. **Compilation:** Compile the source code using the `-d` flag to generate the correct directory structure (e.g., `javac -d . MyClass.java`).
4. **Import Package:** In another program, use the `import packagename.ClassName;` or `import packagename.*;` statement to access the packaged elements.

Worked Example:

MathUtility Package Creation Create a package named `com.utilities` containing a `Calculator` class. Import and use this class in a separate `Main` program located in the default package.

Step 1 and 2: Declare Package and Modifiers

```
// File: Calculator.java
package com.utilities;

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

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

Step 3: Compilation (Conceptual) Executing `javac -d . Calculator.java` creates a directory structure `./com/utilities/` and places `Calculator.class` inside it.

Step 4: Import Package

```
// File: Main.java
import com.utilities.Calculator;

public class Main {
    public static void main(String[] args) {
        // Instantiate class from the imported package
        Calculator calc = new Calculator();

        int sum = calc.add(10, 5);
        int product = calc.multiply(10, 5);

        System.out.println("Sum: " + sum);
        System.out.println("Product: " + product);
    }
}
```

Output:

```
Sum: 15
Product: 50
```

CHAPTER 4

Exception Handling and Multithreading

4.1 Exception Handling

Exceptions in Java are events that disrupt the normal flow of the program. Exception handling ensures that the program can gracefully recover or terminate when an error occurs.

Methodology:

Handling Exceptions with Try and Catch **Steps to handle an exception:**

1. Identify the code segment that may cause a runtime error.
2. Enclose this risky code within a `try` block.
3. Immediately follow the `try` block with one or more `catch` blocks.
4. In the `catch` block, specify the Exception type as a parameter and provide the logic to handle or log the error.

Worked Example:

Demonstrating Multiple Catch Blocks Write a Java program to handle multiple exceptions using different catch blocks.
Solution:

```
public class MultipleCatchDemo {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 10 / 0; // ArithmeticException occurs here
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception:  Division by zero");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index Out of Bounds Exception");
        } catch (Exception e) {
            System.out.println("Parent Exception occurs");
        }
    }
}
```

Output:

```
Arithmetic Exception:  Division by zero
```

Methodology:

Using the Finally Block The `finally` block is used to execute important code such as closing connections or streams, regardless of whether an exception is thrown or caught.
Execution Flow:

1. A `finally` block is placed after the `catch` block(s).
2. If no exception occurs, the `finally` block executes after the `try` block.
3. If an exception occurs, the `finally` block executes after the corresponding `catch` block.
4. Even if an exception is not caught, the `finally` block executes before the program terminates.

Worked Example:

Executing Finally Block Write a program to demonstrate the purpose of the finally block in Java.
Solution:

```
public class FinallyDemo {
    public static void main(String[] args) {
        try {
            int result = 25 / 5;
            System.out.println("Result:  " + result);
        } catch (NullPointerException e) {
            System.out.println("Exception caught");
        } finally {
            System.out.println("Finally block is always executed");
        }
    }
}
```

Output:

```
Result:  5
Finally block is always executed
```

Methodology:

Using Throw and Throws **Steps to explicitly throw and declare exceptions:**

1. Use the `throw` keyword within a method to explicitly throw an exception object (e.g., `throw new ArithmeticException("Error");`).
2. Use the `throws` keyword in the method signature to declare that the method might throw exceptions, forcing the caller to handle them.
3. Multiple exceptions in a `throws` clause are separated by spaces and no commas should be included in block titles to ensure pgfkeys compiles safely.

Worked Example:

Demonstrating Throw and Throws Write a program to demonstrate the use of `throw` and `throws` keywords for a custom validation method.
Solution:

```
public class ThrowThrowsDemo {
    // Declare exception using throws
    static void checkAge(int age) throws ArithmeticException {
```

```
        if (age < 18) {
            // Explicitly throw exception
            throw new ArithmeticException("Access denied");
        } else {
            System.out.println("Access granted");
        }
    }
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (ArithmeticException e) {
            System.out.println("Caught Exception:  " + e.getMessage());
        }
    }
}
```

Output:

```
Caught Exception:  Access denied
```

4.2 Multithreading

Multithreading allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Threads are lightweight sub-processes.

Methodology:

Creating Threads by Extending Thread Class **Steps to implement multithreading using the Thread class:**

1. Create a new class that extends the `java.lang.Thread` class.
2. Override the `run()` method of the Thread class to define the task of the thread.
3. Create an object of the newly created class in the main method.
4. Call the `start()` method on the object to begin execution of the thread.

Worked Example:

Multithreading with Thread Class Write a Java program to demonstrate creation of threads by extending the Thread class.
Solution:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running concurrently.");
    }
}
public class ThreadDemo {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts the execution of the thread
    }
}
```

Output:

```
Thread is running concurrently.
```

Methodology:

Creating Threads by Implementing Runnable **Steps to implement multithreading using the Runnable interface:**

1. Create a class that implements the `java.lang.Runnable` interface.
2. Provide an implementation for the `run()` method.
3. Instantiate this class.
4. Create a `Thread` object, passing the Runnable instance to its constructor.
5. Call the `start()` method on the `Thread` object.

Worked Example:

Concurrent Tasks using Runnable Interface Write a Java program that implements the Runnable interface to perform two concurrent tasks (e.g. printing odd and even numbers).
Solution:

```
class EvenThread implements Runnable {
    public void run() {
        for (int i = 2; i <= 6; i += 2) {
            System.out.println("Even:  " + i);
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}
class OddThread implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i += 2) {
            System.out.println("Odd:  " + i);
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}
public class RunnableDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread(new EvenThread());
        Thread t2 = new Thread(new OddThread());
        t1.start();
        t2.start();
    }
}
```

Worked Example:

Printing Numbers and Alphabets Simultaneously Write a Java program to explain the concept of multithreading by creating two threads that print numbers and alphabets simultaneously.
Solution:

```
class NumberThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
```

```
            System.out.println("Number:  " + i);
            try { Thread.sleep(300); } catch (Exception e) {}
        }
    }
}
class AlphabetThread extends Thread {
    public void run() {
        for (char c = 'A'; c <= 'E'; c++) {
            System.out.println("Alphabet:  " + c);
            try { Thread.sleep(300); } catch (Exception e) {}
        }
    }
}
public class SyncDemo {
    public static void main(String[] args) {
        NumberThread t1 = new NumberThread();
        AlphabetThread t2 = new AlphabetThread();
        t1.start();
        t2.start();
    }
}
```

CHAPTER 5
Collections and File Handling

This chapter focuses on managing data in Java using the Collections Framework, processing text with String manipulation techniques, and persisting data via File I/O and Serialization. The emphasis is on algorithms and problem-solving strategies using these APIs.

5.1 String Manipulation and Regular Expressions

Strings in Java are immutable. For dynamic modifications, `StringBuilder` and `StringBuffer` (thread-safe) are preferred. Regular expressions provide powerful pattern matching capabilities.

Methodology:

String Processing and Pattern Matching **Step 1: Choose the Storage Mechanism** Use `String` for constant text and `StringBuilder` for frequent modifications.
Step 2: String Operations Use methods like `substring()`, `split()`, `indexOf()`, and `replace()` to isolate and modify text.
Step 3: Define Regular Expression Formulate a regex pattern. For instance `^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}$` for basic email validation.
Step 4: Compile and Match Compile the regex using `Pattern.compile(regex)` and instantiate a matcher using `pattern.matcher(text)`. Use `matcher.find()` or `matcher.matches()` to extract or validate.

Worked Example:

Extracting Valid Email Addresses from a Text Block **Problem Statement:** Given a block of text, extract all valid email addresses and format them as a comma-separated string using `StringBuilder`.
Step 1: Input text and regex definition Let the text be: `"Contact us at support@example.com or admin@test.org."`
Regex pattern: `"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}"`
Step 2: Implementation Setup

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

String text = "Contact us at support@example.com or admin@test.org.";
String regex = "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}";
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(text);
StringBuilder results = new StringBuilder();
```

Step 3: Extract and Append

```
while (matcher.find()) {
    if (results.length() > 0) {
        results.append(", ");
    }
    results.append(matcher.group());
```

```
}
System.out.println(results.toString());
```

Output:

```
support@example.com, admin@test.org
```

5.2 Java Collections Framework

The Collections Framework provides pre-packaged data structures.

- **List:** Ordered collection with duplicates (e.g. `ArrayList` and `LinkedList`).
- **Set:** Unordered collection of unique elements (e.g. `HashSet` and `TreeSet`).
- **Map:** Key-value pairs with unique keys (e.g. `HashMap` and `TreeMap`).

Methodology:

Algorithm for Data Aggregation using Maps **Step 1: Initialization** Instantiate a `Map<KeyType, ValueType>` based on sorting needs (`HashMap` for fast lookup or `TreeMap` for sorted keys).
Step 2: Iteration and Population Iterate through the raw data source. For each item determine its classification (Key).
Step 3: Update Logic Check if the key exists using `containsKey(key)`. If it exists retrieve the current value update it and put it back. If it does not exist insert the initial value using `put(key, initialValue)`.
Step 4: Output Iterate over `map.entrySet()` to process or display the aggregated results.

Worked Example:

Calculating Word Frequencies in a Document **Problem Statement:** Count the frequency of each word in an array of words ignoring case and display the results.
Step 1: Initialization

```
import java.util.*;

String[] words = {"apple", "Banana", "Apple", "orange", "banana", "apple"};
Map<String, Integer> frequencyMap = new HashMap<>();
```

Step 2 and 3: Iteration and Update Logic

```
for (String w : words) {
    String word = w.toLowerCase(); // Ignore case
    if (frequencyMap.containsKey(word)) {
        frequencyMap.put(word, frequencyMap.get(word) + 1);
    } else {
        frequencyMap.put(word, 1);
    }
}
```

Step 4: Output Display

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

Output:

```
orange:  1
banana:  2
apple:  3
```

5.3 File Input and Output Operations

Handling files requires choosing between byte streams (for binary data like images) and character streams (for text data).

Methodology:

Algorithm for Reading and Writing Text Files **Step 1: Open Output Stream** Initialize a `BufferedWriter` wrapped around a `FileWriter` to write data. Use try-with-resources to ensure the stream closes automatically.
Step 2: Write Data Use `write(String)` and `newLine()` to write content sequentially.
Step 3: Open Input Stream Initialize a `BufferedReader` wrapped around a `FileReader`.
Step 4: Read Data Iteratively Use a loop to read via `readLine()` until it returns `null`. Process each line as needed.

Worked Example:

Filtering Data from one File to Another **Problem Statement:** Write a list of numbers to a file. Read that file and write only the even numbers to a new file.
Step 1 and 2: Write Initial Data

```
import java.io.*;

try (BufferedWriter bw = new BufferedWriter(new FileWriter("numbers.txt"))) {
    for (int i = 1; i <= 5; i++) {
        bw.write(String.valueOf(i));
        bw.newLine();
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Step 3 and 4: Read Filter and Write

```
try (BufferedReader br = new BufferedReader(new FileReader("numbers.txt"));
     BufferedWriter bw = new BufferedWriter(new FileWriter("even.txt"))) {

    String line;
    while ((line = br.readLine()) != null) {
        int num = Integer.parseInt(line);
        if (num % 2 == 0) {
            bw.write(String.valueOf(num));
            bw.newLine();
        }
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Result: The file `even.txt` will contain 2 and 4.

5.4 Serialization for Object Persistence

Serialization is the process of converting an object state to a byte stream so that the byte stream can be reverted back into a copy of the object.

Methodology:

Object Serialization and Deserialization **Step 1: Implement Serializable** The class of the object to be serialized must implement `java.io.Serializable`. Use the `transient` keyword for fields that should not be saved.
Step 2: Serialize or Write Initialize a `FileOutputStream` and wrap it with an `ObjectOutputStream`. Call `writeObject(obj)`.
Step 3: Deserialize or Read Initialize a `FileInputStream` and wrap it with an `ObjectInputStream`. Call `readObject()` and cast it back to the specific class type.

Worked Example:

Saving and Restoring a User Object **Problem Statement:** Create a `User` class with a username and a password. Serialize the object to a file but ensure the password is not saved. Then deserialize the object to verify.

Step 1: Create Serializable Class

```
import java.io.Serializable;

class User implements Serializable {
    private static final long serialVersionUID = 1L;
    String username;
    transient String password; // Will not be serialized

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
}
```

Step 2: Serialize the Object

```
import java.io.*;

User user = new User("admin", "secret123");
try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("user.ser"))) {
    oos.writeObject(user);
} catch (IOException e) {
    e.printStackTrace();
}
```

Step 3: Deserialize the Object

```
User restoredUser = null;
try (ObjectInputStream ois = new ObjectInputStream(
        new FileInputStream("user.ser"))) {
    restoredUser = (User) ois.readObject();
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}

System.out.println("Username: " + restoredUser.username);
System.out.println("Password: " + restoredUser.password);
```

Output:

```
Username:  admin
Password:  null
```

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Properties of the XOR Operation

The exclusive OR (XOR), denoted by $\oplus$, is a fundamental bitwise operator. The following are its key properties:

$$A \oplus 0 = A \tag{5.1}$$
$$A \oplus A = 0 \tag{5.2}$$
$$A \oplus B \oplus A = B \tag{5.3}$$

- **Identity Property (??):** XORing any value A with 0 results in the value itself.
- **Self-Inverse Property (??):** XORing any value A with itself results in 0.
- **Cancellation Property (??):** Since XOR is commutative and associative, $A \oplus B \oplus A$ rearranges to $(A \oplus A) \oplus B$, which simplifies to $0 \oplus B = B$. This property is frequently used in cryptographic applications and bitwise data swapping.