Java (4343203) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Introduction to Java Programming and Basic Problem Solving

Java is a high-level object-oriented programming language. In this chapter we focus on computational problem-solving using Java's foundational syntax including operators arithmetic expressions control statements and basic iterative algorithms.

1.1 Basic Program Structure and Execution

Methodology:

Writing a Java Console Application To solve computational problems in Java we follow a standard template for writing our algorithmic logic:

1. **Class Definition:** Every Java program must have at least one class.
2. **Main Method:** The entry point of the program execution is always `public static void main(String[] args)`.
3. **Variable Declaration:** Declare variables with appropriate data types (e.g. `int` for integers `double` for decimals).
4. **Computation:** Write the arithmetic or logical expressions to process the data based on your mathematical formula.
5. **Output:** Use `System.out.println()` to display the computed results to the user.

Worked Example:

Calculating Simple Interest **Problem:** Write a program to calculate the simple interest for a principal amount of 5000 rate of interest 8.5 percent and time period of 3 years.
Step-by-Step Solution:

1. **Identify Variables:** We need principal ($P = 5000$) rate ($R = 8.5$) and time ($T = 3$).
2. **Apply Formula:** The computational formula for Simple Interest is $SI = \frac{P \times R \times T}{100}$.
3. **Java Implementation:** Since the rate is a fractional number we will use the `double` data type for all monetary and rate variables to avoid integer division truncation.

```
public class SimpleInterest {
    public static void main(String[] args) {
        double principal = 5000.0;
        double rate = 8.5;
        double time = 3.0;
        double interest = (principal * rate * time) / 100.0;
```

```
        System.out.println("Simple Interest:  " + interest);
    }
}
```

1.2 Decision Making Constructs

Methodology:

Implementing If Else Logic Decision-making statements allow a program to choose different computational paths dynamically based on specific conditions:

1. Formulate a boolean condition using relational operators ($> < == != >= <=$) and logical operators ($\&\& \mid\mid !$).
2. Use the `if (condition)` block to wrap the primary code sequence that must execute when the condition evaluates to true.
3. (Optional) Use `else if (condition)` blocks for evaluating multiple alternative cascading conditions.
4. (Optional) Use the `else` block as a fallback execution path when all preceding conditions evaluate to false.

Worked Example:

Finding Maximum of Three Numbers **Problem:** Write an algorithm and Java program to find the largest of three given integers (15 42 and 27) without using built-in math library functions.
Step-by-Step Solution:

1. Let the input numbers be $A = 15$ $B = 42$ and $C = 27$.
2. Compare A with both B and C. If $A \geq B$ AND $A \geq C$ then A is the maximum.
3. Otherwise if A is not the maximum compare B with A and C. If $B \geq A$ AND $B \geq C$ then B is the maximum.
4. If neither A nor B is the maximum then C must naturally be the largest number.

```
public class MaxThree {
    public static void main(String[] args) {
        int a = 15;
        int b = 42;
        int c = 27;
        int max;

        if (a >= b && a >= c) {
            max = a;
        } else if (b >= a && b >= c) {
            max = b;
        } else {
            max = c;
        }

        System.out.println("Maximum is:  " + max);
    }
}
```

1.3 Looping and Iteration Algorithms

Methodology:

Using the For Loop for Iteration Loops are essential for solving repetitive mathematical sequence generation or accumulation problems. The `for` loop is ideal when the exact number of iterations is known ahead of time.

1. **Initialization:** Declare and set a loop counter variable (e.g. `int i = 1`).
2. **Condition:** Specify the boolean check that determines if the loop continues (e.g. `i <= N`). If false the loop terminates.
3. **Body:** Place the repetitive computational steps (like accumulations or sequence generation) inside the loop block.
4. **Update:** Increment or decrement the counter at the end of each iteration (e.g. `i++` or `i = i + 1`).

Worked Example:

Generating Fibonacci Series **Problem:** Write a program to generate and print the first 8 terms of the Fibonacci sequence: 0 1 1 2 3 5 8 13.
Step-by-Step Solution:

1. A Fibonacci sequence starts with two default terms: $F_1 = 0$ and $F_2 = 1$. Initialize `first = 0` and `second = 1`.
2. Print `first` and `second`.
3. Set up a loop starting from $i = 3$ up to 8 to generate the remaining terms.
4. In each iteration calculate the next mathematical term as the sum of the previous two: `next = first + second`.
5. Print the `next` term.
6. Shift the terms for the next iteration: assign the value of `second` into `first` and the value of `next` into `second`.

```
public class Fibonacci {
    public static void main(String[] args) {
        int n = 8;
        int first = 0;
        int second = 1;

        System.out.print(first + " " + second);

        for (int i = 3; i <= n; i++) {
            int next = first + second;
            System.out.print(" " + next);

            // Shift terms
            first = second;
            second = next;
        }
        System.out.println();
    }
}
```

Worked Example:

Calculating Factorial **Problem:** Write a program to calculate the factorial of a given number $N = 5$. Factorial formula: $N! = N \times (N-1) \times (N-2) \times \cdots \times 1$.

Step-by-Step Solution:

1. Initialize an accumulator variable `fact = 1`. We use 1 instead of 0 because it is the multiplicative identity.
2. Loop a variable i from 1 up to N (where $N = 5$).
3. Inside the loop multiply the current value of `fact` by i and store it back into `fact` (`fact = fact * i`).
4. After the loop terminates the variable `fact` holds the accumulated product which is the final answer.

```
public class Factorial {
    public static void main(String[] args) {
        int num = 5;
        long fact = 1;

        for (int i = 1; i <= num; i++) {
            fact = fact * i;
        }

        System.out.println("Factorial of " + num + " is " + fact);
    }
}
```

CHAPTER 2

Object Oriented Programming Concepts

2.1 Core Principles of OOP

Object-Oriented Programming (OOP) is a paradigm centered around objects rather than functions. It contrasts with Procedure-Oriented Programming (POP), which focuses on functions or sequence of actions.

Methodology:

Key Concepts of OOP

1. **Encapsulation**: Wrapping data (variables) and code (methods) together as a single unit (class) to restrict direct access and ensure data security.
2. **Abstraction**: Hiding internal implementation details and showing only necessary features to the user.
3. **Inheritance**: Acquiring properties and behaviors from another class to promote code reusability.
4. **Polymorphism**: The ability of a variable, function, or object to take on multiple forms (e.g., method overloading).

2.2 Classes and Objects

A class is a blueprint or template, and an object is a real-world instance of a class that holds state and behavior.

Methodology:

Defining a Class and Creating Objects

1. **Define the Class**: Use the `class` keyword followed by the class name.
2. **Declare Fields**: Define instance variables to store the state of the object.
3. **Define Methods**: Write functions inside the class to describe behaviors.
4. **Instantiate Object**: Use the `new` keyword to allocate memory and create an object: `ClassName obj = new ClassName();`.
5. **Access Members**: Use the dot operator (`.`) to access fields and methods: `obj.fieldName` or `obj.methodName()`.

Worked Example:

Rectangle Area Calculator Write a Java program to define a `Rectangle` class with fields for length and width. Include a method to calculate the area and demonstrate object creation.
Solution:

1. **Define the Class and Fields**:

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

2. **Define the Method**:

```
    double calculateArea() {
        return length * width;
    }
}
```

3. **Create the Main Class and Instantiate Object**:

```
public class Main {
    public static void main(String[] args) {
        Rectangle rect1 = new Rectangle();
        rect1.length = 10.5;
        rect1.width = 5.0;

        double area = rect1.calculateArea();
        System.out.println("Area of Rectangle: " + area);
    }
}
```

2.3 Access Modifiers and Keywords

Methodology:

Access Rules in Java

1. **public**: The member is accessible from everywhere.
2. **private**: The member is accessible only within the same class.
3. **protected**: The member is accessible within the same package and subclasses in other packages.
4. **default** (no modifier): The member is accessible only within the same package.

Methodology:

Using this static and final Keywords

1. **`this` Keyword**: A reference to the current object. Used to resolve naming ambiguity between instance variables and parameters.
2. **`static` Keyword**: Denotes that a member belongs to the class rather than instances. Used for memory management (e.g., shared constants or utility methods).
3. **`final` Keyword**: Used to restrict the user. A `final` variable cannot be reassigned (acts as a constant), a `final` method cannot be overridden, and a `final` class cannot be inherited.

Worked Example:

Bank Account Management Demonstrate the use of `private` fields, the `this` keyword, a `static` variable for the bank name, and a `final` variable for the account number.
Solution:

1. **Define the Class with Modifiers**:

```
class BankAccount {
    static String bankName = "State Bank";
    final int accountNumber;
    private double balance;

    BankAccount(int accountNumber, double balance) {
        this.accountNumber = accountNumber; // Using this to resolve ambiguity
        this.balance = balance;
    }
```

2. **Add Methods to Access Private Data**:

```
    void deposit(double amount) {
        if(amount > 0) {
            this.balance += amount;
        }
    }

    void displayAccountDetails() {
        System.out.println("Bank: " + bankName);
        System.out.println("A/C No: " + accountNumber);
        System.out.println("Balance: " + balance);
    }
}
```

3. **Test the Class**:

```
public class BankApp {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount(1001, 5000.0);
        acc.deposit(1500.0);
        acc.displayAccountDetails();
    }
}
```

2.4 Constructors and Passing Objects

Constructors are special methods invoked when an object is created. They are primarily used to initialize the state of an object.

Methodology:

Types of Constructors

1. **Default Constructor**: A constructor with no parameters. If no constructor is defined, Java provides an implicit default constructor.
2. **Parameterized Constructor**: A constructor with arguments to initialize objects with specific values dynamically.
3. **Copy Constructor**: A constructor that initializes a new object using an existing object of the same class.
4. **Passing Object as Parameter**: Objects can be passed to methods or constructors by reference,

allowing them to manipulate the object's data.

Worked Example:

Complex Number Addition Write a Java program to add two complex numbers by passing an object as a parameter to a method. Also demonstrate parameterized and copy constructors.
Solution:

1. **Define the Complex Class and Constructors**:

```
class Complex {
    double real;
    double imag;

    // Parameterized Constructor
    Complex(double real, double imag) {
        this.real = real;
        this.imag = imag;
    }

    // Copy Constructor
    Complex(Complex c) {
        this.real = c.real;
        this.imag = c.imag;
    }
```

2. **Method to Add Objects**:

```
    // Passing object as a parameter
    Complex add(Complex other) {
        double newReal = this.real + other.real;
        double newImag = this.imag + other.imag;
        return new Complex(newReal, newImag);
    }

    void display() {
        System.out.println(real + " + " + imag + "i");
    }
}
```

3. **Execute Logic in Main**:

```
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(3.5, 2.5);
        Complex c2 = new Complex(1.5, 4.5);

        Complex c3 = c1.add(c2); // Passing object c2
        System.out.print("Sum: ");
        c3.display();

        Complex c4 = new Complex(c3); // Using copy constructor
        System.out.print("Copied Object: ");
        c4.display();
    }
}
```

2.5 Method and Constructor Overloading

Overloading is a feature that allows a class to have more than one method or constructor having the same name, if their argument lists are different.

Methodology:

Implementing Overloading

1. **Method Overloading**: Define multiple methods with the same name but different parameter lists (differing by type, number, or sequence of parameters).
2. **Constructor Overloading**: Define multiple constructors in the same class with different parameter lists to initialize objects in various ways.
3. **Resolution**: The Java compiler determines which method or constructor to call based on the arguments passed during invocation.

Worked Example:

Calculating Area using Overloading Demonstrate method overloading by creating methods to calculate the area of a square, rectangle, and circle.
Solution:

1. **Define the Overloaded Methods**:

```
class ShapeArea {
    // Area of Square
    double area(double side) {
        return side * side;
    }

    // Area of Rectangle
    double area(double length, double width) {
        return length * width;
    }

    // Area of Circle (second parameter used to distinguish signature)
    double area(double radius, boolean isCircle) {
        return 3.14159 * radius * radius;
    }
}
```

2. **Test the Overloaded Methods**:

```
public class Main {
    public static void main(String[] args) {
        ShapeArea calc = new ShapeArea();

        System.out.println("Square Area: " + calc.area(5.0));
        System.out.println("Rectangle Area: " + calc.area(4.0, 6.0));
        System.out.println("Circle Area: " + calc.area(3.0, true));
    }
}
```

2.6 String Class and Wrapper Classes

Java provides the `String` class to manipulate character sequences and wrapper classes (like `Integer`, `Double`) to treat primitive data types as objects.

Methodology:

Important String Methods

1. `charAt(int index)`: Returns the character at the specified index.
2. `contains(CharSequence s)`: Checks if the string contains the specified sequence of characters.
3. `format(String format, Object... args)`: Returns a formatted string using the specified format string and arguments.
4. `length()`: Returns the number of characters present in the string.
5. `split(String regex)`: Splits the string around matches of the given regular expression and returns an array of strings.

Worked Example:

String Analysis and Parsing Write a Java program to analyze a sentence using `String` methods and convert a numeric string to an integer using a Wrapper class.
Solution:

1. **String Operations**:

```
public class StringDemo {
    public static void main(String[] args) {
        String text = "Java Programming is fun";

        System.out.println("Length: " + text.length());
        System.out.println("Character at index 5: " + text.charAt(5));
        System.out.println("Contains 'Prog': " + text.contains("Prog"));

        String formatted = String.format("The string is: %s", text);
        System.out.println(formatted);
```

2. **Splitting the String**:

```
        String[] words = text.split(" ");
        System.out.println("Words in string:");
        for(String word : words) {
            System.out.println("- " + word);
        }
```

3. **Using Wrapper Class**:

```
        String numberStr = "1024";
        // Convert string to primitive int using Integer wrapper class
        int num = Integer.parseInt(numberStr);
        System.out.println("Parsed number + 10: " + (num + 10));
    }
}
```

2.7 User Input with Scanner Class

The `Scanner` class in the `java.util` package is used to obtain input of primitive types like `int`, `double`, etc., and strings.

Methodology:

Reading User Input

1. **Import Scanner**: Include `import java.util.Scanner;` at the beginning of the file.
2. **Create Scanner Object**: Initialize it with the standard input stream: `Scanner sc = new Scanner(System.in);`.
3. **Read Data**: Use appropriate methods like `nextInt()`, `nextDouble()`, or `nextLine()` to read data.
4. **Close Scanner**: Release resources by calling `sc.close();` after reading is done.

Worked Example:

Student Grade Calculator Create a program that takes a student's name and marks for three subjects as input from the user, calculates the total and average, and prints the result.
Solution:

1. **Import and Setup**:

```
import java.util.Scanner;

public class GradeCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

2. **Read User Input**:

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

        System.out.print("Enter marks for Subject 1: ");
        int m1 = scanner.nextInt();

        System.out.print("Enter marks for Subject 2: ");
        int m2 = scanner.nextInt();

        System.out.print("Enter marks for Subject 3: ");
        int m3 = scanner.nextInt();
```

3. **Calculate and Display**:

```
        int total = m1 + m2 + m3;
        double average = total / 3.0;

        System.out.println("\n--- Result ---");
        System.out.println("Name: " + name);
        System.out.println("Total Marks: " + total);
        System.out.println("Average: " + average);

        scanner.close(); // Close the scanner
    }
}
```

CHAPTER 3

Unit 3: Inheritance Interfaces and Packages

3.1 Inheritance Basics

Methodology:

Implementing Single and Multilevel Inheritance Inheritance allows a new class (subclass) to acquire the properties and methods of an existing class (superclass). This promotes code reusability.
Steps to Implement Inheritance:

1. **Define the Superclass:** Create the base class containing common fields and methods.
2. **Define the Subclass:** Create the derived class using the `extends` keyword followed by the superclass name.
3. **Add Specific Features:** Add fields and methods specific to the subclass.
4. **Instantiate and Access:** Create an object of the subclass to access both inherited and specific members.

Worked Example:

Computing Total Compensation for Employees **Problem Statement:** Create a base class `Employee` with a basic salary field. Create a subclass `Manager` that inherits `Employee` and adds a bonus field. Compute and display the total compensation for a manager.
Step-by-step Solution:
Step 1: Define the Superclass.

```
class Employee {
    double basicSalary;

    void setBasicSalary(double salary) {
        this.basicSalary = salary;
    }
}
```

Step 2: Define the Subclass.

```
class Manager extends Employee {
    double bonus;

    void setBonus(double b) {
        this.bonus = b;
    }

    double computeTotalCompensation() {
        return basicSalary + bonus;
```

```
    }
}
```

Step 3: Instantiate and Compute.

```
public class Main {
    public static void main(String[] args) {
        Manager mgr = new Manager();
        mgr.setBasicSalary(50000);
        mgr.setBonus(15000);

        System.out.println("Basic Salary: " + mgr.basicSalary);
        System.out.println("Bonus: " + mgr.bonus);
        System.out.println("Total Compensation: "
                           + mgr.computeTotalCompensation());
    }
}
```

Output:

```
Basic Salary: 50000.0
Bonus: 15000.0
Total Compensation: 65000.0
```

3.2 The super Keyword and Constructor Chaining

Methodology:

Utilizing the super Keyword The `super` keyword is a reference variable used to refer to an immediate parent class object.
Algorithmic Steps for Usage:

1. **Constructor Chaining:** Use `super(arguments)` inside the subclass constructor to invoke the parameterized constructor of the superclass. This must be the very first statement in the subclass constructor.
2. **Accessing Hidden Fields:** Use `super.fieldName` to access a superclass variable that is hidden by a subclass variable of the same name.
3. **Invoking Overridden Methods:** Use `super.methodName()` to call a method of the parent class that has been overridden in the child class.

Worked Example:

Calculating Volume and Weight of a Box **Problem Statement:** Design a class `Box` with dimensions width, height, and depth. Design a subclass `BoxWeight` that adds a weight attribute. Use constructor chaining to initialize all properties and calculate the volume.
Step-by-step Solution:
Step 1: Create the Superclass Constructor.

```
class Box {
    double width;
    double height;
    double depth;

    // Parameterized constructor
    Box(double w, double h, double d) {
```

```
        width = w;
        height = h;
        depth = d;
    }

    double volume() {
        return width * height * depth;
    }
}
```

Step 2: Create Subclass and use super().

```
class BoxWeight extends Box {
    double weight;

    // Subclass constructor
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // Call superclass constructor
        weight = m;
    }
}
```

Step 3: Execute in Main.

```
public class DemoSuper {
    public static void main(String[] args) {
        BoxWeight myBox = new BoxWeight(10, 20, 15, 34.3);
        double vol;

        vol = myBox.volume();
        System.out.println("Volume of myBox is " + vol);
        System.out.println("Weight of myBox is " + myBox.weight);
    }
}
```

Output:

```
Volume of myBox is 3000.0
Weight of myBox is 34.3
```

3.3 Method Overriding and Dynamic Method Dispatch

Methodology:

Overriding Methods and Runtime Polymorphism Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime.

Steps to Implement Dynamic Method Dispatch:

1. Define a method in the superclass.
2. Define the exact same method (same name, return type, and parameters) in the subclass.
3. Declare a reference variable of the superclass type.
4. Instantiate a subclass object and assign it to the superclass reference variable (`SuperClass ref = new SubClass();`).

5. Invoke the overridden method using the reference. Java will execute the subclass version based on the actual object type at runtime.

Worked Example:

Computing Interest Rates using Dynamic Dispatch **Problem Statement:** Calculate the simple interest for a principal amount using different interest rates provided by different banks. Create a superclass `Bank` and two subclasses `BankA` and `BankB`. Use dynamic method dispatch to get the rate of interest.

Step-by-step Solution:

Step 1: Define base and derived classes.

```
class Bank {
    double getRateOfInterest() {
        return 0;
    }
}

class BankA extends Bank {
    double getRateOfInterest() {
        return 7.5;
    }
}

class BankB extends Bank {
    double getRateOfInterest() {
        return 8.5;
    }
}
```

Step 2: Utilize dynamic method dispatch to compute interest.

```
public class InterestCalculator {
    public static void main(String[] args) {
        double principal = 10000;
        int time = 3; // years

        // Superclass reference
        Bank b;

        // Dispatch to BankA
        b = new BankA();
        double rateA = b.getRateOfInterest();
        double interestA = (principal * rateA * time) / 100;
        System.out.println("Interest from BankA: " + interestA);

        // Dispatch to BankB
        b = new BankB();
        double rateB = b.getRateOfInterest();
        double interestB = (principal * rateB * time) / 100;
        System.out.println("Interest from BankB: " + interestB);
    }
}
```

Output:

```
Interest from BankA: 2250.0
Interest from BankB: 2550.0
```

3.4 Interfaces and Multiple Inheritance

Methodology:

Defining and Implementing Interfaces Interfaces define a contract of abstract methods. They are used to achieve abstraction and multiple inheritance in Java, since a class can implement multiple interfaces but only extend one class.

Implementation Steps:

1. **Define Interfaces:** Use the `interface` keyword. Declare method signatures without bodies.
2. **Implement Interfaces:** Use the `implements` keyword in a class definition. For multiple interfaces, separate names with commas.
3. **Provide Concrete Methods:** The implementing class must define all methods declared in the interfaces. The methods must be declared as `public`.

Worked Example:

Implementing Multiple Interfaces for Geometric Shapes **Problem Statement:** Create two interfaces: `ShapeArea` with a method `computeArea()` and `ShapePerimeter` with a method `computePerimeter()`. Implement both interfaces in a class `Rectangle` to calculate the area and perimeter.

Step-by-step Solution:

Step 1: Define the interfaces.

```
interface ShapeArea {
    double computeArea();
}

interface ShapePerimeter {
    double computePerimeter();
}
```

Step 2: Implement the interfaces in a class.

```
class Rectangle implements ShapeArea, ShapePerimeter {
    double length;
    double width;

    Rectangle(double l, double w) {
        length = l;
        width = w;
    }

    // Implementing computeArea
    public double computeArea() {
        return length * width;
    }

    // Implementing computePerimeter
    public double computePerimeter() {
        return 2 * (length + width);
    }
}
```

Step 3: Execute.

```
public class InterfaceDemo {
```

```
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10, 5);
        System.out.println("Area: " + rect.computeArea());
        System.out.println("Perimeter: " + rect.computePerimeter());
    }
}
```

Output:

```
Area: 50.0
Perimeter: 30.0
```

3.5 The Object Class and Method Overriding

Methodology:

Overriding toString and equals Methods Every class in Java implicitly extends the `Object` class. Overriding its standard methods is essential for accurate object comparison and string representation.
Steps to Override Object Methods:

1. **Override toString:** Define `public String toString()` to return a concatenated string of the object's properties.
2. **Override equals:** Define `public boolean equals(Object obj)`.
3. Inside `equals`, check if `this == obj` (reference equality).
4. Check if `obj` is an instance of the class and cast it.
5. Compare the logical fields and return `true` if they match.

Worked Example:

Comparing Product Records **Problem Statement:** Create a `Product` class with `id` and `price`. Override `toString()` to display the product details and `equals()` to compare if two products are identical based on their `id`.
Step-by-step Solution:
Step 1: Define class and override toString.

```
class Product {
    int id;
    double price;

    Product(int id, double price) {
        this.id = id;
        this.price = price;
    }

    public String toString() {
        return "Product[ID=" + id + ", Price=" + price + "]";
    }
```

Step 2: Override equals method.

```
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
```

```
        if (obj instanceof Product) {
            Product other = (Product) obj;
            return this.id == other.id;
        }
        return false;
    }
}
```

Step 3: Test Object comparison.

```
public class ObjectDemo {
    public static void main(String[] args) {
        Product p1 = new Product(101, 500.0);
        Product p2 = new Product(102, 600.0);
        Product p3 = new Product(101, 550.0);

        System.out.println(p1.toString());
        System.out.println(p2.toString());

        System.out.println("p1 equals p2? " + p1.equals(p2));
        System.out.println("p1 equals p3? " + p1.equals(p3));
    }
}
```

Output:

```
Product[ID=101, Price=500.0]
Product[ID=102, Price=600.0]
p1 equals p2? false
p1 equals p3? true
```

3.6 Packages and Access Modifiers

Methodology:

Creating and Using Custom Packages Packages are used to group related classes, avoiding name conflicts and managing access control.

Steps to Create and Use a Package:

1. **Declare Package:** At the very top of your source file, add `package pkgName;`.
2. **Define Classes:** Write your classes and make them `public` if they need to be accessed from outside the package.
3. **Compile Directory Structure:** Compile the file into a directory structure matching the package name (e.g., `javac -d . ClassName.java`).
4. **Import Package:** In a different Java file, use `import pkgName.ClassName;` to utilize the packaged classes.

Worked Example:

Organizing Mathematical Operations in a Package **Problem Statement:** Create a package named `mathops` that contains a class `Calculator` with a method to multiply two numbers. Use this package in a separate `Main` class.

Step-by-step Solution:

Step 1: Create the Package Class (Calculator.java).

```
package mathops;

public class Calculator {
    public double multiply(double a, double b) {
        return a * b;
    }
}
```

Step 2: Create the Main Class (Main.java).

```
import mathops.Calculator;

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        double result = calc.multiply(4.5, 2.0);
        System.out.println("Product: " + result);
    }
}
```

Step 3: Compilation and Execution Process.

```
// In the terminal:
// Compile Calculator.java placing the class in a directory 'mathops'
javac -d . Calculator.java

// Compile Main.java
javac Main.java

// Run Main
java Main
```

Output:

```
Product: 9.0
```

CHAPTER 4

Package and Exception Handling

4.1 Creating and Importing User-Defined Packages

Methodology:

Algorithm to Create and Use a Package A package is used to group related classes. Creating and using a package involves the following steps:

1. **Define the Package:** At the very top of the Java source file, declare the package using the keyword `package` followed by the package name.
2. **Define the Class:** Create the class (it must be `public` if it needs to be accessed outside the package) and save the file.
3. **Compile the Package:** Compile the Java file using the `-d` flag to automatically generate the required directory structure: `javac -d . ClassName.java`
4. **Import the Package:** In another Java file, import the class using `import packageName.ClassName;`.
5. **Compile and Execute:** Compile the main program and execute it.

Worked Example:

Creating a Math Package and Using It **Problem Statement:** Create a package named `mathops` containing a `public` class `Calculator` with a method to calculate the factorial of a number. Write a separate main class to import and use this package.

Step 1: Create the Package Class (`Calculator.java`)

```
package mathops;

public class Calculator {
    public int factorial(int n) {
        int fact = 1;
        for(int i = 1; i <= n; i++) {
            fact *= i;
        }
        return fact;
    }
}
```

Step 2: Compile the Package Run the following command in the terminal to create the `mathops` folder containing `Calculator.class`:

```
javac -d . Calculator.java
```

Step 3: Create the Main Class (`MainApp.java`)

```
import mathops.Calculator;

public class MainApp {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.factorial(5);
        System.out.println("Factorial of 5 is: " + result);
    }
}
```

Step 4: Output

```
Factorial of 5 is: 120
```

4.2 Exception Handling Basics

Methodology:

Try-Catch-Finally Mechanism Exception handling prevents abnormal termination of a program. The process is defined by the following block structure:

1. **Identify Risky Code:** Place the code that might generate a runtime error inside a `try { }` block.
2. **Catch the Exception:** Immediately follow the `try` block with one or more `catch(ExceptionClass e) { }` blocks.
3. **Specific to General Catch:** When using multiple catch blocks, always catch the most specific exceptions first (e.g. `ArithmeticException`) followed by general exceptions (e.g. `Exception`).
4. **Execute Cleanup Code:** Add a `finally { }` block at the end. Code in the `finally` block is executed unconditionally regardless of whether an exception occurred or not.

Worked Example:

Handling Multiple Exceptions **Problem Statement:** Write a Java program that takes two integer arguments from the command line, divides the first by the second, and prints the result. Handle cases where the user does not provide enough arguments or attempts to divide by zero.

Step 1: Write the Program

```
public class DivisionApp {
    public static void main(String[] args) {
        try {
            int num1 = Integer.parseInt(args[0]);
            int num2 = Integer.parseInt(args[1]);
            int result = num1 / num2;
            System.out.println("Result: " + result);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Please provide two numbers.");
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero.");
        } catch (NumberFormatException e) {
            System.out.println("Error: Inputs must be valid integers.");
        } finally {
            System.out.println("Execution of try-catch completed.");
        }
    }
}
```

Step 2: Execution and Output *Case 1: Valid input*

```
> java DivisionApp 10 2
Result: 5
Execution of try-catch completed.
```

Case 2: Divide by zero

```
> java DivisionApp 10 0
Error: Cannot divide by zero.
Execution of try-catch completed.
```

Case 3: Missing arguments

```
> java DivisionApp 10
Error: Please provide two numbers.
Execution of try-catch completed.
```

4.3 Throwing Exceptions and Propagating Errors

Methodology:

Using throw and throws Keywords Sometimes it is necessary to explicitly raise an exception or pass the responsibility of handling an exception to the calling method.

1. **Using throw:** The `throw` keyword is used inside a method body to manually trigger a specific exception instance: `throw new ExceptionClass("Error Message");`
2. **Using throws:** The `throws` keyword is used in the method signature to indicate that the method might throw certain checked exceptions delegating the handling to the caller: `returnType methodName(args) throws ExceptionClass1`

Worked Example:

Explicitly Throwing an Exception **Problem Statement:** Create a method `validateAge(int age)` that throws an `IllegalArgumentException` if the age is less than 18. Call this method from `main` and handle the exception.

Step 1: Write the Validation Method

```
public class AgeValidator {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be at least 18.");
        }
        System.out.println("Registration successful!");
    }

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

Step 2: Execution and Output

```
Exception Caught: Age must be at least 18.
```

4.4 Custom User-Defined Exceptions

Methodology:

Algorithm for User-Defined Exceptions Java allows developers to define custom exceptions to represent specific business logic errors.

1. **Create the Exception Class:** Create a new class that extends `Exception` (for checked exceptions) or `RuntimeException` (for unchecked exceptions).
2. **Define Constructors:** Add a default constructor and a parameterized constructor that accepts a `String` message. Pass the message to the superclass constructor using `super(message)`.
3. **Throw the Custom Exception:** Instantiate and `throw` the custom exception in your business logic based on a condition.
4. **Handle the Custom Exception:** Use a `try-catch` block to catch the custom exception and print the error message.

Worked Example:

Implementing InsufficientBalanceException **Problem Statement:** Define a custom checked exception `InsufficientBalanceException`. Create a `BankAccount` class with a `withdraw` method that throws this exception if the withdrawal amount exceeds the balance.

Step 1: Define the Custom Exception

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

Step 2: Create the BankAccount Class

```
public class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException("Withdrawal failed. " +
                "Requested: " + amount + " Available: " + balance);
        }
        balance -= amount;
        System.out.println("Successfully withdrew " + amount +
            ". New balance: " + balance);
    }
}
```

Step 3: Test with Main Method

```
public class BankingApp {
```

```
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.00);
        try {
            account.withdraw(500.00);
            account.withdraw(800.00); // This will cause an exception
        } catch (InsufficientBalanceException e) {
            System.out.println("Transaction Error: " + e.getMessage());
        }
    }
}
```

Step 4: Execution and Output

```
Successfully withdrew 500.0. New balance: 500.0
Transaction Error: Withdrawal failed. Requested: 800.0 Available: 500.0
```

4.5 Static Import

Methodology:

Applying Static Import The static import feature allows the members (fields and methods) defined in a class as `public static` to be used without specifying the class name.

1. **Use the Keyword:** The syntax requires using `import static` followed by the fully qualified name of the class and the member.
2. **Import Specific Member:** `import static java.lang.Math.PI;`
3. **Import All Static Members:** `import static java.lang.Math.*;`
4. **Access Directly:** Use the static member directly (e.g. `PI` instead of `Math.PI`).

Worked Example:

Calculating Circle Area using Static Import **Problem Statement:** Calculate the area and circumference of a circle. Use `static import` for the `Math` class constants and methods.

Step 1: Write the Program

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;

public class CircleGeometry {
    public static void main(String[] args) {
        double radius = 5.0;

        // Notice we do not use Math.PI or Math.pow
        double area = PI * pow(radius, 2);
        double circumference = 2 * PI * radius;

        System.out.println("Radius: " + radius);
        System.out.println("Area: " + area);
        System.out.println("Circumference: " + circumference);
    }
}
```

Step 2: Execution and Output

```
Radius: 5.0
Area: 78.53981633974483
Circumference: 31.41592653589793
```

CHAPTER 5

File Handling and Collections Framework

File handling and the Collections framework are crucial for storing data persistently and managing groups of objects efficiently. In Java, streams are used for file I/O operations, while the Collections framework provides data structures like lists, sets, and maps to store and manipulate object data dynamically.

5.1 File Handling using Byte Streams

Byte streams in Java handle raw binary data. `FileInputStream` and `FileOutputStream` are the fundamental classes used for byte-oriented reading and writing of files.

Methodology:

Writing Data using FileOutputStream To write data to a file using `FileOutputStream`:

1. Import the `java.io.FileOutputStream` and `java.io.IOException` classes.
2. Create a `FileOutputStream` object, passing the file path as a string to its constructor. Handle or declare the `IOException`.
3. Convert the string or data into a byte array. For strings, use the `getBytes()` method.
4. Use the `write(byte[] b)` method to write the byte array to the file.
5. Close the stream using the `close()` method to release system resources.

Worked Example:

Writing a String to a File Write a Java program to create a new text file named `output.txt` and write the text "Java File Handling" into it using `FileOutputStream`.
Solution:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class WriteFileExample {
    public static void main(String[] args) {
        String data = "Java File Handling";
        try {
            // Step 2: Create FileOutputStream
            FileOutputStream fos = new FileOutputStream("output.txt");

            // Step 3: Convert string to byte array
            byte[] byteData = data.getBytes();

            // Step 4: Write to file
            fos.write(byteData);
```

```
            System.out.println("Data written to file successfully.");

            // Step 5: Close the stream
            fos.close();
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Methodology:

Reading Data using FileInputStream To read data from a file using `FileInputStream`:

1. Import the `java.io.FileInputStream` and `java.io.IOException` classes.
2. Create a `FileInputStream` object, providing the file path to read from.
3. Declare an integer variable to hold each byte read.
4. Use a `while` loop to read bytes using the `read()` method until it returns `-1` (indicating the end of the file).
5. Cast the integer byte value to a `char` and process/print it.
6. Close the input stream using the `close()` method.

Worked Example:

Reading from a File Write a Java program to read the contents of `output.txt` byte-by-byte using `FileInputStream` and print it to the console.

Solution:

```
import java.io.FileInputStream;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
        try {
            // Step 2: Create FileInputStream
            FileInputStream fis = new FileInputStream("output.txt");

            // Step 3: Variable to hold byte
            int i;

            System.out.print("File Contents: ");
            // Step 4: Read until end of file
            while ((i = fis.read()) != -1) {
                // Step 5: Cast and print
                System.out.print((char) i);
            }
            System.out.println();

            // Step 6: Close stream
            fis.close();
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
```

```
        }
    }
}
```

5.2 File Handling using Character Streams

Character streams handle 16-bit Unicode characters, making them suitable for reading and writing text files. `FileReader` and `FileWriter` are commonly used for this purpose.

Methodology:

Writing Text using FileWriter To write text data using `FileWriter`:

1. Import `java.io.FileWriter` and `java.io.IOException`.
2. Create a `FileWriter` object with the file name. To append data, use the constructor `FileWriter(String fileName, boolean append)`.
3. Use the `write(String str)` method to write the string directly to the file.
4. Call the `close()` method to save the changes and free resources.

Worked Example:

Writing Text to a File Write a Java program using `FileWriter` to create a text file and write a simple message into it.
Solution:

```
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterDemo {
    public static void main(String[] args) {
        try {
            FileWriter writer = new FileWriter("notes.txt");
            writer.write("Hello! Welcome to Character Streams.");
            writer.close();
            System.out.println("Successfully wrote to the file.");
        } catch (IOException e) {
            System.out.println("An I/O error occurred.");
            e.printStackTrace();
        }
    }
}
```

Methodology:

Reading Text using FileReader To read text data using `FileReader`:

1. Import `java.io.FileReader` and `java.io.IOException`.
2. Create a `FileReader` object pointing to the target text file.
3. Use the `read()` method inside a loop, similar to byte streams, to read characters one by one until it returns `-1`.
4. Cast the integer value to a `char` to retrieve the character.

5. Close the `FileReader` using the `close()` method.

Worked Example:

Reading Text from a File Write a Java program using `FileReader` to read characters from a text file and display them on the console.
Solution:

```
import java.io.FileReader;
import java.io.IOException;

public class FileReaderDemo {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader("notes.txt");
            int character;
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
            System.out.println();
            reader.close();
        } catch (IOException e) {
            System.out.println("An I/O error occurred.");
            e.printStackTrace();
        }
    }
}
```

5.3 Collections Framework: Lists

The Java Collections Framework provides an architecture to store and manipulate a group of objects. The `List` interface represents an ordered collection that allows duplicate elements. The two most common implementations are `ArrayList` and `LinkedList`.

Methodology:

Working with ArrayList `ArrayList` uses a dynamic array to store elements. It is fast for retrieving data but slower for insertions and deletions.

1. Import `java.util.ArrayList`.
2. Create an instance: `ArrayList<Type> list = new ArrayList<>();` (Use wrapper classes like `Integer`, `Double`, `String` for primitive types).
3. Use `add(element)` to append elements to the list.
4. Use `get(index)` to retrieve an element at a specific index.
5. Use `set(index, newElement)` to modify an element.
6. Use `remove(index)` or `remove(Object)` to delete an element.
7. Use `size()` to get the number of elements.

Worked Example:

ArrayList Operations Write a program to create an `ArrayList` of strings, add three city names, modify the second city, remove the first city, and print the size and the contents using a for-each loop.
Solution:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        // Step 2: Create ArrayList
        ArrayList<String> cities = new ArrayList<>();

        // Step 3: Add elements
        cities.add("Ahmedabad");
        cities.add("Surat");
        cities.add("Vadodara");

        // Step 5: Modify element at index 1
        cities.set(1, "Rajkot");

        // Step 6: Remove element at index 0
        cities.remove(0);

        // Step 7: Print size
        System.out.println("List Size: " + cities.size());

        // Iterate using for-each loop
        System.out.println("Cities in the list:");
        for (String city : cities) {
            System.out.println(city);
        }
    }
}
```

Methodology:

Working with LinkedList `LinkedList` uses a doubly linked list structure. It is faster for insertion and deletion compared to `ArrayList` but slower for random access retrieval.

1. Import `java.util.LinkedList`.
2. Create an instance: `LinkedList<Type> list = new LinkedList<>();`.
3. Use standard `List` methods like `add()`, `get()`, `remove()`, and `size()`.
4. Utilize `LinkedList`-specific methods like `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, and `removeLast()` for deque operations.

Worked Example:

LinkedList Operations Write a Java program to create a `LinkedList` of integers. Add elements to the start, end, and middle of the list. Display the elements.
Solution:

```
import java.util.LinkedList;

public class LinkedListExample {
```

```
    public static void main(String[] args) {
        LinkedList<Integer> numbers = new LinkedList<>();

        // Add elements
        numbers.add(20);
        numbers.add(30);

        // Add at the first position
        numbers.addFirst(10);

        // Add at the last position
        numbers.addLast(50);

        // Add at a specific index (index 3)
        numbers.add(3, 40);

        System.out.println("LinkedList Elements:");
        for (Integer num : numbers) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

5.4 Collections Framework: Sets

The `Set` interface represents an unordered collection that does *not* allow duplicate elements.

Methodology:

Working with HashSet `HashSet` uses a hash table for storage. It is the best-performing set implementation but does not guarantee any iteration order.

1. Import `java.util.HashSet`.
2. Create an instance: `HashSet<Type> set = new HashSet<>();`.
3. Use `add(element)` to insert items. Duplicate items will be ignored.
4. Use `remove(element)` to remove a specific item.
5. Use `contains(element)` to check if an item exists in the set.
6. Iterate through the set using a for-each loop, as elements cannot be accessed by an index.

Worked Example:

HashSet Operations Write a program to create a `HashSet` of strings representing colors. Add "Red", "Green", "Blue", and "Red" again. Show that duplicates are ignored by printing all elements.
Solution:

```
import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
        HashSet<String> colors = new HashSet<>();
```

```
        // Add elements
        colors.add("Red");
        colors.add("Green");
        colors.add("Blue");

        // Adding a duplicate element
        colors.add("Red");

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

        // Check for an element
        if (colors.contains("Green")) {
            System.out.println("Green is in the set.");
        }

        // Iterate and print elements
        System.out.println("Colors in the HashSet:");
        for (String color : colors) {
            System.out.println(color);
        }
    }
}
```

5.5 Collections Framework: Maps

The `Map` interface represents an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Methodology:

Working with HashMap `HashMap` stores key-value pairs and uses a hash table for rapid access. It does not maintain order.

1. Import `java.util.HashMap`.
2. Create an instance specifying key and value types: `HashMap<KeyType, ValueType> map = new HashMap<>();`.
3. Use `put(key, value)` to add or update a key-value pair.
4. Use `get(key)` to retrieve the value associated with a specific key.
5. Use `remove(key)` to delete a key-value pair.
6. Use `containsKey(key)` or `containsValue(value)` to check for existence.
7. Iterate over keys using `keySet()` or values using `values()`.

Worked Example:

HashMap Operations Write a Java program to create a `HashMap` mapping student roll numbers (Integer) to their names (String). Add 3 students, retrieve the name of the student with roll number 102, and iterate through the map to print all keys and values.
Solution:

```
import java.util.HashMap;
```

```
public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> students = new HashMap<>();

        // Add key-value pairs
        students.put(101, "Amit");
        students.put(102, "Neha");
        students.put(103, "Rahul");

        // Retrieve a value by key
        String name = students.get(102);
        System.out.println("Student with Roll No 102: " + name);

        // Remove an entry
        students.remove(103);

        // Iterate through the HashMap using keySet()
        System.out.println("All Students:");
        for (Integer rollNo : students.keySet()) {
            System.out.println("Roll No: " + rollNo + ", Name: " + students.get(rollNo));
        }
    }
}
```

Appendix: Key Formulae

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

Mathematical Operations

Factorial

The factorial of a positive integer N, denoted by $N!$, is the product of all positive integers less than or equal to N.

$$N! = N \times (N-1) \times (N-2) \times \cdots \times 1 \tag{5.1}$$

Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. The sequence is defined by the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2 \tag{5.2}$$

with the base cases:

$$F_1 = 0$$
$$F_2 = 1$$

Financial Mathematics

Simple Interest

The simple interest (SI) accumulated on a given principal is calculated using the following formula:

$$SI = \frac{P \times R \times T}{100} \tag{5.3}$$

where:

- P is the principal amount (initial investment or loan amount)
- R is the annual interest rate (in percentage)
- T is the time period (duration)