Problem Solver (DI03016051) - Problem Solver

Antigravity AI

June 4, 2026

Problem Solver

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction to Object Oriented Programming paradigm and JAVA

Object-Oriented Programming (OOP) is a paradigm that organizes software design around data or objects rather than functions and logic. Java is a versatile and robust programming language widely used for building applications based on OOP principles. This chapter covers essential algorithms and problem-solving techniques related to Java fundamentals.

1.1 Introduction to Programming paradigm

Procedural programming focuses on writing sequences of instructions to manipulate data. In contrast, Object-Oriented Programming encapsulates data and behavior into objects. To solve problems using OOP, one must shift from thinking about "functions" to thinking about "entities".

Methodology:

Object Oriented Problem Solving Strategy To model a real-world problem using OOP:

1. **Identify Entities:** Determine the primary nouns in the problem statement. These become your `classes`.
2. **Identify Attributes:** Determine the characteristics or state of each entity. These become the `variables` or fields.
3. **Identify Behaviors:** Determine the actions the entity can perform (verbs). These become the `methods`.
4. **Establish Relationships:** Define how different classes interact with each other.

Worked Example:

Modeling a Simple Bank Account Model a bank account that allows a user to deposit money and check the balance.
Step 1: Identify Entities The primary entity is the Bank Account. Class: `BankAccount`
Step 2: Identify Attributes Characteristics include an account number and a balance. Variables: `accountNumber` (String) and `balance` (double).
Step 3: Identify Behaviors Actions include depositing money and retrieving the balance. Methods: `deposit(amount)` and `getBalance()`.
Step 4: Implementation Structure

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

    void deposit(double amount) {
        balance += amount;
```

```
    }

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

1.2 Basics of Java

Java programs are executed by the Java Virtual Machine (JVM), making them platform-independent. Writing a Java application involves structuring code into classes and methods, starting with a `main` method.

Methodology:

Structuring and Executing a Java Program

1. **Define the Class:** Every Java application must have at least one class defined using the `class` keyword.
2. **Define the Main Method:** The entry point of any standalone Java application is the `main` method. Its signature must strictly be: `public static void main(String[] args)`.
3. **Write Statements:** Place computational logic inside the method blocks.
4. **Compile:** Use `javac Filename.java` to compile the source code into bytecode (`.class` file).
5. **Execute:** Use `java ClassName` to run the bytecode on the JVM.

Worked Example:

Writing a Basic Greeting Program Write a Java program to print a greeting message to the console.
Step 1: Define the Class Choose a descriptive name for the class.

```
public class GreetingApp {
```

Step 2: Define the Main Method

```
    public static void main(String[] args) {
```

Step 3: Write Output Statement Use `System.out.println` to print to standard output.

```
        System.out.println("Welcome to Java Programming!");
    }
}
```

Step 4: Compile and Run Save as `GreetingApp.java`. Compilation: `javac GreetingApp.java` Execution: `java GreetingApp` Output: `Welcome to Java Programming!`

1.3 Basic Language Elements

Java provides various foundational elements necessary for defining logic and storing temporary states during computation.

1.3.1 Data Types Variables and Scope

Java is strongly typed. Every variable must have a declared type that dictates how much memory it occupies and what operations can be performed on it.

Methodology:

Variable Declaration and Scope Management

1. **Determine Data Requirements:** Choose appropriate primitive types: `int` (integers) `double` (floating-point) `boolean` (true/false) `char` (single character).
2. **Declare Variable:** Syntax: `DataType variableName = initialValue;`
3. **Identify Scope:** A variable's scope is defined by the block (enclosed in `{ }`) where it is declared. It cannot be accessed outside this block.

Worked Example:

Calculating Cylinder Volume with Block Scope Calculate the volume of a cylinder. Use a variable for π that is scoped only within the calculation block.

Step 1: Declare Inputs

```
double radius = 5.0;
double height = 10.0;
double volume = 0.0;
```

Step 2: Scoped Calculation

```
{
    // pi is scoped to this block
    double pi = 3.14159;
    volume = pi * radius * radius * height;
}
```

Step 3: Output Result The `volume` variable is accessible here but `pi` is not.

```
System.out.println("Volume: " + volume);
// System.out.println(pi); // This would cause a compilation error
```

1.3.2 Type Conversion

When assigning a value of one data type to another type mismatch errors can occur.

Methodology:

Handling Type Conversion

1. **Implicit Casting (Widening):** Happens automatically when converting a smaller type to a larger type size (e.g. `int` to `double`). No data loss occurs.
2. **Explicit Casting (Narrowing):** Must be done manually when converting a larger type to a smaller type (e.g. `double` to `int`). Syntax: `targetType varName = (targetType) largerVar;`
3. **Handling Precision Loss:** Be aware that narrowing casts will truncate decimal values or cause overflow if the value exceeds the target type's capacity.

Worked Example:

Extracting Fractional Parts using Casting Given a floating-point number 78.95 extract only its fractional part using type conversion.

Step 1: Declare Number

```
double originalNumber = 78.95;
```

Step 2: Explicitly Cast to Integer Cast to `int` to truncate the fractional part.

```
int integerPart = (int) originalNumber; // value becomes 78
```

Step 3: Calculate Fractional Part

```
double fractionalPart = originalNumber - integerPart;
// 78.95 - 78 = 0.95
System.out.println("Fraction: " + fractionalPart);
```

1.3.3 Operators

Operators perform calculations logical comparisons and bitwise manipulations.

Methodology:

Evaluating Expressions using Operators

1. **Arithmetic Operators:** Use `+`, `-`, `*`, `/`, `%` for basic math. Division of two integers results in an integer.
2. **Relational and Logical Operators:** Use `==`, `!=`, `<`, `>` combined with `&&` (AND) `||` (OR) to form complex boolean expressions.
3. **Ternary Operator:** Use `condition ? exprIfTrue : exprIfFalse` for concise conditional assignment.
4. **Operator Precedence:** Use parentheses `()` to enforce a specific evaluation order overriding default precedence.

Worked Example:

Determining Leap Year Determine if a given year is a leap year using a single logical expression. A year is a leap year if it is divisible by 4 but not by 100 or if it is divisible by 400.
Step 1: Define Inputs

```
int year = 2024;
```

Step 2: Formulate Logical Expression

```
boolean isLeap = ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0);
```

Step 3: Use Ternary Operator for Output

```
String result = isLeap ? "Leap Year" : "Not a Leap Year";
System.out.println(year + " is a " + result);
```

1.3.4 Control Statements

Control statements dictate the flow of execution based on conditions and loops.

Methodology:

Designing Loop and Branching Structures

1. **Selection (if-else / switch):** Use when executing code blocks conditionally. Use `switch` for multiple discrete values of a single variable.
2. **Iteration Initialization:** Define the starting state of the loop control variable.
3. **Iteration Condition:** Define the boolean expression that keeps the loop running (`while` or `for`).

4. **Iteration Update:** Increment or modify the loop variable to ensure termination avoiding infinite loops.

Worked Example:

Generating Prime Numbers using Control Flow Find the sum of all prime numbers between 1 and 20.
Step 1: Initialize Loop for Range

```
int sum = 0;
for (int num = 2; num <= 20; num++) {
```

Step 2: Check Primality using Nested Loop Assume the number is prime until proven otherwise.

```
    boolean isPrime = true;
    for (int i = 2; i <= num / 2; i++) {
        if (num % i == 0) {
            isPrime = false;
            break; // Exit inner loop early
        }
    }
```

Step 3: Accumulate Sum

```
    if (isPrime) {
        sum += num;
    }
}
System.out.println("Sum of primes: " + sum); // Output: 77
```

1.3.5 Command Line Arguments

Java allows passing data directly to the program upon execution via the terminal.

Methodology:

Processing Command Line Arguments

1. **Accessing the Array:** Arguments are passed to the `String[] args` parameter of the `main` method.
2. **Check Argument Count:** Always verify the length of the `args` array using `args.length` to prevent `ArrayIndexOutOfBoundsException`.
3. **Parsing Data:** Arguments are always strings. Convert them to numerical types using parsing methods like `Integer.parseInt(args[i])` or `Double.parseDouble(args[i])`.

Worked Example:

Calculating Average from Command Line Write a program that takes numbers as command line arguments and prints their average.
Step 1: Verify Input Length

```
public static void main(String[] args) {
    if (args.length == 0) {
        System.out.println("No arguments provided.");
        return; // Exit program
    }
```

Step 2: Accumulate Sum by Parsing Strings

```
    double sum = 0;
    for (int i = 0; i < args.length; i++) {
        sum += Double.parseDouble(args[i]);
    }
```

Step 3: Calculate and Print Average

```
    double average = sum / args.length;
    System.out.println("Average: " + average);
}
```

Execution: `java AverageCalc 10.5 20.0 5.5` Output: `Average:  12.0`

1.3.6 Java Memory Management

Java manages memory automatically allocating space for variables and objects in different memory regions and cleaning up unused objects using Garbage Collection.

Methodology:

Tracing Stack and Heap Allocation

1. **Stack Memory:** Stores local variables primitive data types and method call frames. Access is extremely fast. Memory is freed when the method block ends.
2. **Heap Memory:** Stores all created objects and class-level variables (using the `new` keyword).
3. **Reference Variables:** When an object is created on the Heap its memory address is stored in a reference variable located on the Stack.
4. **Garbage Collection:** If a Heap object has no reference variables pointing to it it becomes eligible for automatic garbage collection to free up memory.

Worked Example:

Demonstrating Object References Analyze what happens in memory when assigning reference variables.
Step 1: Create Object

```
// 'p1' is on the Stack pointing to a new Point object on the Heap
Point p1 = new Point(10, 20);
```

Step 2: Assign Reference

```
// 'p2' is created on the Stack and points to the SAME Heap object as 'p1'
Point p2 = p1;
```

Step 3: Modify via Second Reference

```
p2.x = 50;
// Because p1 and p2 point to the same object p1.x is now also 50.
System.out.println(p1.x); // Output: 50
```

Step 4: Dereferencing

```
p1 = null;
p2 = null;
// The Point object on the Heap now has 0 references.
// It is eligible for Garbage Collection.
```

CHAPTER 2

Fundamentals of Object Oriented Programming

2.1 Objects Classes and Methods

Object-Oriented Programming (OOP) is a paradigm organized around objects rather than logic. An object encapsulates data and the methods that operate on that data.

Methodology:

Defining a Class and Creating Objects

1. **Define the Class**: Use the `class` keyword followed by the class name.
2. **Declare Instance Variables**: Define the attributes (data members) inside the class.
3. **Instantiate the Object**: Use the `new` keyword followed by the class constructor to create an object. Syntax: `ClassName objectName = new ClassName();`
4. **Access Members**: Use the dot (`.`) operator to access variables and methods of the object. Syntax: `objectName.variableName`

Worked Example:

Calculate Area of a Rectangle using Class and Object **Problem:** Create a class `Rectangle` with properties `length` and `width`. Create an object of this class, assign values, and calculate the area.
Solution:

1. **Define Class and Variables**:

```
class Rectangle {
    int length;
    int width;
}
```

2. **Create Object and Assign Values**:

```
public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(); // Object creation
        rect.length = 10;                 // Assigning values
        rect.width = 5;
```

3. **Calculate Area and Print**:

```
        int area = rect.length * rect.width;
        System.out.println("Area: " + area);
    }
}
```

Output: Area: 50

Methodology:

Creating and Calling Methods

1. **Define Method Signature**: Specify the access modifier, return type, method name, and parameter list. Syntax: `returnType methodName(parameters)`
2. **Write Method Body**: Enclose the logic inside curly braces `{}`. Use the `return` keyword if the return type is not `void`.
3. **Call Method**: Invoke the method using the object reference and pass required arguments. Syntax: `objectName.methodName(arguments);`

Worked Example:

Student Grade Calculation using Methods **Problem:** Create a `Student` class with a method `calculateGrade(int marks)` that returns 'P' for marks ≥ 40 and 'F' otherwise.
Solution:

1. **Define Method inside Class**:

```
class Student {
    char calculateGrade(int marks) {
        if (marks >= 40) {
            return 'P';
        } else {
            return 'F';
        }
    }
}
```

2. **Call Method from Main**:

```
public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        char grade = s1.calculateGrade(75);
        System.out.println("Grade is: " + grade);
    }
}
```

Output: Grade is: P

2.2 Access modifiers this keyword and static keyword

Methodology:

Using Access Modifiers Access modifiers restrict the scope of classes, variables, and methods.

1. **private**: Accessible only within the same class. Used for encapsulation.
2. **default (no modifier)**: Accessible only within the same package.
3. **protected**: Accessible within the same package and subclasses in other packages.
4. **public**: Accessible from anywhere.

To access `private` variables, provide `public` getter and setter methods.

Worked Example:

Encapsulation with Private Variables **Problem:** Secure the `balance` attribute in a `BankAccount` class using private access and provide a method to deposit money safely.
Solution:

1. **Declare Private Variable and Public Method**:

```
class BankAccount {
    private double balance = 0.0; // Hidden from direct access

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

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

2. **Test Encapsulation**:

```
public class Main {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        acc.deposit(500);
        // acc.balance = 1000; // ERROR: balance has private access
        System.out.println("Balance: " + acc.getBalance());
    }
}
```

Output:
Deposited: 500.0
Balance: 500.0

Methodology:

Using the this Keyword The `this` keyword acts as a reference to the current object.

1. **Resolve Variable Hiding**: Use `this.variableName` when local variables (parameters) have the same name as instance variables.

2. **Invoke Current Class Method**: Use `this.methodName()` to explicitly call another method in the same class.

3. **Invoke Current Class Constructor**: Use `this()` to call another constructor from within a constructor.

Worked Example:

Resolving Variable Hiding using this Keyword **Problem:** Initialize instance variables `id` and `name` using a method where parameter names are identical to instance variable names.
Solution:

1. **Use this Keyword to Differentiate**:

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

    void setDetails(int id, String name) {
        this.id = id;     // this.id refers to instance variable
        this.name = name; // this.name refers to instance variable
    }

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

2. **Execute**:

```
public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();
        emp.setDetails(101, "Alice");
        emp.display();
    }
}
```

Output: 101 Alice

Methodology:

Using the static Keyword The `static` keyword indicates that a member belongs to the class itself rather than instances.

1. **Static Variable**: Shared among all instances of a class. Memory is allocated once. Syntax: `static dataType varName;`

2. **Static Method**: Can be called without creating an object. Can only directly access static data. Syntax: `static returnType methodName()`

3. **Static Block**: Used to initialize static data members. Executed once when the class is loaded.

Worked Example:

Counting Objects using Static Variable **Problem:** Track the number of `Item` objects created in a program.
Solution:

1. **Declare Static Counter**:

```
class Item {
    static int count = 0; // Shared across all objects

    Item() {
        count++; // Increment shared counter
    }

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

2. **Create Objects and Check Count**:

```
public class Main {
    public static void main(String[] args) {
        Item i1 = new Item();
        Item i2 = new Item();
        Item i3 = new Item();

        Item.displayCount(); // Calling static method using Class name
    }
}
```

Output: Total Items: 3

2.3 Constructors

Methodology:

Defining Default and Parameterized Constructors A constructor is a special method used to initialize objects. It has the same name as the class and no return type.

1. **Default Constructor**: Takes no parameters. Used to provide default values. Syntax: `ClassName() { }`
2. **Parameterized Constructor**: Takes parameters to initialize variables with specific values at the time of creation. Syntax: `ClassName(type param1) { }`

Worked Example:

Initializing Data using Constructors **Problem:** Create a `Book` class with a parameterized constructor to initialize `title` and `price`.
Solution:

1. **Define Parameterized Constructor**:

```
class Book {
    String title;
```

```
    double price;

    // Parameterized constructor
    Book(String t, double p) {
        title = t;
        price = p;
    }

    void display() {
        System.out.println("Book: " + title + " | Price: " + price);
    }
}
```

2. **Instantiate using Constructor**:

```
public class Main {
    public static void main(String[] args) {
        Book b1 = new Book("Java Programming", 450.50);
        b1.display();
    }
}
```

Output: Book: Java Programming | Price: 450.5

Methodology:

Constructor Overloading A class can have multiple constructors with different parameter lists. This allows object initialization in different ways.

1. **Vary Number of Parameters**: E.g. `ClassA(int x)` and `ClassA(int x, int y)`.
2. **Vary Types of Parameters**: E.g. `ClassA(int x)` and `ClassA(double x)`.

Worked Example:

Box Dimensions Initialization using Overloaded Constructors **Problem:** Create a `Box` class with three constructors: one for a default box one for a cube and one for a general cuboid.
Solution:

1. **Define Overloaded Constructors**:

```
class Box {
    double length, width, height;

    // 1. Default constructor
    Box() {
        length = width = height = 1.0;
    }

    // 2. Constructor for a cube
    Box(double side) {
        length = width = height = side;
    }

    // 3. Constructor for a cuboid
    Box(double l, double w, double h) {
```

```
        length = l; width = w; height = h;
    }

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

2. **Test Overloaded Constructors**:

```
public class Main {
    public static void main(String[] args) {
        Box b1 = new Box();          // Calls default
        Box b2 = new Box(5.0);       // Calls single parameter
        Box b3 = new Box(2.0, 3.0, 4.0); // Calls three parameter

        System.out.println("V1: " + b1.volume());
        System.out.println("V2: " + b2.volume());
        System.out.println("V3: " + b3.volume());
    }
}
```

Output:
V1: 1.0
V2: 125.0
V3: 24.0

2.4 String class StringBuffer class and Wrapper Class

Methodology:

String Class Operations The `String` class creates immutable sequences of characters. Useful methods:

1. `length()`: Returns integer length of string.
2. `charAt(int index)`: Returns character at specified index.
3. `substring(int beginIndex)`: Returns a substring starting from `beginIndex`.
4. `equals(String s)`: Compares content for equality.
5. `toLowerCase()` / `toUpperCase()`: Converts case.

Worked Example:

Counting Vowels in a String **Problem:** Count the number of vowels in a given string using `String` methods.
Solution:

1. **Loop through String and Check Characters**:

```
public class StringExample {
    public static void main(String[] args) {
        String str = "Engineering";
        int count = 0;
        String lowerStr = str.toLowerCase();
```

```
            for (int i = 0; i < lowerStr.length(); i++) {
                char ch = lowerStr.charAt(i);
                if (ch == 'a' || ch == 'e' || ch == 'i' ||
                    ch == 'o' || ch == 'u') {
                    count++;
                }
            }
            System.out.println("Vowels count: " + count);
        }
    }
```

Output: Vowels count: 5

Methodology:

StringBuffer Class Operations The `StringBuffer` class creates mutable (modifiable) sequences of characters, which is memory-efficient for continuous string modifications.

1. `append(String s)`: Adds string to the end.
2. `insert(int offset, String s)`: Inserts string at specified index.
3. `replace(int start, int end, String s)`: Replaces characters from start to end-1.
4. `delete(int start, int end)`: Removes characters from start to end-1.
5. `reverse()`: Reverses the entire string buffer.

Worked Example:

Reversing and Modifying a String **Problem:** Create a `StringBuffer` with text "Hello". Append " World" insert "Java " at index 6 and reverse it.
Solution:

1. **Apply StringBuffer Methods Sequentially**:

```
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        sb.append(" World");
        System.out.println("After append: " + sb);

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

        sb.reverse();
        System.out.println("After reverse: " + sb);
    }
}
```

Output:
After append: Hello World
After insert: Hello Java World
After reverse: dlroW avaJ olleH

Methodology:

Wrapper Class Operations Wrapper classes convert primitive data types into objects.

1. **Autoboxing**: Automatic conversion of primitive type to corresponding wrapper object (e.g., `int` to `Integer`).
2. **Unboxing**: Automatic conversion of wrapper object to corresponding primitive type.
3. **Parsing**: Converting strings to primitives using methods like `Integer.parseInt(String s)` or `Double.parseDouble(String s)`.

Worked Example:

Converting String to Primitive Data Types **Problem:** Convert a string "150" to an integer and a string "45.5" to a double, then calculate their sum.
Solution:

1. **Use Parse Methods of Wrapper Classes**:

```
public class WrapperExample {
    public static void main(String[] args) {
        String numStr1 = "150";
        String numStr2 = "45.5";

        // Parsing Strings to primitives
        int num1 = Integer.parseInt(numStr1);
        double num2 = Double.parseDouble(numStr2);

        double sum = num1 + num2;
        System.out.println("Sum is: " + sum);

        // Autoboxing example
        Integer obj = num1;
        System.out.println("Wrapper Object Value: " + obj);
    }
}
```

Output:
Sum is: 195.5
Wrapper Object Value: 150

CHAPTER 3

Inheritance, Interface and Packages

3.1 Inheritance and its Types

Inheritance is an object-oriented mechanism that allows one class to acquire the properties (fields) and behaviors (methods) of another class. It promotes code reusability and establishes a parent-child relationship between classes.

Methodology:

Implementing Inheritance To implement inheritance in Java:

1. Identify the common attributes and methods, and define them in a **Superclass** (Parent class).
2. Define a **Subclass** (Child class) using the `extends` keyword.
3. Syntax: `class Subclass extends Superclass { // body }`
4. The subclass inherits non-private members of the superclass.
5. Use the `super` keyword to access superclass constructors, methods, or fields when they are hidden or overridden by the subclass.

Worked Example:

Single Inheritance Implementation **Problem:** Create a superclass `Employee` with a salary attribute and a subclass `Programmer` with a bonus attribute. Display both values from the subclass.
Solution Step-by-Step:
Step 1: Define the Superclass

```
class Employee {
    int salary = 40000;
}
```

Step 2: Define the Subclass using extends

```
class Programmer extends Employee {
    int bonus = 10000;
}
```

Step 3: Test the Inheritance

```
public class Main {
    public static void main(String[] args) {
        Programmer p = new Programmer();
        System.out.println("Salary is:  " + p.salary);
        System.out.println("Bonus is:  " + p.bonus);
    }
}
```

Output:
Salary is: 40000
Bonus is: 10000

Methodology:

Types of Inheritance Supported in Classes

1. **Single Inheritance:** One subclass inherits from one superclass.
2. **Multilevel Inheritance:** A subclass inherits from another subclass (e.g. Class C extends Class B, and Class B extends Class A).
3. **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.
4. *Note:* Multiple inheritance (one class extending multiple classes) is NOT supported in Java through classes to avoid the Diamond Problem. It is achieved via interfaces.

Worked Example:

Multilevel Inheritance Implementation **Problem:** Implement a multilevel inheritance hierarchy where `Dog` extends `Mammal`, which in turn extends `Animal`.
Solution:

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

class Mammal extends Animal {
    void walk() { System.out.println("Walking..."); }
}

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

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat(); // Inherited from Animal
        d.walk(); // Inherited from Mammal
        d.bark(); // Own method
    }
}
```

3.2 Polymorphism

Polymorphism means "many forms." It allows objects of different classes to be treated as objects of a common superclass. There are two primary types of polymorphism in Java: Compile-time (Method Overloading) and Run-time (Method Overriding).

Methodology:

Method Overloading Compile-time polymorphism is achieved when multiple methods have the same name but different parameter lists.

1. Create multiple methods with the exact same name within the same class.

2. Ensure each method has a different parameter profile (number of parameters, types of parameters, or order of parameter types).
3. The compiler determines which method to call based on the arguments passed during the method invocation.

Worked Example:

Implementing Method Overloading **Problem:** Create an `Adder` class that overloads the `add` method to sum either two integers or three integers.
Solution:

```
class Adder {
    // Method with 2 parameters
    static int add(int a, int b) {
        return a + b;
    }

    // Method with 3 parameters
    static int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Adder.add(11, 11)); // Output:  22
        System.out.println(Adder.add(11, 11, 11)); // Output:  33
    }
}
```

Methodology:

Method Overriding Run-time polymorphism is achieved when a subclass provides a specific implementation for a method that is already defined in its superclass.

1. Define a method in the superclass.
2. In the subclass, define a method with the **exact same signature** (name, return type, and parameters).
3. Use the `@Override` annotation (optional but recommended) above the subclass method.
4. When an overridden method is called through a superclass reference, Java determines which version of the method to execute based on the actual object type at runtime (Dynamic Method Dispatch).

Worked Example:

Implementing Method Overriding **Problem:** Demonstrate dynamic method dispatch by overriding a `draw` method in `Circle` and `Rectangle` subclasses.
Solution:

```
class Shape {
    void draw() { System.out.println("Drawing a generic shape"); }
}

class Circle extends Shape {
    @Override
```

```
    void draw() { System.out.println("Drawing a Circle"); }
}

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

public class Main {
    public static void main(String[] args) {
        Shape s; // Superclass reference
        s = new Circle(); // Points to Circle object
        s.draw(); // Output:  Drawing a Circle

        s = new Rectangle(); // Points to Rectangle object
        s.draw(); // Output:  Drawing a Rectangle
    }
}
```

3.3 Abstraction

Abstraction is the process of hiding the implementation details and showing only the functionality to the user. In Java, abstraction is achieved using Abstract Classes (0% to 100% abstraction) and Interfaces (100% abstraction).

Methodology:

Abstract Classes and Methods

1. Declare a class using the `abstract` keyword. An abstract class cannot be instantiated directly using `new`.
2. Define **abstract methods** (methods without a body) using the `abstract` keyword.
3. Syntax: `abstract void methodName();`
4. Any subclass that extends the abstract class MUST override and implement all of its abstract methods, or the subclass must also be declared abstract.

Worked Example:

Implementing Abstract Class **Problem:** Create an abstract class `Bank` with an abstract method `getRateOfInterest()`. Implement this method in subclasses `SBI` and `HDFC`.
Solution:

```
abstract class Bank {
    abstract int getRateOfInterest();
}

class SBI extends Bank {
    int getRateOfInterest() { return 7; }
}

class HDFC extends Bank {
    int getRateOfInterest() { return 8; }
}
```

```
public class Main {
    public static void main(String[] args) {
        Bank b = new SBI();
        System.out.println("SBI Rate:  " + b.getRateOfInterest() + "%");
    }
}
```

Methodology:

Interfaces An interface is a blueprint of a class that contains static constants and abstract methods. It is used to achieve full abstraction and multiple inheritance.

1. Declare an interface using the `interface` keyword.
2. By default, all fields are `public static final` and all methods are `public abstract` (prior to Java 8).
3. Use the `implements` keyword in a class to inherit the interface.
4. A class can implement multiple interfaces separated by commas: `class A implements Int1, Int2`.

Worked Example:

Implementing Interfaces **Problem:** Create an interface `Printable` and implement it in a `Document` class. **Solution:**

```
interface Printable {
    void print(); // implicitly public and abstract
}

class Document implements Printable {
    public void print() {
        System.out.println("Document is printing...");
    }
}

public class Main {
    public static void main(String[] args) {
        Printable obj = new Document();
        obj.print();
    }
}
```

3.4 Package

A package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces. Packages prevent naming conflicts, provide access protection, and make searching/locating classes easier.

Methodology:

Creating and Using Packages

1. **Creating a package:** Add the `package` statement as the very first line of your Java source file.
2. Syntax: `package packagename;`
3. Compile the class using the `-d` flag to create the directory structure: `javac -d . MyClass.java`

4. **Using a package:** To use classes from another package, use the `import` keyword.
5. Import specific class: `import packagename.ClassName;`
6. Import all classes: `import packagename.*;`

Worked Example:

Package Implementation **Problem:** Create a class `Calculator` inside a package named `mathutils`. Then, write a `Main` class in a different file to import and use it.
Solution:
File 1: Calculator.java

```
package mathutils; // Defines the package

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

File 2: Main.java

```
import mathutils.Calculator; // Imports the class

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.multiply(5, 4);
        System.out.println("Result:  " + result);
    }
}
```

Compilation Steps (Command Line):

```
> javac -d .  Calculator.java
> javac Main.java
> java Main
```

Output: Result: 20

CHAPTER 4

Exception Handling and Multithreaded Programming

4.1 Basics of Exception and Errors

Methodology:

Try Catch Block Implementation An exception is an abnormal event that disrupts the normal flow of a program. Handling it ensures the program does not crash abruptly.

1. **Try Block:** Enclose the risky code that might generate an exception inside a `try { ... }` block.
2. **Catch Block:** Immediately follow the try block with a `catch (ExceptionClass e) { ... }` block to handle the specific exception.
3. **Execution Flow:** If an exception occurs, the remaining code in the try block is skipped, and control jumps to the matching catch block.

Worked Example:

Handling Divide By Zero Write a program to prevent a crash when dividing a number by zero.
Step-by-step Solution:

1. Place the division logic inside a `try` block.
2. Catch the `ArithmeticException` which is thrown during division by zero.

```
public class DivideByZero {
    public static void main(String[] args) {
        int a = 50;
        int b = 0;
        try {
            int data = a / b; // May throw ArithmeticException
            System.out.println("Result: " + data);
        } catch (ArithmeticException e) {
            System.out.println("Exception handled: Division by zero is not allowed.");
        }
        System.out.println("Rest of the code executes normally.");
    }
}
```

Output:

```
Exception handled: Division by zero is not allowed.
Rest of the code executes normally.
```

Methodology:

Multiple Catch Block Handling A single try block can be followed by multiple catch blocks to handle different types of exceptions differently.

1. Write the `try` block containing code that might throw multiple exception types.
2. Write multiple `catch` blocks, ordered from the most specific exception to the most general exception (e.g. `ArithmeticException` before `Exception`).
3. Only the first matching catch block will be executed.

Worked Example:

Array Index Out Of Bounds Write a program demonstrating multiple catch blocks handling arithmetic and array index errors.

Step-by-step Solution:

1. Define an array of a specific size.
2. Attempt to access an index outside the array bounds and divide by zero in the same block.
3. Catch `ArithmeticException` and `ArrayIndexOutOfBoundsException` separately.

```
public class MultipleCatchDemo {
    public static void main(String[] args) {
        try {
            int[] arr = new int[5];
            arr[5] = 30 / 2; // Throws ArrayIndexOutOfBoundsException
            int result = 30 / 0; // Throws ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic Exception occurs");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ArrayIndexOutOfBounds Exception occurs");
        } catch (Exception e) {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("Execution completed.");
    }
}
```

Output:

```
ArrayIndexOutOfBounds Exception occurs
Execution completed.
```

Methodology:

The Finally Block The `finally` block contains crucial code that must be executed whether an exception occurs or not.

1. Attach a `finally { ... }` block after the `try` or `catch` blocks.
2. Place cleanup code (like closing files or database connections) inside the `finally` block.
3. The `finally` block always executes right before control leaves the try-catch structure.

Worked Example:

Ensuring Execution With Finally Demonstrate the execution of a finally block when an exception is handled.
Step-by-step Solution:

1. Induce a `NullPointerException`.
2. Catch the exception.
3. Add a `finally` block to print a guaranteed message.

```
public class FinallyDemo {
    public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length()); // Throws NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Exception caught: String is null.");
        } finally {
            System.out.println("Finally block is always executed.");
        }
    }
}
```

Output:

```
Exception caught: String is null.
Finally block is always executed.
```

4.2 Basics of Multithreading

Methodology:

Extending Thread Class Multithreading allows concurrent execution of two or more parts of a program. One way to create a thread is by extending the `Thread` class.

1. Create a class that extends the `java.lang.Thread` class.
2. Override the `public void run()` method to define the code that constitutes the new thread.
3. Instantiate the class.
4. Call the `start()` method on the instance to begin execution of the thread.

Worked Example:

Thread Creation via Thread Class Create a multithreaded program by extending the Thread class that prints numbers from 1 to 3.
Step-by-step Solution:

1. Define `MyThread` extending `Thread`.
2. Override `run()` to print a loop.
3. In `main()`, create thread objects and invoke `start()`.

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 3; i++) {
```

```
            System.out.println(Thread.currentThread().getName() + " Value: " + i);
        }
    }
}

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

        t1.setName("Thread-A");
        t2.setName("Thread-B");

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

Output (Order may vary):

```
Thread-A Value: 1
Thread-A Value: 2
Thread-A Value: 3
Thread-B Value: 1
Thread-B Value: 2
Thread-B Value: 3
```

Methodology:

Implementing Runnable Interface The second method to create a thread is by implementing the `Runnable` interface. This is preferred when a class already extends another class.

1. Create a class that implements the `java.lang.Runnable` interface.
2. Provide implementation for the `public void run()` method.
3. Instantiate the class.
4. Pass the instance to a new `Thread` object constructor.
5. Call the `start()` method on the `Thread` object.

Worked Example:

Thread Creation via Runnable Implement the Runnable interface to create a thread that computes the square of a number.

Step-by-step Solution:

1. Define `SquareCalculator` implementing `Runnable`.
2. Put the square logic in the `run()` method.
3. Wrap the runnable instance in a `Thread` and start it.

```
class SquareCalculator implements Runnable {
    int number;

    SquareCalculator(int number) {
```

```
        this.number = number;
    }

    public void run() {
        int square = number * number;
        System.out.println("Square of " + number + " is: " + square);
    }
}

public class RunnableDemo {
    public static void main(String[] args) {
        SquareCalculator calc = new SquareCalculator(5);
        Thread t1 = new Thread(calc);
        t1.start();
    }
}
```

Output:

```
Square of 5 is: 25
```

Methodology:

Thread Synchronization Technique Synchronization prevents thread interference and memory consistency errors when multiple threads share resources.

1. Identify the shared resource or critical section of code accessed by multiple threads.
2. Declare the method or block with the `synchronized` keyword.
3. Only one thread can execute a synchronized method or block on the same object at a time. The other threads are blocked until the lock is released.

Worked Example:

Synchronized Method for Bank Account Demonstrate synchronization by preventing concurrent withdrawals from exceeding the account balance.

Step-by-step Solution:

1. Create a shared `BankAccount` class with a `synchronized` withdraw method.
2. Create a thread class that attempts to withdraw money.
3. Start multiple threads sharing the same account instance.

```
class BankAccount {
    int balance = 100;

    synchronized void withdraw(int amount, String name) {
        if (balance >= amount) {
            System.out.println(name + " is withdrawing...");
            balance = balance - amount;
            System.out.println(name + " completed withdrawal. Balance: " + balance);
        } else {
            System.out.println("Insufficient funds for " + name);
        }
    }
}
```

```
class UserThread extends Thread {
    BankAccount account;

    UserThread(BankAccount account, String name) {
        super(name);
        this.account = account;
    }

    public void run() {
        account.withdraw(80, Thread.currentThread().getName());
    }
}

public class SyncDemo {
    public static void main(String[] args) {
        BankAccount sharedAccount = new BankAccount();
        UserThread user1 = new UserThread(sharedAccount, "User-1");
        UserThread user2 = new UserThread(sharedAccount, "User-2");

        user1.start();
        user2.start();
    }
}
```

Output:

```
User-1 is withdrawing...
User-1 completed withdrawal. Balance: 20
Insufficient funds for User-2
```

CHAPTER 5

File Handling in Java

5.1 Introduction to Streams and IO Classes

In Java, file handling is performed using the Stream API provided in the `java.io` package. A stream is a logical connection to a file that allows reading data (input stream) or writing data (output stream). For computational problem-solving, streams are categorized into Byte Streams (handling 8-bit bytes) and Character Streams (handling 16-bit Unicode characters).

Methodology:

Configuring File IO Streams To successfully read from or write to a file in Java, follow this standard algorithm:

1. **Import the required package:** Include `import java.io.*;` at the beginning of your program.
2. **Choose the Stream Type:**
   - For plain text files, use Character Streams: `FileReader` and `FileWriter`.
   - For binary data, use Byte Streams: `FileInputStream` and `FileOutputStream`.
3. **Wrap with Buffered Streams:** To optimize disk operations, wrap the basic streams in `BufferedReader` or `BufferedWriter`.
4. **Handle Exceptions:** Enclose all I/O operations inside a `try-catch` block to catch `IOException`.
5. **Manage Resources:** Always close the streams using the `close()` method in a `finally` block or utilize Java's try-with-resources statement to prevent resource leaks.

Worked Example:

File Properties Extraction **Problem Statement:** Write a Java program to check if a specific file named `data.txt` exists. If it exists, determine its length in bytes and whether it is readable and writable.
Solution Process:

1. Create a `File` object referencing `data.txt`.
2. Use methods `exists()`, `length()`, `canRead()` and `canWrite()` to extract properties.

Java Implementation:

```
import java.io.File;

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

        if (file.exists()) {
```

```
            System.out.println("File exists!");
            System.out.println("Size in bytes: " + file.length());
            System.out.println("Readable: " + file.canRead());
            System.out.println("Writable: " + file.canWrite());
        } else {
            System.out.println("File does not exist.");
        }
    }
}
```

5.2 Reading and Writing Text and CSV Files

Text files and CSV (Comma Separated Values) files are essential formats for storing tabular data and plain text information. Processing these files requires reading data line by line, manipulating string arrays, and writing formatted text back to the disk.

5.2.1 Handling Text Files

Methodology:

Writing Data to a Text File

1. Declare a `FileWriter` object and pass the file path. Set the second boolean parameter to `true` if you want to append data instead of overwriting.
2. Pass the `FileWriter` object to a `BufferedWriter`.
3. Use the `write(String data)` method to insert text and `newLine()` to move to the next line.
4. Call the `close()` method on the `BufferedWriter` to flush the buffer and save the file.

Methodology:

Reading Data from a Text File

1. Declare a `FileReader` object with the target file path.
2. Pass the `FileReader` object to a `BufferedReader`.
3. Declare a `String` variable (e.g. `line`).
4. Use a `while` loop with the condition `(line = bufferedReader.readLine()) != null` to read the file sequentially.
5. Process the `line` string in the loop body.
6. Call `close()` on the `BufferedReader`.

Worked Example:

Line count in a Text File **Problem Statement:** Write a Java program that reads a text file named `input.txt` and counts the total number of lines present in the file.
Solution Process:

1. Initialize a line counter to 0.
2. Open `input.txt` using `BufferedReader`.
3. Read line by line in a loop. Increment the counter for every non-null line read.

4. Print the counter.

Java Implementation:

```
import java.io.*;

public class LineCounter {
    public static void main(String[] args) {
        String filePath = "input.txt";
        int count = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            while (br.readLine() != null) {
                count++;
            }
            System.out.println("Total number of lines: " + count);
        } catch (IOException e) {
            System.out.println("Error reading the file: " + e.getMessage());
        }
    }
}
```

5.2.2 Processing CSV Files

A CSV file stores data as plain text where each line represents a data record and each record consists of fields separated by commas.

Methodology:

Extracting and Computing Data from CSV

1. Open the CSV file using `BufferedReader`.
2. If the first line is a header (e.g. `ID,Name,Salary`), read it once using `readLine()` before entering the data processing loop to skip it.
3. Inside a `while` loop, read each subsequent line.
4. Use the `String.split(",")` method on the read line. This returns an array of `String` objects representing the columns.
5. Extract the specific index required from the array (e.g. `columns[2]` for the third column).
6. Convert the extracted string to a numerical type using wrapper classes like `Integer.parseInt()` or `Double.parseDouble()` to perform arithmetic computations.
7. Accumulate or process the computed values and close the stream.

Worked Example:

Calculating Average Score from CSV **Problem Statement:** A CSV file named `students.csv` contains records in the format `RollNo,Name,Score`. Write a Java program to read this file, skip the header row, compute the total score of all students, and calculate the class average.
Solution Process:

1. Initialize `totalScore = 0.0` and `studentCount = 0`.
2. Use `BufferedReader` to open `students.csv`.

3. Read the first line and discard it (header row).
4. Loop through the remaining lines. Split each line by the comma delimiter `","`.
5. Extract the score which is located at index 2 of the resulting array.
6. Parse the score string to a double using `Double.parseDouble()`.
7. Add the parsed score to `totalScore` and increment `studentCount`.
8. Compute `Average = totalScore / studentCount` and print the result.

Java Implementation:

```
import java.io.*;

public class CSVProcessor {
    public static void main(String[] args) {
        String file = "students.csv";
        double totalScore = 0.0;
        int studentCount = 0;

        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            // Read and skip the header
            String line = br.readLine();

            // Read data lines
            while ((line = br.readLine()) != null) {
                // Split by comma
                String[] columns = line.split(",");

                // Ensure the row has exactly 3 columns
                if (columns.length == 3) {
                    double score = Double.parseDouble(columns[2]);
                    totalScore += score;
                    studentCount++;
                }
            }

            if (studentCount > 0) {
                double average = totalScore / studentCount;
                System.out.println("Total Students: " + studentCount);
                System.out.println("Average Score: " + average);
            } else {
                System.out.println("No student records found.");
            }

        } catch (IOException e) {
            System.out.println("I/O Error: " + e.getMessage());
        } catch (NumberFormatException e) {
            System.out.println("Error parsing the score. Check CSV data format.");
        }
    }
}
```

Appendix: Key Formulae

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

Unit 1: DC Circuits and Network Theorems

- **Equivalent Resistance (Series)**:

$$R_{eq} = R_1 + R_2 + \cdots + R_n$$

- **Equivalent Resistance (Parallel)**:

$$R_{parallel} = \frac{R_1 \times R_2}{R_1 + R_2}$$

- **Kirchhoff's Current Law (KCL)**:

$$\sum I_{in} = \sum I_{out}$$

- **Ohm's Law**:

$$I = \frac{V}{R}$$

- **Delta to Star Transformation**:

$$R_1 = \frac{R_B \times R_C}{R_A + R_B + R_C}$$

- **Current Division Rule**:

$$I_1 = I_{total} \times \frac{R_2}{R_1 + R_2}$$

- **Thevenin's Theorem (Load Current)**:

$$I_L = \frac{V_{th}}{R_{th} + R_L}$$

- **Maximum Power Transfer Theorem**:

$$P_{max} = \frac{V_{th}^2}{4R_{th}}$$

- **Norton's Theorem (Norton Resistance)**:

$$R_N = R_{th}$$

Unit 4: Resonance and Coupled Circuits

- **Inductive Reactance**:

$$X_L = 2\pi f L$$

- **Capacitive Reactance**:

$$X_C = \frac{1}{2\pi f C}$$

- **Resonant Frequency (Series RLC)**:
$$f_r = \frac{1}{2\pi\sqrt{LC}}$$
- **Resonant Frequency (Parallel RLC with coil resistance)**:
$$f_r = \frac{1}{2\pi}\sqrt{\frac{1}{LC} - \frac{R^2}{L^2}}$$
- **Quality Factor (Q)**:
$$Q_0 = \frac{1}{R}\sqrt{\frac{L}{C}} = \frac{2\pi f_r L}{R}$$
- **Quality Factor in terms of Reactance**:
$$Q_C = \frac{X_C}{R_C}, \quad Q_L = \frac{X_L}{R_L}$$
- **Bandwidth**:
$$BW = f_2 - f_1 = \frac{f_r}{Q_0} = \frac{R}{2\pi L}$$
- **Dynamic Impedance (Parallel Resonance)**:
$$Z_{dyn} = \frac{L}{CR}$$
- **Resonant Current**:
$$I_r = \frac{V}{Z_{dyn}} \quad \text{(Parallel)} \quad \text{or} \quad I_r = \frac{V}{R} \quad \text{(Series)}$$
- **Equivalent Inductance (Series Aiding)**:
$$L_{eq(aid)} = L_1 + L_2 + 2M$$
- **Equivalent Inductance (Series Opposing)**:
$$L_{eq(opp)} = L_1 + L_2 - 2M$$
- **Mutual Inductance and Coefficient of Coupling**:
$$M = k\sqrt{L_1 L_2} \implies k = \frac{M}{\sqrt{L_1 L_2}}$$

Unit 5: Filters and Attenuators

- **Characteristic Impedance**:
$$Z_1 Z_2 = R_0^2 = K^2$$
- **Attenuation (Decibels)**:
$$D = 10\log_{10}\left(\frac{P_{in}}{P_{out}}\right) \text{ dB} = 20\log_{10}\left(\frac{V_{in}}{V_{out}}\right) \text{ dB}$$
- **Conversion between Nepers and Decibels**:
$$1 \text{ dB} = 0.1151 \text{ Np} \quad \text{and} \quad 1 \text{ Np} = 8.686 \text{ dB}$$

Unit 6: Transient Analysis and Laplace Transforms

- **General Transient Equation for DC Excitation**:

$$x(t) = x(\infty) + [x(0) - x(\infty)]e^{-t/\tau}$$

- **Time Constant**:

$$\tau = RC \quad \text{(for RC circuits)}$$

$$\tau = \frac{L}{R} \quad \text{(for RL circuits)}$$

- **Initial Conditions**:

$$v_C(0^+) = v_C(0^-)$$

$$i_L(0^+) = i_L(0^-)$$

- **Voltage-Current Relationships (Inductor and Capacitor)**:

$$v_L(t) = L\frac{di(t)}{dt}$$

$$i_C(t) = C\frac{dv_C(t)}{dt}$$

- **Transfer Function**:

$$H(s) = \frac{V_o(s)}{V_i(s)} = K\frac{(s - z_1)(s - z_2)\dots(s - z_m)}{(s - p_1)(s - p_2)\dots(s - p_n)}$$