G.S. Mandal's
Maharashtra Institute of Technology, Chh. Sambhajinagar
Department of Emerging Sciences and Technology

LAB MANUAL

CSD/AIDS 273 Lab Object Oriented Programming using Java

Maharashtra Institute of Technology, Chh. Sambhajinagar
NH-211, MIT Campus, Satara Village Road, Chh. Sambhajinagar- 431 010 (M.S.);
India. Phone: (0240) 2375222; Fax: (0240) 2376618, E-mail:
principalmitt@mit.asia Website: www.btech.mit.asia

Program Outcomes

PO1: Apply knowledge of mathematics, science, and engineering fundamentals to solve problems in Computer science and Engineering.

PO2: Identify, formulate and analyze complex problems.

PO3: Design system components or processes to meet the desired needs within realistic constraints for the public health and safety, cultural, societal and environmental considerations.

PO4: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data for valid conclusions.

PO5: Select and apply modern engineering tools to solve the complex engineering problem.

PO6: Apply knowledge to assess contemporary issues.

PO7: Understand the impact of engineering solutions in a global, economic, environmental, and societal context.

PO8: Apply ethical principles and commit to professional ethics and responsibilities
.

PO9: Work effectively as an individual, and as a member or leader in diverse teams and in multidisciplinary settings.

PO10: Communicate effectively in both verbal and written form.

PO11: Demonstrate knowledge and apply engineering and management principles to manage projects and in multi-disciplinary environment.

PO12: To engage in life-long learning to adapt to the technological changes.

Course Outcomes

Course: CSD/AIDS 273 Lab Object Oriented Programming using Java

After completing the course students will be able to,

CO1:	**Apply basic concepts Object Oriented Programming in Java for problem**
CO2:	**Apply exception handling and multithreading.**

LIST OF EXPERIMENTS

Course Code: CSD 273

Course Title: Lab Object Oriented Programming using Java

Sr. No.	Name of the Experiment
	Prerequisite: Basic understanding of Object-Oriented Programming (OOP) concepts like: • Classes and objects. • Inheritance, polymorphism, encapsulation, and abstraction. (These will be covered in detail while learning Java.) • Familiarity with text editors (like Notepad++) or Integrated Development Environments (IDEs) such as VS Code or IntelliJ IDEA. • Basic command-line usage for compiling and running programs.
	Objective: 1. To develop versatile, platform-independent applications ranging from desktop software to mobile apps and web applications. 2. Learn to design modular, reusable, and maintainable code using Java's object-oriented principles.
1	Write a program that inputs a word and a sentence. Find the given word in the sentence entered.
2	Write a program to store five student's marks along with roll number in an array. Display marks of a particular roll number.
3	Write a class Stack to push number, pop number and also to check stack empty or full. (Use OOPs concepts and constructor overloading to assign default size of stack to 5 or user can change the stack size while creating object)
4	Create a class, Bank Account, with fields account Number, account Holder Name, balance and interest Rate and a method deposit() that adds an amount to the balance. Create a subclass Savings Account that extends Bank Account and adds a field minimum Balance and a method withdraw() that subtracts an amount from the balance. Create a subclass Fixed Deposit Account that extends Savings Account and adds a field term and a method get Interest() that returns the interest earned on the account. Create an object of the Bank Account class and call the deposit() method. Create an object of the Savings Account class and call the deposit() and withdraw() methods. Create an object of the Fixed Deposit Account class and call the deposit().
5	Write a program to implement compile time polymorphism.

6	Write a program to implement run time polymorphism.
7	Write a program to demonstrate the use of interfaces and packages.
8	Write a program to handle Run Time Exceptions in Java
9	Write a program to throw Negative Number Exception (which is a user defined exception) if user enters a negative number as input.
10	Write a program to demonstrate multithreading in Java.

DOs and DON'Ts in Laboratory:

1. Make entry in the Log Book as soon as you enter the Laboratory.
2. All the students should sit according to their roll numbers starting from their left to right.
3. All the students are supposed to enter the terminal number in the log book.
4. Do not change the terminal on which you are working.
5. All the students are expected to get at least the algorithm of the program/concept to be implemented.
6. Strictly observe the instructions given by the teacher/Lab Instructor.
7. Do not disturb machine Hardware / Software Setup.

Instruction for Laboratory Teachers:

1. Submission related to whatever lab work has been completed should be done during the next lab session along with signing the index.
2. The promptness of submission should be encouraged by way of marking and evaluation patterns that will benefit the sincere students.
3. Continuous assessment in the prescribed format must be followed.

Installation of Java on System

1. Download Java

Visit the official Java website: Oracle Java Downloads.

Choose the latest version of the Java Development Kit (JDK) for your operating system (Windows, macOS, or Linux).

---

2. Install Java

a. For Windows

Run the Installer:

Double-click the downloaded .exe file.

Follow the Setup Wizard:

Accept the license agreement.

Choose the installation directory (or leave it as default).

Complete the Installation:

Click Next and wait for the installation to finish.

b. For macOS

Run the Installer:

Double-click the downloaded .dmg file.

Follow the prompts to install the JDK.

Verify Installation:

Open the terminal and type: java -version

If installed, it will show the Java version.

---

3. Set Environment Variables (For Windows)

Locate the JDK Path: Typically: C:\Program Files\Java\jdk<version>\bin

Edit Environment Variables:

Search for Environment Variables in the Start Menu.

Under "System Properties," click Environment Variables.

In "System Variables," find Path and click Edit.

Add the JDK bin directory path to the list.

Test Configuration:

Open the Command Prompt and type: javac -version

It should display the JDK version.

---

4. **Install any Text editor** like Notepad++, Sublime, VS Code or simply use Notepad

Practical 1

Aim: Write a program that inputs a word and a sentence. Find the given word in the sentence entered

Objective	CO	PO
To understand the usage of the String and StringBuffer classes and perform string manipulations like finding a word in a sentence, modifying strings, and exploring string methods.	CO1	PO1, PO2, PO12

Theory:

String Class

The String class in Java represents a sequence of characters. Strings are immutable, meaning once a String object is created, it cannot be modified. Operations that seem to modify a string actually create a new String object.

Key methods of the String class:

1. **length()**: Returns the length of the string.
    - Example: "hello".length() returns 5.
2. **toUpperCase()**: Converts all characters in the string to uppercase.
    - Example: "hello".toUpperCase() returns "HELLO".
3. **toLowerCase()**: Converts all characters in the string to lowercase.
    - Example: "HELLO".toLowerCase() returns "hello".
4. **contains(CharSequence)**: Checks if the string contains a specified sequence of characters.
    - Example: "hello world".contains("world") returns true.
5. **indexOf(String)**: Finds the first occurrence of a substring in the string.
    - Example: "hello world".indexOf("world") returns 6.
6. **substring(int, int)**: Extracts a substring from the string.
    - Example: "hello world".substring(0, 5) returns "hello".
7. **equals(Object)**: Compares two strings for equality.
    - Example: "hello".equals("hello") returns true.

StringBuffer Class

The StringBuffer class is used to create modifiable (mutable) strings. It is thread-safe, meaning it can be safely used in a multithreaded environment.

Key methods of the StringBuffer class:

1. **append(String)**: Adds a string to the end of the current StringBuffer.
    - Example: new StringBuffer("hello").append(" world") results in "hello world".
2. **insert(int, String)**: Inserts a string at a specified index.

    - Example: new StringBuffer("hello").insert(5, " world") results in "hello world".
3. **delete(int, int)**: Removes characters from a specified range.
    - Example: new StringBuffer("hello world").delete(5, 6) results in "helloworld".
4. **reverse()**: Reverses the characters in the StringBuffer.
    - Example: new StringBuffer("hello").reverse() results in "olleh".
5. **replace(int, int, String)**: Replaces characters in a specified range with a new string.
    - Example: new StringBuffer("hello").replace(0, 5, "hi") results in "hi".
6. **length()**: Returns the length of the StringBuffer.
    - Example: new StringBuffer("hello").length() returns 5

Program & Output:

Write a program that inputs a word and a sentence. Find the given word in the sentence entered.

Conclusion:

This program demonstrates how to use the String and StringBuffer classes for various string operations. It highlights the immutability of String objects and the mutability of StringBuffer objects.

Viva Quetions:

1. Explain the difference between String and StringBuffer in Java.
2. Why are strings in Java immutable? How does this affect performance?
3. What is the output of the indexOf method if the word is not found in the sentence?
4. Explain the significance of the reverse() method in StringBuffer. Provide an example where it can be practically useful.
5. How does the substring method differ from the replace method in the String class?

Practical 2

Aim: Write a program to store five student's marks along with roll number in an array. Display marks of a particular roll number.

Objective	CO	PO
To write a Java program that: 1. Stores the marks of five students along with their roll numbers using arrays. 2. Displays the marks of a particular student based on the roll number entered by the user.	CO1	PO1, PO2, PO12

Theory:

Introduction to Arrays in Java

An **array** in Java is a collection of similar data types stored in contiguous memory locations. It is a data structure that allows you to store multiple values in a single variable. The key points about arrays in Java are:

- Arrays are fixed in size once initialized.
- Each element in an array is accessed using its index, starting from 0.
- Arrays can store primitive data types (like int, float, etc.) as well as objects.

Syntax of Arrays

```
// Declaration and initialization
int[] arrayName = new int[size];
// Example
int[] marks = new int[5]; // Array to store 5 integers
```

Program Explanation

1. Create two arrays:
    - One for storing roll numbers (rollNumbers).
    - One for storing marks of the students (marks).
2. Use a loop to initialize the arrays with some values.
3. Prompt the user to input a roll number to retrieve the corresponding marks.
4. Use a loop to search for the roll number in the rollNumbers array and display the marks.
5. Handle cases where the roll number does not exist.

Step-by-Step Execution

1. **Initialize Arrays**:
    - rollNumbers: Contains roll numbers [101, 102, 103, 104, 105].
    - marks: Corresponding marks [85, 90, 78, 88, 95].

2. **User Input**:
    - User is prompted to input a roll number.
3. **Search in Arrays**:
    - The program searches the rollNumbers array to find the index of the input roll number.
    - If found, the corresponding marks are retrieved from the marks array.
4. **Output**:
    - If roll number exists, the marks are displayed.
    - If roll number does not exist, an appropriate message is displayed.

Expected Output

1. Enter roll number to get the marks: 103

Marks for roll number 103: 78

2. Enter roll number to get the marks: 106

Roll number 106 not found.

Program & Output:

Write a program to store five student's marks along with roll number in an array. Display marks of a particular roll number.

Conclusion

This program demonstrates the use of arrays to store and retrieve data efficiently in Java. By combining arrays with loops and conditional statements, we can manage and process collections of data in a structured manner.

Answer the following questions:

1. What is an array in Java, and how is it different from a regular variable?
2. How do you declare and initialize an array in Java? Provide examples.
3. What are the key properties of an array in Java?
4. Explain the difference between a one-dimensional and a two-dimensional array in Java.
5. How are arrays stored in memory in Java?

Practical 3

Aim: Write a class Stack to push number, pop number and to check stack empty or full. (Use OOPs concepts and constructor overloading to assign default size of stack to 5 or user can change the stack size while creating object)

Objective	CO	PO
1 To learn about constructor overloading and how it can be used to provide default and custom initialization. 2.To create a stack class with push, pop, isEmpty, and isFull methods. 3.To manage stack operations and handle cases where the stack is full or empty. 4.To reinforce the importance of encapsulation and abstraction in OOP.	CO1	PO1, PO2, PO12

Theory:

In object-oriented programming (OOP), a stack is a collection of elements that follows the Last In, First Out (LIFO) principle. This means that the most recently added element is the first one to be removed. To implement a stack using Java, we utilize the concepts of encapsulation, abstraction, and constructor overloading.

- **Object-Oriented Programming Concepts**
  1. **Encapsulation:** Encapsulation is the mechanism of bundling the data (variables) and methods (functions) that operate on the data into a single unit, typically a class. In this context, the stack class encapsulates the stack size, the stack itself (as an array), and the top index.
  2. **Abstraction:** Abstraction involves exposing only the necessary details and hiding the implementation details from the user. The stack class provides methods like push(), pop(), isEmpty(), and isFull(), abstracting the internal workings of these operations.
  3. **Constructor Overloading:** Constructor overloading allows a class to have more than one constructor, each with different parameter lists. This enables the creation of objects in different ways. In our stack implementation, we use constructor overloading to allow the stack to be initialized with a default size of 5 or a custom size specified by the user.

- **Stack Class Implementation**

The stack class includes the following components:

Private Members:

int size: The size of the stack.

int[] stack: An array to store the stack elements.

int top: The index of the top element in the stack.

Constructors:

public Stack(): This constructor initializes the stack with a default size of 5.

public Stack(int stackSize): This constructor initializes the stack with a user-specified size.

Public Methods:

public void push(int value): Adds an element to the stack if it is not full.

public int pop(): Removes and returns the top element from the stack if it is not empty.

public boolean isEmpty(): Checks if the stack is empty.

public boolean isFull(): Checks if the stack is full.

Expected Output:

Pushed 1 to the stack.

Pushed 2 to the stack.

Pushed 3 to the stack.

Pushed 4 to the stack.

Pushed 5 to the stack.

Stack is full. Cannot push 6.

Popped 5 from the stack.

Popped 4 from the stack.

Pushed 10 to the stack.

Pushed 20 to the stack.

Conclusion

This program demonstrates the use of object-oriented programming concepts to create a stack class. It shows how to use constructor overloading to provide default and custom initialization of stack size. The program also handles basic stack operations such as pushing and popping elements, and checking if the stack is empty or full.

Viva Questions

1. What is constructor overloading, and how is it used in this program?

2. Can you explain the concept of encapsulation and how it is applied in the Stack class?
3. Why is it important to check if the stack is full before pushing an element and if it is empty before popping an element?
4. How does the isEmpty and isFull functions help in managing the stack operations?
5. What are the advantages of using an array to implement the stack in this program?

Practical 4

Aim: Implementation of a Banking System Using Inheritance in Java

Objective	CO	PO
1. To understand and implement the concepts of inheritance, method overriding, and object-oriented programming in Java by creating a banking system involving a base class and derived classes. This will enhance students' understanding of code reusability and encapsulation.	CO1	PO1, PO2, PO12

Theory:

Object-Oriented Programming (OOP) promotes reusability and modular design. The "Bank Account" example demonstrates inheritance, where common attributes and methods are defined in a base class and extended in derived classes to implement specific features. The hierarchy includes:
A base class BankAccount for general account operations.
A subclass SavingsAccount that adds withdrawal functionality.
Another subclass FixedDepositAccount that calculates interest based on the deposit term.

Pseudocode:

1. **Define the BankAccount Class:**
    a. Declare the fields:
        i. accountNumber: stores the account number.
        ii. accountHolderName: stores the name of the account holder.
        iii. balance: stores the current balance of the account.
        iv. interestRate: stores the interest rate applicable to the account.
    b. Implement the setDetails(accountNumber, accountHolderName, balance, interestRate) method to set the field values.
    c. Implement the deposit(amount) method:
        i. Add the given amount to the balance.
        ii. Print a message showing the deposited amount and the updated balance.
2. **Define the SavingsAccount Class (Extends BankAccount):**
    a. Declare an additional field:
        i. minimumBalance: stores the minimum balance that must be maintained.
    b. Implement the setMinimumBalance(minimumBalance) method to set the field value.
    c. Implement the withdraw(amount) method:

i. Check if withdrawing the amount will result in a balance below minimumBalance.
ii. If valid, subtract the amount from balance and print the new balance.
iii. Otherwise, print an error message indicating insufficient balance.

3. **Define the FixedDepositAccount Class (Extends SavingsAccount):**
    a. Declare an additional field:
        i. term: stores the duration of the fixed deposit in months.
    b. Implement the setTerm(term) method to set the field value.
    c. Implement the getInterest() method:
        i. Calculate the interest using the formula: (balance * interestRate * term) / 1200.
        ii. Print the interest earned and return the calculated value.
4. **Implement the Main Class:**
    a. Create an object of the BankAccount class:
        i. Call setDetails() and deposit() methods to test the functionality.
    b. Create an object of the SavingsAccount class:
        i. Call setDetails(), setMinimumBalance(), deposit(), and withdraw() methods to test the functionality.
    c. Create an object of the FixedDepositAccount class:
        i. Call setDetails(), setMinimumBalance(), setTerm(), deposit(), and getInterest() methods to test the functionality.

Expected Output:

Deposited: 5000.0, New Balance: 15000.0
Deposited: 3000.0, New Balance: 23000.0
Withdrawn: 1000.0, Remaining Balance: 22000.0
Deposited: 10000.0, New Balance: 60000.0
Interest Earned: 3000.0

Conclusion:

Gain practical experience in implementing inheritance, method overriding, and encapsulation in Java. The hierarchical structure of classes demonstrates the reusability of code, while the practical application of methods like deposit(), withdraw(), and getInterest() highlights real-world banking operations. This exercise reinforces the importance of structured programming and the power of OOP in solving complex problems efficiently.

Viva Questions

1. What is inheritance, and why is it used?
2. How does method overriding differ from method overloading?
3. Why do we use setters instead of constructors in this implementation?
4. How can encapsulation be achieved in Java?
5. What are the advantages of using a superclass-subclass relationship?

Practical 5

Aim: Write a program to implement compile time polymorphism.

Objective	CO	PO
• Understand the concept of compile-time polymorphism in Java. • Implement method overloading to demonstrate compile-time polymorphism. • Differentiate between compile-time and runtime polymorphism.	CO1	PO1, PO2, PO12

Theory:

Compile-time polymorphism, also known as static polymorphism, occurs when the method to be called is determined at compile time. This is achieved through method overloading and operator overloading. Method overloading allows multiple methods with the same name but different parameter lists to coexist in the same scope. The compiler differentiates these methods based on the number and types of arguments passed

Pseudocode

1. Create a class Calculator.
2. Define multiple add methods with the same name but different parameter lists:
    a. One method takes two integers.
    b. Another method takes three integers.
    c. A third method takes two floating-point numbers.
3. In the main method:
    a. Create an instance of the Calculator class.
    b. Call all overloaded methods with appropriate arguments.
4. Print the results.

Expected Output:

Sum of two integers: 15

Sum of three integers: 6

Sum of two doubles: 6.0

Conclusion:

Compile-time polymorphism is implemented successfully using method overloading. The method selection is based on the parameters passed during the method call, which is resolved at compile

time. This showcases the flexibility and power of Java's polymorphic behavior in improving code readability and reusability.

Viva Questions:

1. What is polymorphism in Java?
2. What is the difference between compile-time and runtime polymorphism?
3. How does method overloading differ from method overriding?
4. Why is method overloading considered compile-time polymorphism?
5. Can we overload methods by changing only the return type?

Practical 6

Aim: Program to Implement Runtime Polymorphism in Java

Objective	CO	PO
• Understand the concept of runtime polymorphism in Java. • Implement **method overriding** to demonstrate runtime polymorphism. • Illustrate dynamic method dispatch where the method to be executed is determined at runtime.	CO1	PO1, PO2, PO12

Theory

Polymorphism allows objects to take multiple forms. Runtime polymorphism is achieved through **method overriding**. It occurs when a subclass provides a specific implementation for a method already defined in its superclass.

Key points about runtime polymorphism:

1. The method in the child class must have the **same signature** as the method in the parent class.
2. A reference variable of the parent class can refer to an object of the child class (dynamic method dispatch).
3. The method to be executed is determined based on the actual object type, not the reference type.

Pseudocode

1. Create a parent class Animal with a method sound().
2. Create two child classes Dog and Cat that extend Animal and override the sound() method.
3. In the main method:
    a. Create references of type Animal and assign objects of Dog and Cat.
    b. Call the sound() method using these references.
4. Observe the behavior based on the actual object type.

Expected Output:

Dog barks

Cat meows

Conclusion:

Runtime polymorphism is successfully demonstrated using method overriding. The method executed is based on the actual object type, even though the reference is of the parent class. This highlights the flexibility and dynamic behavior of Java's polymorphism.

Viva Questions:

1. What is runtime polymorphism?
2. What is the difference between method overloading and method overriding?
3. How does the JVM determine which method to execute in runtime polymorphism?
4. Why do we need the @Override annotation in method overriding?
5. Can we override a static method in Java? Why or why not?
6. What is the role of dynamic method dispatch in runtime polymorphism?

Practical 7

Aim: Write a program to demonstrate the use of interfaces and packages.

Objective	CO	PO
1. To understand and implement the use of interfaces and packages in Java, demonstrating modularity, abstraction, and code reusability.	CO1	PO1, PO2, PO12

Theory:

1. Interface

An interface in Java is a blueprint for a class. It is used to specify a set of methods that a class must implement. An interface ensures a contract that the implementing class adheres to. It is particularly useful for achieving abstraction and multiple inheritance in Java.

Key Features of an Interface:

Abstract Methods: All methods in an interface are implicitly public and abstract.

Constants: Variables in an interface are implicitly public, static, and final.

Multiple Inheritance: A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources.

Default Methods: Introduced in Java 8, these are methods with a default implementation.

Static Methods: Introduced in Java 8, these methods can be called on the interface itself.

Syntax:

```
public interface InterfaceName {
    void method1(); // Abstract method
    int method2(int param);
}
```

2. Package

A package in Java is a namespace for organizing classes and interfaces. It helps to group related classes together and provides controlled access. Packages prevent naming conflicts and enhance code modularity.

Types of Packages:

Built-in Packages: Provided by Java (e.g., java.util, java.io, java.net).

User-defined Packages: Created by the user to organize their own classes.

Syntax for Creating a Package:

```
package packageName;
public class ClassName {
    // Class code here
```

}

Accessing Packages:

Importing a Specific Class:

import packageName.ClassName;

Importing All Classes in a Package:

import packageName.*;

Pseudo code:

Step 1: Create a Package

Create a new folder for your project.

Define a package in a file. For example, mypackage.

Step 2: Define the Interface

Inside the package, create an interface with some abstract methods.

Step 3: Implement the Interface in a Class

Create a class in the same or a different package.

Implement the interface in this class.

Step 4: Create a Driver Program

Write a main class in another file.

Use the implemented class to demonstrate the functionality of the interface.

Steps to Execute

1. **Compile** the files:
    a. Navigate to the directory containing mypackage.
    b. Use the javac command to compile all files:

        javac mypackage/*.java MainProgram.java

2. **Run** the program:
    a. Use the java command to execute the main class: java MainProgram

Expected Output:

Message: Hello, World!

Sum: 30

Explanation of the Output:

1. **Message:**
    a. The method displayMessage() from the interface is invoked. The message "Hello, World!" is passed as an argument, and it gets printed with the prefix "Message: ".

2. **Sum:**
    a. The method addNumbers() from the interface is invoked with arguments 10 and 20. It returns their sum, which is 30, and it gets printed with the prefix "Sum: ".

Conclusion:

- **Interfaces** provide a contract that classes can implement, promoting abstraction and flexibility in code.
- **Packages** allow better organization, reusability, and namespace management in large Java applications.
- By combining interfaces and packages, modular and maintainable software can be developed efficiently.

Viva Questions

1. What is the purpose of an interface in Java?
2. How do packages help in organizing Java programs?
3. Can a class implement multiple interfaces? Justify your answer.
4. Explain the difference between implements and extends in Java.
5. How do you compile and run Java programs using packages?

Practical 8

Aim: Program to Handle Runtime Exceptions in Java

Objective	CO	PO
1. Understand the concept of runtime exceptions in Java. 2. Learn how to use try-catch blocks to handle runtime exceptions. 3. Demonstrate error handling to ensure smooth program execution even when exceptions occur.	CO2	PO1, PO2, PO3, PO12

Theory:

- In Java, **runtime exceptions** are a subclass of the Exception class under the RuntimeException hierarchy. These exceptions are unchecked, meaning they are not checked at compile time. Common runtime exceptions include:
- ArithmeticException (e.g., division by zero)
- NullPointerException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- **Handling Runtime Exceptions**:
- Use try-catch blocks to catch and handle exceptions gracefully.
- Provide meaningful messages or recovery mechanisms for the exceptions

Pseudo Code:

- 1. Initialize an array with predefined values.
- Inside a try block:
- Attempt to access an invalid index in the array.
- Perform an arithmetic operation that causes an exception (e.g., division by zero).
- Use multiple catch blocks to handle specific exceptions.
- Provide a default catch block for any unanticipated exceptions.
- Print appropriate error messages for each exception.

Expected Output:

Error: Attempted to access an invalid index in the array.

Execution completed, whether an exception occurred or not.

If the first exception is fixed, the second exception will occur:

Error: Division by zero is not allowed.
Execution completed, whether an exception occurred or not.

Conclusion:

The program demonstrates handling runtime exceptions using try-catch blocks. By catching specific exceptions like ArrayIndexOutOfBoundsException and ArithmeticException, the program prevents crashes and continues execution smoothly. The finally block ensures that cleanup or final statements are executed regardless of whether an exception occurred.

Viva Questions:

1. What are runtime exceptions in Java? How are they different from checked exceptions?
2. Can we handle multiple exceptions in the same try block?
3. What is the purpose of the finally block?
4. What happens if an exception occurs but no catch block is defined for it?
5. Can a try block exist without a catch block? Explain.
6. What is the difference between Throwable, Error, and Exception in Java?

Practical 9

Aim: Write a program to throw NegativeNumberException (which is a user defined exception) if user enters a negative number as input.

Objective	CO	PO
1. Understand how to create and use custom (user-defined) exceptions in Java. 2. Learn how to throw and handle custom exceptions in specific situations. 3. Demonstrate exception handling for invalid inputs, such as negative numbers.	CO2	PO1, PO2, PO3, PO12

Theory:

In Java, **user-defined exceptions** are custom exceptions created by extending the Exception or RuntimeException class. These exceptions allow developers to handle specific application scenarios that are not covered by Java's built-in exceptions.

Key Steps:

1. Create a class extending Exception (checked exception) or RuntimeException (unchecked exception).
2. Add a constructor to pass a custom error message.
3. Use the throw keyword to raise the exception in the program when the specified condition is met.
4. Catch and handle the exception using try-catch blocks.

Pseudo Code:

1. Create a custom exception class NegativeNumberException extending Exception.
2. In the class constructor, allow a custom error message to be passed.
3. Create a main class with a method to read an integer input from the user.
4. If the input is negative, throw a NegativeNumberException.
5. Catch the exception and print an appropriate error message.
6. Allow the program to continue after handling the exception.

Expected Output:

Case 1: Positive number input

Enter a number: 5

The entered number is valid: 5

Program execution completed.

Case 2: Negative number input

Enter a number: -3
Exception caught: Negative numbers are not allowed: -3
Program execution completed.

Conclusion:

The program demonstrates how to create and handle a user-defined exception (NegativeNumberException). By extending the Exception class, we can define specific error-handling logic for invalid inputs like negative numbers. This approach improves code clarity and robustness by allowing developers to create application-specific exceptions.

Viva Questions

1. What is a user-defined exception in Java?
2. How do you create a custom exception?
3. Why do we need user-defined exceptions when Java provides built-in exceptions?
4. What is the difference between throw and throws?
5. Can we use RuntimeException instead of Exception for custom exceptions? Why or why not?
6. What happens if we don't catch a checked exception?

Practical 10

Aim: Write a program to demonstrate multithreading in Java.

Objective	CO	PO
1. Understand the concept of multithreading in Java. 2. Learn how to create and manage threads using the Thread class and the Runnable interface. 3. Demonstrate simultaneous execution of multiple threads to improve program efficiency.	CO2	PO1, PO2, PO3, PO12

Theory:

1. **Multithreading** in Java allows multiple threads to execute simultaneously within a single program. Threads are lightweight sub-processes that run independently, sharing the same memory space.
2. Key Points:
    a. A thread in Java can be created in two ways:
    b. By extending the Thread class.
    c. By implementing the Runnable interface.
    d. The start() method is used to initiate a thread, which internally calls the run() method.
    e. Threads can run concurrently, and the order of execution is managed by the Java Virtual Machine (JVM).

Pseudo Code:

1. Create a Thread by Extending the Thread Class:
2. Define a class MyThread that extends Thread.
3. Override the run() method.
4. In the run() method:
    a. Write a loop that performs some task and prints thread-specific messages.
    b. Use Thread.sleep() to pause the thread for a short duration.
5. Create a Thread by Implementing the Runnable Interface:
6. Define a class MyRunnable that implements the Runnable interface.
7. Implement the run() method.
8. In the run() method:
    a. Write a similar loop to perform a task.
    b. Use Thread.sleep() for delays.

9. Main Method:
10. Create an instance of MyThread.
    a. Set a name for the thread (optional).
    b. Start the thread using start() method.
11. Create an instance of MyRunnable and wrap it in a Thread object.
    a. Set a name for the thread.
    b. Start the thread using start().
12. In the main thread:
    a. Write another loop to perform a task and print "Main Thread" messages.
    b. Use Thread.sleep() for delays.

Expected Output:

Thread-1 - Count: 1
Thread-2 - Count: 1
Main Thread - Count: 1
Thread-1 - Count: 2
Thread-2 - Count: 2
Main Thread - Count: 2
Thread-1 - Count: 3
Thread-2 - Count: 3
Main Thread - Count: 3
Thread-1 - Count: 4
Thread-2 - Count: 4
Main Thread - Count: 4
Thread-1 - Count: 5
Thread-2 - Count: 5
Main Thread - Count: 5

Conclusion:

The program demonstrates multithreading in Java by creating threads using both the Thread class and the Runnable interface. Threads execute concurrently, and their execution order is managed by the JVM. This approach improves program efficiency, especially in scenarios where multiple tasks can run simultaneously.

Viva Questions

1. What is multithreading in Java?
2. How do you create a thread in Java?
3. What is the difference between extending the Thread class and implementing the Runnable interface?
4. What is the role of the start() and run() methods in threading?
5. How does the sleep() method affect thread execution?
6. What happens if a thread is interrupted while it is sleeping?
7. Explain the advantages of multithreading in Java.
8. How does the JVM decide the execution order of threads?