**Q1. Basic Java Program with Input/Output**

**Aim:**  
To write a Java program that accepts student details (name, roll number, and marks) using console input and displays the details in a formatted output.

**Theory:**  
Input and output are the foundation of user interaction in Java.

* **Input:** The Scanner class (from java.util package) is commonly used for reading input from the keyboard.
* **Output:** Java uses System.out.print() and System.out.println() for displaying data.
* **Formatted Output:** The printf() method provides better control over the appearance of numbers and text.  
  This program demonstrates how Java reads and displays data of different types and prints them in an organized manner.

Code:

import java.util.Scanner;

class StudentDetails {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter Name: ");

String name = sc.nextLine();

System.out.print("Enter Roll No: ");

int rollNo = sc.nextInt();

System.out.print("Enter Marks: ");

float marks = sc.nextFloat();

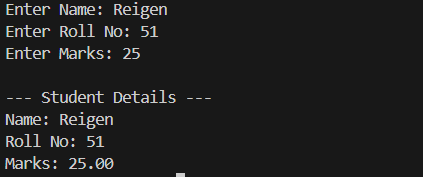
System.out.println("\n--- Student Details ---");

System.out.printf("Name: %s\nRoll No: %d\nMarks: %.2f\n", name, rollNo, marks);

}

}

Output:



Q2. **Decision-Making System for Library**

**Aim:**  
To develop a Java program for managing library book categories using conditional statements to check book availability.

**Theory:**  
Decision-making statements allow a program to choose actions based on conditions.

* The **if-else** statement executes a block of code only if a certain condition is true.
* The **switch** statement is used for selecting one option from multiple choices.  
  This concept is useful in applications like library systems, where the system must decide which message or data to display based on user input.

Code:

import java.util.Scanner;

class LibrarySystem {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter category (Fiction / Science / History): ");

String category = sc.nextLine();

if (category.equalsIgnoreCase("Fiction"))

System.out.println("Fiction books available: 10");

else if (category.equalsIgnoreCase("Science"))

System.out.println("Science books available: 5");

else if (category.equalsIgnoreCase("History"))

System.out.println("History books available: 8");

else

System.out.println("Invalid category.");

}

}

Output:

Q3. **Flight Reservation System using Loops**

**Aim:**  
To design a reservation system using loops that allows users to book or cancel flight tickets multiple times until they exit.

**Theory:**  
Loops enable repetitive execution of code blocks.

* The **while loop** runs until a condition becomes false.
* The **for loop** is used for counting iterations.  
  In this experiment, loops allow continuous user interaction, so the user can repeatedly book or cancel tickets without restarting the program. This models a real-world interactive system.

Code:

import java.util.Scanner;

class FlightReservation {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int seats = 5;

while (true) {

System.out.println("1. Book Ticket 2. Cancel Ticket 3. Exit");

int choice = sc.nextInt();

if (choice == 1 && seats > 0) {

seats--;

System.out.println("Ticket booked! Remaining seats: " + seats);

} else if (choice == 2) {

seats++;

System.out.println("Ticket cancelled. Seats available: " + seats);

} else if (choice == 3)

break;

else

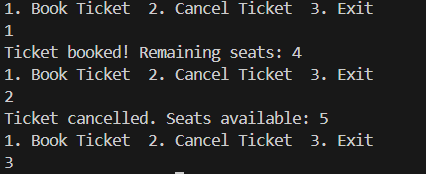
System.out.println("Invalid or no seats available.");

}

}

}

Output:



Q4. **Array-Based Passenger Booking System**

**Aim:**  
To implement a 2D array that stores and displays booking details such as passenger name, seat number, and class.

**Theory:**  
Arrays are used to store multiple values of the same type in contiguous memory locations.  
A **2D array** can be imagined as a table (rows and columns) that holds structured data like a passenger list.  
Using loops with arrays allows systematic data storage and retrieval, which is vital in booking and reservation systems.

Code:

import java.util.Scanner;

class PassengerBooking {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String[][] bookings = new String[3][3]; // name, seat no, class

for (int i = 0; i < 3; i++) {

System.out.print("Enter Name: ");

bookings[i][0] = sc.next();

System.out.print("Enter Seat No: ");

bookings[i][1] = sc.next();

System.out.print("Enter Class: ");

bookings[i][2] = sc.next();

}

System.out.println("\n--- Booking Details ---");

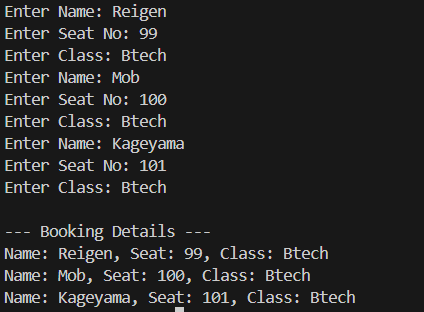
for (String[] b : bookings)

System.out.println("Name: " + b[0] + ", Seat: " + b[1] + ", Class: " + b[2]);

}

}

Output:



Q5. **Banking Transaction Tracker using Strings**

**Aim:**  
To develop a program that records banking transactions using String and StringBuilder, allowing the user to append and view the transaction history.

**Theory:**  
Strings in Java are immutable (cannot be changed once created), so for continuous text modification, **StringBuilder** is preferred.  
It allows operations like append(), insert(), and delete() efficiently.  
This experiment demonstrates dynamic string manipulation to maintain a transaction log — an important concept in applications like digital banking systems.

Code:

import java.util.Scanner;

class BankingTransaction {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

StringBuilder history = new StringBuilder();

while (true) {

System.out.println("1. Deposit 2. Withdraw 3. Show History 4. Exit");

int ch = sc.nextInt();

if (ch == 1) {

System.out.print("Amount: ");

int amt = sc.nextInt();

history.append("Deposited: ").append(amt).append("\n");

} else if (ch == 2) {

System.out.print("Amount: ");

int amt = sc.nextInt();

history.append("Withdrawn: ").append(amt).append("\n");

} else if (ch == 3)

System.out.println(history.toString());

else

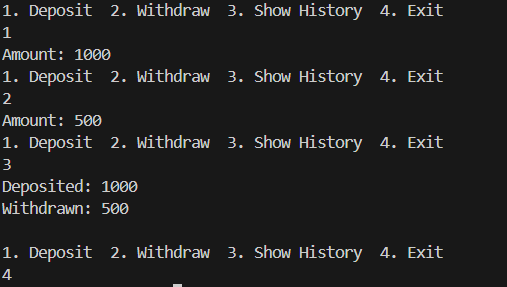
break;

}

}

}

Output:



Q6. **File I/O for Customer Accounts**

**Aim:**  
To read and write customer account details into a text file using file handling in Java.

**Theory:**  
File handling enables data to be permanently stored and retrieved.  
Java provides classes such as:

* **FileWriter** and **BufferedWriter** for writing data into files.
* **FileReader** and **BufferedReader** for reading data from files.  
  This concept is crucial for developing systems that store data persistently, like banking systems, reservation apps, and employee databases.

Code:

import java.io.\*;

import java.util.Scanner;

class CustomerFileIO {

public static void main(String[] args) throws Exception {

Scanner sc = new Scanner(System.in);

System.out.print("Enter Name: ");

String name = sc.nextLine();

System.out.print("Enter Balance: ");

double bal = sc.nextDouble();

FileWriter fw = new FileWriter("customer.txt");

fw.write("Name: " + name + "\nBalance: " + bal);

fw.close();

BufferedReader br = new BufferedReader(new FileReader("customer.txt"));

String line;

System.out.println("\nFile Content:");

while ((line = br.readLine()) != null)

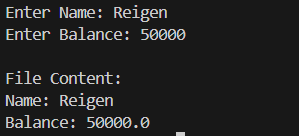
System.out.println(line);

br.close();

}

}

Output:



Q7. **Shopping Cart with Price Formatting**

**Aim:**  
To build a shopping cart system that calculates total cost, GST, and displays final price using proper numeric formatting.

**Theory:**  
Price calculation often involves floating-point arithmetic and requires formatted output.  
The **DecimalFormat** class (from java.text package) helps format numerical values (e.g., two decimal points).  
This experiment simulates an e-commerce cart, where users input product prices, and the system automatically calculates total price and applicable tax (GST).

Code:

import java.text.DecimalFormat;

import java.util.Scanner;

class ShoppingCart {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

DecimalFormat df = new DecimalFormat("#.00");

double total = 0;

for (int i = 1; i <= 3; i++) {

System.out.print("Enter price of item " + i + ": ");

total += sc.nextDouble();

}

double gst = total \* 0.18;

double finalPrice = total + gst;

System.out.println("\nTotal: ₹" + df.format(total));

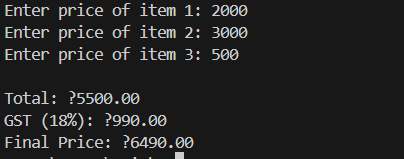
System.out.println("GST (18%): ₹" + df.format(gst));

System.out.println("Final Price: ₹" + df.format(finalPrice));

}

}

Output:



Q8. **Date and Pattern Matching for Orders**

**Aim:**  
To write a program that extracts specific details like Order ID from text using Regular Expressions (Regex) and displays the current date.

**Theory:**  
Regular expressions (Regex) are patterns used for text searching and validation.  
In Java:

* **Pattern** class defines the regular expression.
* **Matcher** class searches for patterns within strings.  
  Combined with **date handling** using the java.time package, this experiment helps in extracting structured information from unstructured data, useful in invoice or order management systems.

Code:

import java.util.regex.\*;

import java.time.\*;

class OrderPattern {

public static void main(String[] args) {

String order = "Customer: Riya, OrderID: 12345, Date: 2025-10-13";

Pattern p = Pattern.compile("OrderID: (\\d+)");

Matcher m = p.matcher(order);

if (m.find()) System.out.println("Order ID: " + m.group(1));

LocalDate today = LocalDate.now();

System.out.println("Current Date: " + today);

}

}

Output:



Q9. **GUI Application with Swing for Personal Assistant**

**Aim:**  
To create a simple graphical user interface (GUI) using Java Swing that allows users to enter and save tasks.

**Theory:**  
Swing is a part of Java’s javax.swing package and is used to build window-based applications.  
Key components include:

* **JFrame**: The main window.
* **JLabel**: Displays text.
* **JTextField**: Allows user input.
* **JButton**: Performs an action when clicked.  
  This experiment introduces event-driven programming, where the program reacts to user events such as button clicks.

Code:

import javax.swing.\*;

class PersonalAssistant {

public static void main(String[] args) {

JFrame frame = new JFrame("Personal Assistant");

JLabel label = new JLabel("Enter Task:");

JTextField field = new JTextField(15);

JButton btn = new JButton("Save");

JPanel panel = new JPanel();

panel.add(label);

panel.add(field);

panel.add(btn);

frame.add(panel);

frame.setSize(300, 150);

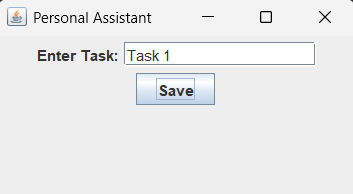
frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

}

Output:



Q10. **Event-Based Exception Handling in Personal Assistant App**

**Aim:**  
To develop a GUI application that demonstrates event handling and exception handling for invalid inputs.

**Theory:**

* **Event Handling:** Java uses interfaces like ActionListener to perform actions when an event (like a button click) occurs.
* **Exception Handling:** The try-catch mechanism handles runtime errors gracefully without crashing the program.  
  This experiment combines user interface and error handling, teaching how to make applications both interactive and robust.

Code:

import javax.swing.\*;

import java.awt.event.\*;

class AssistantException {

public static void main(String[] args) {

JFrame f = new JFrame("Event Handler");

JTextField field = new JTextField(10);

JButton btn = new JButton("Check");

btn.addActionListener(e -> {

try {

int num = Integer.parseInt(field.getText());

JOptionPane.showMessageDialog(f, "Number: " + num);

} catch (NumberFormatException ex) {

JOptionPane.showMessageDialog(f, "Invalid input!");

}

});

JPanel p = new JPanel();

p.add(field);

p.add(btn);

f.add(p);

f.setSize(250, 120);

f.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

f.setVisible(true);

}

}

Output:

