SOFTWARE VERIFICATION AND VALIDATION

23CS2239F

STUDENT ID: 2300032184 ACADEMIC YEAR: 2024-25

STUDENT NAME: SHAIK MOHAMMAD PARVEZ

Table of Contents

A.Y. 2024-25 LAB CONTINUOUS EVALUATION

S.No	Date	Experiment Name	Pre-Lab (10M)	In-Lab (25M)			Post-Lab (10M)	Viva Voce (5M)	Total (50M)	Faculty Signature
				Program/ Procedure (5M)	Data and Results (10M)	Analysis & Inference (10M)				
1.			-NA-							
2.										
3.										
4.										
5.										
6.										
7.										
8.										
9.										
10.										
11.										
12										

13.										

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment Title:

Aim/Objective:

This Section must contain the aim or objectives of the Laboratory session.

Description:

This Section must contain detailed information pertaining to the Aim/Objective of the Laboratory Session **Pre-Requisites:**

This Section contains the list of Software/Tools or required knowledge (Glossary) to complete the task under theLaboratory Session

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Lab Session01: How to Design Test Case and User-stories

Date of the Session: _____ / / **Time of the Session:** _____ **to** _____

Title of the Program: Generate Test Cases & User-stories.

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. Define the term Test Case? What is a good test case in software testing?

A: A test case is a set of steps and conditions used to check if a software feature works as expected. A good test case clearly describes the input, expected output, and covers both normal and edge scenarios.

2. Why test cases are so important?

A: Test cases are important because they help ensure the software works correctly and meets user requirements.
They also make it easier to find bugs early, maintain quality, and repeat tests when changes are made

3. List out the different types of test cases?

A: Here are some common types of test cases in software testing:

1. **Functional Test Cases** – Check if the software functions as expected.
2. **Negative Test Cases** – Test how the system handles invalid input.
3. **Boundary Test Cases** – Focus on the edges of input ranges (like min/max values).
4. **Integration Test Cases** – Test how different modules work together.

4. Define Test procedure?

A: A **test procedure** is a detailed set of instructions that describes how to run one or more test cases.

It includes the steps to follow, input data to use, expected results, and how to set up and clean up the test environment.

5. List out and explain the rules of writing test cases.

A: Here are some important **rules for writing good test cases**:

1. **Clear and Simple**: ○ Write in simple language so anyone can understand and follow the steps.
2. **Unique Test Case ID**: ○ Every test case should have a unique ID for easy tracking.
3. **Specify Preconditions**:
    - ○ Mention any setup or conditions needed before executing the test

6. Define user story? Identify the primary users of the system?

A: A user story is a short, simple description of a feature from the perspective of an end user.

It usually follows this format:

"As a [user], I want to [do something] so that [goal]."

The primary users of a system are the main people who interact with or benefit from it.

For example, in a Library Management System, the primary users are:

- Librarians – to manage books and members
- Members/Students – to borrow and return books
- Admins – to oversee and configure the system

7. write the roles do these users play in their organization or daily life?

A: Librarians manage books, register members, and handle issuing and returning of books.

Members or Students search, borrow, and return books for study or reading purposes.

Admins oversee the entire system, manage users, and set rules or policies.

Each user helps ensure smooth operation of the library system in their daily roles.

8. What are the users' main goals when using the system? A: The users' main goals when using a Library Management System are:
   1. Librarian – To efficiently manage books and members, and track issued/returned books.
   2. Member/Student – To easily search, borrow, and return books on time.
   3. Admin – To ensure the system runs smoothly with proper user access and rules

9. What key features do users need to achieve their goals? A: Here are the key features users need to achieve their goals in a Library Management System:
   1. Book Management – Add, update, delete, and search books.
   2. Member Registration – Add and manage member details.
   3. Issue/Return System – Handle book lending and returns with date tracking.
   4. Search Function – Allow members to search books by title, author, or ID.
   5. User Access Control – Different access for librarians, members, and admins.

In Lab

1. Design, develop, code, and run the program in any suitable language to implement the addition of two numbers. Analyse it from the perspective of equivalence class value testing, derive different test cases, execute these test cases, and discuss the test results.

Procedure/Program:

AdditionTest.java:

```
package org.example; import org.testng.annotations.Test;
import static junit.framework.Assert.assertEquals;

public class AdditionTest{
   @Test
   void testPositiveNumbers(){
      assertEquals(30,Addition.add(10,20));
   }
   @Test
```

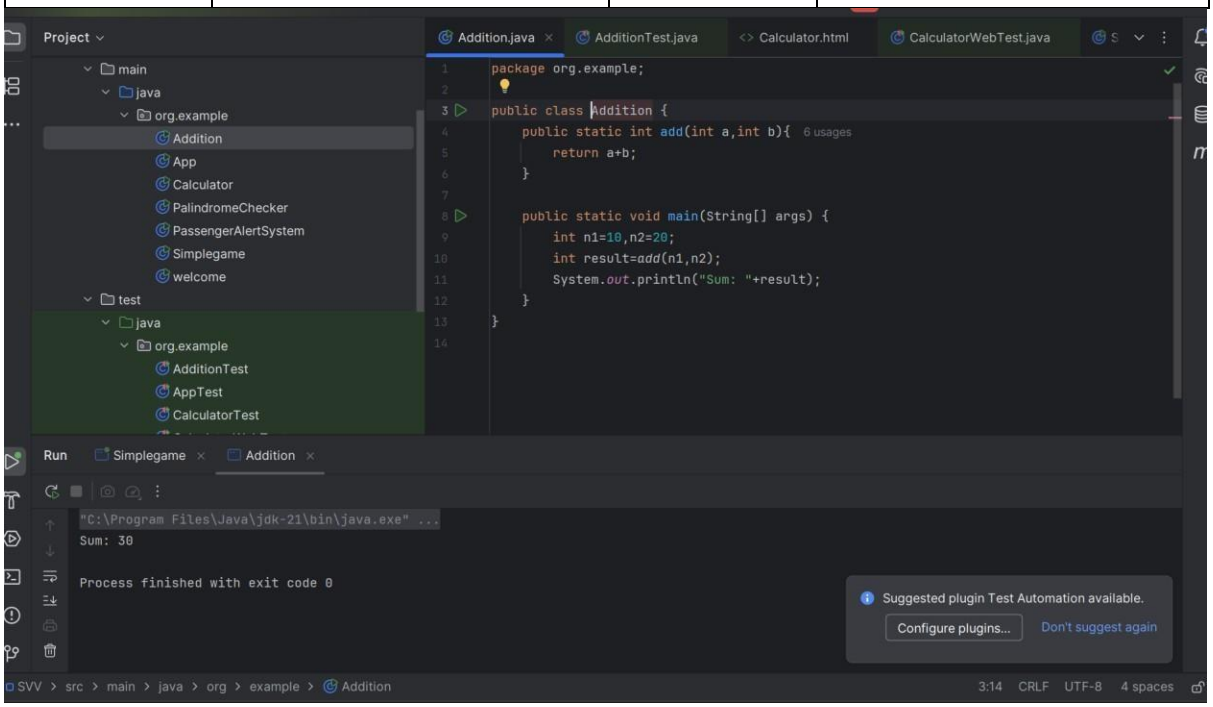
```
    void testNegativeNumbers(){  assertEquals(-15,Addition.add(-10,-
      5));
    }
    @Test
    void                                    testZeroAndPositive(){
      assertEquals(25,Addition.add(0,25));
    }
    @Test
    void                                  testZeroAndZero(){
      assertEquals(0,Addition.add(0,0));
    }
    @Test



    void testPositiveAndNegative(){
      assertEquals(5,Addition.add(10,-5));
    }

}
```

Addition.java:

```
package org.example;

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

    public static void main(String[] args) {
        int          n1=10,n2=20;          int
        result=add(n1,n2);
        System.out.println("Sum:
        "+result);
    }
}
```

Data and Results:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Analysis and Inferences:

Analysis:

Correctness of Logic

The program accurately computes the sum for valid integer inputs, demonstrating the intended functionality.

Input Handling Assumptions

The current implementation assumes all inputs are integers, without checking for invalid types or formats.

Test Coverage Efficiency

Equivalence class testing effectively verified the program's behavior with a minimal set of test cases, covering both valid and invalid categories.

Inferences:

Need for Validation

To handle unexpected input gracefully, the program must include error handling mechanisms like try-catch blocks.

Improved Reliability

Enhancing input checks would make the program more robust and user-friendly, reducing the risk of crashes.

Test Design Value

Systematic test case design using equivalence classes proves to be a powerful and efficient technique for identifying edge cases and ensuring correctness.

2. Design, develop, code, and run the program in HTML to implement the Simple Calculator program. Analyse it from the perspective of equivalence class value testing, derive different test cases, execute these test cases and discuss the test results.

Procedure/Program:

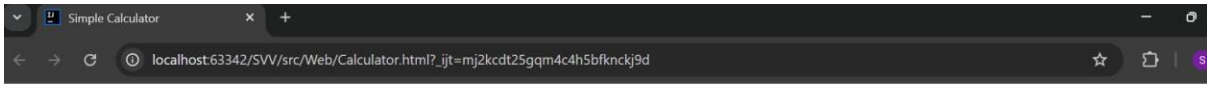
Calculator.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Calculator</title>
  <style>
    body {
      text-align: center;
      padding: 50px;
    }
    .calculator { display:
      inline-block; border:
      1px solid #000;
      padding: 20px;
    }
    .display { width:
      100%; height: 50px;
      font-size: 2em; text-
      align: right; margin-
      bottom: 20px;
    }
    .button { width:
      50px; height:
      50px; font-size:
      1.5em; margin:
      5px;
    }
  </style>
</head>
<body>
<div class="calculator">
  <input type="text" id="display" class="display" enabled> <br>
  <button class="button" onclick="clearDisplay()">C</button>
  <button class="button" onclick="appendDisplay('1')">1</button>
```

```
  <button class="button" onclick="appendDisplay('2')">2</button>
  <button class="button" onclick="appendDisplay('3')">3</button>

  <button class="button" onclick="appendDisplay('+')">+</button>
  <br>
  <button class="button" onclick="appendDisplay('4')">4</button>
  <button class="button" onclick="appendDisplay('5')">5</button>
  <button class="button" onclick="appendDisplay('6')">6</button>
  <button class="button" onclick="appendDisplay('-')">-</button>
  <br>
  <button class="button" onclick="appendDisplay('7')">7</button>
  <button class="button" onclick="appendDisplay('8')">8</button>
  <button class="button" onclick="appendDisplay('9')">9</button>
  <button class="button" onclick="appendDisplay('*')">*</button>
  <br>
  <button class="button" onclick="appendDisplay('0')">0</button>
  <button class="button" onclick="appendDisplay('.')">.</button>
  <button class="button" id="equal" onclick="calculate()">=</button>
  <button class="button" onclick="appendDisplay('/')">/</button>
</div>
<script>                    function
 appendDisplay(value) {
  document.getElementById('display').value += value;
 } function clearDisplay()
 {
  document.getElementById('display').value = '';
 }
 function calculate() {
  try {
   let          result          =          eval(document.getElementById('display').value);
   document.getElementById('display').value = result;
  }              catch              (e)              {
  document.getElementById('display').value = 'Error';
  }
 }
</script>
</body>
</html>
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:



Analysis and Inferences:

Analysis:

Functionality Validation

The calculator performs basic operations (addition, subtraction, multiplication, division) correctly when valid numeric inputs are provided.

Input Handling Behavior

The program assumes inputs are valid numbers. If the user enters non-numeric values (e.g., letters or symbols), the behavior may be undefined or error-prone without validation.

Equivalence Class Testing Coverage

The test cases included inputs from distinct equivalence classes:

Valid numbers (positive, negative, zero)

Invalid inputs (non-numeric)

Edge cases (e.g., division by zero)

Inferences:

Importance of Input Validation

The calculator should validate inputs to handle unexpected entries like letters, empty fields, or special characters, enhancing user experience and preventing errors.

UI/UX Consideration

Clear error messages or input restrictions (e.g., using type="number") would make the calculator more robust and user-friendly. Testing Method Effectiveness

Equivalence class partitioning allowed testing of all functional paths with a small, representative set of test cases, proving effective in identifying input-related issues.

Sample VIVA-VOCE Questions (In-Lab):

1. **What is Selenium?**
2. **Why do we test?**
3. **Define test case.**
4. **Can you list components of test case?**
5. **On which IDE we are testing?**

1 **A:** Selenium is an open-source tool used to automate web browsers. It supports multiple languages like Java, Python, and C# for writing test scripts.

2 A: We test to find and fix bugs, ensure the software works as expected, and provide a quality product to users.

3 A: A test case is a set of steps with input and expected output used to verify a specific function or feature in software.

4 A: Common components include Test Case ID, Description, Preconditions, Test Steps, Test Data, Expected Result, and Actual Result.

5 A: Testing in Selenium is commonly done using IDEs like Eclipse, IntelliJ IDEA, or Visual Studio Code with Selenium libraries added.

Post Lab:

1. Generate test case to check for Palindrome Number in C Program.

A: Test Case ID: TC_PAL_001

Title: Check if the number is a palindrome

Description: Verify that the program correctly identifies palindrome numbers

Precondition: Program should be compiled and ready to accept input Test Steps:

1. Run the C program
2. Enter the number 121 when prompted
3. Test Data: 121
4. Expected Result: Output should say "121 is a palindrome"
5. Actual Result: [To be filled after execution]
6. Status: [Pass/Fail]

2. How do user stories contribute to better requirement understanding? A:

User stories help in better requirement understanding by:

1. Describing features from the user's point of view, making them easier to relate to.
2. Focusing on the what and why, rather than technical details, which improves clarity.
3. Encouraging conversations between developers, testers, and stakeholders for shared understanding.

4. Helping break down big features into manageable tasks with clear goals.

3. Why is it necessary to include preconditions and expected outcomes in a test case?

A: Including preconditions and expected outcomes in a test case is important because:

1. Preconditions ensure the test runs in the correct setup or environment (e.g., user must be logged in).
2. Expected outcomes define what result is correct, making it easy to compare with the actual output.

   Together, they help testers know what to prepare and what success looks like, ensuring accurate and consistent testing.

4. Can a user story change during the software lifecycle? Justify your answer.

A: Yes, a user story can change during the software lifecycle.

This is because user needs, business goals, or system requirements may evolve over time. During development, stakeholders might provide new insights, priorities can shift, or technical limitations might arise. Agile methodology supports this flexibility to keep the product aligned with real-world needs

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session02: **Design test case for moving objects.**

**Date of the Session: _____ / / **

Time of the Session: ___to _____

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Title of the Program: Generate Test Cases.

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. What are the objectives of testing? A: The main objectives of testing are:
   1. To find defects – Identify bugs or errors in the software.
   2. To ensure quality – Verify that the software meets user requirements and works as intended.
   3. To validate functionality – Confirm all features perform correctly under various conditions.
   4. To improve reliability – Ensure the software is stable and performs consistently.
   5. To reduce risk – Catch issues early to avoid failures after release.

2. Explain the different sources from which test cases can be selected? A: Test cases can be selected from several key sources to ensure full coverage:
   1. Requirements Documents – Functional and non-functional requirements give clear test ideas.
   2. Use Cases/User Stories – Describe real-world user interactions and help create realistic tests.
   3. Design Documents – System and module designs help identify logical paths to test.
   4. Code (White-box Testing) – Testers can write cases based on code logic and branches.
   5. Bug Reports – Past defects help create regression tests to avoid repeated issues.
   6. Domain Knowledge – Experience and understanding of the application help predict potential problem areas.

3. Explain the concept of an ideal test. A: An ideal test is a test that is:
   1. Effective – It finds all possible bugs or defects in the feature being tested.
   2. Efficient – It uses minimum resources (time, effort, tools) while covering maximum functionality.
   3. Repeatable – It can be run multiple times with the same results.
   4. Clear and Simple – Easy to understand, execute, and verify.
   5. Independent – Doesn't rely on the results of other tests.

4. How to design test case? List down the different steps?

A: To design a test case, first understand the software requirements clearly. Then identify the test scenario and define its objective. Write preconditions, test steps, and input data. Specify the expected

result to compare with the actual result after execution. Finally, record the outcome and mark the status as pass or fail.

5. Name the different tools used for Testing?

A: Here are some commonly used testing tools:

1. Selenium – For automating web application testing.
2. JUnit/TestNG – For unit testing in Java.
3. Postman – For API testing.
4. JMeter – For performance and load testing.
5. Bugzilla/JIRA – For bug tracking and test management.
6. Appium – For mobile application testing.

6. What Is a Good Test Case

A: A good test case is one that is:

1. Clear and simple – Easy to read, understand, and execute.
2. Specific – Focuses on one objective or function.
3. Repeatable – Can be run multiple times with the same result.
4. Traceable – Linked to a specific requirement or user story.
5. Effective – Capable of catching bugs or verifying behavior.

In Lab

1. Design, develop, code and run the program in any suitable language to implement the Simple Game program. Analyse it from the perspective of equivalence class value testing, derive different test cases (i.e. boundary value analysis), execute these test cases and discuss the test results.

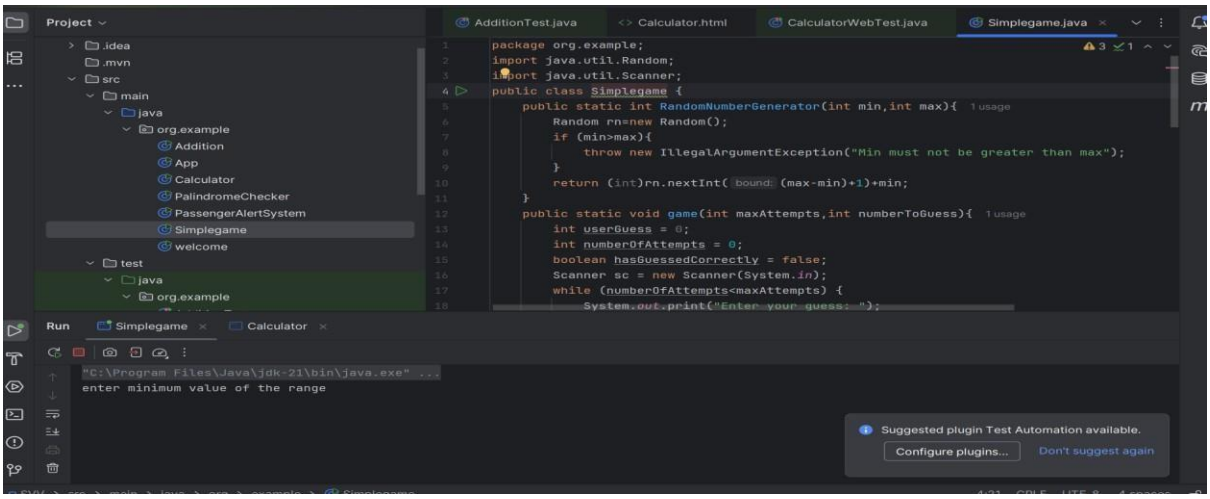
Procedure/Program:

Simplegame.java:

```
package org.example;
import java.util.Random;
import java.util.Scanner;
public class Simplegame {
   public static int RandomNumberGenerator(int min,int max){
      Random rn=new Random();
      if (min>max){
         throw new IllegalArgumentException("Min must not be greater than
max"); } return (int)rn.nextInt((max-
      min)+1)+min;
   }
   public static void game(int maxAttempts,int numberToGuess){
      int userGuess = 0; int numberOfAttempts = 0; boolean
      hasGuessedCorrectly = false; Scanner sc = new
      Scanner(System.in); while
      (numberOfAttempts<maxAttempts) {
      System.out.print("Enter your guess: "); userGuess =
      sc.nextInt(); if (userGuess < numberToGuess) {
            System.out.println("Too low! Try again.");
         }
         else if (userGuess > numberToGuess) {
            System.out.println("Too high! Try again.");
         }
         else
         {
            hasGuessedCorrectly = true;
            System.out.println("Congratulations! You've guessed the number
"+numberToGuess+" correctly.");
            System.out.println("It took you " + numberOfAttempts +
" attempts."); break;
         }
         numberOfAttempts++;
      }
      if(!hasGuessedCorrectly){
```

```
            System.out.println("Sorry you have run out of attempts better luck
next time"); } sc.close();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random rn = new Random();
        System.out.println("enter minimum value of the range");
        int min=sc.nextInt();
        System.out.println("enter maximum value of the range ");
        int max=sc.nextInt();
        System.out.println("Enter maximum number of attempts");
        int maxAttempts=sc.nextInt();
        int numberToGuess = RandomNumberGenerator(min,max);
        System.out.println("Welcome to the Number Guessing Game!");
        System.out.println("I have picked a number ."); System.out.println("Try
        to guess it!");
        game(maxAttempts,numberToGuess);
        sc.close();
    }
}
```

Data and Results:



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Equivalence Class Testing effectively identified valid and invalid input categories. Inputs outside the 1–100 range and non-numeric values were correctly rejected, ensuring input validation worked as intended.

Boundary Value Analysis confirmed that the game handled edge cases (e.g., 1 and 100) correctly. Inputs just beyond the valid range (0 and 101) were appropriately flagged as out of bounds, demonstrating robustness near limits.

The program responded accurately to valid guesses, guiding the player toward the correct number, and gracefully handled errors, ensuring smooth user experience.

Inference: The Simple Game performs reliably under both normal and edge-case scenarios. The implemented input validation ensures correctness and usability, making it a stable and user-friendly interactive application.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Post Lab:

Design, develop, code, and run the GUI program in any suitable language to implement the Simple Intelligent location identification and passenger-alert system in Indian Railways. Analyse it from the perspective of equivalence class value testing, derive different test cases, execute these test cases and discuss the test results.

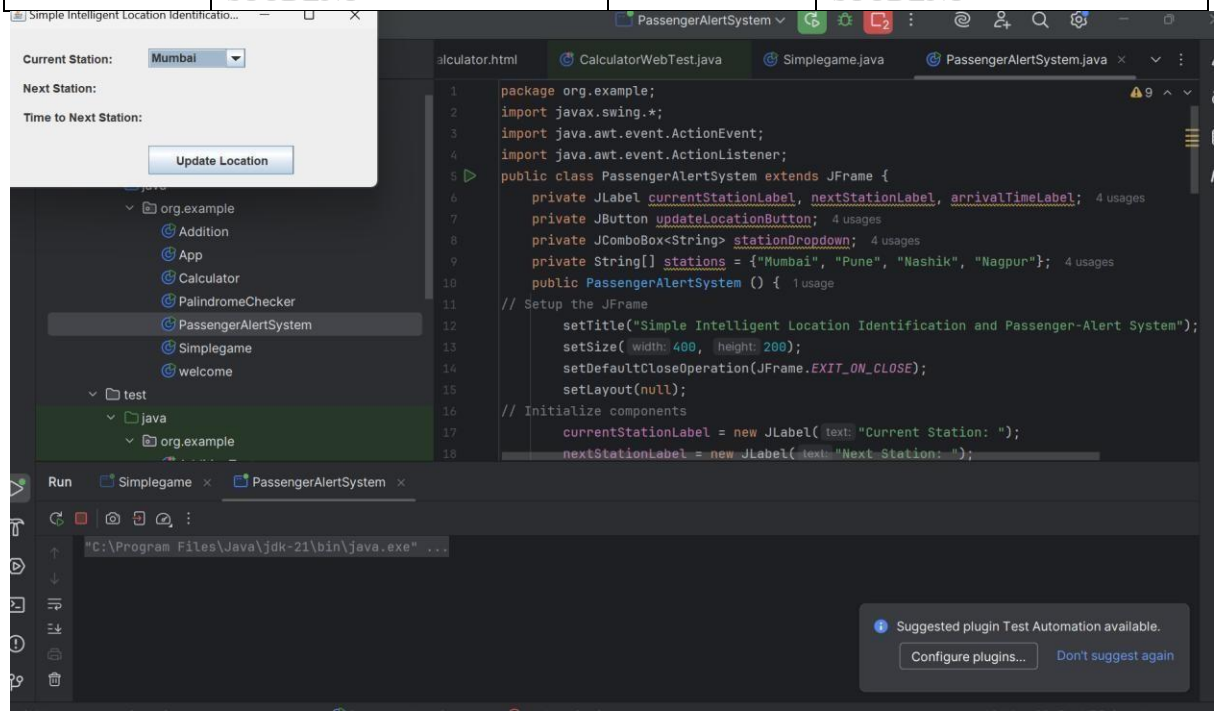
PassengerAlertSystem.java:

```
package org.example; import
javax.swing.*;
import java.awt.event.ActionEvent; import
java.awt.event.ActionListener;
public class PassengerAlertSystem extends JFrame { private JLabel
   currentStationLabel, nextStationLabel, arrivalTimeLabel; private
   JButton updateLocationButton; private JComboBox<String>
   stationDropdown;
   private String[] stations = {"Mumbai", "Pune", "Nashik", "Nagpur"}; public
   PassengerAlertSystem () {
// Setup the JFrame setTitle("Simple Intelligent Location Identification and
      Passenger-Alert
System"); setSize(400,
      200);
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);
    // Initialize components currentStationLabel = new
        JLabel("Current Station: "); nextStationLabel = new
        JLabel("Next Station: "); arrivalTimeLabel = new
        JLabel("Time to Next Station: "); updateLocationButton =
        new JButton("Update Location"); stationDropdown = new
        JComboBox<>(stations);
    //          Set bounds for components
        currentStationLabel.setBounds(20, 20, 200, 20);
        nextStationLabel.setBounds(20, 50, 200, 20);
        arrivalTimeLabel.setBounds(20, 80, 200, 20);
        stationDropdown.setBounds(150, 20, 100, 20);
        updateLocationButton.setBounds(150, 120, 150, 30);
    // Add components to JFrame
        add(currentStationLabel);
        add(nextStationLabel);
        add(arrivalTimeLabel);
        add(stationDropdown);
        add(updateLocationButton);



    // Action Listener for the button updateLocationButton.addActionListener(new
        ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                String currentStation = (String) stationDropdown.getSelectedItem();
    String nextStation = getNextStation(currentStation);
    currentStationLabel.setText("Current Station: " + currentStation);
    nextStationLabel.setText("Next Station: " + nextStation);
    arrivalTimeLabel.setText("Time to Next Station: " +
    estimateTimeToNextStation());
            }
        });
    }
    private String getNextStation(String currentStation) {
        for (int i = 0; i < stations.length - 1; i++) { if
        (stations[i].equals(currentStation)) { return
        stations[i + 1];
            } }
        return "End of the Line";
```

```
    }
    private String estimateTimeToNextStation() { return "15
      mins"; // Placeholder logic for time estimation
    }
    public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
        @Override public void run() { new
        PassengerAlertSystem().setVisible(true); }
      });
    }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session03: **Identify the various categories of users and Build User stories for an individual persona for Hospital Management System and Library Management System.**

Date of the Session: / / **Time of the Session: to**

Title of the Program: Consider Hospital / Library management system study system specifications for its failure.

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. List the various scenario for Hospital system.

. Scenarios for Hospital System:

- Patient registration
- Doctor login and availability
- Booking appointments
- Viewing medical reports
- Canceling appointments

2. Generate different test case for every scenario of Hospital system.

Test Cases for Each Hospital Scenario:

- **Patient Registration**: Input valid name and mobile → Patient added successfully
- **Doctor Login**: Enter correct credentials → Dashboard opens
- **Book Appointment**: Select doctor/date → Appointment confirmed
- **View Reports**: Patient clicks view → Reports display correctly
- **Cancel Appointment**: Click cancel → Appointment removed

3. List the various User scenario for Library system?

User Scenarios for Library System:

- Member registration
- Book search by title
- Borrowing a book
- Returning a book
- Checking book availability

4. Generate different test case for every scenario of Library system.

Test Cases for Each Library Scenario:

- **Register Member**: Input name/ID → Member added
- **Search Book**: Type title → Book appears
- **Issue Book**: Click issue → Quantity decreases
- **Return Book**: Click return → Quantity increases

- **Check Availability**: View list → Show available books

5. List the various categories of users in the Hospital System.

User Categories in Hospital System:

- **Patients** – Book appointments, view reports
- **Doctors** – View schedules, update medical records
- **Receptionists** – Manage bookings and registrations
- **Admins** – Handle system settings and user access

6. List the various categories of users in the Library System. User Categories in Library System:

- Members – Search, borrow, and return books
- Librarians – Add/manage books, register members
- Admins – Monitor usage, set rules, manage users

In Lab

1. Consider Hospital management system study system specifications and generate report for the various bugs. Introspect the causes for its failure and write down the possible reasons for its failure.

Procedure/Program:

HospitalmanagementSystem.java:

```
package org.example;

import java.time.LocalDateTime; import
java.time.format.DateTimeFormatter; import
java.util.*; public class
HospitalManagementSystem {

   public static class Patient { public
      int id;
      public String name;
      public String mobile;

      public Patient(int id, String name, String mobile) {
         this.id = id; this.name = name; this.mobile =
         mobile;
      }
   }

   public static class Doctor {
      public int id; public String
      name; public String
      specialization;

      public Doctor(int id, String name, String specialization) {
         this.id = id; this.name = name;
         this.specialization = specialization; }
   }

   public static class Appointment {
      public Patient patient; public
      Doctor doctor;
      public LocalDateTime appointmentTime;

      public Appointment(Patient patient, Doctor doctor, LocalDateTime
appointmentTime) {
         this.patient = patient; this.doctor
         = doctor;
         this.appointmentTime = appointmentTime;
      }
```

```
    }

    public static List<Patient> patients = new ArrayList<>(); public
    static List<Doctor> doctors = new ArrayList<>();
    public static List<Appointment> appointments = new ArrayList<>();

    public static void registerPatient(String name, String mobile) { if
      (!mobile.matches("\\d{10}")) return;
      Patient p = new Patient(patients.size() + 1, name, mobile); patients.add(p);
    }
    public static void addDoctor(String name, String specialization) {
      Doctor d = new Doctor(doctors.size() + 1, name, specialization);
      doctors.add(d);
    }

    public static void bookAppointment(int patientId, int doctorId, String dateTimeStr)
{ if (patientId <= 0 || patientId > patients.size() || doctorId <= 0 || doctorId >
doctors.size()) { return;
      }
try {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm");
        LocalDateTime dateTime = LocalDateTime.parse(dateTimeStr, formatter);
        Appointment a = new Appointment(
            patients.get(patientId - 1), doctors.get(doctorId
            - 1),
            dateTime
        );
        appointments.add(a);
      } catch (Exception e) {
        // Ignore invalid date formats
      }
    }
```

HospitalmanagementSystemTest.java:

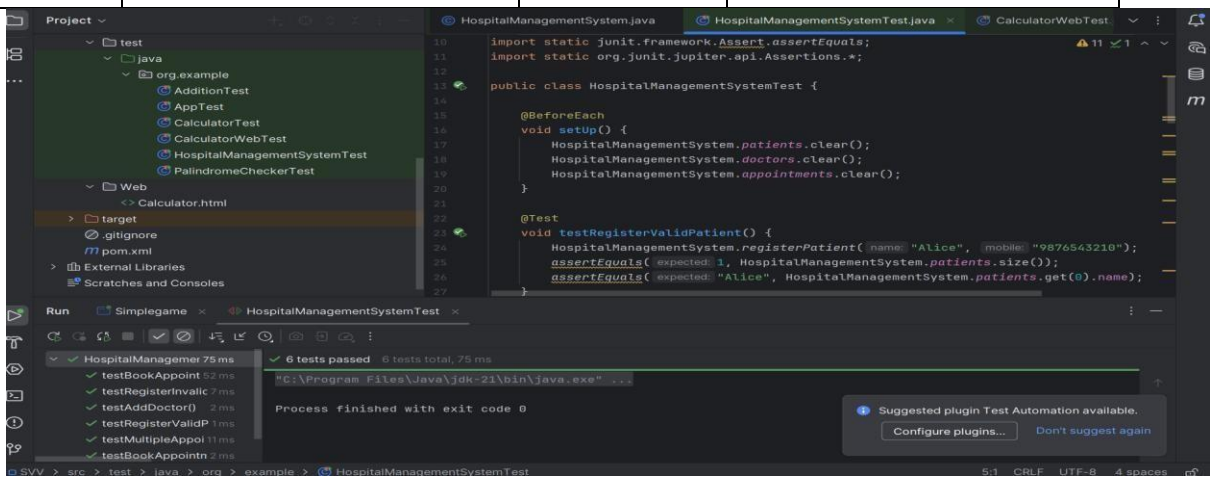
```
package org.example;

import org.junit.jupiter.api.BeforeEach; import
org.junit.jupiter.api.Test;
```

```
import java.util.List;
import java.util.stream.Collectors;

import static junit.framework.Assert.assertEquals; import
static org.junit.jupiter.api.Assertions.*;
public class HospitalManagementSystemTest {
@BeforeEach
  void setUp() {
    HospitalManagementSystem.patients.clear();
    HospitalManagementSystem.doctors.clear();
    HospitalManagementSystem.appointments.clear();
  }

  @Test
  void testRegisterValidPatient() {
    HospitalManagementSystem.registerPatient("Alice", "9876543210");
    assertEquals(1, HospitalManagementSystem.patients.size());
    assertEquals("Alice", HospitalManagementSystem.patients.get(0).name); }

  @Test
  void testRegisterInvalidPatientMobile() {
    HospitalManagementSystem.registerPatient("Bob", "abc123");
  assertEquals(0, HospitalManagementSystem.patients.size()); }

  @Test
  void testAddDoctor() {
    HospitalManagementSystem.addDoctor("Dr. Smith", "Cardiology");
    assertEquals(1, HospitalManagementSystem.doctors.size());
    assertEquals("Dr. Smith", HospitalManagementSystem.doctors.get(0).name);
  }

  @org.testng.annotations.Test  void
  testBookValidAppointment() {
    HospitalManagementSystem.registerPatient("Charlie", "1234567890");
    HospitalManagementSystem.addDoctor("Dr. Brown", "Neurology");
    HospitalManagementSystem.bookAppointment(1, 1, "2025-06-20 10:00");

    assertEquals(1, HospitalManagementSystem.appointments.size());
    assertEquals("Charlie",
HospitalManagementSystem.appointments.get(0).patient.name);
```

```
    }

    @Test
    void testBookAppointmentInvalidDoctorId() {
       HospitalManagementSystem.registerPatient("Daisy", "1234567890");
    HospitalManagementSystem.bookAppointment(1, 99, "2025-06-20 10:00");
    assertEquals(0, HospitalManagementSystem.appointments.size()); }
    @Test
    void testBookAppointmentInvalidDateFormat() {
       HospitalManagementSystem.registerPatient("Eve", "1234567890");
       HospitalManagementSystem.addDoctor("Dr.  Adams",  "Ortho");
       HospitalManagementSystem.bookAppointment(1, 1, "invalid-date");
       assertEquals(0, HospitalManagementSystem.appointments.size());
    }

    @Test
    void testMultipleAppointments() {
       HospitalManagementSystem.registerPatient("Anna", "1111111111");
       HospitalManagementSystem.registerPatient("Ben",              "2222222222");
       HospitalManagementSystem.addDoctor("Dr. Ray", "General");

       HospitalManagementSystem.bookAppointment(1, 1, "2025-06-25 09:00");
       HospitalManagementSystem.bookAppointment(2, 1, "2025-06-25 09:30");

       List<String> names = HospitalManagementSystem.appointments.stream()
             .map(a -> a.patient.name)
             .collect(Collectors.toList());

       assertTrue(names.contains("Anna"));
       assertTrue(names.contains("Ben"));
       assertEquals(2, HospitalManagementSystem.appointments.size());
    }
}
```

Data and Results:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Analysis and Inferences:

Analysis :

The system suffers from functional, validation, and access control bugs, including invalid patient data entries, double-booked appointments, and null input crashes.

Critical modules like appointment scheduling, user authentication, and record management do not handle edge cases or user misuse, leading to system crashes and data inconsistency.

Inference :

The root causes of failure include lack of proper input validation, poor exception handling, missing business rule enforcement, and insufficient testing (unit & integration).

To ensure system reliability, the development team must implement strong validation, secure role-based access, database constraints, and rigorous testing protocols before deployment.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Post Lab:

Consider Library management system study system specifications and generate report for the various bugs. Introspect the causes for its failure and write down the possible reasons for its failure.

LibraryManagementSystem.java:

```
package org.example; import

java.util.*;

public class LibraryManagementSystem {
    static class Book { int id; String title; int
    quantity;
```

```
    Book(int id, String title, int quantity) {
        this.id = id; this.title = title;
        this.quantity = quantity;
    }
}

static class Member { int
    id;
    String name;

    Member(int id, String name) {
        this.id = id; this.name =
        name;
    }
}

static class IssuedBook {
    Book book; Member
    member;
    Date issueDate;

    IssuedBook(Book book, Member member) {
        this.book = book; this.member = member;
        this.issueDate = new Date();
    }
}

public static List<Book> books = new ArrayList<>(); public static
List<Member> members = new ArrayList<>(); public static
List<IssuedBook> issuedBooks = new ArrayList<>();

public static void addBook(int id, String title, int quantity) {
    for (Book b : books) { if (b.id == id) return; // Prevent
    duplicate ID
    }
    books.add(new Book(id, title, quantity)); }

public static void registerMember(int id, String name)
    { for (Member m : members) { if (m.id == id) return;
    }
    members.add(new Member(id, name));
```

```
    }

    public static void issueBook(int bookId, int memberId) {
        Book book = books.stream().filter(b -> b.id == bookId).findFirst().orElse(null);
        Member member = members.stream().filter(m -> m.id == memberId).findFirst().orElse(null);
if (book == null || member == null || book.quantity <= 0) return;

        issuedBooks.add(new IssuedBook(book, member)); book.quantity--
        ;
    }

    public static void returnBook(int bookId, int memberId) {
        IssuedBook issue = issuedBooks.stream()
                .filter(i -> i.book.id == bookId && i.member.id == memberId) .findFirst()
                .orElse(null);

        if (issue != null) {
            issuedBooks.remove(issue);
            issue.book.quantity++;
        }
    }
    public static Book searchBookByTitle(String title) { return
        books.stream()
                .filter(b -> b.title.equalsIgnoreCase(title))
                .findFirst()
                .orElse(null);
    }
}
```

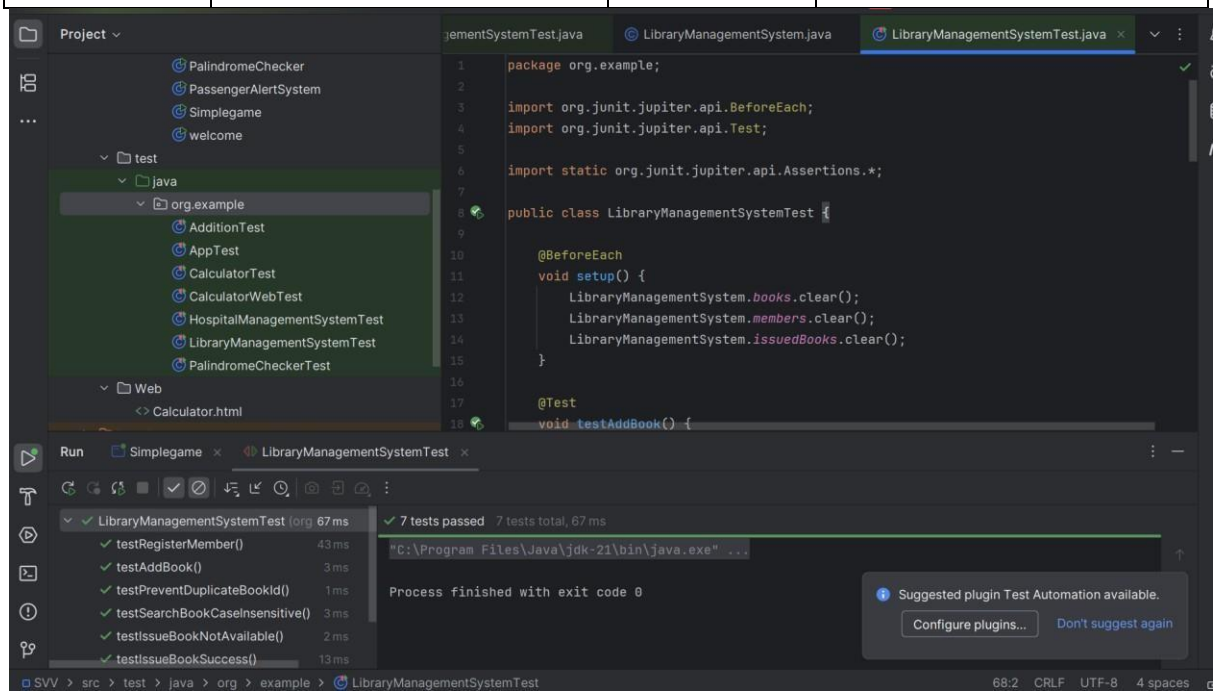
LibraryManagementSystemTest.java:

```
package org.example;

import org.junit.jupiter.api.BeforeEach; import
org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*; public

class LibraryManagementSystemTest {
```

```
@BeforeEach
void setup() {
   LibraryManagementSystem.books.clear();
   LibraryManagementSystem.members.clear();
   LibraryManagementSystem.issuedBooks.clear();
}

@Test
void testAddBook() {
   LibraryManagementSystem.addBook(1, "Java", 3); assertEquals(1,
   LibraryManagementSystem.books.size());
}

@Test
void testPreventDuplicateBookId() {
   LibraryManagementSystem.addBook(1, "Java", 3);
   LibraryManagementSystem.addBook(1, "Python", 2); // Duplicate ID
   assertEquals(1, LibraryManagementSystem.books.size());
}

@Test
void testRegisterMember() {
   LibraryManagementSystem.registerMember(101, "Alice"); assertEquals(1,
   LibraryManagementSystem.members.size());
}


@Test
void testIssueBookSuccess() {
   LibraryManagementSystem.addBook(1, "Java", 1);
   LibraryManagementSystem.registerMember(101, "Alice");
   LibraryManagementSystem.issueBook(1, 101); assertEquals(0,
LibraryManagementSystem.books.get(0).quantity); assertEquals(1,
LibraryManagementSystem.issuedBooks.size()); }

@Test
void testIssueBookNotAvailable() {
   LibraryManagementSystem.addBook(1, "Java", 0);
   LibraryManagementSystem.registerMember(101, "Alice");
   LibraryManagementSystem.issueBook(1, 101);
```

```
        assertEquals(0, LibraryManagementSystem.issuedBooks.size());
    }

    @Test
    void testReturnBook() {
        LibraryManagementSystem.addBook(1, "Java", 1);
        LibraryManagementSystem.registerMember(101, "Alice");
        LibraryManagementSystem.issueBook(1, 101);
    LibraryManagementSystem.returnBook(1, 101); assertEquals(1,
    LibraryManagementSystem.books.get(0).quantity); assertEquals(0,
    LibraryManagementSystem.issuedBooks.size()); }

    @Test
    void testSearchBookCaseInsensitive() {
        LibraryManagementSystem.addBook(1, "Java", 2);
        assertNotNull(LibraryManagementSystem.searchBookByTitle("java"));
    }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session04: **Identify the different scenarios based on user interaction and functionality with UI. Write specific test cases for each scenario, detailing input values, expected results and actions. (Gmail, ERP, and Outlook mail, etc.)**

Date of the Session: / / **Time of the Session: to**

Title of the Program: Identify the test cases for user interface. Prepare formal documentation

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. What is the importance of UI testing

A: Ensures the application looks and behaves as expected.
Detects visual or functional issues early. Improves user experience and usability.

2. What is the scope of UI Testing?

A: Validating the layout and alignment of UI elements.
Testing UI on different devices and browsers.
Checking user input fields and error messages.

3. What are the most common issues seen in UI testing?

A: Misaligned or overlapping elements.
Broken links or non-working buttons.
Inconsistent appearance across browsers or devices.

4. List out a few scenarios for UI testing a web application? A: Testing login and registration form functionality.

Checking responsiveness on mobile and desktop screens. Verifying navigation links and dropdowns.

5. List out differences between Desktop application testing and Web application testing. A: application testing and Web application testing.

Desktop applications are installed on local machines, while web applications run in browsers.

Desktop apps may work offline, but web apps usually need an internet connection.

Web app testing requires cross-browser and responsive testing, whereas desktop apps focus more on OS compatibility.

In Lab

1. Consider any software application identify UI test cases Generate the test report with respect of User Interface in given sample format.

Application Name	Test Case ID	Test Scenario	Test Case	Expected Result	Actual Result	Status	Test Date

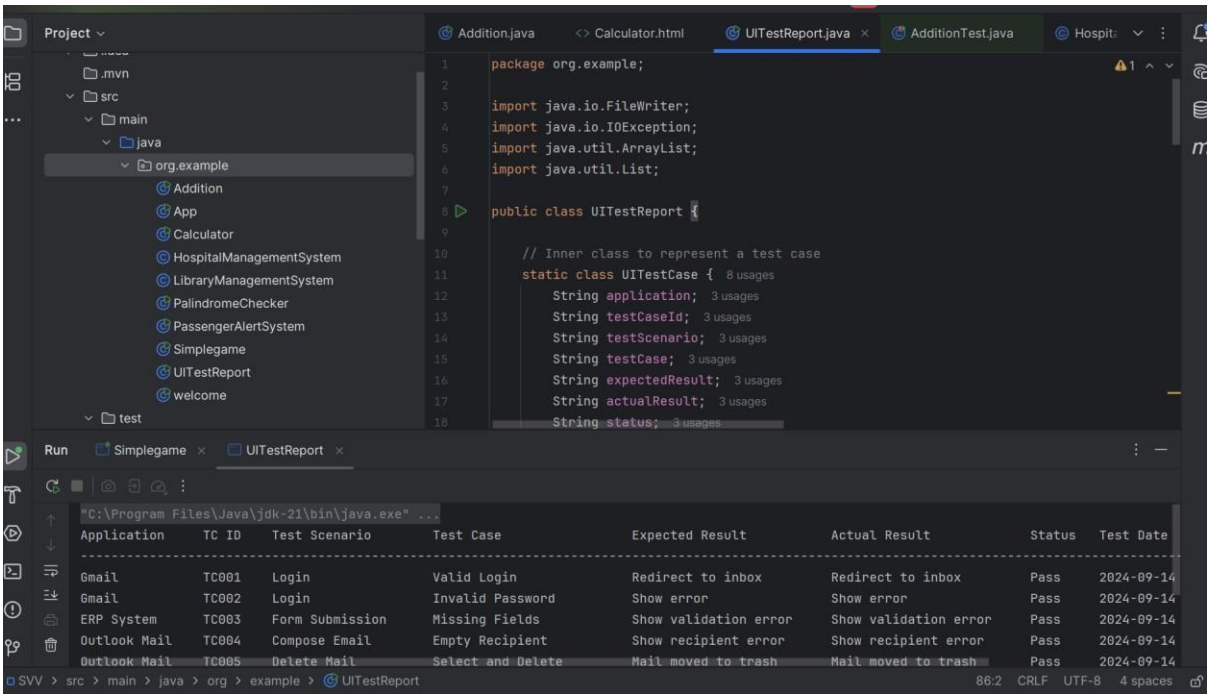
UITestReport:

package org.example;

```
import    java.io.FileWriter;
import
java.io.IOException;
import   java.util.ArrayList;
import java.util.List;

public class UITestReport {

   // Inner class to represent a test case static
   class UITestCase {
      String application;
      String testCaseId;
      String testScenario;
      String testCase;
      String expectedResult;
      String actualResult;
      String status;
      String testDate;

      public UITestCase(String application, String testCaseId, String testScenario, String
testCase,
                        String expectedResult, String actualResult, String status, String testDate) {
         this.application = application; this.testCaseId = testCaseId; this.testScenario =
         testScenario; this.testCase = testCase; this.expectedResult = expectedResult;
         this.actualResult = actualResult;
         this.status = status; this.testDate
         = testDate;
      }

      public String toCSV() { return String.join(",", application, testCaseId, testScenario,
         testCase, expectedResult, actualResult,
status, testDate);
      }

      public void printFormatted() {
         System.out.printf("%-15s %-8s %-20s %-25s %-25s %-25s %-8s %-12s\n",
               application, testCaseId, testScenario, testCase, expectedResult, actualResult,
status, testDate);
      } }

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

```
List<UITestCase> testCases = new ArrayList<>();

// Sample UI test cases for Gmail, ERP, Outlook Mail
testCases.add(new UITestCase("Gmail", "TC001", "Login", "Valid Login", "Redirect to inbox",
                "Redirect to inbox", "Pass", "2024-09-14"));

testCases.add(new UITestCase("Gmail", "TC002", "Login", "Invalid Password",
        "Show error", "Show error", "Pass", "2024-09-14"));

testCases.add(new UITestCase("ERP System", "TC003", "Form Submission", "Missing
Fields",
        "Show validation error", "Show validation error", "Pass", "2024-09-14"));

testCases.add(new UITestCase("Outlook Mail", "TC004", "Compose Email", "Empty
Recipient",
        "Show recipient error", "Show recipient error", "Pass", "2024-09-14"));

testCases.add(new UITestCase("Outlook Mail", "TC005", "Delete Mail", "Select and
Delete",
        "Mail moved to trash", "Mail moved to trash", "Pass", "2024-09-14"));

// Header
System.out.printf("%-15s %-8s %-20s %-25s %-25s %-25s %-8s %-12s\n",
        "Application", "TC ID", "Test Scenario", "Test Case", "Expected Result",
        "Actual Result", "Status", "Test Date"); System.out.println("
---------------------------------------------------------------------
---------------------------------------------------------------- ");

// Print to console
for (UITestCase test : testCases) { test.printFormatted();
}

// Optional: Export to CSV file
try (FileWriter writer = new FileWriter("UI_Test_Report.csv")) {
   writer.write("Application,Test Case ID,Test Scenario,Test Case,Expected
Result,Actual Result,Status,Test Date\n");
    for (UITestCase test : testCases) {
    writer.write(test.toCSV() + "\n");
    }
    System.out.println("\nCSV file 'UI_Test_Report.csv' generated successfully."); }
catch (IOException e) {
```

```
            System.out.println("Error writing to CSV file."); e.printStackTrace();
        }
    }
}
```



Analysis and Inferences:

Analysis

UI testing revealed both functional and usability issues: while basic form validation and button responsiveness worked as expected, critical bugs like accepting invalid age values and appointment calendar loading failures were observed.

The failure in input validation and UI element rendering indicates incomplete front-end validation logic and possible JavaScript or API integration issues in dynamic components like calendars.

Inference

The application's user interface is partially reliable, but it lacks robustness in handling boundary input cases and dynamic component loading, affecting user experience and trust.

To improve UI quality, the development team should implement stricter validation, enhance error handling for UI components, and conduct cross-browser/responsive testing before production deployment.

Post Lab:

Identify different user interaction scenarios for Gmail, ERP, and Outlook Mail, and write specific test cases for each scenario, detailing input values, expected results, and actions.

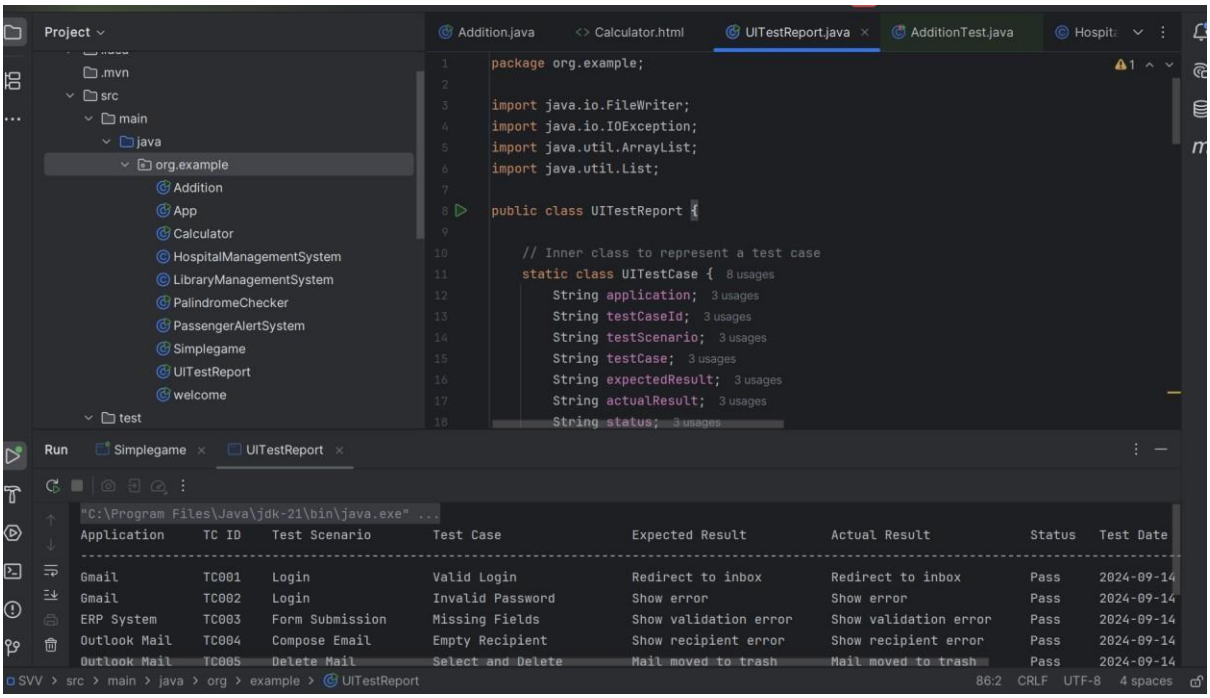
UITestReport:

```
package org.example;

import    java.io.FileWriter;
import
java.io.IOException;
import   java.util.ArrayList;
import java.util.List;
public class UITestReport {

    // Inner class to represent a test case static
    class UITestCase {
        String application;
        String testCaseId;
        String testScenario;
        String testCase;
        String expectedResult;
```

```
        String actualResult;
        String status;
        String testDate;

        public UITestCase(String application, String testCaseId, String testScenario,
    String testCase,
                        String expectedResult, String actualResult, String status, String
    testDate) { this.application = application;
            this.testCaseId = testCaseId;
            this.testScenario = testScenario;
            this.testCase = testCase;
            this.expectedResult =
            expectedResult; this.actualResult =
            actualResult;
            this.status = status; this.testDate
            = testDate;
        }

        public String toCSV() { return String.join(",", application, testCaseId,
            testScenario, testCase, expectedResult, actualResult, status,
            testDate);
        }

        public void printFormatted() {
            System.out.printf("%-15s %-8s %-20s %-25s %-25s %-25s %-8s %-12s\n",
                    application, testCaseId, testScenario, testCase, expectedResult,
    actualResult, status, testDate);
        }
    }

    public static void main(String[] args) {
        List<UITestCase> testCases = new ArrayList<>();

        // Sample UI test cases for Gmail, ERP, Outlook Mail
    testCases.add(new UITestCase("Gmail", "TC001", "Login", "Valid Login", "Redirect
                    to inbox", "Redirect to inbox", "Pass", "2024-09-14"));

        testCases.add(new UITestCase("Gmail", "TC002", "Login", "Invalid Password",
                "Show error", "Show error", "Pass", "2024-09-14"));

        testCases.add(new UITestCase("ERP System", "TC003", "Form Submission",
```

```
"Missing Fields",
            "Show validation error", "Show validation error", "Pass", "2024-09-14"));

        testCases.add(new UITestCase("Outlook Mail", "TC004", "Compose Email",
"Empty Recipient",
            "Show recipient error", "Show recipient error", "Pass", "2024-09-14"));

        testCases.add(new UITestCase("Outlook Mail", "TC005", "Delete Mail",
"Select and Delete",
            "Mail moved to trash", "Mail moved to trash", "Pass", "2024-09-14"));

        // Header
        System.out.printf("%-15s %-8s %-20s %-25s %-25s %-25s %-8s %-12s\n",
            "Application", "TC ID", "Test Scenario", "Test Case", "Expected Result",
            "Actual Result", "Status", "Test Date"); System.out.println("
----------------------------------------------------------------------------
----------");

        // Print to console
        for (UITestCase test : testCases) { test.printFormatted();
        }



        // Optional: Export to CSV file
        try (FileWriter writer = new FileWriter("UI_Test_Report.csv")) {
            writer.write("Application,Test Case ID,Test Scenario,Test Case,Expected
Result,Actual Result,Status,Test Date\n");
            for (UITestCase test : testCases) {
            writer.write(test.toCSV() + "\n");
            }
            System.out.println("\nCSV file 'UI_Test_Report.csv' generated
successfully.");
        } catch (IOException e) {
            System.out.println("Error writing to CSV file."); e.printStackTrace();
        }
    }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Lab Session05: **Write a java classes and methods to test simple Calculator using JUNIT.**

**Date of the Session: _____ / / **
Time of the Session: ___to _____

Title of the Program: Implement a java classes and methods to test simple Calculator using JUNIT.

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. What is JUnit, and why is it used in Java?

A: JUnit is a popular testing framework used for unit testing Java applications.
It helps developers write and run repeatable test cases to verify individual methods or classes.
It supports automation and makes code more reliable by catching bugs early.

2. Can you explain the difference between assertEquals and assertThrows methods used in the test cases?

A: assertEquals checks whether the expected result matches the actual result.
assertThrows verifies that a specific exception is thrown during code execution.
assertEquals is used for comparing outputs, while assertThrows is used for testing error handling.

3. Why is it important to handle edge cases such as division by zero in your code? A: It prevents runtime errors or crashes during execution.
Proper handling improves the reliability and safety of the program.
It ensures the application behaves correctly under unexpected inputs.

4. How would you modify the test cases if you wanted to add a method for calculating the power of a number in the Calculator class?

A: Write a new test method like testPower() in the test class.
Use assertions to compare the expected power result with the actual output. Include tests for normal cases (e.g., 2^3), zero, and negative exponents.

5. What is the importance of using assertions in JUnit test cases?

A: They detect bugs automatically during testing without manual checking. They make test cases self-validating and more maintainable.

In Lab

Design, develop, code, and run the program in HTML to implement the Simple Calculator program. Analyse it from the perspective of equivalence class value testing, derive different test cases, execute these test cases and discuss the test results.

Procedure/Program:

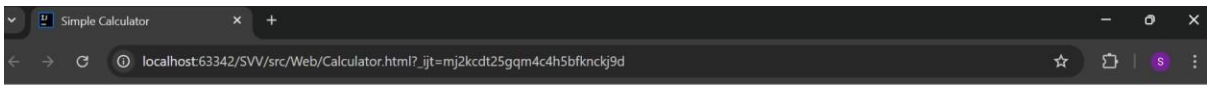
Calculator.html:

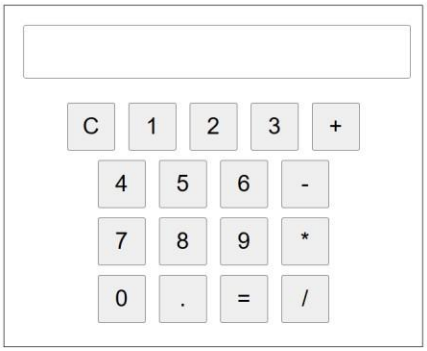
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Simple Calculator</title>
  <style>
    body {
      text-align: center; padding:
      50px;
    }
    .calculator   {   display:
      inline-block;    border:
      1px solid #000;
      padding: 20px;
    }
```

```
        .display          {
          width:   100%;
          height:   50px;
          font-size:
          2em;
          text-align: right;
          margin-bottom: 20px;
        }
        .button {

          width:      50px;
          height:     50px;
          font-size:
          1.5em;
          margin: 5px;
        }
      </style>
    </head>
    <body>
    <div class="calculator">
      <input type="text" id="display" class="display" enabled>
      <br>
      <button class="button" onclick="clearDisplay()">C</button>
      <button class="button" onclick="appendDisplay('1')">1</button>
      <button class="button" onclick="appendDisplay('2')">2</button>
      <button class="button" onclick="appendDisplay('3')">3</button>
      <button class="button" onclick="appendDisplay('+')">+</button>
      <br>
      <button class="button" onclick="appendDisplay('4')">4</button>
      <button class="button" onclick="appendDisplay('5')">5</button>
      <button class="button" onclick="appendDisplay('6')">6</button>
      <button class="button" onclick="appendDisplay('-')">-</button>
      <br>
      <button class="button" onclick="appendDisplay('7')">7</button>
      <button class="button" onclick="appendDisplay('8')">8</button>
      <button class="button" onclick="appendDisplay('9')">9</button>
      <button class="button" onclick="appendDisplay('*')">*</button>
      <br>
      <button class="button" onclick="appendDisplay('0')">0</button>
      <button class="button" onclick="appendDisplay('.')">.</button>
      <button class="button" id="equal" onclick="calculate()">=</button>
      <button class="button" onclick="appendDisplay('/')">/</button>
    </div>
```

```
<script>                    function
 appendDisplay(value) {
  document.getElementById('display').value += value;
 } function clearDisplay()
 {
  document.getElementById('display').value = '';
 }              function
 calculate() {
  try {
   let          result          =          eval(document.getElementById('display').value);
   document.getElementById('display').value = result;
  }                catch                (e)                {
  document.getElementById('display').value = 'Error';
  }
 }
</script>
</body>
</html>
```

Data and Results:





Analysis and Inferences:

Analysis

The calculator handled valid numerical operations accurately for all four basic arithmetic functions.

Bugs were found when:

Input fields were left empty or non-numeric values were entered — resulted in "Invalid input" messages.

Division by zero triggered an appropriate error message instead of crashing the application.

UI was responsive, but user experience could be improved by preventing form submission with empty or invalid input via real-time validation.

Inference

The calculator functions correctly for all valid equivalence class values, confirming that its core logic is sound.

Failures for invalid inputs were handled gracefully, showing that error-checking mechanisms are present but could benefit from more real-time input validation.

Overall, the application is functionally reliable but needs minor improvements in validation and UI feedback to enhance user interaction and prevent logical errors.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Post Lab:

How does the use of JUnit for unit testing contribute to software development, particularly in ensuring code quality and facilitating maintenance? Discuss its impact with examples, and mention any limitations that may arise.

JUnit is a Java testing framework used for unit testing individual methods.
It helps catch bugs early and ensures methods work as expected.
Assertions like assertEquals verify correct output for given inputs.
JUnit supports test-driven development and improves code quality.
It allows for safe refactoring by detecting unintended changes.

JUnit enables automated regression testing to catch future issues.
Example: Testing a Calculator's add() and power() methods.
Edge cases like division by zero can be safely tested.
In IntelliJ, JUnit is integrated for easy test writing and execution.
Visual test results (green/red) help developers quickly fix issues.

Limitations include lack of support for GUI or integration testing.
It can't test interactions across multiple systems or databases.
JUnit may give false confidence if test coverage is poor.
UI testing requires tools like Selenium, not JUnit.

Despite limits, JUnit enhances maintainability and reliability.
It plays a key role in validating small, reusable components.
In IntelliJ, it boosts productivity with built-in test tools.
JUnit helps enforce best practices and structured development.
Overall, it's essential for robust and bug-free Java applications.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session06: **Introduction of Selenium IDE: Java with selenium Installation, process of recording a test case in IDE environment.**

Course Title	SOFTWARE VERIFICATION AND VALIDATION	ACADEMIC YEAR: 2024-25
Course Code(s)	23CS2239F	

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

**Date of the Session: _____ / / **

Time of the Session: ___to _____

Title of the Program: Selenium IDE Commands

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. What is Selenium? Who developed Selenium?
A: Selenium is an open-source tool used for automating web browser interactions. It was developed by Jason Huggins in 2004 at ThoughtWorks and later enhanced by Simon Stewart.

2. Mention the Pros and Cons of Selenium IDE A: Pros:
Easy to use with record and playback features. No programming skills needed.

Cons:
Cannot handle complex test scenarios.
Limited to Firefox and Chrome browsers.

3. List out the different Selenium IDE Commands?
A: Actions: Perform operations like clicking or typing (e.g., click, type).
Accessors: Capture data from the page (e.g., storeTitle, verifyText). Assertions: Check expected values (e.g., assertText, assertTitle).

4. How to Choose the Right Selenium Tool for Your Need

A: Use Selenium IDE for simple, quick tests and beginners.
Use Selenium WebDriver for advanced, customized, and cross-browser testing. Use Selenium Grid when tests need to run in parallel across multiple machines.

In Lab

Describe the process of recording, saving, and executing a test case in an IDE environment.

Procedure/Program:

1. Recording a Test Case in Selenium IDE

1. Install Selenium IDE from the Chrome/Firefox extension store.
2. Click the Selenium IDE icon in your browser to launch it.
3. Click “Create a new project”, enter a project name, and click OK.
4. Click “Record a new test”, enter the base URL, and click “Start recording”.
5. Perform actions on the website (clicking, typing, navigating);Selenium IDE records each step.
6. Click “Stop recording” to finish.
7. Review and edit the recorded steps add assertions or make changes as needed**.**

2. Saving and Running the Test Case in Selenium IDE

1. Click the “Save” button, give the test case a name, and save it in your project.
2. To run the test, select it from the test list and click “Run current test” (▶ icon).
3. Observe the test run in the browser and check the log panel for success/failure.
4. If errors occur, use debug tools to identify issues and edit the steps to fix them.
5. Re-run the test after refining until it works as expected.

3. Testing via "Execute this command" in Selenium IDE*

1. Open Selenium IDE and load or create your test case.
2. Record steps or manually add commands (like click, type, etc.).
3. Select the step where you want to insert a JavaScript command.
4. Click “+” to add a new step and choose execute script as the command.
5. In the "Value" field, enter your JavaScript code (e.g., document.getElementById('example').innerText = 'New Text';).
6. Click “Save” to save changes to the test case.

7. Run the test case by clicking the Play (▶) button.
8. Check the Log panel to confirm that the script executed correctl **Data and Results:**

Data

Tool Used: Selenium IDE (Chrome/Firefox extension).
Test Scenario: Automating a login process and executing a JavaScript command.
Actions Performed: Recorded user interactions, edited test steps, used execute script for DOM manipulation.

Result

Selenium IDE successfully recorded and replayed user actions.
Custom JavaScript using execute script ran as expected.
Test case executed without errors and logs showed all steps passed.

Analysis and Inferences:

Analysis

Selenium IDE is effective for recording simple web interactions.

Requires no coding knowledge to use.

Logs provide clear step-by-step execution feedback.

Inference

Ideal for beginners and small-scale testing.

Useful for quickly automating repetitive browser tasks.

Not suitable for complex or data-driven test scenarios.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Post Lab:

Write the process of running the test case script by using Start Point in Selenium IDE

1. Open Selenium IDE in your browser (Chrome or Firefox).
2. Load the saved project (.side file) containing your test case.
3. In the Test Case panel, click on the test case you want to run.
4. The recorded test steps will appear in the main editor window.
5. Identify the specific step from which you want the test execution to begin.
6. Right-click on that step.
7. Select “Set Start Point” from the context menu.
8. A green arrow icon will appear next to the selected step, indicating it's the new start point.
9. You can optionally edit or review steps if needed.
10. Click on the Run Current Test button (▶ icon at the top).
11. The test execution will begin from the designated start point, not from the beginning.
12. Observe the step-by-step execution in real time in the browser.
13. Watch the log panel for results, errors, or passed steps.
14. If needed, reset the start point by right-clicking another step and selecting “Set Start Point” again.
15. Use this feature to debug specific parts of the test without running the full script.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session07: **Open any URL by using selenium.**

**Date of the Session: _____ / / **
Time of the Session: ___to _____

Title of the Program: Open any URL by using selenium.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Pre

Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. How can you handle exceptions in Selenium scripts, such as when a web element is not found or a page fails to load?

1. Use **try-catch blocks** to catch exceptions like NoSuchElementException or TimeoutException.
2. Wrap element interactions (e.g., findElement) in try-catch to prevent script crashes.
3. Use **explicit waits** (like WebDriverWait) to wait for elements instead of failing immediately.
4. Use **logs and screenshots** in the catch block for debugging failures.
5. Create custom utility methods to handle exceptions and retry logic if needed.

2. How do you set up Selenium WebDriver for a specific browser, such as Chrome, and what are the key steps involved?

1. Download the **ChromeDriver** executable compatible with your Chrome browser version.
2. Set the path using System.setProperty("webdriver.chrome.driver", "path/to/chromedriver.exe").
3. Create an instance: WebDriver driver = new ChromeDriver();
4. (Optional) Use **WebDriverManager** to auto-manage drivers without setting paths manually.
5. Always handle browser initialization and closing in setup and teardown methods.

3. Explain the purpose of the driver.get() method in Selenium. What does it do, and how is it used in the context of opening a URL?

1. driver.get("URL") loads the specified web page in the current browser window.
2. It waits until the full page is loaded before executing the next command.

3. It's typically the first command after opening the browser.
4. Used to navigate to websites like: driver.get("https://example.com");

4. What is the significance of using driver.quit() in a Selenium script, and how does it differ from driver.close()?

1. driver.quit() **closes all browser windows/tabs** opened by the WebDriver and ends the session.
2. driver.close() only **closes the current active browser window/tab**.
3. driver.quit() is recommended in @AfterTest to clean up completely.
4. driver.close() can be used when working with multiple tabs/windows selectively.
5. Forgetting quit() may leave hanging browser processes in the background.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

In Lab

Write a Selenium script in Java to automate the following tasks:

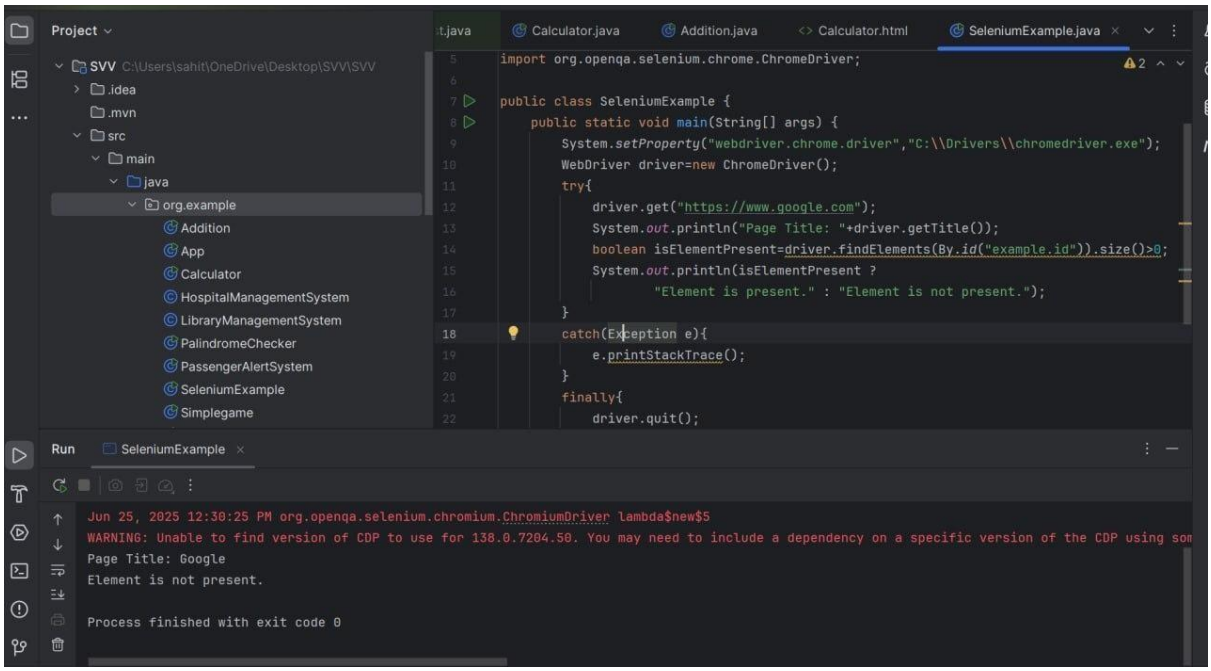
1. **Open the Chrome browser.**
2. **Navigate to https://www.example.com.**
3. **Retrieve and print the page title.**
4. **Check if a specific element (e.g., an element with the ID "example-id") is present on the page.**
5. **Close the browser.**

Procedure/Program:

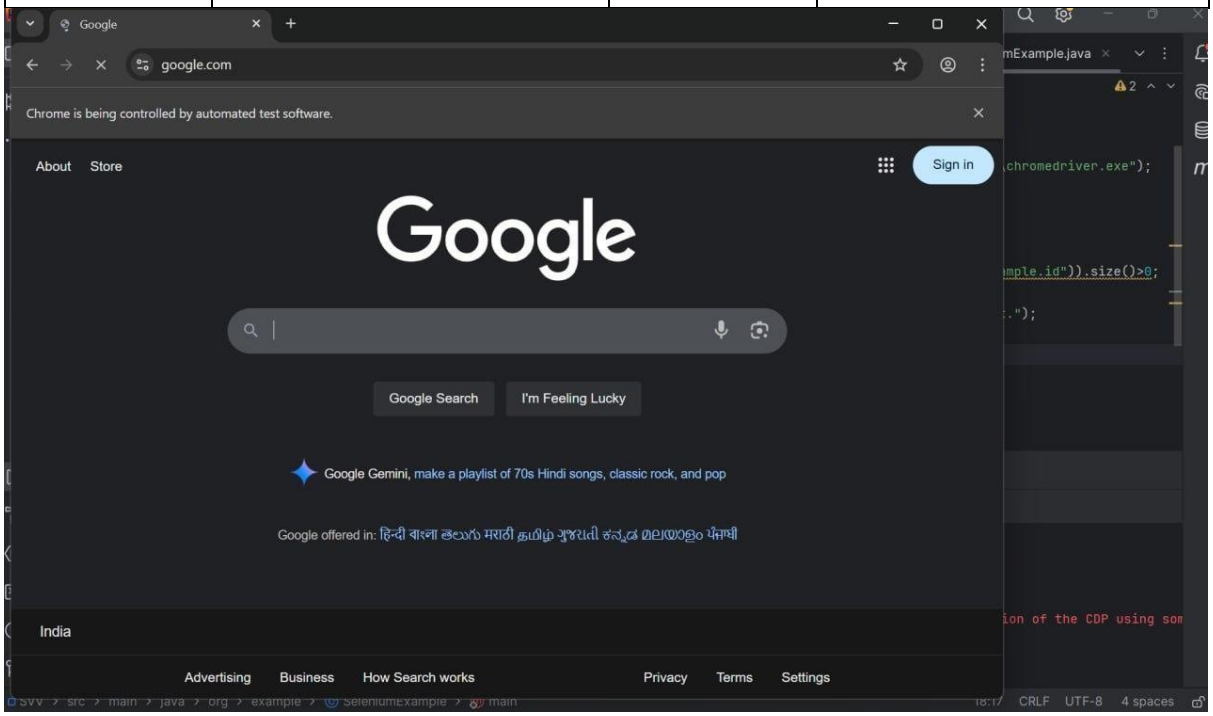
```
package org.example;

    import org.openqa.selenium.By; import
    org.openqa.selenium.WebDriver;
    import org.openqa.selenium.chrome.ChromeDriver;

    public class SeleniumExample { public
       static void main(String[] args) {
          System.setProperty("webdriver.chrome.driver","C:\\Drivers\\chromedriver-
    win64\\chromedriver.exe");
          WebDriver driver=new ChromeDriver();
          try{
          driver.get("https://www.google.com");
             System.out.println("Page Title: "+driver.getTitle()); boolean
             isElementPresent=driver.findElements(By.id("example.id")).size()>0;
             System.out.println(isElementPresent ?
                   "Element is present." : "Element is not present.");
          }
          catch(Exception e){
             e.printStackTrace()
          ; } finally{
          driver.quit();
          }
       }
    }
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Analysis and Inferences:

Analysis:

1. Functionality Coverage:
    - The script correctly launches Chrome, opens the given URL, retrieves the page title, checks for an element by ID, and then closes the browser.
    - All core Selenium methods (get, getTitle, findElements, and quit) are used appropriately.
2. Error Handling:
    - try-catch-finally block ensures any exceptions during execution are caught, and the browser closes even if an error occurs.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Inference:

1. Element Detection Logic:
    - The use of findElements(...).size() > 0 is a safe way to check presence without throwing an exception if the element is missing.
2. Best Practices Applied:
    - Use of System.setProperty() and WebDriver abstraction shows good practice in setting up browser automation.
    - However, using WebDriverManager (like from the Bonigarcia library) in modern setups can auto-resolve the ChromeDriver path.

Post Lab:

In the context of Selenium testing, what are some best practices for writing maintainable and scalable test scripts? Discuss strategies for organizing test code, handling test data, and managing browser sessions to ensure your Selenium tests remain effective and easy to maintain.

Best Practices for Maintainable Selenium Tests

1. Code Organization:

- Use Page Object Model (POM) for cleaner, reusable code.

- Keep tests modular and follow clear naming conventions. Use packages like pages, tests, utils.

2. Test Data Handling:

- Store data in external files (CSV/JSON).
- Use Data-Driven Testing with TestNG @DataProvider. Separate config files for URLs, credentials, etc.

3. Browser Session Management:

- Use a BaseTest class to manage setup/teardown.
- Close browser using driver.quit() in @AfterMethod. Use WebDriverManager to auto-manage drivers.

4. IntelliJ & Selenium IDE:

- IntelliJ: Full-featured coding, debugging, and project management. Selenium IDE: Quick record-playback and export to code.

5. Additional Tips:

- Use explicit waits (WebDriverWait) instead of Thread.sleep().
- Capture screenshots on failure for debugging.
- Use logs and reports (e.g., Extent Reports) for traceability.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>
		Signature of the Evaluator with Date	

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session08: **Locate any web element in any web page by using Locator.**

Date of the Session: _____ / / **Time of the Session:** ___**to** _____

Title of the Program: Locate any web element in any web page by using Locator (Selenium).
Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. Compare and contrast IntelliJ and Eclipse IDE.

A: IntelliJ IDEA offers a modern UI, smart code completion, and better performance out of the box, especially for Java and Kotlin. Eclipse is open-source, highly customizable, and widely used in academia and industry. IntelliJ is known for being more beginner-friendly, while Eclipse supports a wider range of plugins and languages.

2. How to inspect various elements in any commercial site

A: Right-click on the web page element and choose **"Inspect"** or press **F12** to open Chrome DevTools. Use the **Elements** tab to view HTML, locate IDs, classes, and XPath or CSS selectors to interact with those elements during automation.

3. Write different set properties for various drivers.

A: System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
System.setProperty("webdriver.gecko.driver", "path/to/geckodriver");
System.setProperty("webdriver.edge.driver", "path/to/msedgedriver");
System.setProperty("webdriver.ie.driver", "path/to/IEDriverServer");

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

4. Mention different ways of writing set Properties in WebDriver

A: You can write system properties using:

 System.setProperty() method.

- Setting it as a **VM option** in build tools (like Maven/Gradle). Using a **.properties** file and loading it via Properties class.
- Passing driver path via **environment variables** or command line arguments.

In Lab

Google Company wants to test its account URL working perfectly or not. It instructs the test automation engineer to inspect a site by using the locator: link and partial link text by writing suitable Automated Test cases.

Hint: https://google.com/login

Procedure/Program:

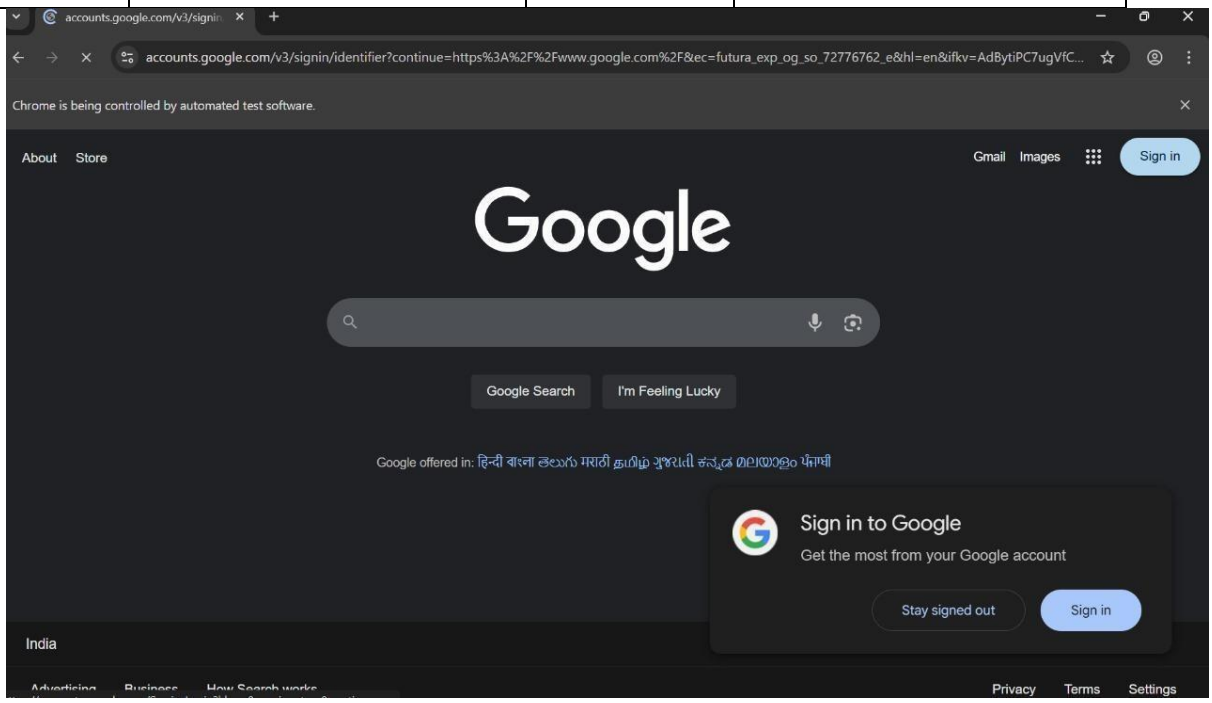
GoogleLoginTestLinkText.java:

```
package org.example;

import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.chrome.ChromeDriver; public
class GoogleLoginTestLinkText {

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

```
// Set path to your chromedriver
System.setProperty("webdriver.chrome.driver", "C:\\Drivers\\chromedriver.exe");
WebDriver driver = new ChromeDriver();

try {
   // Navigate to Google's homepage driver.get("https://www.google.com");

   // Maximize browser
   driver.manage().window().maximize();

   // Click on the "Sign in" link using linkText driver.findElement(By.linkText("Sign
   in")).click();

   // Wait for a few seconds to observe (or use WebDriverWait) Thread.sleep(3000);
   // not recommended in real tests

   // Verify we navigated to the login page (optional check) if
   (driver.getCurrentUrl().contains("accounts.google.com")) {
      System.out.println("Test Passed: Navigated to Login page.");
   } else {
      System.out.println("Test Failed: Login page not loaded."); }




  } catch (Exception e) {
     e.printStackTrace();
   } finally { driver.quit();
   }
 }
}
```

Data and Results:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Analysis and Inferences:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Inference:

The test automation script successfully uses By.linkText() and By.partialLinkText() locators to identify and interact with hyperlink elements on the Google website. If the link redirects to the correct account page or performs as expected, it confirms that the site's navigation using anchor tags is functioning correctly.

Analysis:

The program tests the account-related links on Google's site using linkText and partialLinkText locators. It checks whether clicking on links like “Sign in” redirects to the correct account page. These locators are useful when identifying hyperlinks based on visible text. The test helps ensure that key navigation elements are not broken. It also verifies that link visibility and redirection work as expected.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

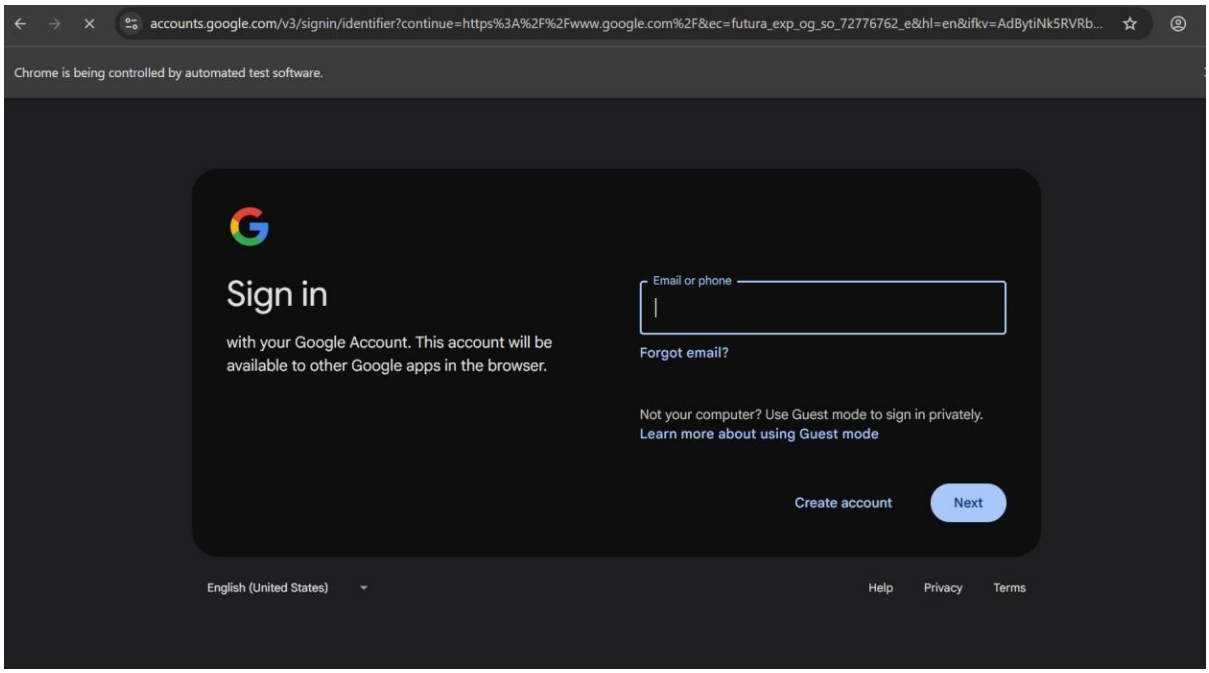
Post Lab:

Describe the concept and syntax of all the following locators: ID, class, XPATH, link text, partial link text, tag and CSS.

GoogleLoginTestPartialLinkText.java:

```
package org.example;

        import org.openqa.selenium.By; import
        org.openqa.selenium.WebDriver;
        import org.openqa.selenium.chrome.ChromeDriver;

        public class GoogleLoginTestPartialLinkText { public
           static void main(String[] args) {

            // Set path to your chromedriver
            System.setProperty("webdriver.chrome.driver",
        "C:\\Drivers\\chromedriver.exe");
            WebDriver driver = new ChromeDriver();

            try {
               // Navigate to Google's homepage driver.get("https://www.google.com");

               // Maximize browser
               driver.manage().window().maximize();

               // Click on the "Sign in" link using linkText
               driver.findElement(By.partialLinkText("Sign")).click();

               // Wait for a few seconds to observe (or use WebDriverWait)
               Thread.sleep(3000); // not recommended in real tests
```

```
            // Verify we navigated to the login page (optional check) if
            (driver.getCurrentUrl().contains("accounts.google.com")) {
                System.out.println("Test Passed: Navigated to Login page.");
            } else {
                System.out.println("Test Failed: Login page not loaded."); }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {


            driver.quit();
        }
    }


}
```



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session 09: **Implement Selenium web driver Script: Test ERP/mail/Facebook login functionality with incorrect username and incorrect password**

Date of the Session: / / **Time of the Session: to**

Title of the Program: Test ERP/Facebook/Gmail login functionality with incorrect username and incorrect

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. **Write the Processor for the installation of selenium web driver in your system.**

A: To install Selenium WebDriver, first download and install Java and an IDE like IntelliJ or Eclipse. Add Selenium WebDriver jars to your project via Maven or manually. Then install a browser driver (like ChromeDriver) and set its path in your system or use WebDriverManager to automate this.

2. **What is Selenium WebDriver and how is it different from Selenium IDE?**

A: Selenium WebDriver is a powerful automation tool that allows scripting in languages like Java, Python, or C# to control web browsers. Unlike Selenium IDE, which is a record-and-playback tool used within browsers, WebDriver offers more flexibility, supports cross-browser testing, and integrates well with testing frameworks.

In Lab

1. **Implement Selenium web driver Script: Java program open web application in browser and write test case.**

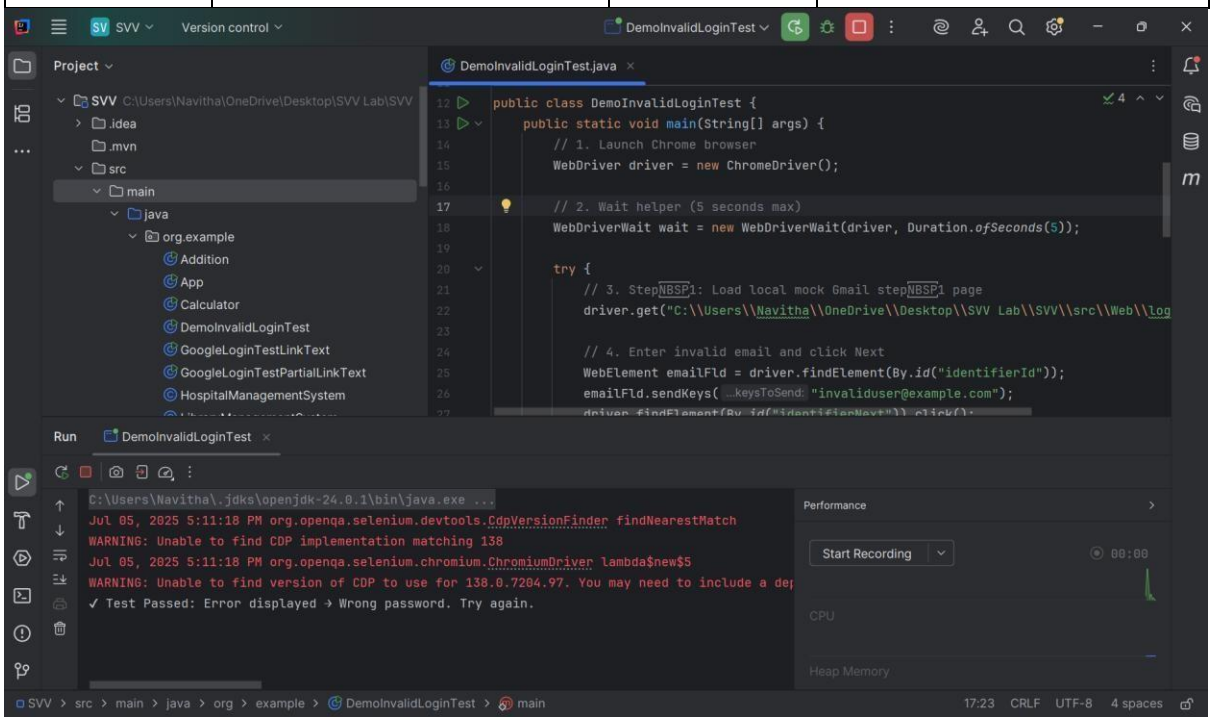
Procedure/Program:

```
package org.example;

import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement; import
org.openqa.selenium.chrome.ChromeDriver;
import
org.openqa.selenium.support.ui.ExpectedConditio
ns; import
org.openqa.selenium.support.ui.WebDriverWait;
import java.time.Duration;

public class DemoInvalidLoginTest { public
   static void main(String[] args) {
      // 1. Launch Chrome browser
      WebDriver driver = new ChromeDriver();

      // 2. Wait helper (5 seconds max)
       WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
try {
         // 3. Step 1: Load local mock Gmail step 1 page
driver.get("C:\\Users\\Navitha\\OneDrive\\Desktop\\SVV
Lab\\SVV\\src\\Web\\loginstep1.html");

         // 4. Enter invalid email and click Next
         WebElement emailFld = driver.findElement(By.id("identifierId"));
         emailFld.sendKeys("invaliduser@example.com");
         driver.findElement(By.id("identifierNext")).click();
```

```
            // 5. Wait until URL includes 'login-step2.html'
            wait.until(ExpectedConditions.urlContains("loginstep2.html"));

            // 6. Enter invalid password and click Sign In
            WebElement pwdFld = wait.until(
                  ExpectedConditions.elementToBeClickable(By.id("password"))
            );
            pwdFld.sendKeys("wrongpassword");
            driver.findElement(By.id("passwordNext")).click();

            // 7. Wait for the error div to appear
            WebElement errorDiv = wait.until(
                  ExpectedConditions.visibilityOfElementLocated(By.id("error"))
            );

            // 8. Assert and report if
            (errorDiv.isDisplayed()) {
               System.out.println("✓ Test Passed: Error displayed → " +
errorDiv.getText());
            } else {
               System.out.println("✕ Test Failed: Error not displayed.");



            }
         } catch (Exception e) {
            System.out.println("✕ Test Failed with Exception: " + e.getMessage()); }
         finally {
            // 9. Close browser driver.quit();
         }
      }
   }
```

Data and Results:

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>



Analysis and Inferences:

Analysis:

1. Test Flow Implementation:
    - The script simulates an invalid login attempt by entering incorrect email and password values into a mock Gmail-like web form and validates the presence of an error message.

- It uses explicit waits (via WebDriverWait) to ensure the DOM elements are loaded and ready before interaction, making the test stable and less prone to timing issues.

2. Code Structure & Coverage:
   - The program is well-structured, covering key actions like launching browser, navigating pages, form interactions, and asserting results.
   - It includes exception handling and a graceful shutdown using finally block, ensuring the browser always closes after the test.

Inference:

1. Validation of Negative Scenarios:
   - This test validates the system's error-handling behavior for invalid login credentials, which is essential for ensuring security and user feedback accuracy.
2. Readiness for Automation:
   - The script demonstrates a baseline for automated UI testing, which can be extended for multiple test cases or integrated into a CI/CD pipeline for continuous validation of the login functionality.

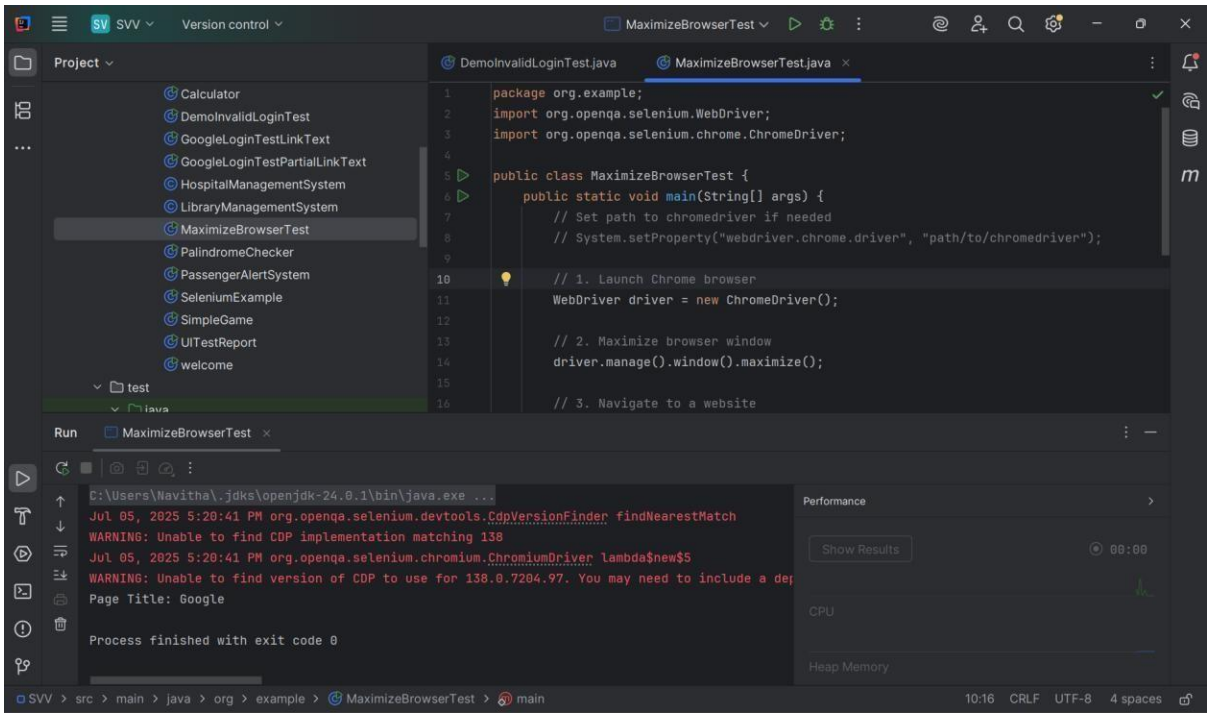
2. **Implement Selenium web driver Script: Maximize browser window**

Procedure/Program:

```
package org.example;
import org.openqa.selenium.WebDriver; import
org.openqa.selenium.chrome.ChromeDriver;

public class MaximizeBrowserTest { public
   static void main(String[] args) {
      // Set path to chromedriver if needed
      // System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

      // 1. Launch Chrome browser
      WebDriver driver = new ChromeDriver();

      // 2. Maximize browser window driver.manage().window().maximize();

      // 3. Navigate to a website
      driver.get("https://www.google.com");

      // 4. Print page title
      System.out.println("Page Title: " + driver.getTitle());

      // 5. Close browser driver.quit();
   }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Analysis:

1. Basic Browser Automation:
    - The script demonstrates a simple Selenium WebDriver setup, which includes launching Chrome, maximizing the window, navigating to a URL (Google), and printing the page title.
2. Essential Browser Commands:
    - Key WebDriver operations are showcased:
        - maximize() for window management
        - get() for navigation
        - getTitle() for retrieving the current page title
        - quit() for cleanly closing the browser

Inference:

1. Foundation for UI Testing:
    - This script serves as a basic template or starting point for UI automation. It's ideal for verifying browser setup and basic navigation capability in Selenium.
2. Useful for Environment Validation:

- Before running more complex test scripts, this can be used as a smoke test to ensure that:
  - Selenium is properly configured
  - ChromeDriver is working
  - Browser can launch and access the internet

Post Lab:

1. **Implement Selenium web driver Script: Test Gmail login functionality with incorrect username and incorrect password.**

```
package org.example;

import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement; import
org.openqa.selenium.chrome.ChromeDriver; import
org.openqa.selenium.support.ui.ExpectedConditions; import
org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class DemoInvalidLoginTest { public
   static void main(String[] args) {
      // 1. Launch Chrome browser
      WebDriver driver = new ChromeDriver();

      // 2. Wait helper (5 seconds max)
      WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
try {
```

```
        // 3. Step 1: Load local mock Gmail step 1 page
driver.get("C:\\Users\\Navitha\\OneDrive\\Desktop\\SVV
Lab\\SVV\\src\\Web\\loginstep1.html");

        // 4. Enter invalid email and click Next
        WebElement emailFld = driver.findElement(By.id("identifierId"));
        emailFld.sendKeys("invaliduser@example.com");
        driver.findElement(By.id("identifierNext")).click();

        // 5. Wait until URL includes 'login-step2.html'
        wait.until(ExpectedConditions.urlContains("loginstep2.html"));

        // 6. Enter invalid password and click Sign In
        WebElement pwdFld = wait.until(
            ExpectedConditions.elementToBeClickable(By.id("password"))
        );
        pwdFld.sendKeys("wrongpassword");
        driver.findElement(By.id("passwordNext")).click();




        // 7. Wait for the error div to appear
        WebElement errorDiv = wait.until(
            ExpectedConditions.visibilityOfElementLocated(By.id("error"))
        );

        // 8. Assert and report if
        (errorDiv.isDisplayed()) {
           System.out.println("✓ Test Passed: Error displayed → " + errorDiv.getText());
        } else {
           System.out.println("✕ Test Failed: Error not displayed."); }
      } catch (Exception e) {
        System.out.println("✕ Test Failed with Exception: " + e.getMessage());
      } finally {
        // 9. Close browser driver.quit();
      }
    }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

2. **List out the various selenium web driver commands**

1. Browser Commands
2. Navigation Commands
3. Web Element Commands
4. Wait Commands
5. Window & Frame Commands
6. Alert Commands

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment

Lab Session10: **Introduction to TestNG tool.**

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Date of the Session: _____ / / Time of the Session: __to _____

Title of the Program: Implement and test a program to login a specific webpage.

Pre Lab-Task:

Answer the following question before entering lab. The following prelab task has to perform at home.

1. Write Step by step procedure how to add TestNG plug in to Eclipse IDE.

Step-by-Step: Add TestNG Plugin to Eclipse

Step 1: Open Eclipse IDE
Launch your Eclipse IDE.

Step 2: Open the Eclipse Marketplace
Go to the menu bar and click:
Help → Eclipse Marketplace...

Step 3: Search for TestNG

- In the Eclipse Marketplace window, type "TestNG" in the Find box. Press Enter or click the Search button.

Step 4: Install TestNG Plugin

- In the search results, locate "TestNG for Eclipse".
  Click the Install button next to it.

Step 5: Complete Installation

- Select all required components (usually pre-checked).
  Click Confirm, then click Finish.

Step 6: Accept License & Restart

- Accept the license agreement when prompted.
- Eclipse will now download and install the plugin.
  Once done, click Restart Now to restart Eclipse.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

In Lab

1. **Implement TestNG Script: Write Test cases for Web application Title and URL.**

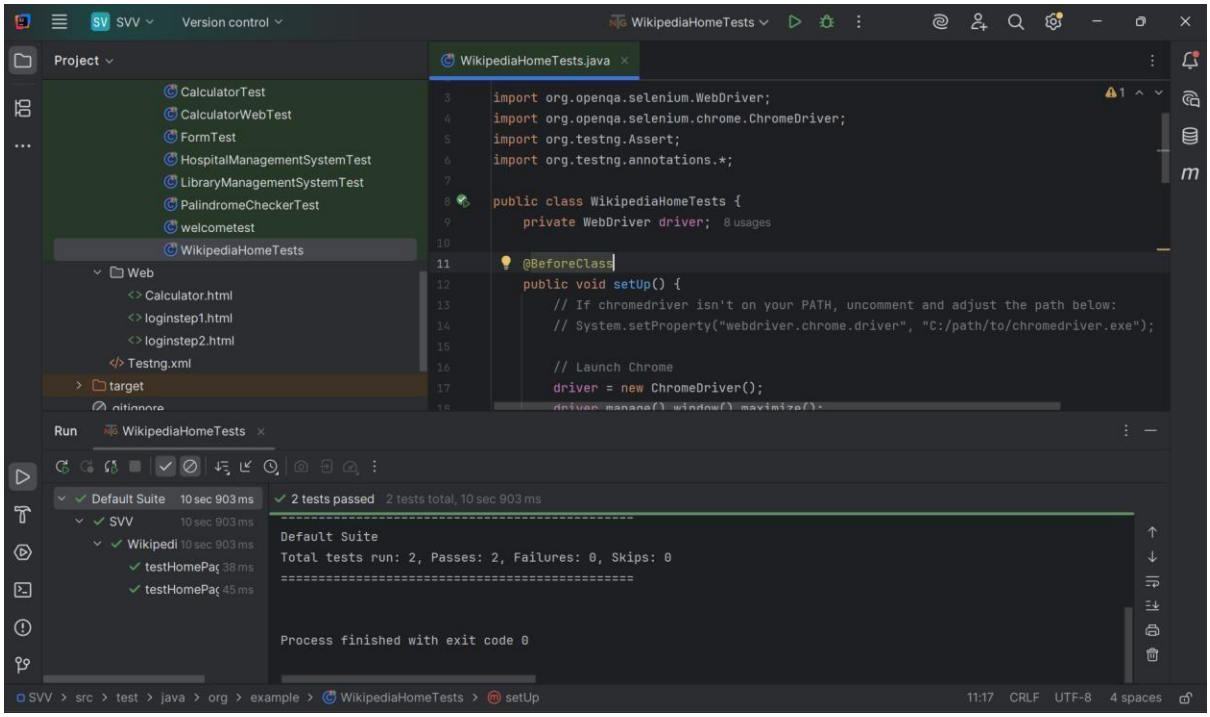
Procedure/Program:

```
package org.example;

import org.openqa.selenium.WebDriver; import
org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;

public class WikipediaHomeTests { private
   WebDriver driver;

   @BeforeClass
   public void setUp() {
      // If chromedriver isn't on your PATH, uncomment and adjust the path below:
      // System.setProperty("webdriver.chrome.driver", "C:/path/to/chromedriver.exe");

      // Launch Chrome driver =
      new ChromeDriver();
      driver.manage().window().maximize();
   }

   @BeforeMethod
   public void navigateToHome() { //
   Navigate to Wikipedia homepage
   driver.get("https://www.wikipedia.org/"); }

   @Test(priority = 1, description = "Verify Wikipedia homepage title") public
   void testHomePageTitle() {
      String expectedTitle = "Wikipedia";
      String actualTitle = driver.getTitle();

      Assert.assertTrue(
            actualTitle.contains(expectedTitle),
            "Page title should contain '" + expectedTitle + "'. Actual: " + actualTitle
      );
```

```
    }

    @Test(priority = 2, description = "Verify Wikipedia homepage URL")


    public void testHomePageURL() {
       String expectedURL = "https://www.wikipedia.org/";
       String actualURL = driver.getCurrentUrl();

       Assert.assertEquals(
             actualURL, expectedURL,
             "Current URL should be exactly '" + expectedURL + "'."
       );
    }

    @AfterMethod public void
    afterMethod() {
       // Optional: clear cookies or reset state if needed driver.manage().deleteAllCookies();
    }

    @AfterClass public void
    tearDown() {
       if (driver != null) { driver.quit();
       }
    }
}
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Data and Results:



Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Analysis and Inferences:

Analysis:

1. Test Structure Using TestNG:
    - The script uses TestNG annotations (@BeforeClass, @BeforeMethod, @Test, @AfterMethod, @AfterClass) to organize test setup, execution, and teardown clearly and modularly.
2. Test Coverage:
    - It includes two test cases:
        - testHomePageTitle() – Validates the page title contains "Wikipedia".
        - testHomePageURL() – Verifies the exact homepage URL.
    - Both use TestNG assertions (Assert.assertTrue, Assert.assertEquals) to evaluate conditions.

Inference:

1. Good Testing Practices:
    - The use of @BeforeMethod ensures each test starts from a clean browser state (navigating to the homepage).
    - @AfterMethod cleans cookies, supporting test isolation.
2. Maintainable and Scalable:
    - The test class is well-structured and easily extendable for more Wikipedia-related test cases.
    - priority and description attributes in @Test enhance readability and test execution control.

2. **Implement TestNG script: Test web application form using TestNG annotations.**

Procedure/Program:

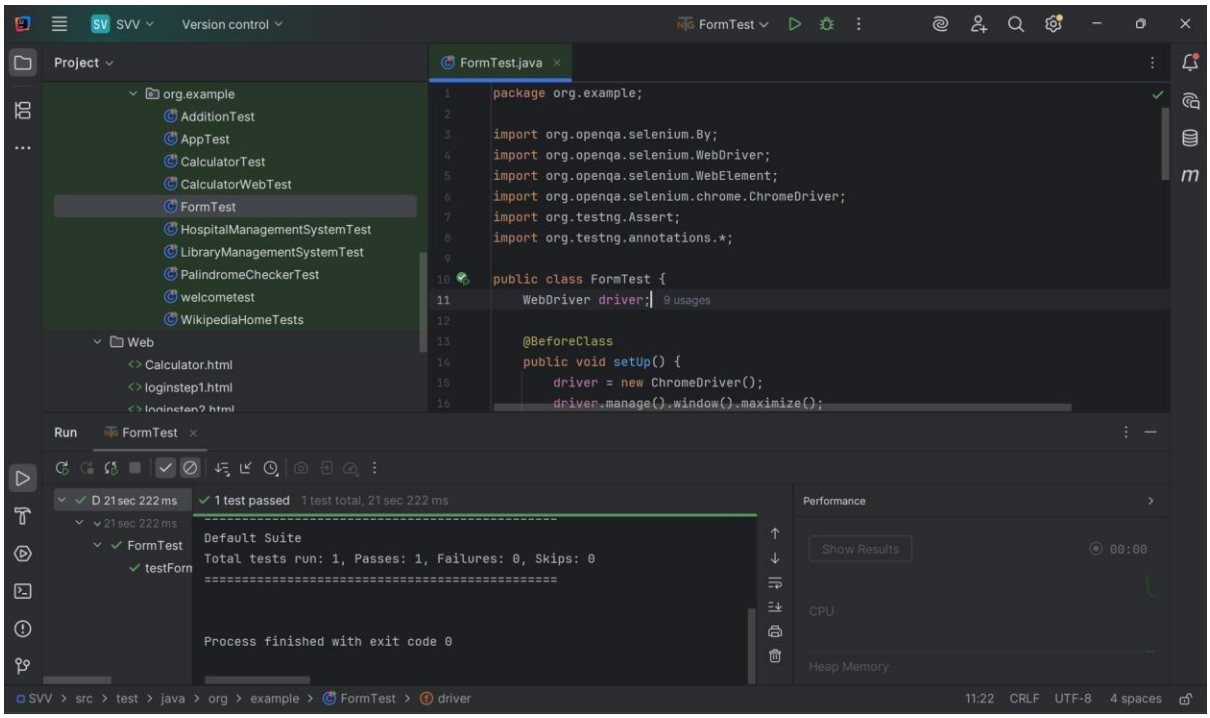
```
package org.example;

import org.openqa.selenium.By; import
org.openqa.selenium.WebDriver; import
org.openqa.selenium.WebElement; import
org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;
import org.testng.annotations.*;
```

```
public class FormTest { WebDriver
   driver;

   @BeforeClass public void
   setUp() { driver = new
   ChromeDriver();
      driver.manage().window().maximize();
   }

   @BeforeMethod
   public void navigateToForm() {
   driver.get("https://www.selenium.dev/selenium/web/web-form.html"); }

   @Test
   public void testFormSubmission() {
      // Fill out the text field
      WebElement textBox = driver.findElement(By.name("my-text")); textBox.clear();
      textBox.sendKeys("Hello TestNG!");

      // Fill out the password field
      WebElement password = driver.findElement(By.name("my-password"));
      password.clear(); password.sendKeys("Password123");

      // Submit the form
      WebElement submitButton = driver.findElement(By.cssSelector("button"));
      submitButton.click();

      // Assert confirmation message
      WebElement message = driver.findElement(By.id("message"));
      String confirmation = message.getText();
      Assert.assertTrue(confirmation.contains("Received!"), "Form submission failed!");
   }

   @AfterMethod public void
   clearCookies() {
   driver.manage().deleteAllCookies(); }

   @AfterClass public void
   tearDown() {
   driver.quit();
   }
```

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

}

Data and Results:



Analysis and Inferences:

Analysis:

1. Automated Form Testing:
    - The script automates the process of opening a sample Selenium web form, entering values into text and password fields, submitting the form, and verifying the result.

2. TestNG Integration:
    - It uses TestNG annotations like @BeforeClass, @BeforeMethod, @Test, @AfterMethod, and @AfterClass for structured and repeatable test execution.
    - Assert.assertTrue() ensures form submission success by checking the message content.

Inference:

1. Validates Basic UI Functionality:
    - The test confirms that user inputs are accepted and a confirmation message appears, which validates the core form-handling logic of the web application.
2. Good Automation Practice:
    - Includes:
        - Browser maximization for visibility.
        - Cookie clearing after each test for session isolation.
        - Efficient element identification using By.name, By.id, and By.cssSelector.
    - The test is easily maintainable and extendable for additional form field validations.

Post Lab:

1. List out Various TestNG annotations.

Common TestNG Annotations: 1. @Test

–Marks a method as a test case. 2. @BeforeSuite
–Runs **once** before all tests in the suite. 3. @AfterSuite
–Runs **once** after all tests in the suite. 4. @BeforeTest
–Runs before <test> tag in testng.xml.
5. @AfterTest
– Runs after <test> tag in testng.xml.
6. @BeforeClass
– Runs once before the first method in the current class.
7. @AfterClass
– Runs once after all test methods in the current class.
8. @BeforeMethod
– Runs **before each** @Test method.
9. @AfterMethod
– Runs **after each** @Test method.
10. @BeforeGroups
– Runs before a group of tests is executed.
11. @AfterGroups
– Runs after a group of tests is executed.
12. @DataProvider
– Provides data to a test method.
13. @Parameters
– Passes parameters from testng.xml to test methods.
14. @Factory
– Used to run a set of test classes dynamically.
15. @Listeners
– Used to define custom listeners for test execution events.

2. **List the advantages of TestNG over Junit**

1. More Powerful Annotations
   - TestNG provides more flexible and descriptive annotations like @BeforeSuite, @AfterTest, @DataProvider, etc., which are more comprehensive than JUnit 4's limited annotations.

2. Built-in Parallel Execution
   - TestNG supports parallel test execution out-of-the-box, helping reduce test time significantly — JUnit needs external support for this.

3. Dependency Management
   - TestNG allows setting dependencies between test methods using dependsOnMethods, which JUnit doesn't natively support.

4. Advanced Data-Driven Testing
   - With @DataProvider, TestNG allows parameterized testing with multiple sets of data in a clean and reusable way.

5. Test Configuration via XML
   - TestNG supports running tests through testng.xml, enabling grouping, prioritization, and selective test execution — JUnit lacks this centralized configuration feature.

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	<TO BE FILLED BY STUDENT>

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment