

PL/SQL Exercises

QUESTION 1:-

Creation of **CUSTOMER** Table

```
INSERT INTO customers VALUES (1, 'John', 'Doe', TO_DATE('1950-01-01', 'YYYY-MM-DD'), 15000, 'N');
```

```
INSERT INTO customers VALUES (2, 'Jane', 'Smith', TO_DATE('1985-06-10', 'YYYY-MM-DD'), 8000, 'N');
```

The screenshot shows the Oracle Live SQL interface. On the left, the Navigator pane shows the 'My Schema' containing a table named 'CUSTOMERS'. The table's columns are listed: CUSTOMER_ID, FIRST_NAME, LAST_NAME, DOB, BALANCE, and ISVIP. The main editor shows a SQL query that selects all columns from the 'CUSTOMERS' table. The 'Script output' pane displays the results of the query, showing two rows of data for the 'CUSTOMERS' table. The output includes the customer ID, first name, last name, date of birth, balance, and whether they are a VIP.

CUSTOMER_ID	FIRST_NAME	LAST_NAME	DOB	BALANCE	ISVIP
1	John	Doe	01/01/1950, 05:30:00 AM	15000	N
2	Jane	Smith	06/10/1985, 05:30:00 AM	8000	N

Creation of **LOAN** Table

```
INSERT INTO loans VALUES (101, 1, 9.5, SYSDATE + 10);
```

```
INSERT INTO loans VALUES (102, 2, 10.0, SYSDATE + 40);
```

The screenshot shows the Oracle Live SQL interface. On the left, the Navigator pane shows the 'My Schema' containing a table named 'LOANS'. The table's columns are listed: LOAN_ID, CUSTOMER_ID, INTEREST_RATE, and DUE_DATE. The main editor shows a SQL query that selects all columns from the 'LOANS' table. The 'Script output' pane displays the results of the query, showing two rows of data for the 'LOANS' table. The output includes the loan ID, customer ID, interest rate, and due date.

LOAN_ID	CUSTOMER_ID	INTEREST_RATE	DUE_DATE
101	1	9.5	07/09/2025, 11:24:54 PM
102	2	10	08/08/2025, 11:24:54 PM

Exercise 1: Control Structures

Scenario 1: The bank wants to apply a discount to loan interest rates for customers above 60 years old.

- **Question:** Write a PL/SQL block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates.

Solution:-



```
SET SERVEROUTPUT ON;

DECLARE
    v_rows PLS_INTEGER;
BEGIN
    UPDATE loans l
    SET    interest_rate = interest_rate * 0.99
    WHERE EXISTS (
        SELECT 1
        FROM    customers c
        WHERE   c.customer_id = l.customer_id
        AND     TRUNC(MONTHS_BETWEEN(SYSDATE, c.dob)/12) > 60
    );

    v_rows := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(v_rows || ' loan(s) discounted. ');
    COMMIT;
END;
/
```

OUTPUT:-

Query result Script output DBMS output Explain Plan SQL history


```
SQL> DECLARE
      v_rows PLS_INTEGER;
      BEGIN
      UPDATE loans l...
```

[Show more...](#)

1 loan(s) discounted.

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.015



Activate Windows

After Senorio 1 the updated table are as follows

```
SELECT * FROM loans;
```

```
SELECT * FROM customers;
```

LOAN_ID CUSTOMER_ID INTEREST_RATE DUE_DATE

101	1	9.41	07/09/2025, 11:24:54 PM
102	2	10	08/08/2025, 11:24:54 PM

Elapsed: 00:00:00.003

2 rows selected.

CUSTOMER_ID FIRST_NAME LAST_NAME DOB BALANCE ISVIP

1	John	Doe	01/01/1950, 05:30:00 AM	15000	N
2	Jane	Smith	06/10/1985, 05:30:00 AM	8000	N

Elapsed: 00:00:00.002

2 rows selected.

Scenario 2: A customer can be promoted to VIP status based on their balance.

- **Question:** Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over \$10,000.

Solution:-

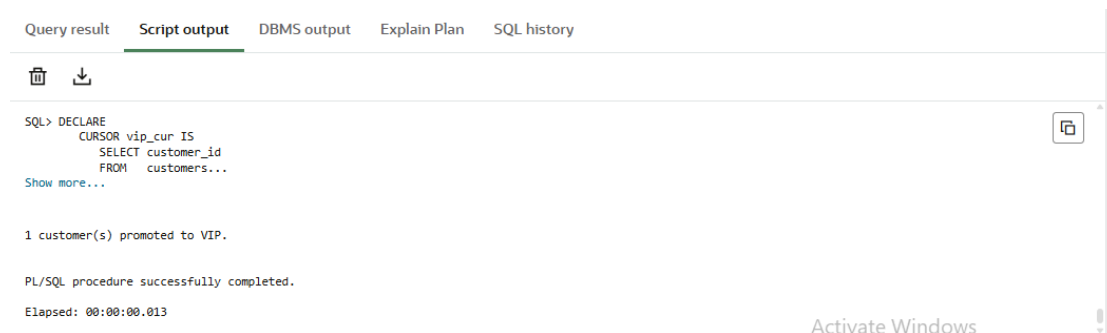
```
SET SERVEROUTPUT ON;

DECLARE
    CURSOR vip_cur IS
        SELECT customer_id
        FROM   customers
        WHERE  balance > 10000;

    v_count PLS_INTEGER := 0;
BEGIN
    FOR rec IN vip_cur LOOP
        UPDATE customers
        SET    isvip = 'Y'
        WHERE customer_id = rec.customer_id;
        v_count := v_count + 1;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE(v_count || ' customer(s) promoted to VIP. ');
    COMMIT;
END;
/
```

OUTPUT:-



Query result **Script output** DBMS output Explain Plan SQL history

SQL> DECLARE
CURSOR vip_cur IS
SELECT customer_id
FROM customers...
Show more...

1 customer(s) promoted to VIP.

PL/SQL procedure successfully completed.

Elapsed: 00:00:00.013

Activate Windows

After Scenario 2 the updated table is as follows

```
SELECT customer_id, first_name, balance, isvip FROM customers;
```

```
SQL> SELECT customer_id, first_name, balance, isvip FROM customers
```

CUSTOMER_ID	FIRST_NAME	BALANCE	ISVIP
1	John	15000	Y
2	Jane	8000	N

Elapsed: 00:00:00.004
2 rows selected.

Scenario 3: The bank wants to send reminders to customers whose loans are due within the next 30 days.

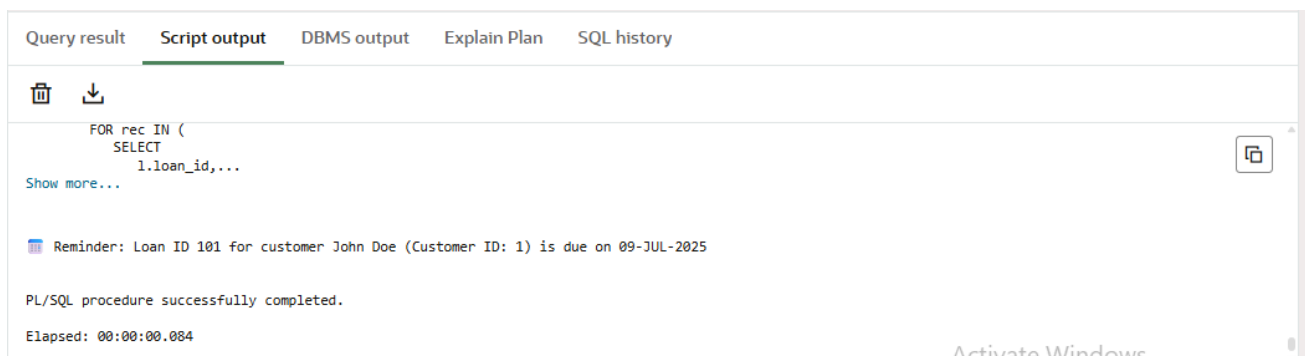
- **Question:** Write a PL/SQL block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

Solution:-

```
SET SERVEROUTPUT ON;

BEGIN
    FOR rec IN (
        SELECT
            l.loan_id,
            l.due_date,
            c.customer_id,
            c.first_name || ' ' || c.last_name AS full_name
        FROM
            loans l
        JOIN
            customers c ON c.customer_id = l.customer_id
        WHERE
            l.due_date BETWEEN SYSDATE AND SYSDATE + 30
        ORDER BY
            l.due_date
    ) LOOP
        DBMS_OUTPUT.PUT_LINE(
            '17 Reminder: Loan ID ' || rec.loan_id ||
            ' for customer ' || rec.full_name ||
            ' (Customer ID: ' || rec.customer_id ||
            ') is due on ' || TO_CHAR(rec.due_date, 'DD-MON-YYYY')
        );
    END LOOP;
END;
/
```

OUTPUT:-



The screenshot shows the SQL Developer interface with the 'Script output' tab selected. The output displays the following:

```
FOR rec IN (
  SELECT
    l.loan_id,...
```

Below the SQL code, a reminder message is shown:

```
Reminder: Loan ID 101 for customer John Doe (Customer ID: 1) is due on 09-JUL-2025
```

The message is preceded by a small icon. Below the reminder, the following text is displayed:

```
PL/SQL procedure successfully completed.
Elapsed: 00:00:00.084
```

The bottom right corner of the window shows the 'Activate Windows' watermark.

Question 3:-

Build a minimal schema & sample data

-- 1A. Core tables

```
CREATE TABLE customers (  
  customer_id  NUMBER PRIMARY KEY,  
  first_name   VARCHAR2(50),  
  last_name    VARCHAR2(50)  
);  
  
CREATE TABLE accounts (  
  account_id   NUMBER PRIMARY KEY,  
  customer_id  NUMBER REFERENCES customers(customer_id),  
  account_type VARCHAR2(15),  -- 'SAVINGS' or 'CHECKING'  
  balance      NUMBER(15,2)  
);  
  
CREATE TABLE departments (  
  department_id NUMBER PRIMARY KEY,  
  dept_name     VARCHAR2(50)  
);  
  
CREATE TABLE employees (  
  employee_id   NUMBER PRIMARY KEY,  
  first_name    VARCHAR2(50),  
  last_name     VARCHAR2(50),  
  department_id NUMBER REFERENCES departments(department_id),  
  salary        NUMBER(15,2)  
);
```

-- 1B. Sample data (tiny but enough to test)

```
INSERT INTO customers VALUES (1, 'John', 'Doe');  
INSERT INTO customers VALUES (2, 'Jane', 'Smith');  
  
INSERT INTO accounts VALUES (1001, 1, 'SAVINGS', 5000);  
INSERT INTO accounts VALUES (1002, 1, 'CHECKING', 2000);  
INSERT INTO accounts VALUES (1003, 2, 'SAVINGS', 12000);  
  
INSERT INTO departments VALUES (10, 'Operations');  
INSERT INTO departments VALUES (20, 'IT');  
  
INSERT INTO employees VALUES (101, 'Alice', 'Green', 10, 60000);  
INSERT INTO employees VALUES (102, 'Bob', 'Brown', 10, 55000);  
INSERT INTO employees VALUES (103, 'Carol', 'White', 20, 70000);  
  
COMMIT;
```

Scenario 1: The bank needs to process monthly interest for all savings accounts.

- **Question:** Write a stored procedure **ProcessMonthlyInterest** that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance.

Solution:-

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest
IS
    v_rows PLS_INTEGER;
BEGIN
    UPDATE accounts
    SET     balance = balance * 1.01          -- +1 %
    WHERE  account_type = 'SAVINGS';

    v_rows := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(v_rows || ' savings account(s) credited with monthly
interest.');
```

COMMIT;

END;

/

Output:-

Procedure PROCESSMONTHLYINTEREST compiled

Elapsed: 00:00:00.025

Test call:-

```
SET SERVEROUTPUT ON;
EXEC ProcessMonthlyInterest;

-- Verify
SELECT account_id, balance FROM accounts WHERE account_type =
'SAVINGS';
```

Output:-

ACCOUNT_ID BALANCE

1001 5050

1003 12120

Elapsed: 00:00:00.005

2 rows selected.

Scenario 2: The bank wants to implement a bonus scheme for employees based on their performance.

- **Question:** Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

Solution:-

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (  
    p_dept_id    IN    employees.department_id%TYPE,  
    p_bonus_pct  IN    NUMBER                -- e.g. pass 5 for 5 %  
)  
IS  
    v_rows PLS_INTEGER;  
BEGIN  
    UPDATE employees  
    SET     salary = salary * (1 + p_bonus_pct/100)  
    WHERE  department_id = p_dept_id;  
  
    v_rows := SQL%ROWCOUNT;  
    DBMS_OUTPUT.PUT_LINE(v_rows || ' employee(s) received a ' || p_bonus_pct  
    || '% bonus in department ' || p_dept_id || '.');  
    COMMIT;  
END;  
/
```

Output:-

Procedure UPDATEEMPLOYEEBONUS compiled

Elapsed: 00:00:00.022

Test call:-

```
SET SERVEROUTPUT ON;  
EXEC UpdateEmployeeBonus(p_dept_id => 10, p_bonus_pct => 5);  
  
-- Verify  
SELECT employee_id, department_id, salary FROM employees WHERE  
department_id = 10;
```

Output:-

EMPLOYEE_ID	DEPARTMENT_ID	SALARY
101	10	63000
102	10	57750

Elapsed: 00:00:00.005

2 rows selected.

Scenario 3: Customers should be able to transfer funds between their accounts.

- **Question:** Write a stored procedure **TransferFunds** that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

Solution:-

```
CREATE OR REPLACE PROCEDURE TransferFunds (
    p_from_acct IN accounts.account_id%TYPE,
    p_to_acct   IN accounts.account_id%TYPE,
    p_amount    IN NUMBER
)
IS
    v_from_bal NUMBER;
BEGIN
    -- 1. Get current balance of source account
    SELECT balance
    INTO   v_from_bal
    FROM   accounts
    WHERE  account_id = p_from_acct
    FOR UPDATE;

    -- 2. Check sufficient funds
    IF v_from_bal < p_amount THEN
        RAISE_APPLICATION_ERROR(-20001,
            'Insufficient balance in account ' || p_from_acct);
    END IF;

    -- 3. Debit source, credit destination
    UPDATE accounts
    SET    balance = balance - p_amount
    WHERE  account_id = p_from_acct;

    UPDATE accounts
    SET    balance = balance + p_amount
    WHERE  account_id = p_to_acct;

    DBMS_OUTPUT.PUT_LINE('Transferred ' || p_amount ||
        ' from ' || p_from_acct ||
        ' to ' || p_to_acct || '.');

    COMMIT;
END;
/
```

Output:-

Procedure TRANSFERFUNDS compiled

Elapsed: 00:00:00.021

Test call:-

```
SET SERVEROUTPUT ON;

-- Successful transfer
EXEC TransferFunds(1002, 1001, 500);

-- Attempt transfer that will fail (not enough money)
BEGIN
    TransferFunds(1002, 1001, 100000);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

/

-- Verify balances
SELECT account_id, balance FROM accounts ORDER BY account_id;
```

Output:-

ACCOUNT_ID BALANCE	

1001	5550
1002	1500
1003	12120

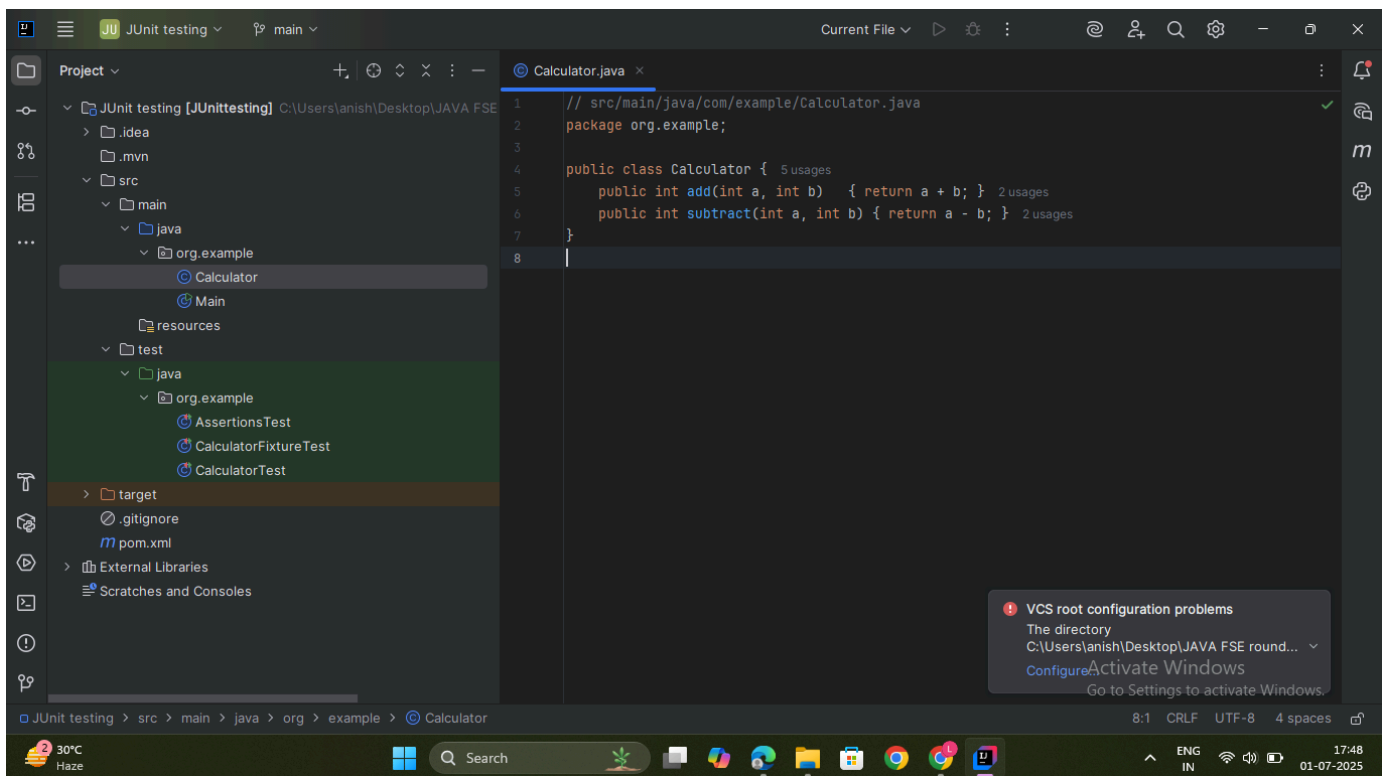
Elapsed: 00:00:00.003
3 rows selected.

JUnit Testing Exercises

Exercise 1: Setting Up JUnit

Scenario: You need to set up JUnit in your Java project to start writing unit tests.

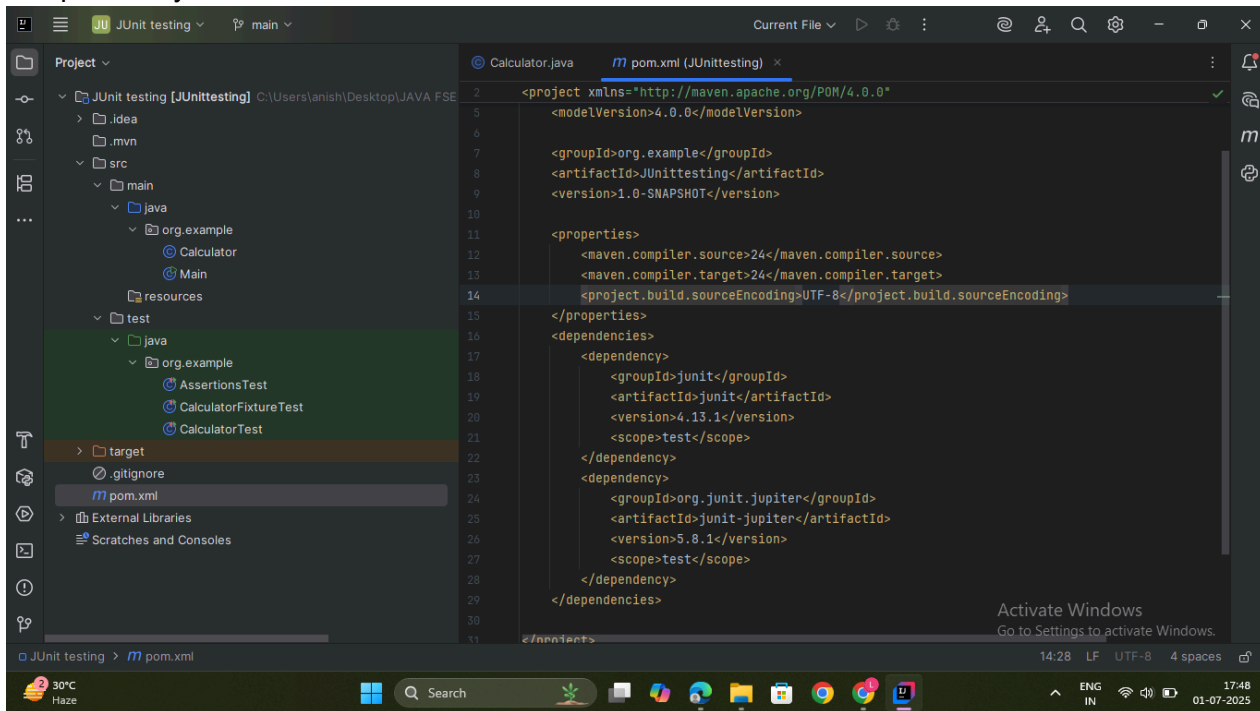
Steps: 1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).



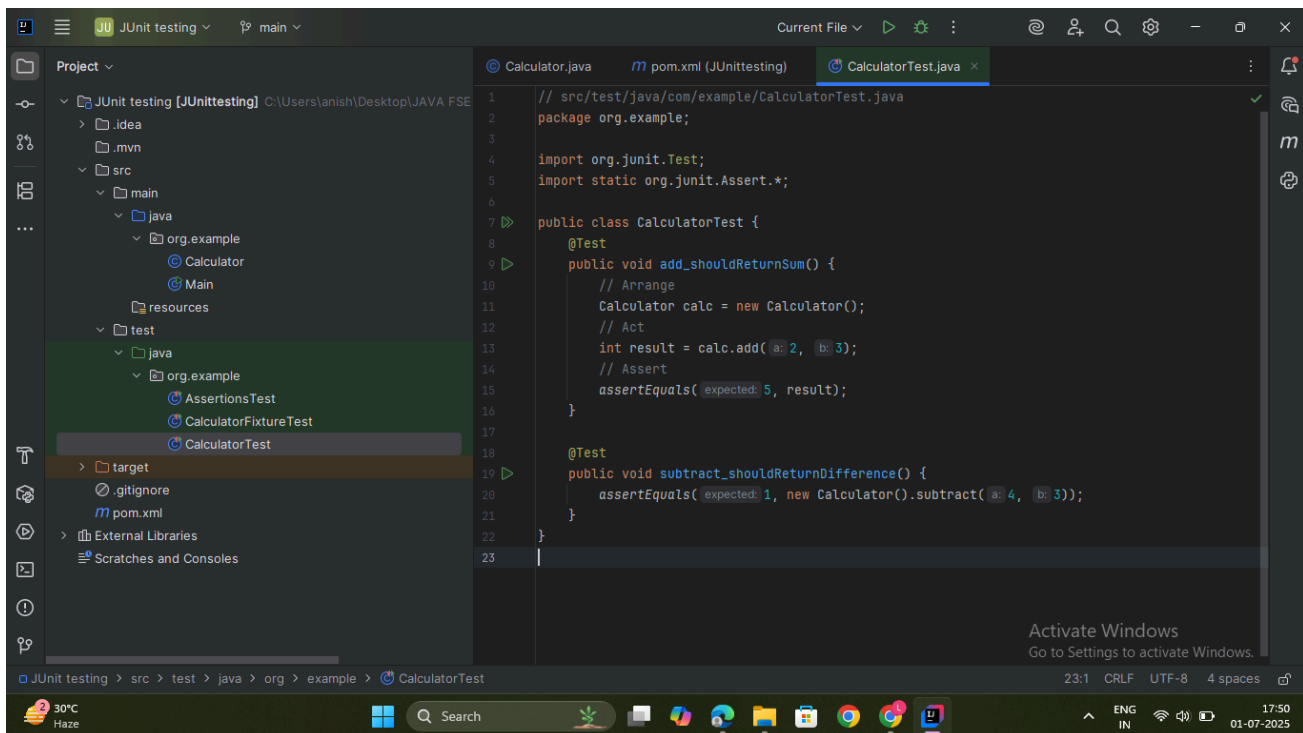
2. Add JUnit dependency to your project. If you are using Maven, add the following to your

pom.xml:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.13.2</version>
<scope>test</scope>
</dependency>
```



3. Create a new test class in your project.



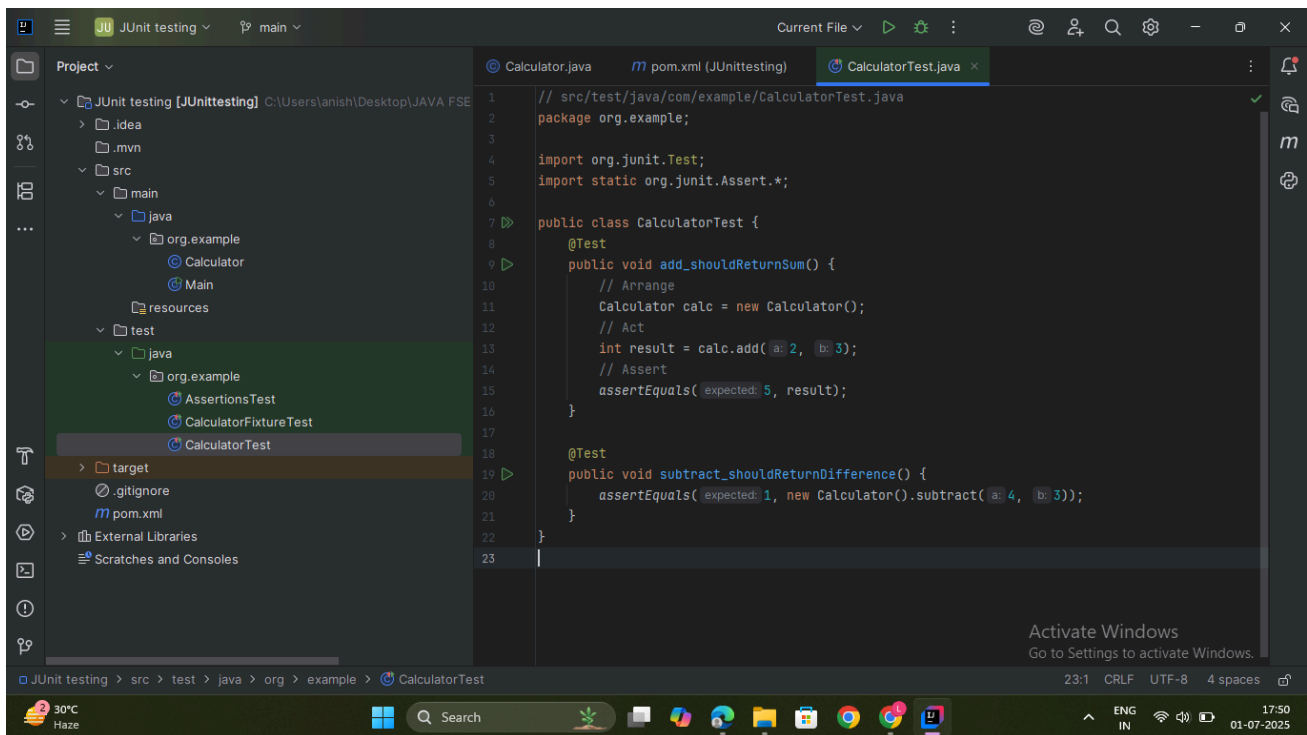
Exercise 2: Writing Basic JUnit Tests

Scenario:

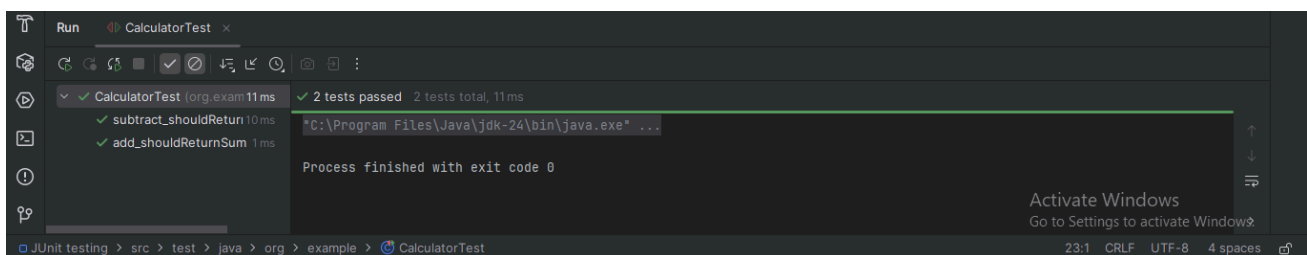
You need to write basic JUnit tests for a simple Java class.

Steps:

1. Create a new Java class with some methods to test.
2. Write JUnit tests for these methods.



Output:-



Exercise 3: Assertions in JUnit

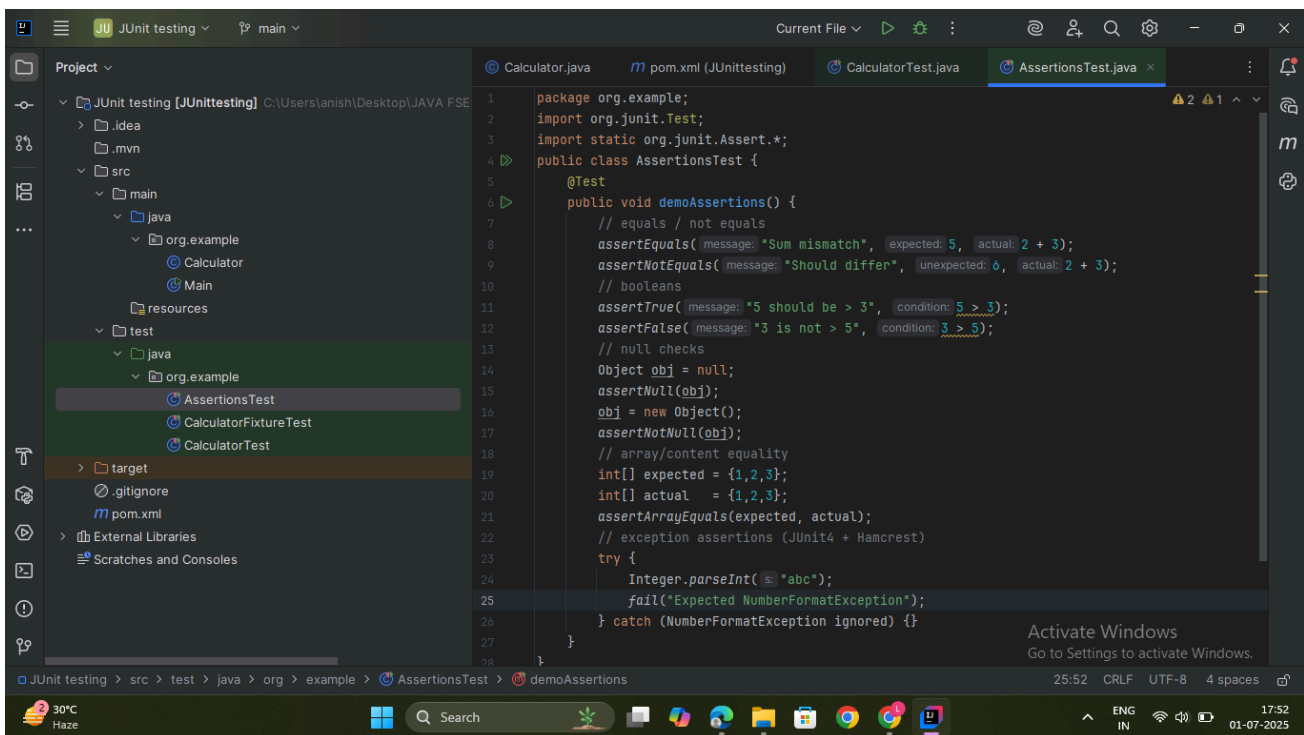
Scenario:

You need to use different assertions in JUnit to validate your test results. Steps:

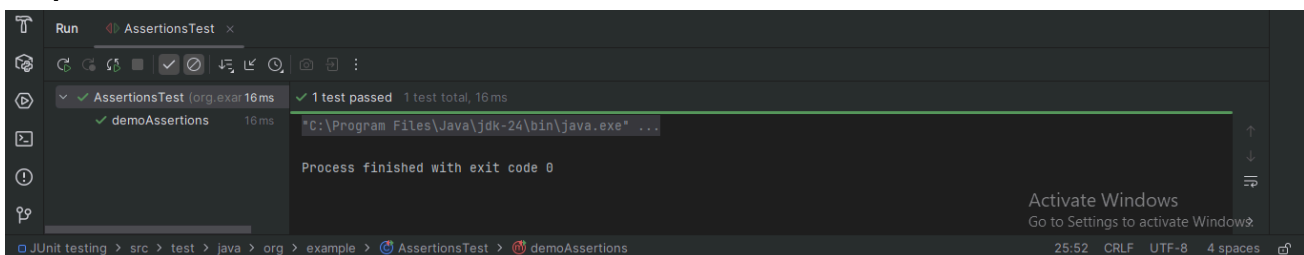
1. Write tests using various JUnit assertions.

Solution Code:

```
public class AssertionsTest {
    @Test
    public void testAssertions() {
        // Assert equals
        assertEquals(5, 2 + 3);
        // Assert true
        assertTrue(5 > 3);
        // Assert false
        assertFalse(5 < 3);
        // Assert null
        assertNull(null);
        // Assert not null
        assertNotNull(new Object());
    }
}
```



Output:-



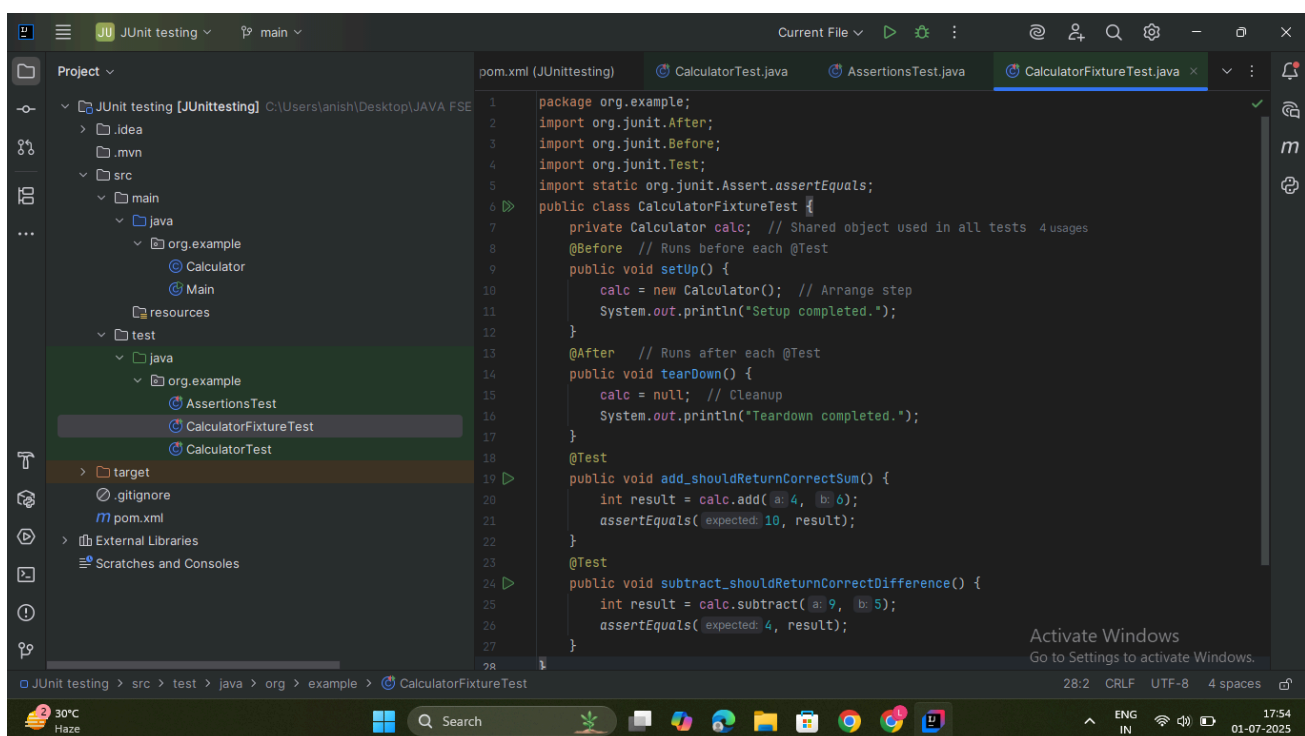
Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit

Scenario:

You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.

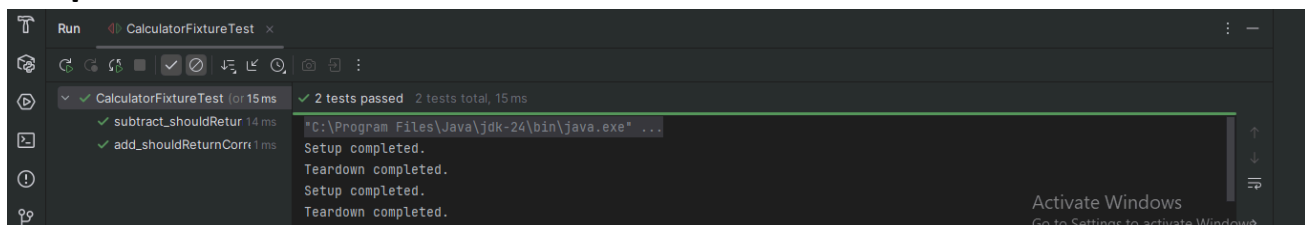
Steps:

1. Write tests using the AAA pattern.
2. Use `@Before` and `@After` annotations for setup and teardown methods.



```
1 package org.example;
2 import org.junit.After;
3 import org.junit.Before;
4 import org.junit.Test;
5 import static org.junit.Assert.assertEquals;
6 public class CalculatorFixtureTest {
7     private Calculator calc; // Shared object used in all tests 4 usages
8     @Before // Runs before each @Test
9     public void setUp() {
10         calc = new Calculator(); // Arrange step
11         System.out.println("Setup completed.");
12     }
13     @After // Runs after each @Test
14     public void tearDown() {
15         calc = null; // Cleanup
16         System.out.println("Teardown completed.");
17     }
18     @Test
19     public void add_shouldReturnCorrectSum() {
20         int result = calc.add(a: 4, b: 6);
21         assertEquals("expected: 10, result);", result);
22     }
23     @Test
24     public void subtract_shouldReturnCorrectDifference() {
25         int result = calc.subtract(a: 9, b: 5);
26         assertEquals("expected: 4, result);", result);
27     }
28 }
```

Output:-



```
Run CalculatorFixtureTest x
2 tests passed 2 tests total, 15 ms
[C:\Program Files\Java\jdk-24\bin\java.exe] ...
Setup completed.
Teardown completed.
Setup completed.
Teardown completed.
```

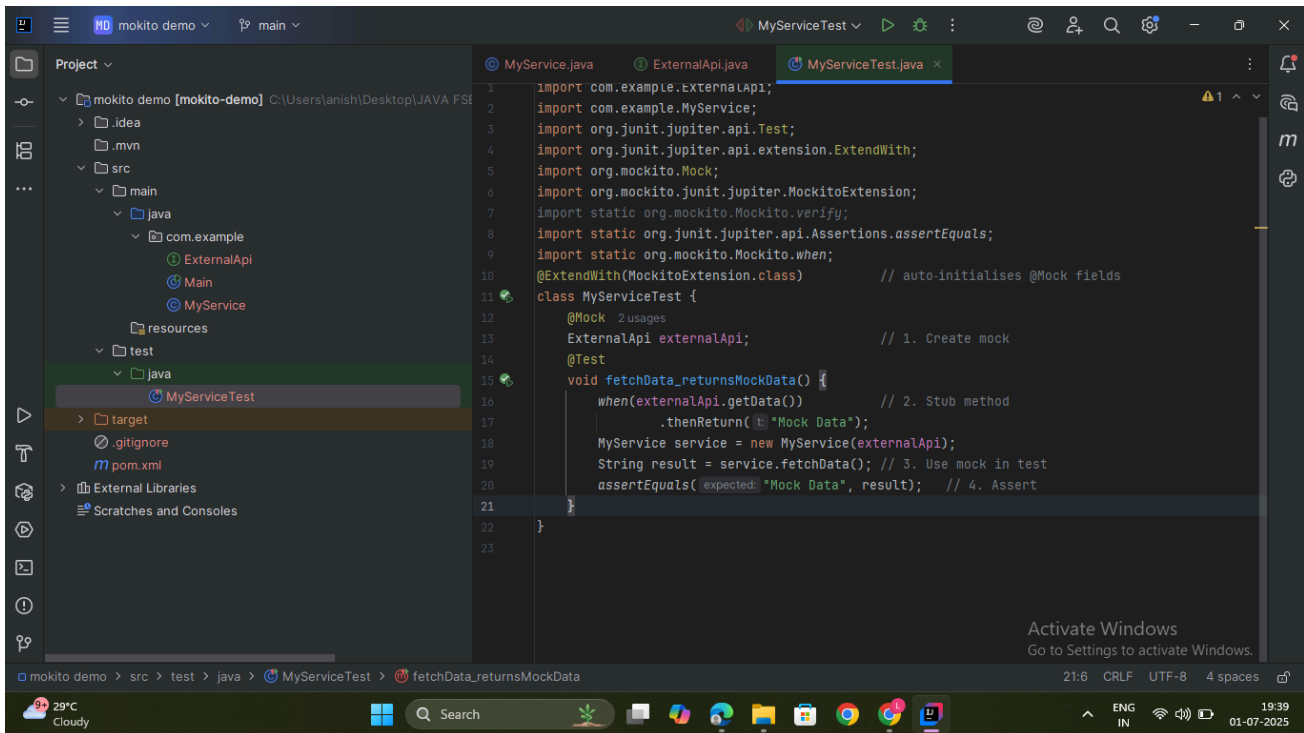
Mockito Hands-On Exercises

Exercise 1: Mocking and Stubbing

Scenario: You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

Steps:

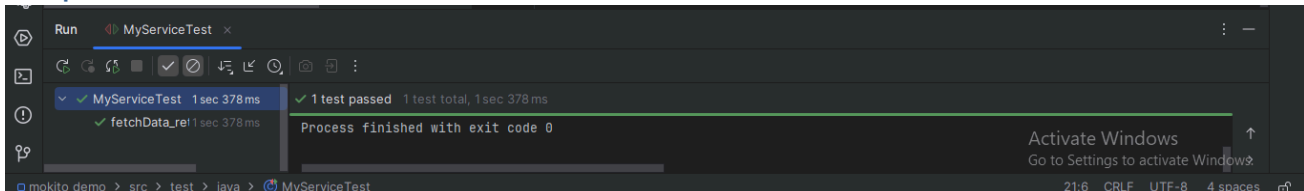
1. Create a mock object for the external API.
2. Stub the methods to return predefined values.
3. Write a test case that uses the mock object.



The screenshot shows an IDE with a project named 'mokito demo'. The project structure includes a 'test' directory with a 'java' subdirectory containing 'MyServiceTest'. The code in 'MyServiceTest.java' is as follows:

```
1 import com.example.ExternalApi;
2 import com.example.MyService;
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.extension.ExtendWith;
5 import org.mockito.Mock;
6 import org.mockito.junit.jupiter.MockitoExtension;
7 import static org.mockito.Mockito.verify;
8 import static org.junit.jupiter.api.Assertions.assertEquals;
9 import static org.mockito.Mockito.when;
10 @ExtendWith(MockitoExtension.class) // auto-initialises @Mock fields
11 class MyServiceTest {
12     @Mock // 2 usages
13     ExternalApi externalApi; // 1. Create mock
14     @Test
15     void fetchData_returnsMockData() {
16         when(externalApi.getData()) // 2. Stub method
17             .thenReturn("Mock Data");
18         MyService service = new MyService(externalApi);
19         String result = service.fetchData(); // 3. Use mock in test
20         assertEquals("expected: 'Mock Data', result"); // 4. Assert
21     }
22 }
```

Output:-



The screenshot shows the 'Run' window of the IDE. The test 'MyServiceTest' has passed. The output is as follows:

```
Run MyServiceTest
MyServiceTest 1 sec 378 ms
fetchData_re 1 sec 378 ms
1 test passed 1 test total, 1 sec 378 ms
Process finished with exit code 0
```

Exercise 2: Verifying Interactions

Scenario:

You need to ensure that a method is called with specific arguments.

Steps:

1. Create a mock object.
2. Call the method with specific arguments.
3. Verify the interaction.

```
1 import com.example.ExternalApi;
2 import com.example.MyService;
3 import org.junit.jupiter.api.Test;
4 import org.junit.jupiter.api.extension.ExtendWith;
5 import org.mockito.Mock;
6 import org.mockito.junit.jupiter.MockitoExtension;
7 import static org.mockito.Mockito.verify;
8 import static org.junit.jupiter.api.Assertions.assertEquals;
9 import static org.mockito.Mockito.when;
10 @ExtendWith(MockitoExtension.class) // auto-initialises @Mock fields
11 class MyServiceTest {
12     @Mock // 4 usages
13     ExternalApi externalApi; // 1. Create mock
14     @Test
15     void fetchData_returnsMockData() {
16         when(externalApi.getData()) { // 2. Stub method
17             .thenReturn("Mock Data");
18         }
19         MyService service = new MyService(externalApi);
20         String result = service.fetchData(); // 3. Use mock in test
21         assertEquals("Mock Data", result); // 4. Assert
22     }
23     @Test
24     void fetchData_callsExternalApiOnce() {
25         MyService service = new MyService(externalApi);
26         service.fetchData(); // call method under test
27         verify(externalApi).getData(); // Did we call getData() exactly once?
28     }
29 }
```

Output:-

```
Run MyServiceTest
MyServiceTest 1 sec 499 ms
  ✓ fetchData_callsExternalApiOnce 1 sec 477 ms
  ✓ fetchData_returnsMockData 22 ms
2 tests passed 2 tests total, 1 sec 499 ms
Process finished with exit code 0
Activate Windows
Go to Settings to activate Windows
```