

## WEEK-2

### PL/SQL,JUnit\_Basic Testing,Mockito exercises &

### SL4 Logging

### PL-SQL

#### 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.

**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.

**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.

#### CODE:

```
SET SERVEROUTPUT ON;

BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE loans';
EXCEPTION WHEN OTHERS THEN NULL;
END;

/

BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE customers';
```

```

EXCEPTION WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE customers (
    cust_id  NUMBER PRIMARY KEY,
    age      NUMBER,
    balance  NUMBER,
    vip_flag VARCHAR2(5)
);

CREATE TABLE loans (
    loan_id  NUMBER PRIMARY KEY,
    cust_id  NUMBER,
    int_rate NUMBER,
    due_on   DATE,
    FOREIGN KEY (cust_id) REFERENCES customers(cust_id)
);

INSERT INTO customers VALUES (1, 65, 12000, 'FALSE');
INSERT INTO customers VALUES (2, 45, 8000, 'FALSE');
INSERT INTO customers VALUES (3, 70, 15000, 'FALSE');


INSERT INTO loans VALUES (101, 1, 10, TO_DATE('04-JUL-2025','DD-MON-YYYY'));
INSERT INTO loans VALUES (102, 2, 9, TO_DATE('01-SEP-2025','DD-MON-YYYY'));
INSERT INTO loans VALUES (103, 3, 8, TO_DATE('29-JUN-2025','DD-MON-YYYY'));

COMMIT;

BEGIN

FOR loan_rec IN (
    SELECT l.loan_id, l.cust_id, l.int_rate
    FROM loans l
    JOIN customers c ON l.cust_id = c.cust_id
    WHERE c.age > 60
)

```

LOOP

UPDATE loans

SET int\_rate = int\_rate - 1

WHERE loan\_id = loan\_rec.loan\_id;

DBMS\_OUTPUT.PUT\_LINE(

'Scenario 1: 1% interest discount applied on Loan ' || loan\_rec.loan\_id ||

' (Customer ID ' || loan\_rec.cust\_id || ' )'

);

END LOOP;

FOR cust\_rec IN (

SELECT cust\_id, balance FROM customers

WHERE balance > 10000

)

LOOP

UPDATE customers

SET vip\_flag = 'TRUE'

WHERE cust\_id = cust\_rec.cust\_id;

DBMS\_OUTPUT.PUT\_LINE(

' Scenario 2: VIP status set for Customer ' || cust\_rec.cust\_id ||

' (Balance: \$' || cust\_rec.balance || ' )'

);

END LOOP;

FOR due\_rec IN (

SELECT loan\_id, cust\_id, due\_on

FROM loans

WHERE due\_on BETWEEN SYSDATE AND SYSDATE + 30

)

LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Scenario 3: Reminder - Loan ' || due\_rec.loan\_id ||

```

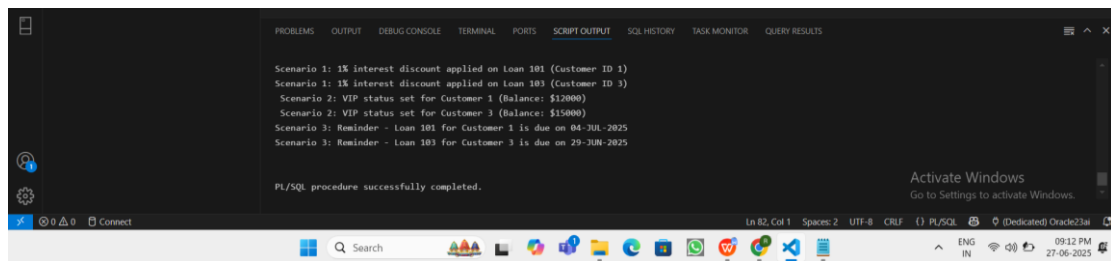
        ' for Customer ' || due_rec.cust_id ||
        ' is due on ' || TO_CHAR(due_rec.due_on, 'DD-MON-YYYY')
    );
END LOOP;

COMMIT;

END;
/

```

## OUTPUT:



Scenario 1: 1% interest discount applied on Loan 101 (Customer ID 1)

Scenario 1: 1% interest discount applied on Loan 103 (Customer ID 3)

Scenario 2: VIP status set for Customer 1 (Balance: \$12000)

Scenario 2: VIP status set for Customer 3 (Balance: \$15000)

Scenario 3: Reminder - Loan 101 for Customer 1 is due on 04-JUL-2025

Scenario 3: Reminder - Loan 103 for Customer 3 is due on 29-JUN-2025

PL/SQL procedure successfully completed.

## **Exercise 3: Stored Procedures**

**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.

**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.

**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.

### **CODE:**

```
SET SERVEROUTPUT ON;

BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE accounts';
EXCEPTION WHEN OTHERS THEN NULL;
END;

/

BEGIN
    EXECUTE IMMEDIATE 'DROP TABLE employees';
EXCEPTION WHEN OTHERS THEN NULL;
END;
```

/

```
CREATE TABLE accounts (  
    account_id  NUMBER PRIMARY KEY,  
    customer_id NUMBER,  
    balance     NUMBER,  
    account_type VARCHAR2(20)  
);
```

```
CREATE TABLE employees (  
    emp_id  NUMBER PRIMARY KEY,  
    name    VARCHAR2(50),  
    department VARCHAR2(50),  
    salary  NUMBER  
);
```

```
INSERT INTO accounts VALUES (101, 1, 10000, 'SAVINGS');  
INSERT INTO accounts VALUES (102, 2, 15000, 'CURRENT');  
INSERT INTO accounts VALUES (103, 3, 20000, 'SAVINGS');  
INSERT INTO employees VALUES (1, 'Ravi', 'Sales', 40000);  
INSERT INTO employees VALUES (2, 'Sneha', 'Finance', 45000);  
INSERT INTO employees VALUES (3, 'Ajith', 'Sales', 42000);
```

```
COMMIT;
```

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS  
BEGIN
```

```
    UPDATE accounts  
    SET balance = balance + (balance * 0.01)  
    WHERE UPPER(account_type) = 'SAVINGS';
```

```
    DBMS_OUTPUT.PUT_LINE('Interest applied to all savings accounts.');
```

```
    COMMIT;
```

```
END;
```

/

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (  
    p_dept    IN VARCHAR2,  
    p_bonus_pct IN NUMBER  
) IS  
BEGIN  
    UPDATE employees  
    SET salary = salary + (salary * p_bonus_pct / 100)  
    WHERE LOWER(department) = LOWER(p_dept);  
  
    DBMS_OUTPUT.PUT_LINE('Bonus of ' || p_bonus_pct || '% applied to ' || p_dept  
    || ' department.');
```

COMMIT;

END;

/

```
CREATE OR REPLACE PROCEDURE TransferFunds (  
    p_from_account IN NUMBER,  
    p_to_account   IN NUMBER,  
    p_amount       IN NUMBER  
) IS  
    v_balance NUMBER;  
BEGIN  
  
    SELECT balance INTO v_balance  
    FROM accounts  
    WHERE account_id = p_from_account;  
  
    IF v_balance < p_amount THEN  
        RAISE_APPLICATION_ERROR(-20001, 'Not enough balance in source account.');
```

END IF;

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account;

DBMS\_OUTPUT.PUT\_LINE('₹' || p\_amount || ' transferred from Account ' ||  
p\_from\_account || ' to Account ' || p\_to\_account);

COMMIT;

END;

/

BEGIN

DBMS\_OUTPUT.PUT\_LINE('----- Executing ProcessMonthlyInterest -----');

ProcessMonthlyInterest;

DBMS\_OUTPUT.PUT\_LINE('----- Executing UpdateEmployeeBonus (Sales, 10%) -----');

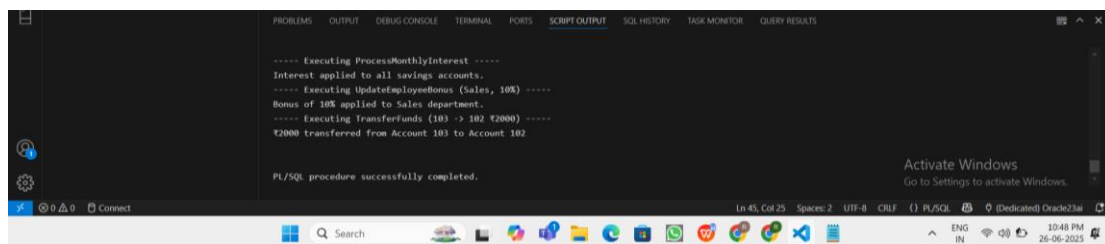
UpdateEmployeeBonus('Sales', 10);

DBMS\_OUTPUT.PUT\_LINE('----- Executing TransferFunds (103 -> 102 ₹2000) -----');

TransferFunds(103, 102, 2000);

END;

## OUTPUT



```
----- Executing ProcessMonthlyInterest -----  
Interest applied to all savings accounts.  
----- Executing UpdateEmployeeBonus (Sales, 10%) -----  
Bonus of 10% applied to Sales department.  
----- Executing TransferFunds (103 -> 102 ₹2000) -----  
₹2000 transferred from Account 103 to Account 102  
  
PL/SQL procedure successfully completed.
```



# JUnit Basic Testing

## 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

CODE:

### **Calculator.java**

```
package com.example;

public class Calculator {

    public int add(int a, int b) {

        System.out.println("Adding numbers: " + a + " + " + b);

        int result = a + b;

        System.out.println("Computed result: " + result);

        return result;

    }

}
```

```

public int subtract(int a, int b) {
    System.out.println("Subtracting numbers: " + a + " - " + b);
    int result = a - b;
    System.out.println("Computed result: " + result);
    return result;
}

public int multiply(int a, int b) {
    System.out.println("Multiplying numbers: " + a + " * " + b);
    int result = a * b;
    System.out.println("Computed result: " + result);
    return result;
}
}

```

## CalculatorTest.java

```

package com.example;

import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        System.out.println("---- Testing add() ----");
        int result = calc.add(2, 3);
        assertEquals(5, result);
        System.out.println("Test add() passed.\n");
    }

    @Test
    public void testSubtract() {
        Calculator calc = new Calculator();
        System.out.println("---- Testing subtract() ----");
        int result = calc.subtract(10, 4);
    }
}

```

```

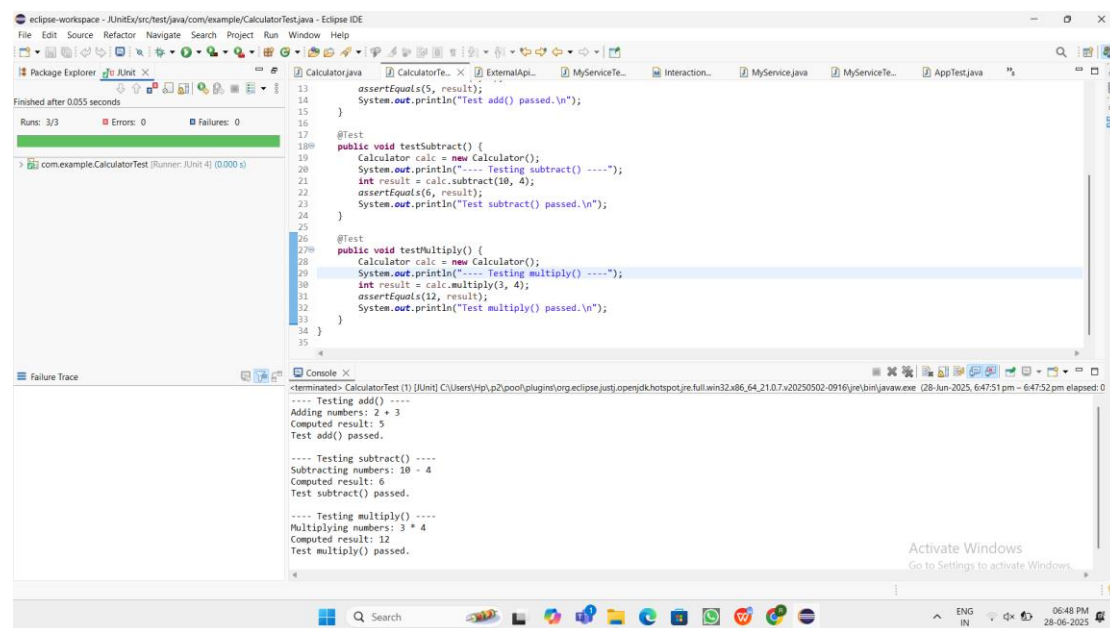
assertEquals(6, result);

System.out.println("Test subtract() passed.\n");
}

@Test
public void testMultiply() {
    Calculator calc = new Calculator();
    System.out.println("---- Testing multiply() ----");
    int result = calc.multiply(3, 4);
    assertEquals(12, result);
    System.out.println("Test multiply() passed.\n");
}

```

## 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());  
    }  
}
```

CODE:

### **AssertionsTest.java**

```
package com.example;  
import org.junit.Test;  
import static org.junit.Assert.*;  
public class AssertionsTest {  
    @Test  
    public void testAssertions() {
```

```

assertEquals(5, 2 + 3);

System.out.println("assertEquals passed");

assertTrue(5 > 3);

System.out.println("assertTrue passed");

assertFalse(5 < 3);

System.out.println("assertFalse passed");

assertNull(null);

System.out.println("assertNull passed");

assertNotNull(new Object());

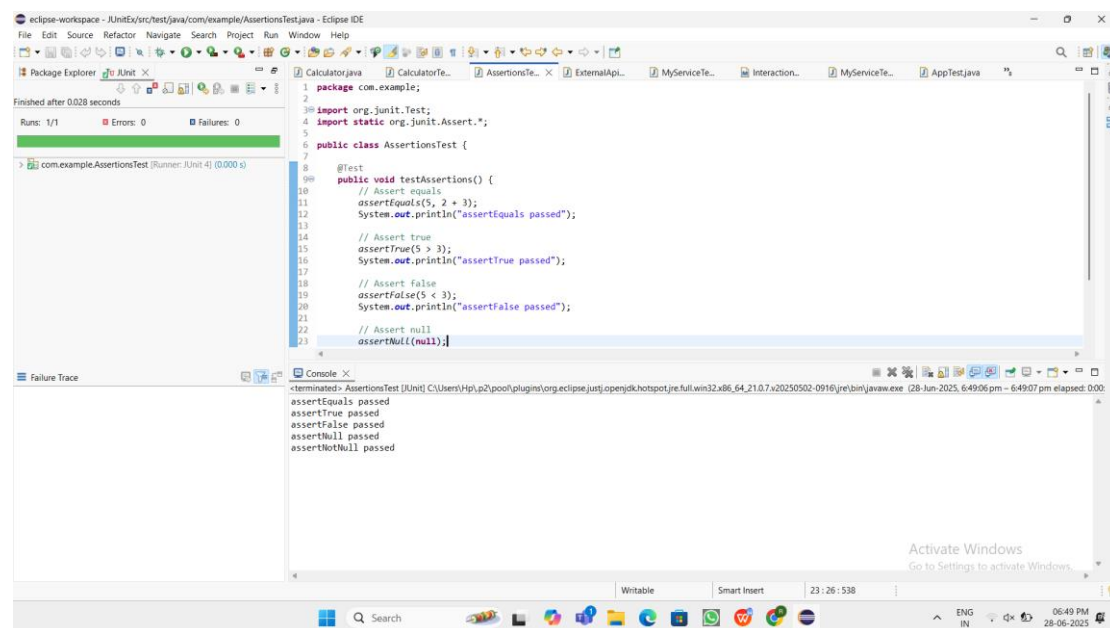
System.out.println("assertNotNull passed");

}

}

```

## 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.

CODE:

**CalculatorTestAAA.java**

```
package com.example;
```

```
import org.junit.Before;
```

```
import org.junit.After;
```

```
import org.junit.Test;
```

```
import static org.junit.Assert.*;
```

```
public class CalculatorTestAAA {
```

```
    private Calculator calculator;
```

```
    @Before
```

```
    public void setUp() {
```

```
        calculator = new Calculator();
```

```
        System.out.println("setUp: Calculator instance created");
```

```
    }
```

@After

```
public void tearDown() {
```

```
System.out.println("tearDown: Test completed\n");
```

```
}@Test
```

```
public void testAddition() {
```

```
int a = 10;
```

```
int b = 5;
```

```
int result = calculator.add(a, b);
```

```
assertEquals(15, result);
```

```
System.out.println("testAddition: " + a + " + " + b + " = " + result);
```

```
}
```

@Test

```
public void testSubtraction() {
```

```
int a = 8;
```

```
int b = 3;
```

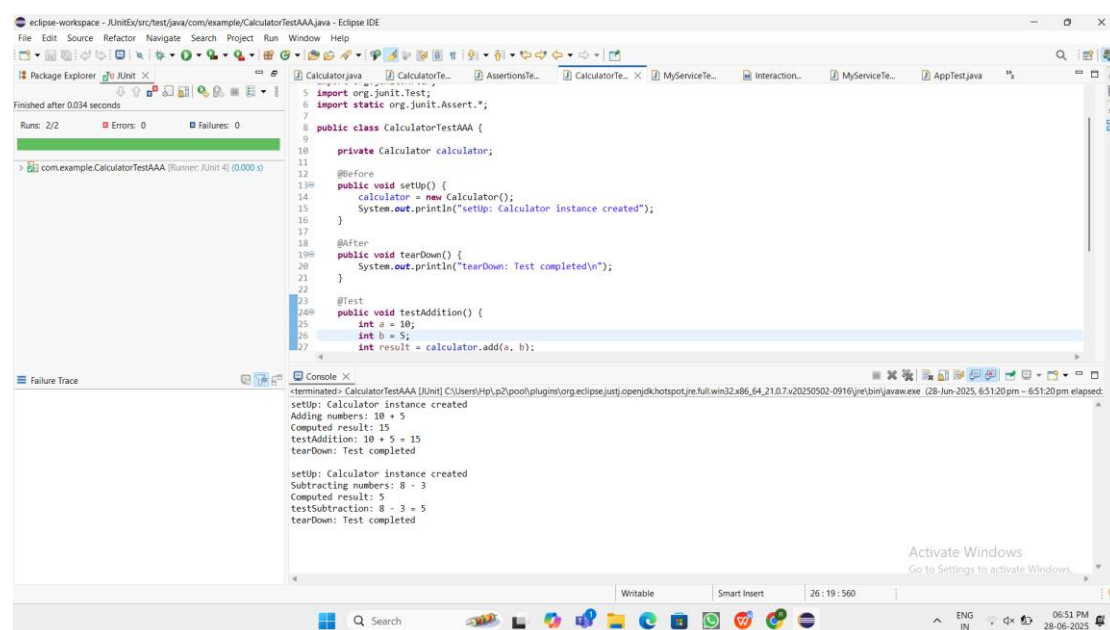
```
int result = calculator.subtract(a, b);
```

```
assertEquals(5, result);
```

```
System.out.println("testSubtraction: " + a + " - " + b + " = " + result);
```

```
}
```

## OUTPUT:



The screenshot shows the Eclipse IDE interface. The main editor displays the `CalculatorTestAAA.java` file with the following code:

```
5 import org.junit.Test;
6 import static org.junit.Assert.*;
7
8 public class CalculatorTestAAA {
9
10     private Calculator calculator;
11
12     @Before
13     public void setUp() {
14         calculator = new Calculator();
15         System.out.println("setUp: Calculator instance created");
16     }
17
18     @After
19     public void tearDown() {
20         System.out.println("tearDown: Test completed\n");
21     }
22
23     @Test
24     public void testAddition() {
25         int a = 10;
26         int b = 5;
27         int result = calculator.add(a, b);
```

The Package Explorer on the left shows the project structure, and the Console at the bottom displays the output of the test execution:

```
<terminated> CalculatorTestAAA [JUnit C:\Users\Hp\p2\pool\plugins\org.eclipse.jdt.openjdk.hotspot.jre.full.win32.x86_64_21.0.7.v20250502-0916\jre\bin\javaw.exe (28-Jun-2025, 6:51:20 pm - 6:51:20 pm elapsed):
setUp: Calculator instance created
Adding numbers: 10 + 5
Computed result: 15
testAddition: 10 + 5 = 15
tearDown: Test completed

setUp: Calculator instance created
Subtracting numbers: 8 - 3
Computed result: 5
testSubtraction: 8 - 3 = 5
tearDown: Test completed
```

The status bar at the bottom indicates the current time is 26:19:560 and the date is 28-06-2025.

# 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.

Solution Code:

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

public class MyServiceTest {

    @Test
    public void testExternalApi() {
        ExternalApi mockApi = Mockito.mock(ExternalApi.class);
        when(mockApi.getData()).thenReturn("Mock Data");

        MyService service = new MyService(mockApi);

        String result = service.fetchData();
        assertEquals("Mock Data", result);
    }
}
```

CODE:

### **ExternalApi.java**

```
package com.example.MockDemo;

public interface ExternalApi {

    String getData();

}
```



## MyService.java

```
package com.example.MockDemo;

public class MyService {
    private final ExternalApi externalApi;

    public MyService(ExternalApi externalApi) {
        this.externalApi = externalApi;
    }

    public String fetchData() {
        return externalApi.getData();
    }
}
```

## MyServiceTest.java

```
package com.example.MockDemo;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

public class MyServiceTest {

    @Test
    public void testFetchData() {
        // Arrange (Mock setup)
        ExternalApi mockApi = mock(ExternalApi.class);
        when(mockApi.getData()).thenReturn("Mock Data");
    }
}
```

```

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

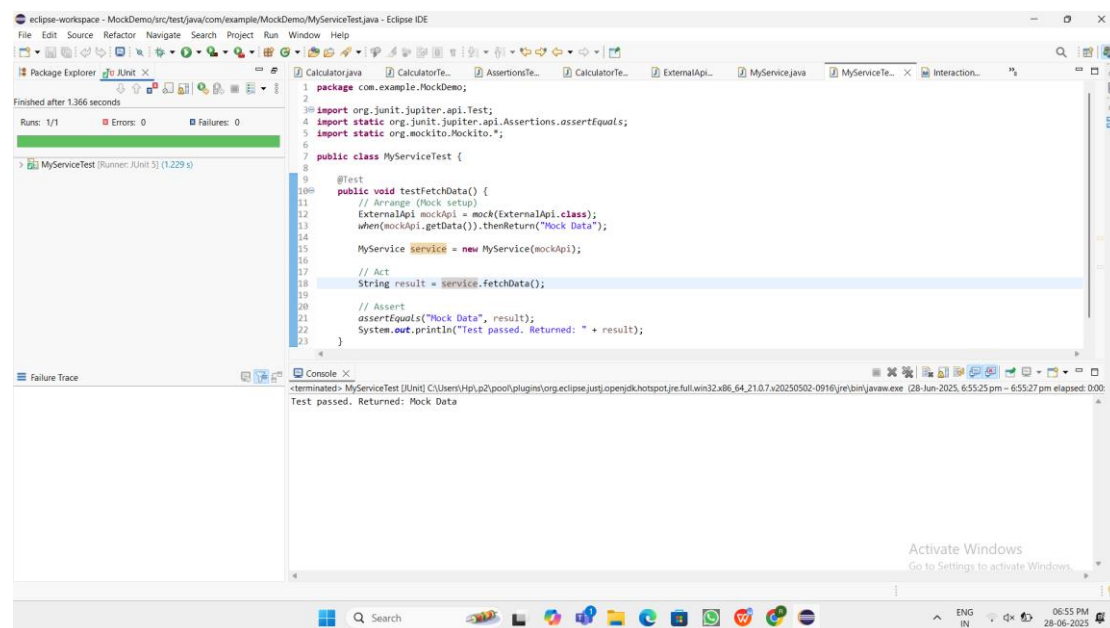
System.out.println("Test passed. Returned: " + result);

}

}

```

## OUTPUT:



## **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.

Solution Code:

```
import static org.mockito.Mockito.*; import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

    @Test

    public void testVerifyInteraction() {

        ExternalApi mockApi = Mockito.mock(ExternalApi.class);

        MyService service = new MyService(mockApi);

        service.fetchData();

        verify(mockApi).getData();
    }
}
```

CODE:

### **ExternalApi.java**

```
package com.example.InteractionVerifier;

public interface ExternalApi {

    String getData();

}
```

## **MyService.java**

```
package com.example.InteractionVerifier;

public class MyService {
    private ExternalApi externalApi;

    public MyService(ExternalApi externalApi) {
        this.externalApi = externalApi;
    }

    public String fetchData() {
        return externalApi.getData();
    }
}
```

## **MyServiceTest.java**

```
package com.example.InteractionVerifier;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.*;

public class MyServiceTest {

    @Test
    public void testVerifyInteraction() {
        ExternalApi mockApi = mock(ExternalApi.class);
```

```
when(mockApi.getData()).thenReturn("Mocked Interaction Data");
```

```
MyService service = new MyService(mockApi);
```

```
String result = service.fetchData();
```

```
assertEquals("Mocked Interaction Data", result);
```

```
System.out.println("fetchData() returned: " + result);
```

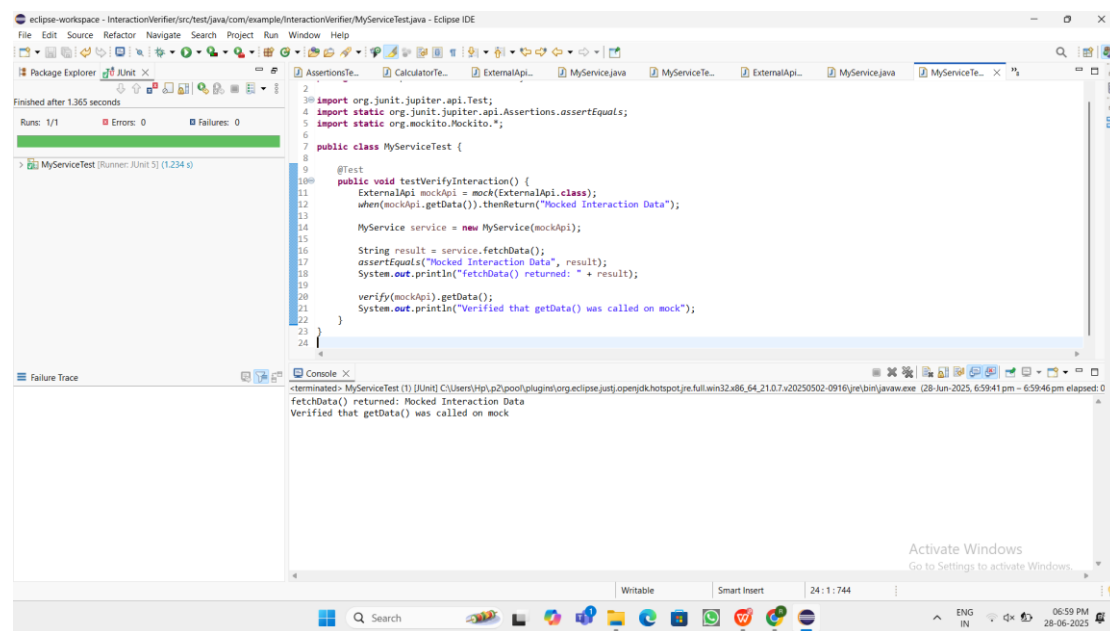
```
verify(mockApi).getData();
```

```
System.out.println("Verified that getData() was called on mock");
```

```
}
```

```
}
```

## OUTPUT:



# Logging using SLF4J

## **Exercise 1: Logging Error Messages and Warning Levels**

Task: Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

Step-by-Step Solution:

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

```
<dependency>  
<groupId>org.slf4j</groupId>  
<artifactId>slf4j-api</artifactId>  
<version>1.7.30</version>  
</dependency>  
<dependency>  
<groupId>ch.qos.logback</groupId>  
<artifactId>logback-classic</artifactId>  
<version>1.2.3</version>  
</dependency>
```

2. Create a Java class that uses SLF4J for logging:

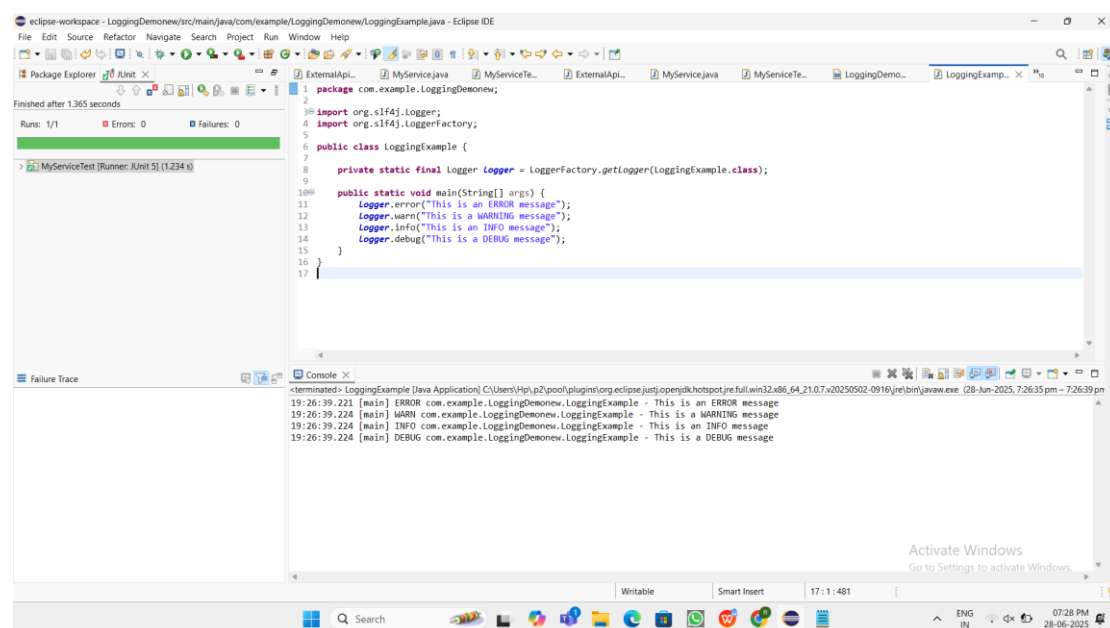
```
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
public class LoggingExample {  
    private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);  
    public static void main(String[] args) {  
        logger.error("This is an error message");  
        logger.warn("This is a warning message");  
    }  
}
```

CODE:

## LoggingExample.java

```
package com.example.LoggingDemonew;  
  
import org.slf4j.Logger;  
  
import org.slf4j.LoggerFactory;  
  
public class LoggingExample {  
  
    private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);  
  
    public static void main(String[] args) {  
  
        logger.error("This is an ERROR message");  
  
        logger.warn("This is a WARNING message");  
  
        logger.info("This is an INFO message");  
  
        logger.debug("This is a DEBUG message");  
  
    }  
  
}
```

OUTPUT:



The screenshot shows the Eclipse IDE interface. The main editor displays the source code of `LoggingExample.java`. The Package Explorer on the left shows the project structure. The Console window at the bottom displays the output of the program, showing four log messages: ERROR, WARN, INFO, and DEBUG. The messages are as follows:

```
<main> com.example.LoggingDemonew.LoggingExample - This is an ERROR message  
19:26:39.224 [main] WARN com.example.LoggingDemonew.LoggingExample - This is a WARNING message  
19:26:39.224 [main] INFO com.example.LoggingDemonew.LoggingExample - This is an INFO message  
19:26:39.224 [main] DEBUG com.example.LoggingDemonew.LoggingExample - This is a DEBUG message
```