WEEK 2

Exercise 1: Control Structures

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

o Ques on: 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.

SET SERVEROUTPUT ON;

BEGIN

FOR cust IN (

SELECT CustomerID, LoanID

FROM Customers

WHERE Age > 60

) LOOP

UPDATE Loans

SET InterestRate = InterestRate - 1

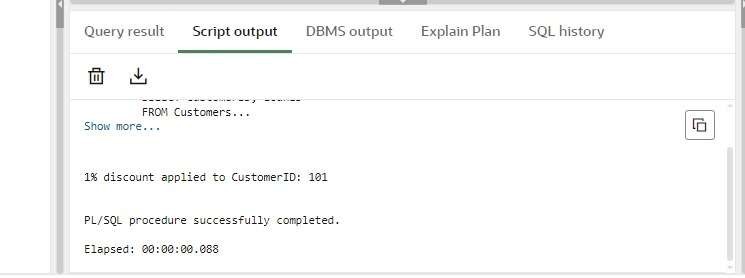
WHERE LoanID = cust.LoanID;

DBMS\_OUTPUT.PUT\_LINE('1% discount applied to CustomerID: ' || cust.CustomerID); END LOOP;

COMMIT;

END;

/



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

o Ques on: 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.

SET SERVEROUTPUT ON;

BEGIN -- Loop through customers with balance over 10,000

FOR cust IN (

SELECT CustomerID

FROM Customers

WHERE Balance > 10000

) LOOP -- Update IsVIP flag to 'Y' (TRUE)

UPDATE Customers

SET IsVIP = 'Y'

WHERE CustomerID = cust.CustomerID; -- Print confirma on message

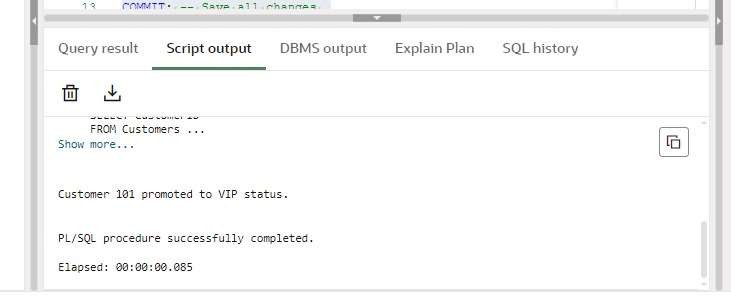
DBMS\_OUTPUT.PUT\_LINE('Customer ' || cust.CustomerID || ' promoted to VIP status.');

END LOOP;

COMMIT; -- Save all changes

END;

/



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

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

-- Drop the Loans table if it already exists

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Loans';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/ -- Create Loans table with correct columns

CREATE TABLE Loans (

LoanID NUMBER,

CustomerID NUMBER,

DueDate DATE

); -- Insert test data

INSERT INTO Loans VALUES (201, 101, SYSDATE + 10); -- Due soon

INSERT INTO Loans VALUES (202, 102, SYSDATE + 35); -- Too late

INSERT INTO Loans VALUES (203, 103, SYSDATE + 5); -- Due soon

COMMIT; -- Show reminders for loans due in next 30 days

BEGIN

FOR due\_rec IN (

SELECT LoanID, CustomerID, DueDate

FROM Loans

WHERE DueDate <= SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Loan ' || due\_rec.LoanID ||

' for Customer ' || due\_rec.CustomerID ||

' is due on ' || TO\_CHAR(due\_rec.DueDate, 'DD-MON-YYYY')

);

END LOOP;

END;

/



Exercise 3: Stored Procedures

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

o Ques on: 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.

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

Balance NUMBER,

AccountType VARCHAR2(20)

);

INSERT INTO Accounts VALUES (101, 1000, 'SAVINGS');

INSERT INTO Accounts VALUES (102, 2000, 'CHECKING');

INSERT INTO Accounts VALUES (103, 3000, 'SAVINGS');

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'SAVINGS'; -- Op onal: count updated rows

DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT || ' savings accounts updated.');

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

END;

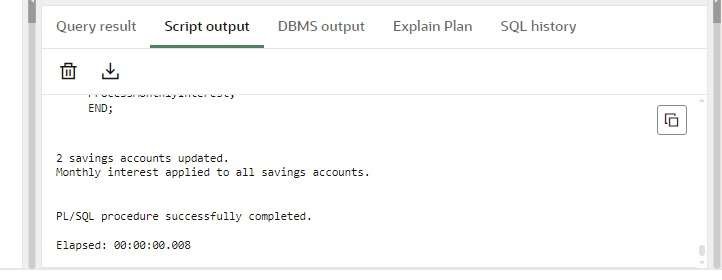
/

SET SERVEROUTPUT ON; BEGIN

ProcessMonthlyInterest;

END;

/



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

o Ques on: Write a stored procedure UpdateEmployeeBonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Employees';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

Name VARCHAR2(50),

Department VARCHAR2(30),

Salary NUMBER

);

INSERT INTO Employees VALUES (101, 'Alice', 'HR', 40000);

INSERT INTO Employees VALUES (102, 'Bob', 'HR', 45000);

INSERT INTO Employees VALUES (103, 'Charlie', 'IT', 60000);

INSERT INTO Employees VALUES (104, 'David', 'IT', 65000);

INSERT INTO Employees VALUES (105, 'Eva', 'Finance', 50000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus( p\_department IN VARCHAR2, p\_bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_bonus\_percent / 100)

WHERE Department = p\_department;

DBMS\_OUTPUT.PUT\_LINE('Bonus of ' || p\_bonus\_percent || '% applied to ' || p\_department || ' department.');

END;

/

BEGIN

UpdateEmployeeBonus('IT', 10);

END;

/

SELECT \* FROM Employees;



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

o Ques on: 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.

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE Accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

Balance NUMBER

);

INSERT INTO Accounts VALUES (101, 1, 5000); -- Source

INSERT INTO Accounts VALUES (102, 1, 2000); -- Target

COMMIT;

CREATE OR REPLACE PROCEDURE TransferFunds( p\_from\_account IN NUMBER, p\_to\_account IN NUMBER, p\_amount IN NUMBER

) IS

v\_balance NUMBER; insufficient\_funds EXCEPTION;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_from\_account

FOR UPDATE;

IF v\_balance < p\_amount THEN

RAISE insufficient\_funds;

END IF;

UPDATE Accounts

SET Balance = Balance - p\_amount

WHERE AccountID = p\_from\_account;

UPDATE Accounts

SET Balance = Balance + p\_amount

WHERE AccountID = p\_to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || p\_amount || ' from Account ' || p\_from\_account || ' to

Account ' || p\_to\_account);

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: Insufficient funds in Account ' || p\_from\_account);

ROLLBACK;

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Transfer failed: One or both accounts do not exist.');

ROLLBACK;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

ROLLBACK;

END;

/

BEGIN

TransferFunds(101, 102, 1500);

END;

/

SELECT \* FROM Accounts;



1. Setting Up JUnit Scenario:

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

Code:

CalculateMultiplication.java public class CalculateMultiplication

{ public int multiplication(int n, int m)

{

return n\*m;

}

}

CalculateMultiplyTest.java import org.junit.jupiter.api.Test; import static org.junit.jupiter.api.Assertions.assertEquals; public class CalculateMultiplyTest

{

@Test

void testMultiply()

{

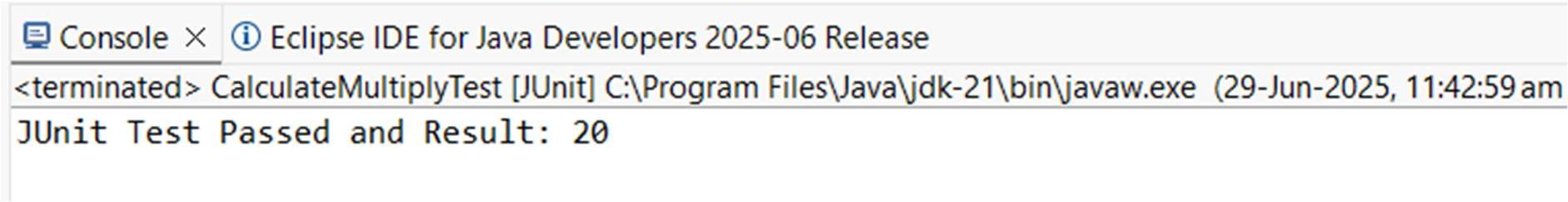
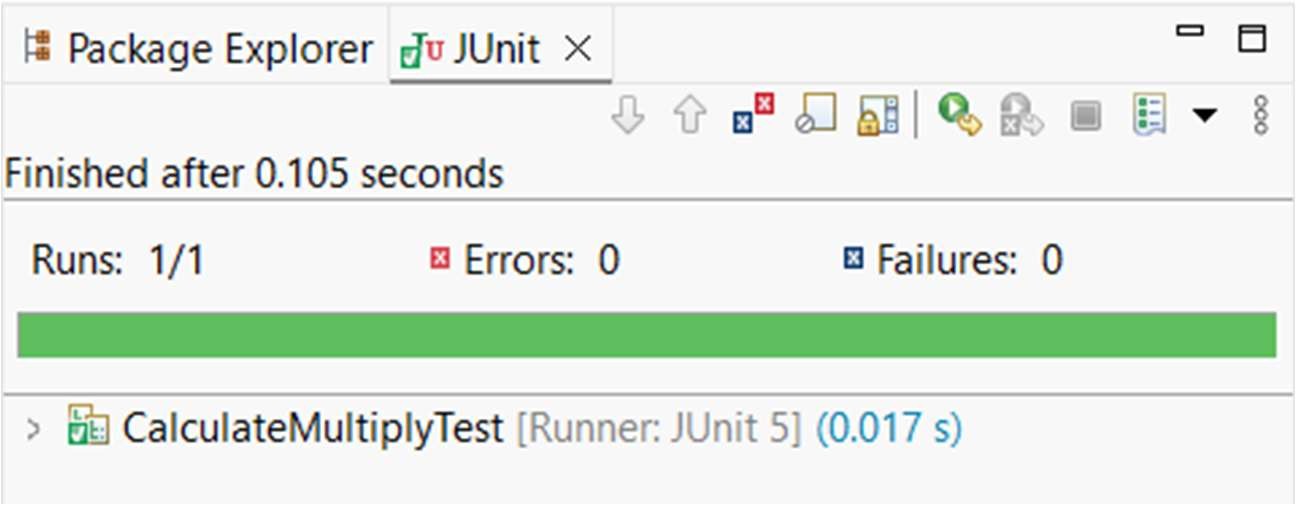
CalculateMultiplication c = new CalculateMultiplication(); int result = c.multiplication(4, 5); assertEquals(20, result);

System.out.print("JUnit Test Passed and Result: "+result);

}

}

Output:



2. Assertions in JUnit Scenario:

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

Code:

AssertionsTest.java

import static org.junit.jupiter.api.Assertions.\*; import org.junit.jupiter.api.Test; 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());

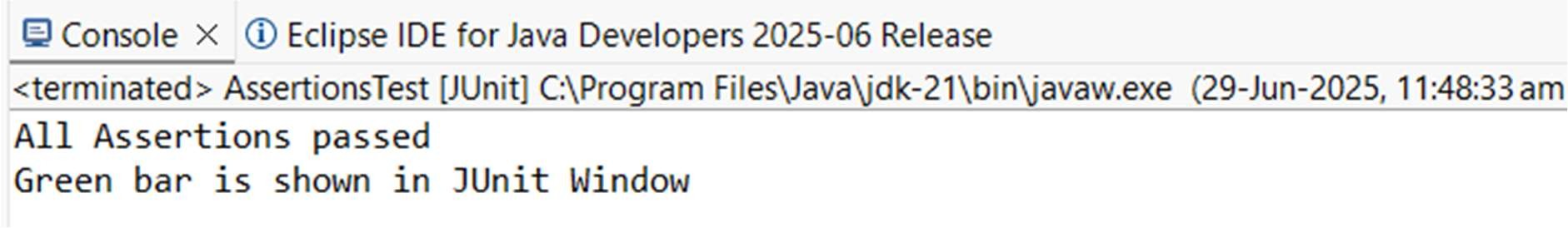
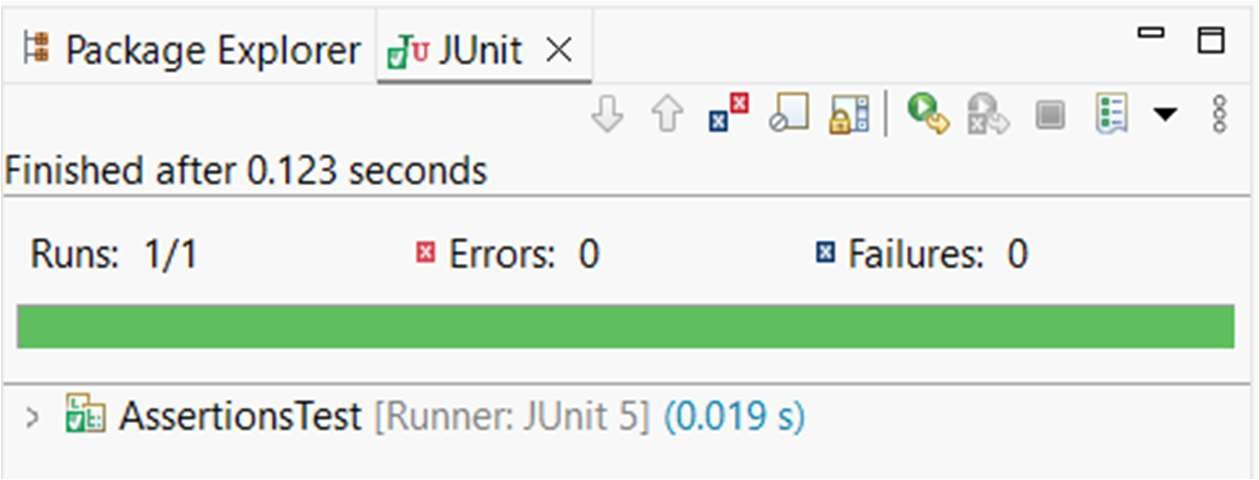
System.out.println("All Assertions passed");

System.out.println("Green bar is shown in JUnit Window");

}

}

Output:



3. 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.

Code:

ArithmeticOperations.java public class ArithmeticOperations

{

public int add(int a, int b)

{

return a+b;

}

public int divide(int a, int b)

{

return a/b; // Note: This will throw ArithmeticException if b=0

}

}

ArithmeticOperationsTest.java import org.junit.jupiter.api.\*;

import static org.junit.jupiter.api.Assertions.\*; public class ArithmeticOperationsTest

{

ArithmeticOperations a;

@BeforeEach // Setup method public void setup()

{

a = new ArithmeticOperations(); // Arrange

System.out.println("Setup done");

}

@AfterEach // Teardown method public void teardown()

{

a = null; // Shown for learning

System.out.println("Teardown done");

}

@Test

public void testAddition()

{

// Act

int result = a.add(10, 20);

// Assert

assertEquals(30, result);

}

@Test

public void testDivision()

{

// Act

int result = a.divide(20, 5);

// Assert assertEquals(4, result);

}

@Test

public void testDivideByZero()

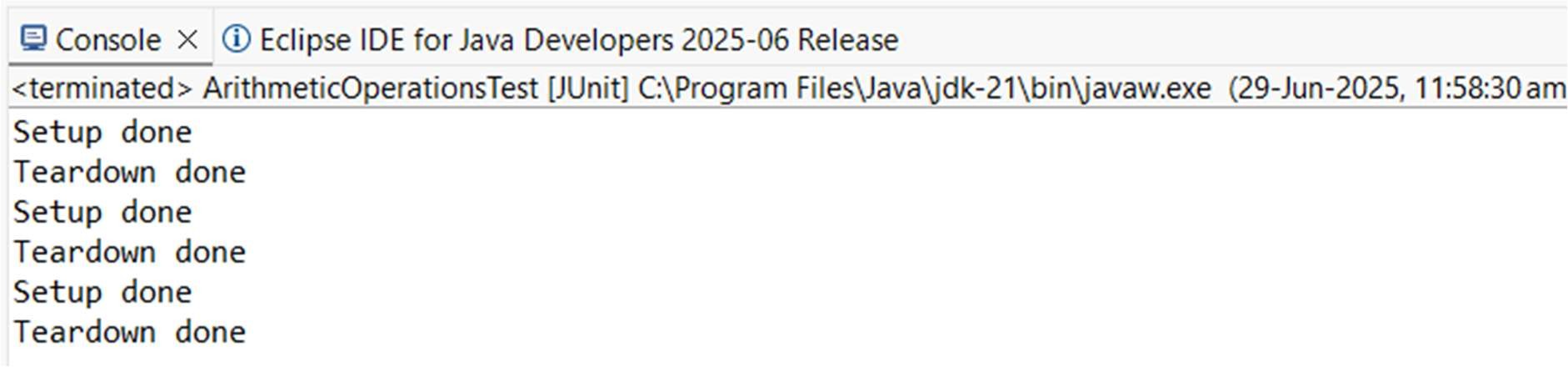
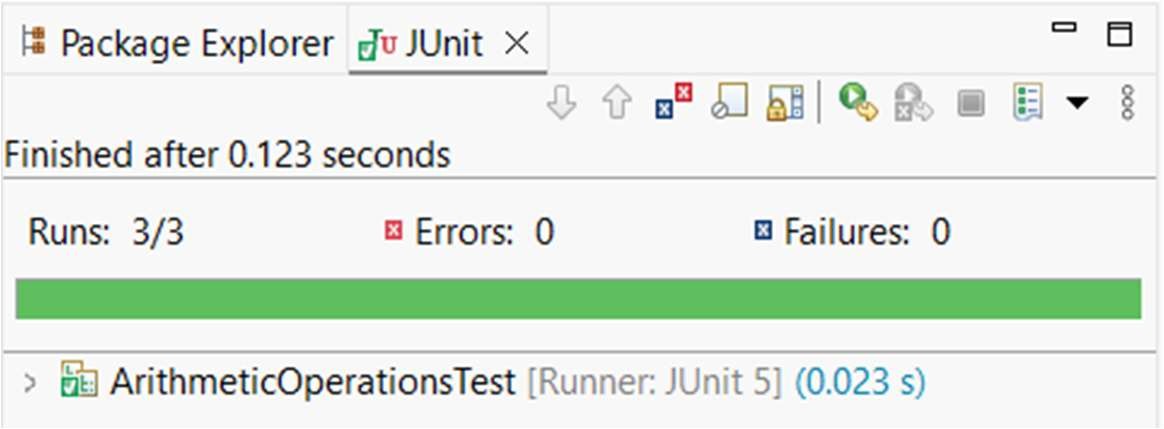
{

// Act + Assert (expecting exception) assertThrows(ArithmeticException.class, () -> a.divide(10, 0));

}

}

Output:



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.

Code:

ExternalAPI.java public interface ExternalAPI

{

String getData();

}

MyService.java public class MyService

{ private ExternalAPI api; // Constructor injection

public MyService(ExternalAPI api)

{ this.api = api;

}

public String fetchData()

{ return api.getData(); // Uses the API

}

}

MyServiceTestClass.java import static org.mockito.Mockito.\*; import static org.junit.jupiter.api.Assertions.\*; import org.junit.jupiter.api.Test; import org.mockito.Mockito; public class MyServiceTestClass

{

@Test

public void testExternalApi()

{

// Step 1: Mock the External API

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

// Step 2: Stub the method getData() to return "Mock Data" when(mockApi.getData()).thenReturn("Mock Data");

// Step 3: Pass mock to the service and call the method

MyService service = new MyService(mockApi);

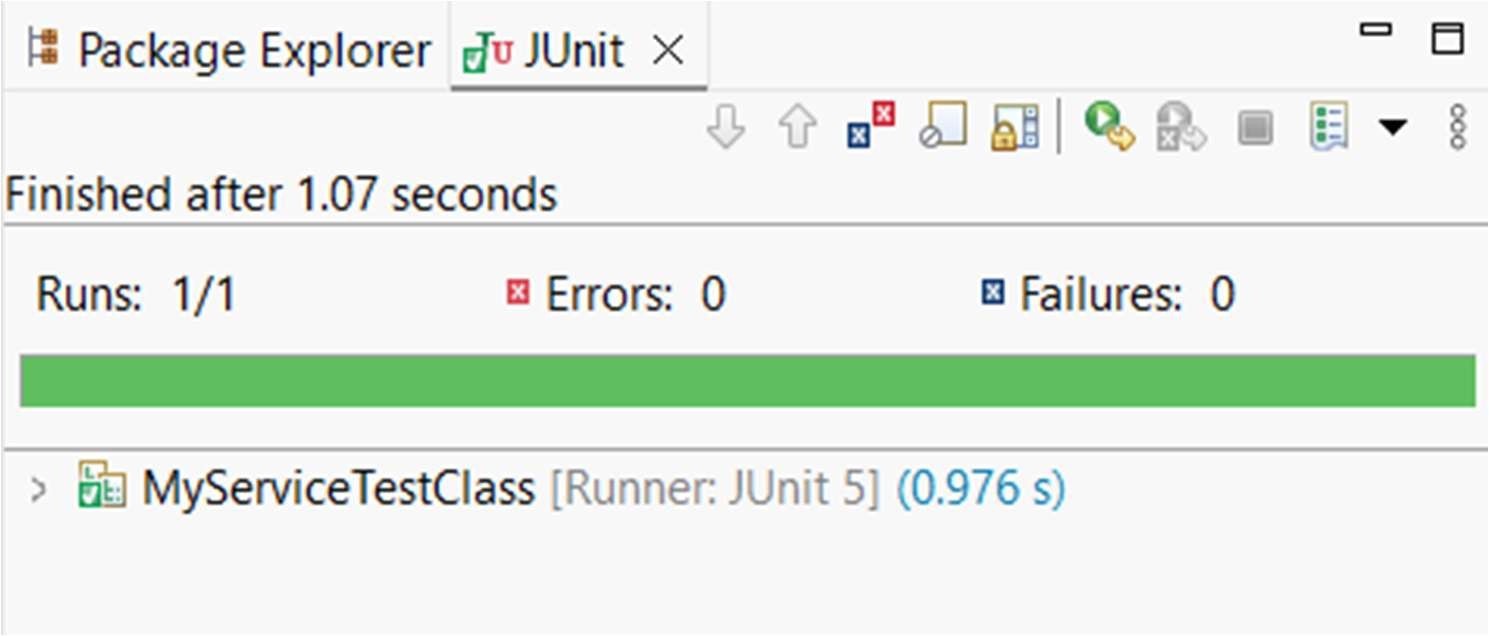
String result = service.fetchData();

// Step 4: Assert the result assertEquals("Mock Data", result);

}

}

Output:



2. Verifying Interactions

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

Code:

ExternalAPI.java public interface ExternalAPI

{

String getData();

}

MyService.java public class MyService

{ private ExternalAPI api; // Constructor injection

public MyService(ExternalAPI api)

{

this.api = api;

}

public String fetchData()

{ return api.getData(); // Uses the API

}

}

MyServiceTestClass.java import static org.mockito.Mockito.\*; import static org.junit.jupiter.api.Assertions.\*; import org.junit.jupiter.api.Test; import org.mockito.Mockito; public class MyServiceTestClass

{ @Test public void testVerifyInteraction()

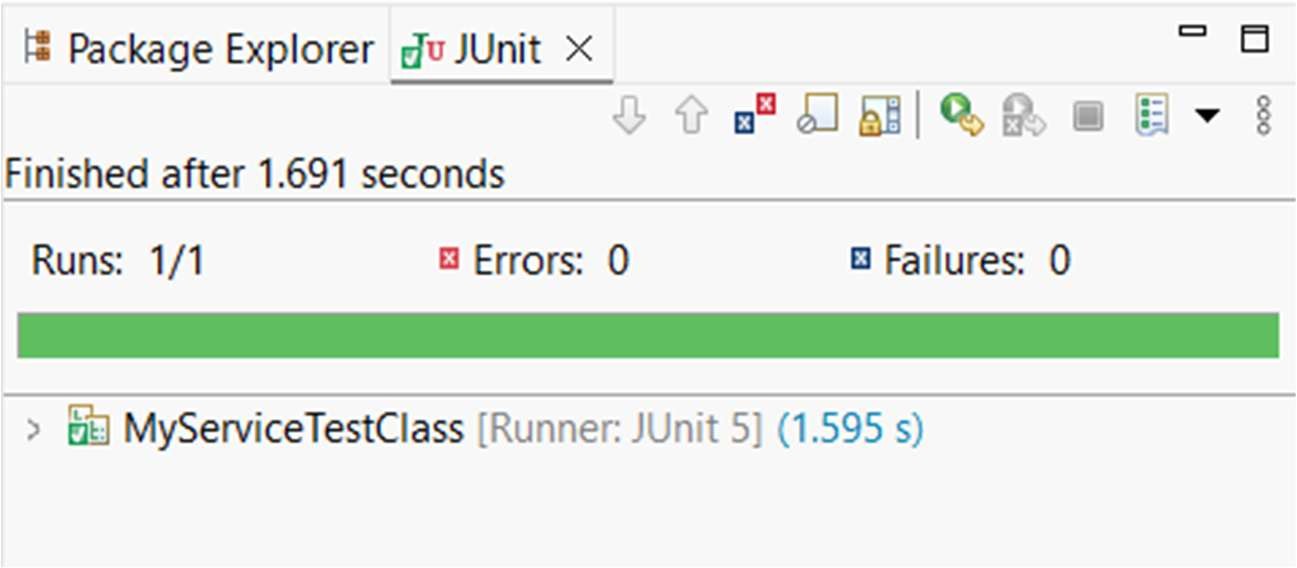
{

ExternalAPI mockApi = Mockito.mock(ExternalAPI.class); MyService service = new MyService(mockApi); service.fetchData(); // calling method verify(mockApi).getData(); // verifying the method was called

}

}

Output:



1. Logging Error Messages and Warning Levels Task:

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

Code:

LoggingExample.java

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");

}

}

Output:

