**WEEK – 02**

**PL/SQL EXERCISES**

**Exercise 1: Control Structures**

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

o 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.

o 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.

o 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:**

**-- Create Tables**

CREATE TABLE customers (

customer\_id INT,

name VARCHAR(100),

age INT,

balance DECIMAL(10, 2),

is\_vip BIT,

interest\_rate DECIMAL(4, 2)

);

CREATE TABLE loans (

loan\_id INT,

customer\_id INT,

due\_date DATE,

loan\_amount DECIMAL(10, 2)

);

**-- Insert Sample Data**

INSERT INTO customers VALUES (1, 'John Doe', 65, 5000.00, 0, 6.5);

INSERT INTO customers VALUES (2, 'Jane Smith', 45, 12000.00, 0, 7.2);

INSERT INTO customers VALUES (3, 'Alex Brown', 70, 8000.00, 0, 8.0);

-- OneCompiler supports GETDATE(), but not arithmetic directly with DATE

-- So use DATEADD instead

INSERT INTO loans VALUES (101, 1, DATEADD(DAY, 10, GETDATE()), 20000.00);

INSERT INTO loans VALUES (102, 2, DATEADD(DAY, 40, GETDATE()), 15000.00);

INSERT INTO loans VALUES (103, 3, DATEADD(DAY, 5, GETDATE()), 30000.00);

**-- Scenario 1 - Apply 1% Discount to Interest Rate for Customers Aged Above 60**

UPDATE customers

SET interest\_rate = interest\_rate - 1

WHERE age > 60;

-- Step 4: Scenario 2 - Set IsVIP Flag to True for Customers with Balance > 10000

UPDATE customers

SET is\_vip = 1

WHERE balance > 10000;

**-- Scenario 3 - Print Loan Reminders Due Within Next 30 Days**

SELECT

c.name AS Customer\_Name,

l.loan\_id AS Loan\_ID,

l.due\_date AS Due\_Date,

'Reminder: Your loan is due soon!' AS Message

FROM loans l

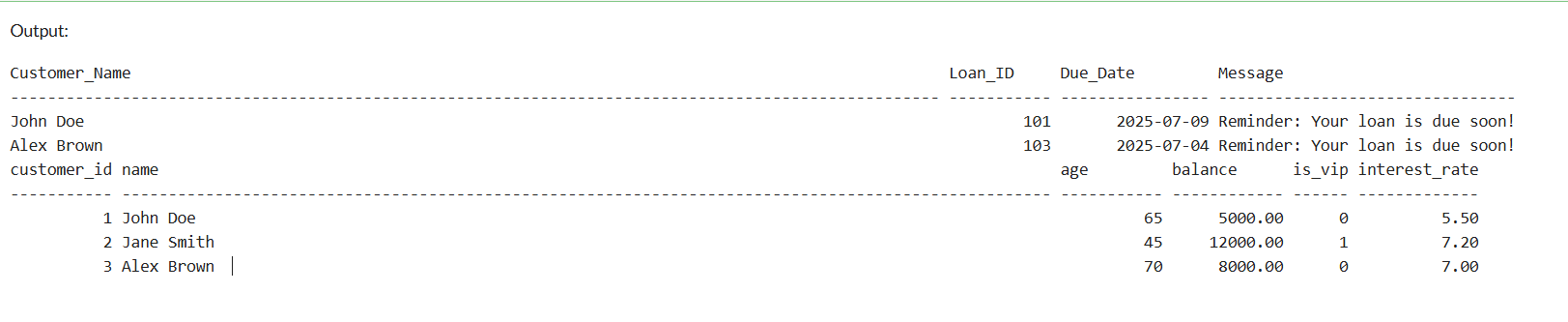
JOIN customers c ON c.customer\_id = l.customer\_id

WHERE l.due\_date <= DATEADD(DAY, 30, GETDATE());

**-- View Updated Customer Data**

SELECT \* FROM customers;

**Output:**



**Explanation:**

The above code demonstrates the use of SQL control structures such as conditional updates and data filtering to simulate real-world banking scenarios. It involves:

1. **Interest Discount**: Customers aged above 60 receive a 1% reduction in their loan interest rate, encouraging senior citizen benefits.
2. **VIP Promotion**: Customers maintaining a balance above $10,000 are marked as VIPs using a flag update, supporting customer segmentation.
3. **Loan Due Reminders**: Customers with loan due dates within the next 30 days are identified and issued reminder messages, improving customer service and reducing payment defaults.

These tasks are executed using UPDATE and SELECT queries with WHERE clauses, showcasing how SQL can handle dynamic, rule-based operations in financial systems.

**Exercise 3: Stored Procedures**

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

o 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.

o 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.

o 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:**

**-- Customers table**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100),

balance DECIMAL(10, 2)

);

**-- Employees table**

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

name VARCHAR(100),

department VARCHAR(50),

salary DECIMAL(10, 2)

);

**-- Accounts table**

CREATE TABLE accounts (

account\_id INT PRIMARY KEY,

customer\_id INT,

balance DECIMAL(10, 2)

);

**-- Sample customers**

INSERT INTO customers VALUES

(1, 'Alice', 1000.00),

(2, 'Bob', 1500.00),

(3, 'Charlie', 2000.00);

**-- Sample employees**

INSERT INTO employees VALUES

(101, 'John', 'HR', 50000),

(102, 'Jane', 'HR', 55000),

(103, 'Tom', 'IT', 60000);

**-- Sample accounts**

INSERT INTO accounts VALUES

(1, 1, 1000.00),

(2, 2, 1500.00),

(3, 3, 2000.00);

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE customers

SET balance = balance + (balance \* 0.01);

END;

/

BEGIN

ProcessMonthlyInterest;

END;

/

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

dept\_name IN VARCHAR2,

bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE employees

SET salary = salary + (salary \* (bonus\_percent / 100))

WHERE department = dept\_name;

END;

/

BEGIN

UpdateEmployeeBonus('HR', 10);

END;

/

CREATE OR REPLACE PROCEDURE TransferFunds (

source\_account IN NUMBER,

destination\_account IN NUMBER,

amount IN NUMBER

) IS

source\_balance NUMBER(10,2);

BEGIN

SELECT balance INTO source\_balance

FROM accounts

WHERE account\_id = source\_account;

IF source\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance for transfer.');

END IF;

UPDATE accounts

SET balance = balance - amount

WHERE account\_id = source\_account;

UPDATE accounts

SET balance = balance + amount

WHERE account\_id = destination\_account;

END;

/

BEGIN

TransferFunds(1, 2, 500);

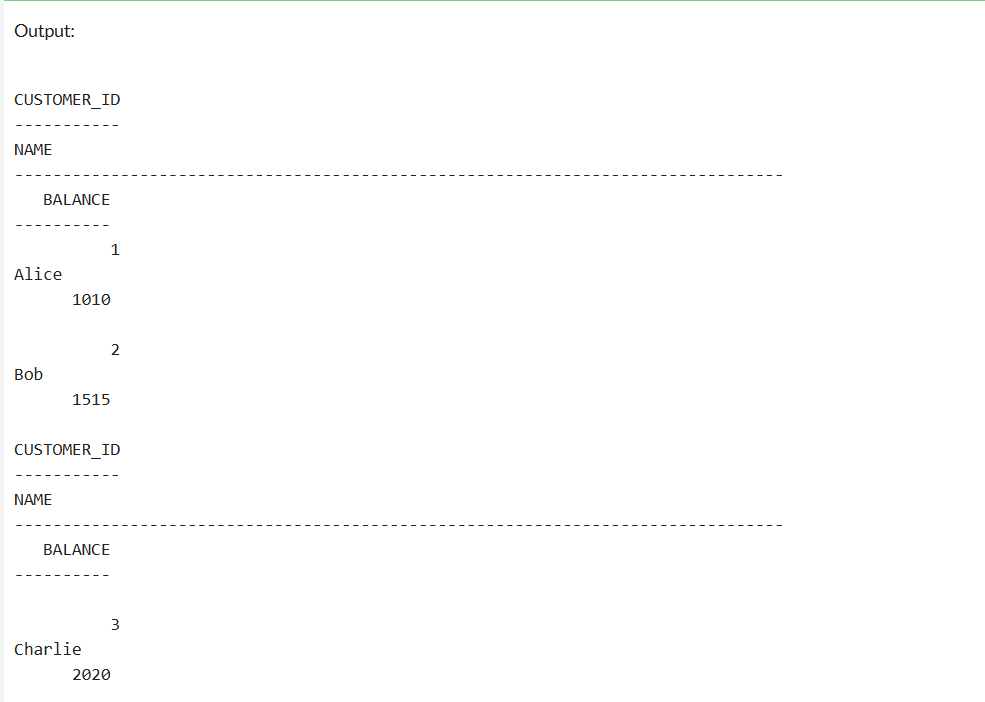
END;

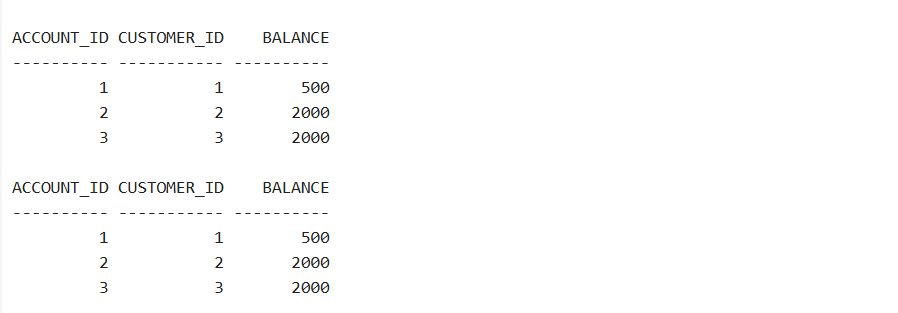
/

SELECT \* FROM customers;

SELECT \* FROM employees;

SELECT \* FROM accounts;

**Output:**

****

**Explanation:**

The above code demonstrates the use of stored procedures in PL/SQL to perform routine banking operations efficiently. Stored procedures help encapsulate logic, improve performance, and ensure consistent data processing.

**Scenario 1:** ProcessMonthlyInterest

A stored procedure named ProcessMonthlyInterest is created to add 1% monthly interest to the balance of all customers. It updates each customer's balance by applying the formula:  
balance = balance + (balance \* 0.01)

**Scenario 2:** UpdateEmployeeBonus

The UpdateEmployeeBonus procedure updates the salaries of employees in a given department by adding a bonus percentage passed as a parameter. It helps implement a performance-based bonus scheme efficiently across departments.

**Scenario 3:** TransferFunds

The TransferFunds procedure enables secure fund transfer between two accounts. It checks if the source account has sufficient balance, and only then performs the transfer. Otherwise, it raises an error message to prevent overdraft.

These procedures reflect real-world applications in the banking sector and illustrate how PL/SQL can be used to build secure, reusable database logic.

**TDD using JUnit5 and Mockito - JUnit\_Basic 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: junit junit 4.13.2 test

3. Create a new test class in your project.

**Solution:**

**TemperatureConverter.java:**

package com.exampletemp;

public class TemperatureConverter {

// Celsius to Fahrenheit

public double celsiusToFahrenheit(double celsius) {

return (celsius \* 9/5) + 32;

}

// Fahrenheit to Celsius

public double fahrenheitToCelsius(double fahrenheit) {

return (fahrenheit - 32) \* 5/9;

}

}

**TemperatureConverterTest.java:**

package com.exampletemp;

import org.junit.Test;

import static org.junit.Assert.\*;

public class TemperatureConverterTest {

@Test

public void testCelsiusToFahrenheit() {

TemperatureConverter tc = new TemperatureConverter();

double result = tc.celsiusToFahrenheit(0);

System.out.println("0°C to F: " + result);

assertEquals(32.0, result, 0.001);

}

@Test

public void testFahrenheitToCelsius() {

TemperatureConverter tc = new TemperatureConverter();

double result = tc.fahrenheitToCelsius(212);

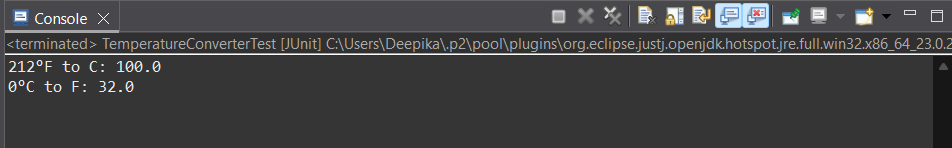
System.out.println("212°F to C: " + result);

assertEquals(100.0, result, 0.001);

}

}

**Output:**

****

**Explanation:**

* Implemented a TemperatureConverter class with methods to convert between Celsius and Fahrenheit.
* Used standard formulas for accurate temperature conversions.
* Created a TemperatureConverterTest class using JUnit.
* Validated conversion results with assertEquals() to ensure correctness.
* Ensured reliability and accuracy of the logic through unit testing.

**Exercise 3: Assertions in JUnit**

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

**Solution:**

**AssertionsTest.java:**

package com.example;

import org.junit.jupiter.api.Test;

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

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

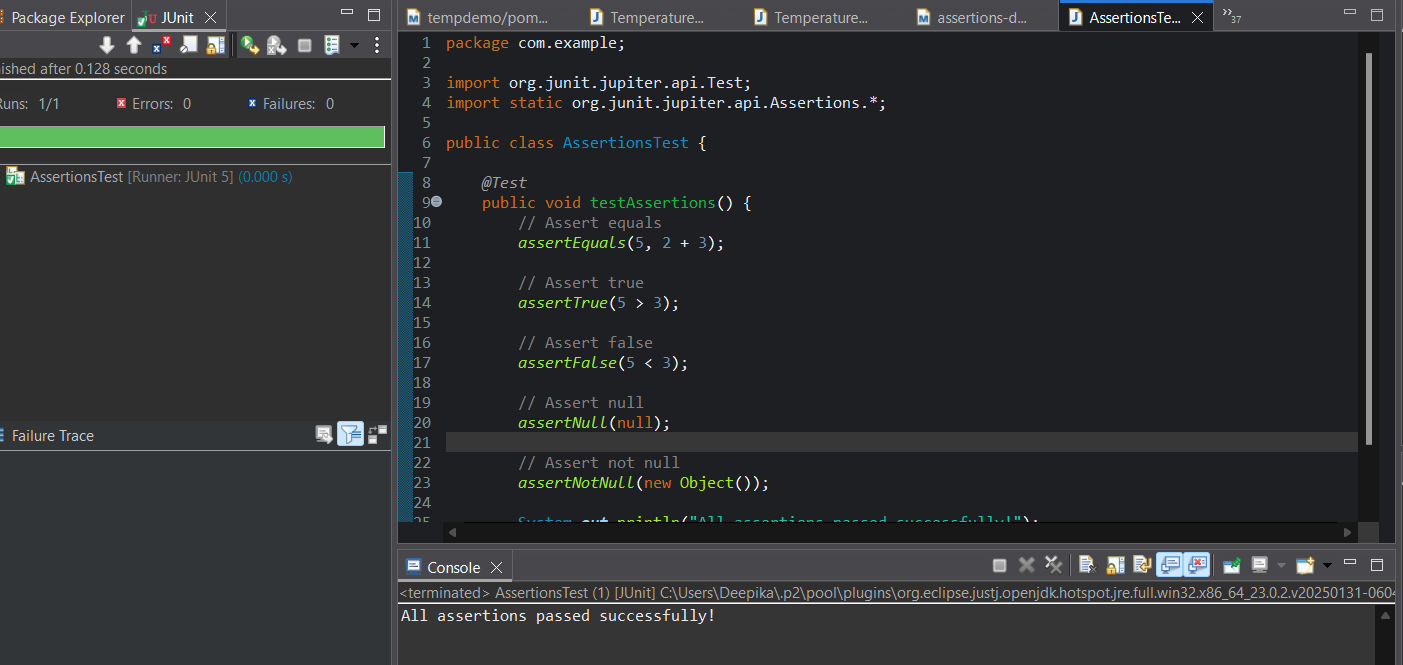
assertNotNull(new Object());

System.out.println("All assertions passed successfully!");

}

}

**Output:**



**Explanation:**

* Created a JUnit test class named AssertionsTest to demonstrate basic assertions.
* Used assertEquals, assertTrue, and assertFalse to validate expected conditions.
* Tested null and not null values using assertNull() and assertNotNull().
* The test ensures correctness of simple logic and object states.
* Successfully ran all tests in Eclipse using JUnit 5, verifying behavior with a green test bar.

**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.

**Solution:**

**BankAccount.java:**

package com.example;

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) {

if (amount <= balance)

balance -= amount;

}

public double getBalance() {

return balance;

}

}

**BankAccountTest.java:**

package com.example;

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

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

public class BankAccountTest {

BankAccount account;

@BeforeEach

public void setUp() {

// Arrange

account = new BankAccount(1000.0);

System.out.println("Setup: Account created with balance ₹1000");

}

@AfterEach

public void tearDown() {

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

}

@Test

public void testDeposit() {

// Act

account.deposit(500.0);

// Assert

assertEquals(1500.0, account.getBalance(), 0.001);

}

@Test

public void testWithdraw() {

// Act

account.withdraw(400.0);

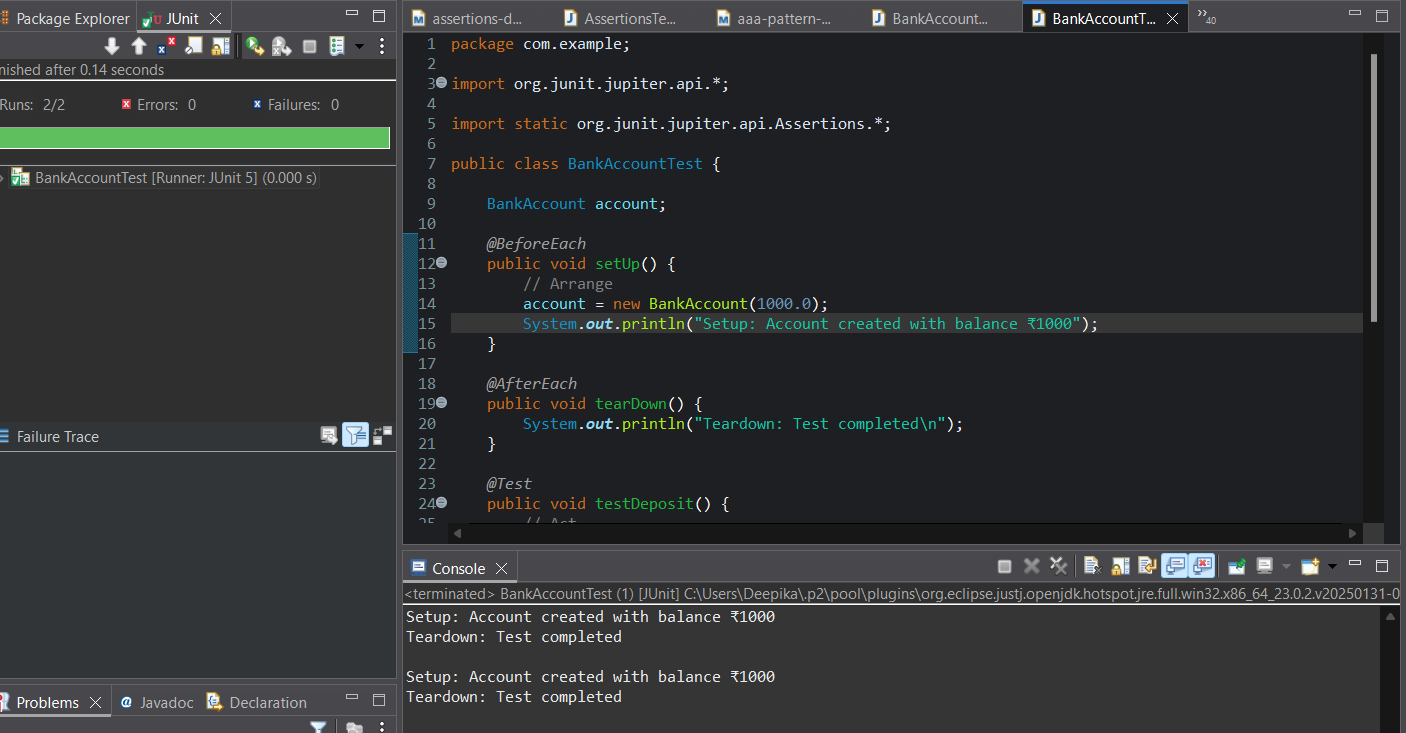
// Assert

assertEquals(600.0, account.getBalance(), 0.001);

}

}

**Output:**

****

**Explanation:**

* Created a BankAccount class with deposit and withdraw methods.
* Followed Arrange-Act-Assert (AAA) pattern for test clarity.
* Used @BeforeEach to initialize a fresh account before each test.
* Used @AfterEach to display teardown messages after tests.
* Verified correct account behavior using assertEquals() in JUnit 5.

**TDD using JUnit5 and Mockito - Mockito 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.

**Solution:**

**Dependencies to be added in pom.xml:**

<dependencies>

<!-- JUnit 5 -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

</dependencies>

**ExternalApi.java:**

package com.examplemockstub;

public interface ExternalApi {

String getData();

}

**MyService.java:**

package com.examplemockstub;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**MyServiceTest.java:**

package com.examplemockstub;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

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

public class MyServiceTest {

@Test

public void testExternalApi() {

// Step 1: Create mock

ExternalApi mockApi = mock(ExternalApi.class);

// Step 2: Stub method

when(mockApi.getData()).thenReturn("Mock Data");

// Step 3: Use mock in service

MyService service = new MyService(mockApi);

String result = service.fetchData();

// Step 4: Assert

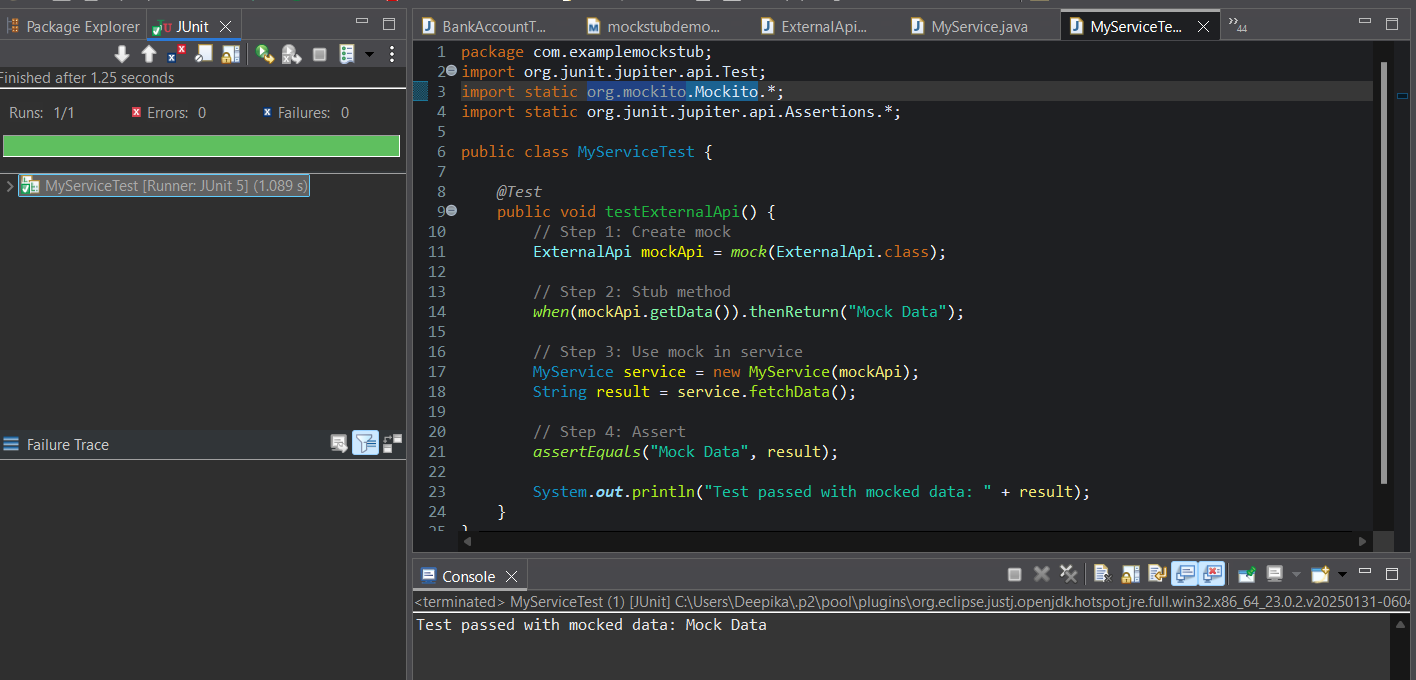
assertEquals("Mock Data", result);

System.out.println("Test passed with mocked data: " + result);

}

}

**Output:**

****

**Explanation:**

* Created a mock of ExternalApi using Mockito for isolation.
* Stubbed the getData() method to return "Mock Data" instead of calling the real API.
* Injected the mock into MyService to simulate real behavior.
* Verified output using assertEquals() in a JUnit test.
* Ensured service logic worked correctly without needing a real API call.

**Exercise 2: Verifying Interactions**

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

**Solution:**

**Dependencies added in pom.xml:**

<dependencies>

<!-- JUnit 5 -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

</dependencies>

**ExternalApi.java:**

package mockverifydemo;

public interface ExternalApi {

String getData();

}

**MyService.java:**

package mockverifydemo;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData(); // This will be mocked

}

}

**MyServiceTest.java:**

package mockverifydemo;

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

// Step 1: Create mock

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

// Step 2: Inject into service

MyService service = new MyService(mockApi);

// Step 3: Call the method

service.fetchData();

// Step 4: Verify interaction

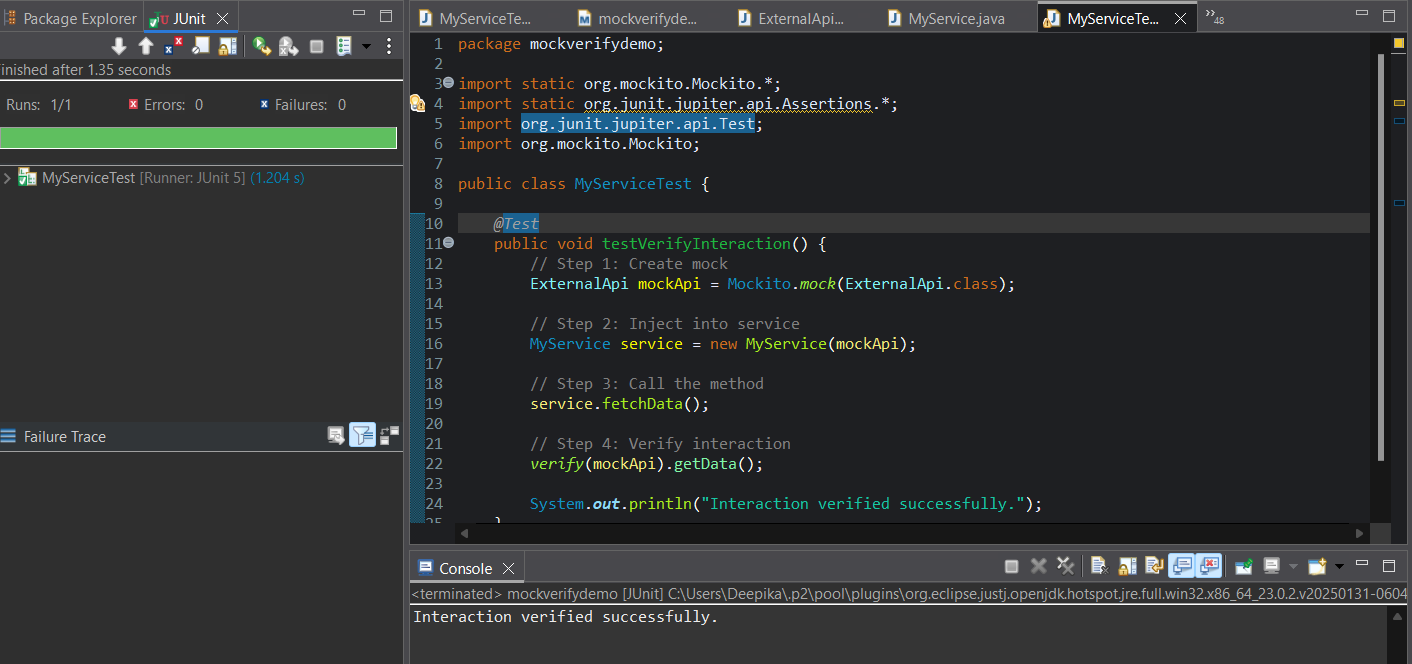
verify(mockApi).getData();

System.out.println("Interaction verified successfully.");

}

}

**Output:**

****

**Explanation:**

* Created a mock object using Mockito for an external dependency (ExternalApi).
* Injected the mock into the class under test (MyService).
* Called a method on MyService that internally calls the mock.
* Used verify() to check if the mock method was called as expected.
* Demonstrated interaction verification using Mockito for unit test accuracy.

**SLF4J logging framework - SL4J Logging exercises**

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

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

**Solution:**

**Dependencies added in pom.xml:**

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<!-- Logback (SLF4J Implementation) -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

**LoggingExample.java:**

package logging\_demo;

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");

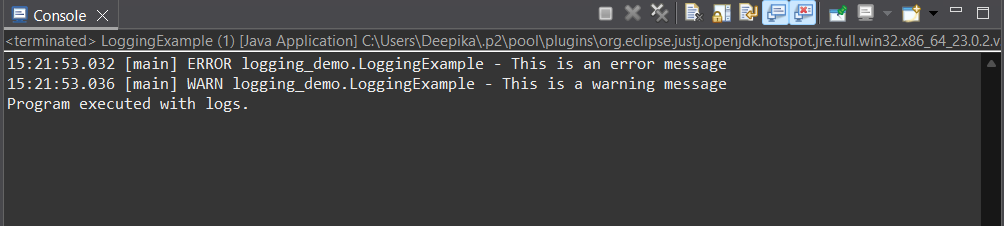
logger.warn("This is a warning message");

System.out.println("Program executed with logs.");

}

}

**Output:**



**Explanation:**

* Added **SLF4J and Logback** dependencies to pom.xml to enable logging.
* Created a simple LoggingExample class using LoggerFactory to get a logger instance.
* Used logger.error() and logger.warn() to demonstrate error and warning level logs.
* Maven handles dependency management and automatically configures Logback.
* Demonstrated clean and consistent logging, useful for debugging and monitoring Java applications.