**Week 2 Hands-on exercises:**

PL/SQL Programming :

**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.

**Code :**

CREATE TABLE Customers (

CustomerID NUMBER,

Name VARCHAR2(50),

Age NUMBER,

LoanInterestRate NUMBER

);

INSERT INTO Customers VALUES (101, 'John', 65, 10);

INSERT INTO Customers VALUES (102, 'Alice', 45, 9);

INSERT INTO Customers VALUES (103, 'David', 70, 8);

COMMIT;

BEGIN

FOR rec IN (SELECT CustomerID FROM Customers WHERE Age > 60) LOOP

UPDATE Customers

SET LoanInterestRate = LoanInterestRate - 1

WHERE CustomerID = rec.CustomerID;

END LOOP;

COMMIT;

END;

/

SELECT \* from Customers;

**Output :**

A screenshot of a computer

AI-generated content may be incorrect.

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.

**Code :**

CREATE TABLE Customers (

CustomerID NUMBER,

Name VARCHAR2(50),

Age NUMBER,

Balance NUMBER,

LoanInterestRate NUMBER,

IsVIP VARCHAR2(5)

);

INSERT INTO Customers VALUES (1, 'Alice', 35, 15000, 9.0, 'FALSE');

INSERT INTO Customers VALUES (2, 'Bob', 62, 8000, 10.0, 'FALSE');

INSERT INTO Customers VALUES (3, 'Charlie', 45, 12000, 8.5, 'FALSE');

COMMIT;

BEGIN

FOR rec IN (SELECT CustomerID FROM Customers WHERE Balance > 10000) LOOP

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

END LOOP;

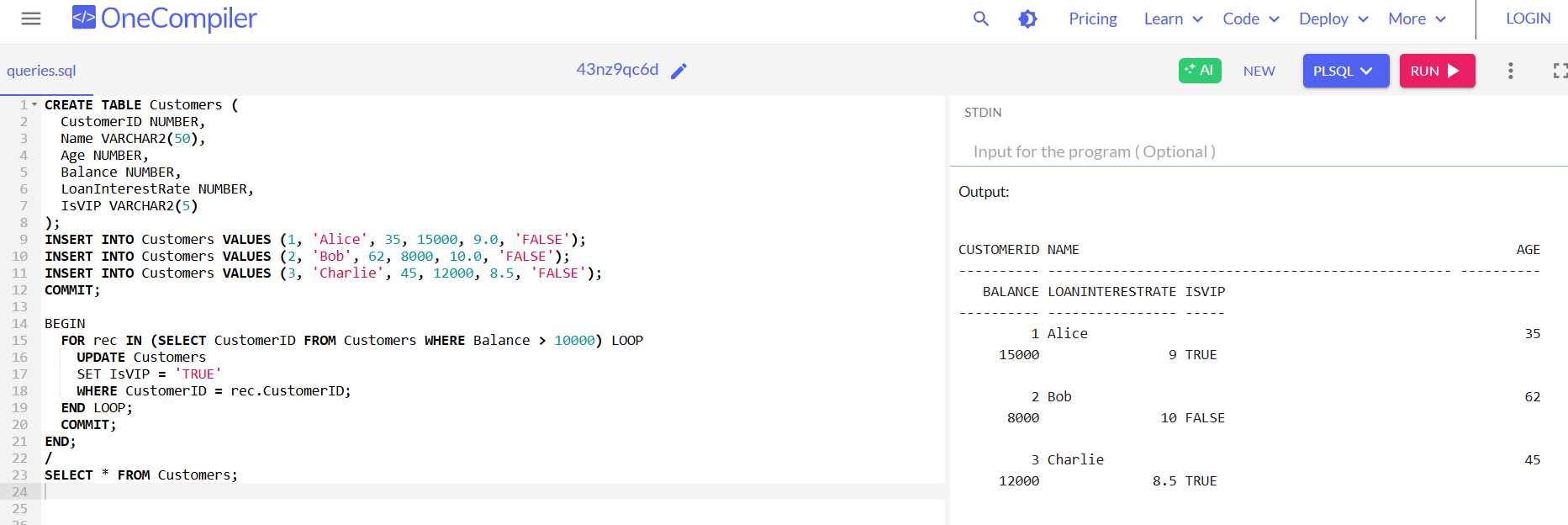
COMMIT;

END;

/

SELECT \* FROM Customers;

**Output :**

****

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.

**Code :**

CREATE TABLE Loans (

LoanID NUMBER,

CustomerID NUMBER,

DueDate DATE

);

INSERT INTO Loans VALUES (201, 1, SYSDATE + 10);

INSERT INTO Loans VALUES (202, 2, SYSDATE + 25);

INSERT INTO Loans VALUES (203, 3, SYSDATE + 45);

COMMIT;

SET SERVEROUTPUT ON;

BEGIN

FOR rec IN (

SELECT LoanID, CustomerID, DueDate

FROM Loans

WHERE DueDate <= SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID ||

' for Customer ' || rec.CustomerID ||

' is due on ' || rec.DueDate);

END LOOP;

END;

/

**Output :**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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.

**Code :**

CREATE TABLE Accounts (

AccountID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER

);

INSERT INTO Accounts VALUES (1, 'Savings', 1000);

INSERT INTO Accounts VALUES (2, 'Savings', 1500);

INSERT INTO Accounts VALUES (3, 'Current', 2000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR acc IN (SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings') LOOP

UPDATE Accounts

SET Balance = acc.Balance + (acc.Balance \* 0.01)

WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

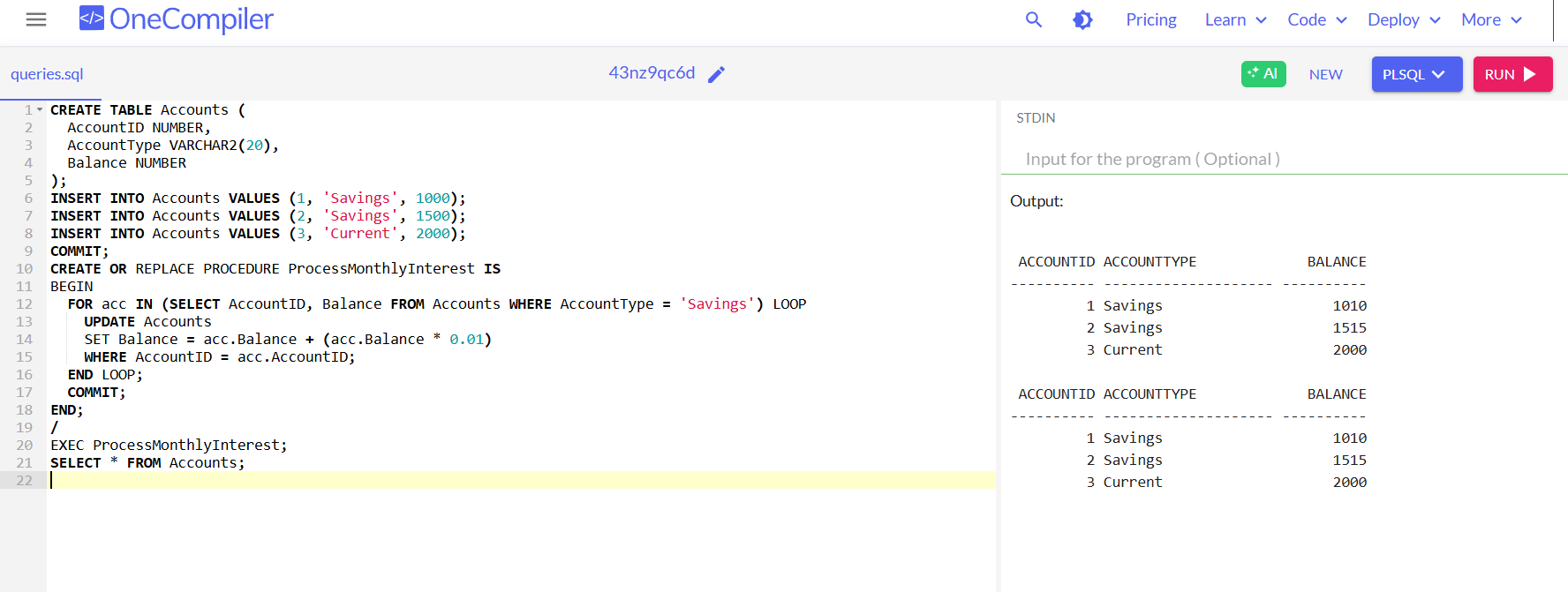
END;

/

EXEC ProcessMonthlyInterest;

SELECT \* FROM Accounts;

**Output :**

****

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.

**Code :**

SET SERVEROUTPUT ON;

CREATE TABLE Employees (

EmployeeID NUMBER,

DepartmentID NUMBER,

Salary NUMBER

);

INSERT INTO Employees VALUES (1, 101, 20000);

INSERT INTO Employees VALUES (2, 101, 25000);

INSERT INTO Employees VALUES (3, 102, 30000);

COMMIT;

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_DeptID IN NUMBER,

p\_BonusPercent IN NUMBER

) IS

v\_Count NUMBER := 0;

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* p\_BonusPercent / 100)

WHERE DepartmentID = p\_DeptID;

v\_Count := SQL%ROWCOUNT;

COMMIT;

END;

/

EXEC UpdateEmployeeBonus(101, 10);

SELECT \* FROM Employees;

**Output :**



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.

**Code :**

CREATE TABLE Accounts (

AccountID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER

);

INSERT INTO Accounts VALUES (4, 'Savings', 5000);

INSERT INTO Accounts VALUES (5, 'Savings', 3000);

COMMIT;

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_FromAccount IN NUMBER,

p\_ToAccount IN NUMBER,

p\_Amount IN NUMBER

) IS

v\_Balance NUMBER;

BEGIN

SELECT Balance INTO v\_Balance FROM Accounts WHERE AccountID = p\_FromAccount;

IF v\_Balance >= p\_Amount THEN

UPDATE Accounts SET Balance = Balance - p\_Amount WHERE AccountID = p\_FromAccount;

UPDATE Accounts SET Balance = Balance + p\_Amount WHERE AccountID = p\_ToAccount;

COMMIT;

ELSE

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance.');

END IF;

END;

/

EXEC TransferFunds(4, 5, 1000);

SELECT \* FROM Accounts;

**Output :**

****

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:

<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**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

**CalculatorTest.java**

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

int result = calc.add(2, 3);

assertEquals(5, result);

}

}

**Output :**

A screenshot of a computer

AI-generated content may be incorrect.

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

import org.junit.jupiter.api.Test;

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

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3, "2 + 3 should equal 5");

assertTrue(5 > 3, "5 should be greater than 3");

assertFalse(5 < 3, "5 should not be less than 3");

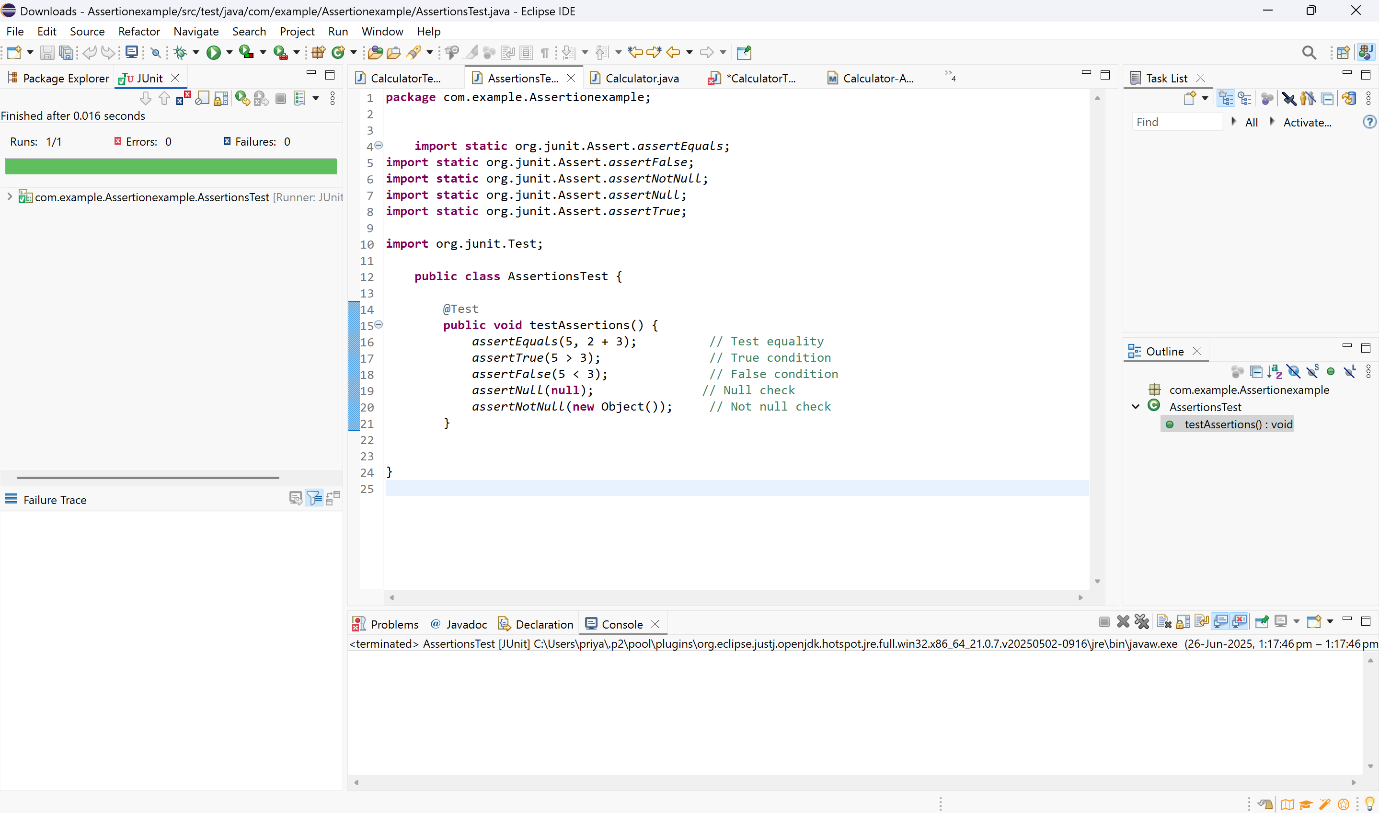
assertNull(null, "Object should be null");

assertNotNull(new Object(), "Object should not be null");

}

}

**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 :**

Calculator.java

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

CalculatorTest.java

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.Test;

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

public class CalculatorTest {

private Calculator calculator;

@BeforeEach

public void setUp() {

calculator = new Calculator();

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

}

@AfterEach

public void tearDown() {

calculator = null;

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

}

@Test

public void testAdd() {

int a = 2;

int b = 3;

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

assertEquals(5, result, "Addition should be 5");

}

@Test

public void testSubtract() {

int a = 7;

int b = 4;

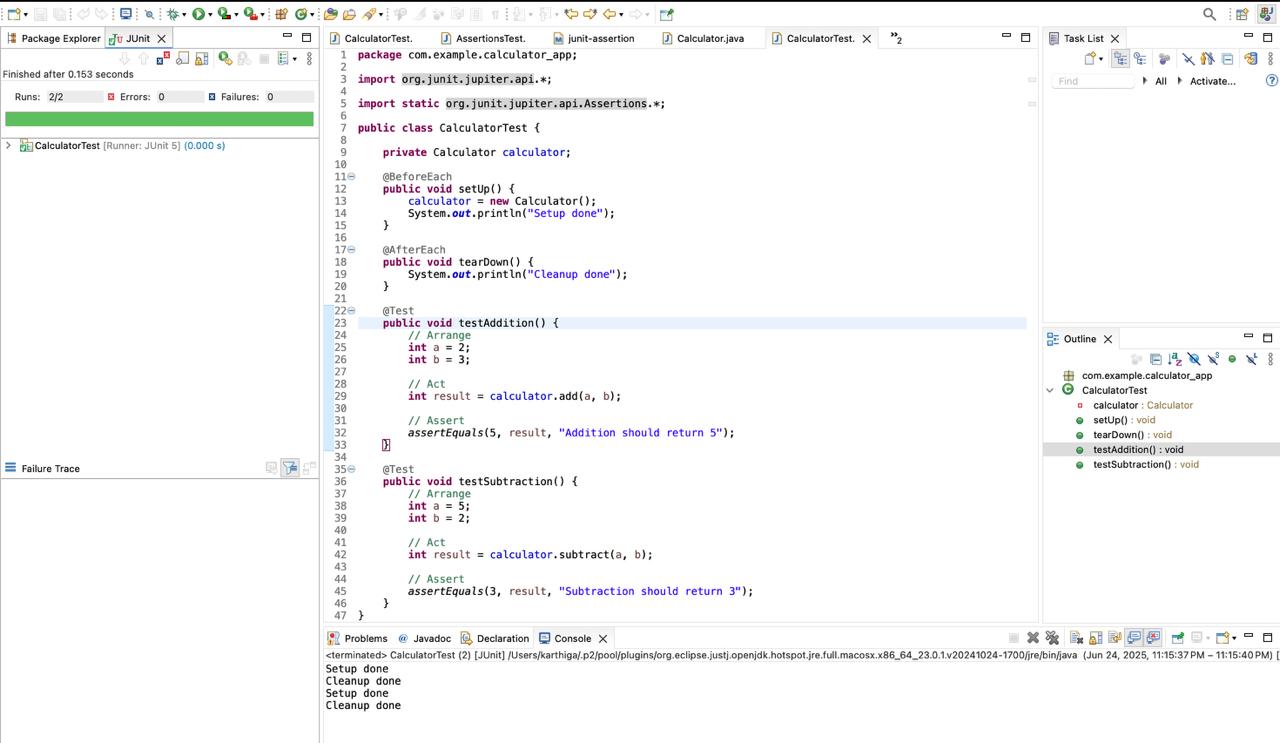
int result = calculator.subtract(a, b);

assertEquals(3, result, "Subtraction should be 3");

}

}

**Output :**



**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.

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 :**

Create the usercontrollertest:

Package controller;  
  
import model.User;  
import org.junit.jupiter.api.Test;  
import org.mockito.injectmocks;  
import org.mockito.Mock;  
  
import static org.junit.jupiter.api.Assertions.\*;  
import static org.mockito.Mockito.\*;  
  
import org.junit.jupiter.api.extension.extendwith;  
import org.mockito.junit.jupiter.mockitoextension;  
import org.springframework.http.responseentity;  
import service.userservice;  
  
@extendwith(mockitoextension.class)  
public class usercontrollertest {  
  
 @Mock  
 private userservice userservice;  
  
 @injectmocks  
 private usercontroller usercontroller;  
  
 @Test  
 void getuser\_shouldreturnuser\_whenuserexists() {  
 User user = new User();  
 user.setid(1L);  
 user.setname("Test User");  
  
 *when*(userservice.getuserbyid(1L)).thenreturn(user);  
  
 responseentity<User> response = usercontroller.getuser(1L);  
  
 *assertequals*(200, response.getstatuscodevalue());  
 *assertnotnull*(response.getbody());  
 *assertequals*("Test User", response.getbody().getname());  
 }  
  
 @Test  
 void getuser\_shouldreturnnotfound\_whenuserdoesnotexist() {  
 *when*(userservice.getuserbyid(1L)).thenreturn(null);  
  
 responseentity<User> response = usercontroller.getuser(1L);  
  
 *assertequals*(404, response.getstatuscodevalue());  
 *assertnull*(response.getbody());  
 }  
}

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 :**

Create the UserServiceTest:

package com.example.demo.service;

import com.example.demo.model.User;

import com.example.demo.repository.UserRepository;

import org.junit.jupiter.api.Test;

import org.mockito.InjectMocks;

import org.mockito.Mock;

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

import static org.mockito.Mockito.\*;

import java.util.Optional;

import org.junit.jupiter.api.extension.ExtendWith;

import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)

public class UserServiceTest {

@Mock

private UserRepository userRepository;

@InjectMocks

private UserService userService;

@Test

void getUserById\_ShouldReturnUser\_WhenUserExists() {

User user = new User();

user.setId(1L);

user.setName("Mocked User");

when(userRepository.findById(1L)).thenReturn(Optional.of(user));

User result = userService.getUserById(1L);

assertNotNull(result);

assertEquals("Mocked User", result.getName());

verify(userRepository, times(1)).findById(1L);

}

@Test

void getUserById\_ShouldReturnNull\_WhenUserDoesNotExist() {

when(userRepository.findById(1L)).thenReturn(Optional.empty());

User result = userService.getUserById(1L);

assertNull(result);

verify(userRepository, times(1)).findById(1L);

}

}

SLF4J logging framework :

SL4J Logging exercise :

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 org.example;

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");

}

}

pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<artifactId>logging-demo</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>17</maven.compiler.source>

<maven.compiler.target>17</maven.compiler.target>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<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>

</dependencies>

</project>

**Output :**

