**Week 2-Hands on Exercise:**

**Skill: PL/SQL programming**

**Schema Created:**

CREATE TABLE Customers (

    CustomerID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    DOB DATE,

    Balance NUMBER,

    LastModified DATE

);

CREATE TABLE Accounts (

    AccountID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    AccountType VARCHAR2(20),

    Balance NUMBER,

    LastModified DATE,

    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Transactions (

    TransactionID NUMBER PRIMARY KEY,

    AccountID NUMBER,

    TransactionDate DATE,

    Amount NUMBER,

    TransactionType VARCHAR2(10),

    FOREIGN KEY (AccountID) REFERENCES Accounts (AccountID)

);

CREATE TABLE Loans (

    LoanID NUMBER PRIMARY KEY,

    CustomerID NUMBER,

    LoanAmount NUMBER,

    InterestRate NUMBER,

    StartDate DATE,

    EndDate DATE,

    FOREIGN KEY (CustomerID) REFERENCES Customers (CustomerID)

);

CREATE TABLE Employees (

    EmployeeID NUMBER PRIMARY KEY,

    Name VARCHAR2(100),

    Position VARCHAR2(50),

    Salary NUMBER,

    Department VARCHAR2(50),

    HireDate DATE

);

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (2, 2, 3000, 4.5, SYSDATE, SYSDATE + 15);

COMMIT;

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (1, 'John Doe', TO\_DATE ('1985-05-15', 'YYYY-MM-DD'), 1000, SYSDATE);

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (2, 'Jane Smith', TO\_DATE ('1990-07-20', 'YYYY-MM-DD'), 1500, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (1, 1, 'Savings', 1000, SYSDATE);

INSERT INTO Accounts (AccountID, CustomerID, AccountType, Balance, LastModified)

VALUES (2, 2, 'Checking', 1500, SYSDATE);

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (1, 1, SYSDATE, 200, 'Deposit');

INSERT INTO Transactions (TransactionID, AccountID, TransactionDate, Amount, TransactionType)

VALUES (2, 2, SYSDATE, 300, 'Withdrawal');

INSERT INTO Loans (LoanID, CustomerID, LoanAmount, InterestRate, StartDate, EndDate)

VALUES (1, 1, 5000, 5, SYSDATE, ADD\_MONTHS (SYSDATE, 60));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (1, 'Alice Johnson', 'Manager', 70000, 'HR', TO\_DATE ('2015-06-15', 'YYYY-MM-DD'));

INSERT INTO Employees (EmployeeID, Name, Position, Salary, Department, HireDate)

VALUES (2, 'Bob Brown', 'Developer', 60000, 'IT', TO\_DATE ('2017-03-20', 'YYYY-MM-DD'));

**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.

**Scenario 1:**

BEGIN

    FOR customer IN (

        SELECT c.CustomerID, c.DOB, l.LoanID, l.InterestRate

        FROM Customers c

        JOIN Loans l ON c.CustomerID = l.CustomerID

    )

    LOOP

        IF MONTHS\_BETWEEN(SYSDATE, customer.DOB) / 12 > 60 THEN

            UPDATE Loans

            SET InterestRate = customer.InterestRate - 1

            WHERE LoanID = customer.LoanID;

        END IF;

    END LOOP;

    COMMIT;

END;

**Scenario 2:**

ALTER TABLE Customers ADD IsVIP CHAR(1);

BEGIN

    FOR customer IN (SELECT CustomerID, Balance FROM Customers)

    LOOP

        IF customer.Balance > 10000 THEN

            UPDATE Customers

            SET IsVIP = 'Y'

            WHERE CustomerID = customer.CustomerID;

        END IF;

    END LOOP;

    COMMIT;

END;

**Scenario 3:**

BEGIN

    FOR loan\_rec IN (

        SELECT l.LoanID, l.CustomerID, l.EndDate, c.Name

        FROM Loans l

        JOIN Customers c ON l.CustomerID = c.CustomerID

        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30

    )

    LOOP

        DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan for customer ' || loan\_rec.Name ||

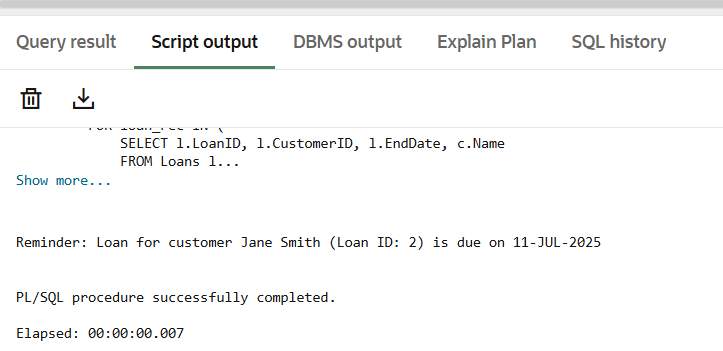
                             ' (Loan ID: ' || loan\_rec.LoanID ||

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

    END LOOP;

END;

**Output:**

****

**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.

**Scenario 1:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

COMMIT;

END;

**Scenario 2:**

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;

COMMIT;

END;

EXEC UpdateEmployeeBonus('IT', 10); -- 10% bonus for IT department

**Scenario 3:**

CREATE OR REPLACE PROCEDURE TransferFunds (

source\_account\_id IN NUMBER,

dest\_account\_id IN NUMBER,

transfer\_amount IN NUMBER

) IS

insufficient\_funds EXCEPTION;

BEGIN

DECLARE

source\_balance NUMBER;

BEGIN

SELECT Balance INTO source\_balance

FROM Accounts

WHERE AccountID = source\_account\_id;

IF source\_balance < transfer\_amount THEN

RAISE insufficient\_funds;

END IF;

END;

-- Deduct from source

UPDATE Accounts

SET Balance = Balance - transfer\_amount

WHERE AccountID = source\_account\_id;

-- Add to destination

UPDATE Accounts

SET Balance = Balance + transfer\_amount

WHERE AccountID = dest\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds in source account.');

ROLLBACK;

WHEN OTHERS THEN

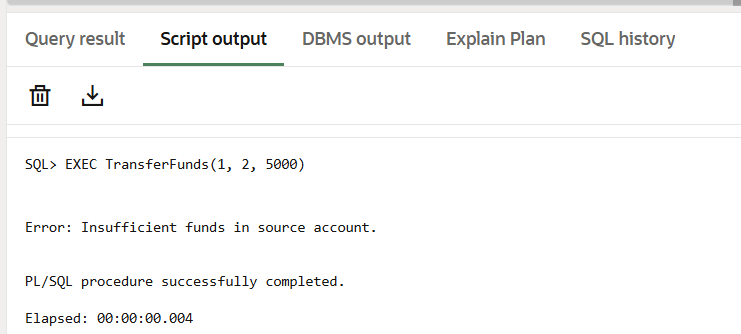
DBMS\_OUTPUT.PUT\_LINE('Unexpected error: ' || SQLERRM);

ROLLBACK;

END;

EXEC TransferFunds(1, 2, 500); -- Transfer $5000 from Account 1 to Account 2

**Output:**

****

**Skill: 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

**Solution Code:**

//Create a new class for example simple calculator

package com.example.demo;

public class CalculatorApp {

public int add(int a, int b) {

System.*out*.println("Addtion of given numbers is "+(a+b));

return a+b;

}

}

//Create a Test Class

package com.example.demo;

import org.junit.jupiter.api.Test;

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

public class AppTest {

@Test

void testAddition() {

System.*out*.println("Running testAddition...");

CalculatorApp calc=new CalculatorApp();

*assertEquals*(4, calc.add(2, 2));

}

@Test

void testStringNotEmpty() {

System.*out*.println("Running testStringNotEmpty...");

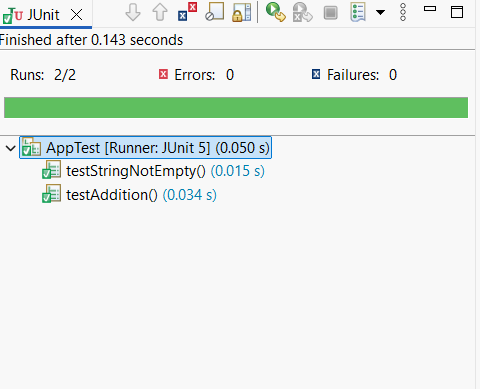
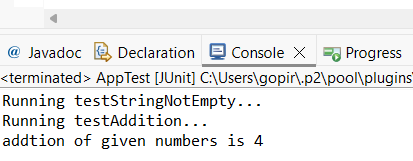
String name = "JUnit5";

*assertFalse*(name.isEmpty());

}

}

**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:**

package com.example;

import org.junit.jupiter.api.Test;

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

public class AssertionTest {

@Test

public void testAssertions() {

// Assert equals

System.*out*.println("Testing assertEquals...");

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

// Assert true

System.*out*.println("Testing assertTrue...");

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

// Assert false

System.*out*.println("Testing assertFalse...");

*assertFalse* (5 < 3, "5 is not less than 3");

// Assert null

System.*out*.println("Testing assertNull...");

*assertNull* (null, "Expected null value");

// Assert not null

System.*out*.println("Testing assertNotNull...");

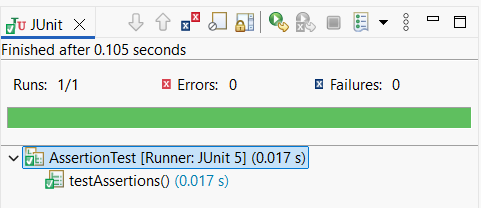
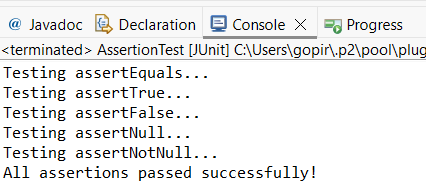
*assertNotNull* (new Object (), "Expected non-null object");

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

}

}

**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

**Solution Code:**

//Create a class bank account

package com.example;

public class BankAccount {

private String owner;

private double balance;

public BankAccount(String owner, double balance) {

this.owner = owner;

this.balance = balance;

}

public void deposit(double amount) {

if (amount > 0)

this.balance += amount;

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance)

this.balance -= amount;

}

public double getBalance() {

return balance;

}

}

//Create a test class bankaccounttest

package com.example;

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 BankAccountTest {

private BankAccount account;

@BeforeEach

public void setUp() {

System.*out*.println("Opening new bank account...");

account = new BankAccount("Uma", 1000.0);

}

@AfterEach

public void tearDown() {

System.*out*.println("Closing bank account...");

account = null;

}

@Test

public void testDeposit() {

double depositAmount = 500.0;

account.deposit (depositAmount);

*assertEquals*(1500.0, account.getBalance(), "Balance should be 1500 after deposit");

}

@Test

public void testWithdraw() {

double withdrawAmount = 400.0;

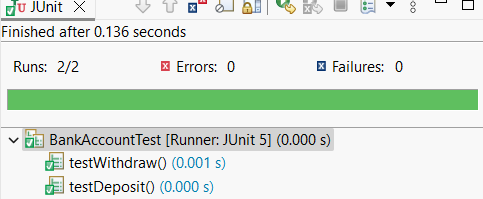
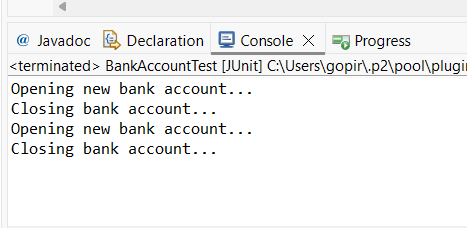
account.withdraw(withdrawAmount);

*assertEquals*(600.0, account.getBalance(), "Balance should be 600 after withdrawal");

}

}

**Output:**

****

**Skill: Mockito**

**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:**

//Create external api class

public interface ExternalApi {

String getData();

}

//Create myservive class

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

//test my service class

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 testExternalApi() {

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

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

MyService service = new MyService(mockApi);

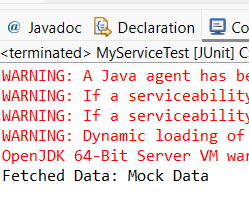
String result = service.fetchData();

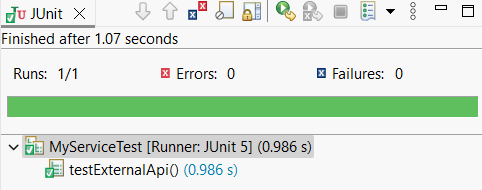
System.*out*.println("Fetched Data: " + result);

*assertEquals*("Mock Data", 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:**

package com.example.uma;

public class MyService {

private final ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public void fetchData() {

api.getData();

}

}

package com.example.uma;

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);

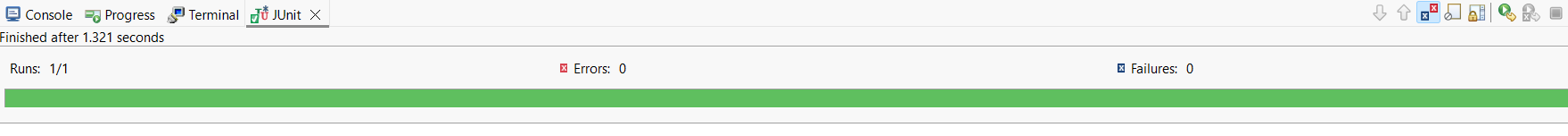
service.fetchData();

*verify*(mockApi).getData();

}

}

**Output:**



**Skill: Slf4J logging**

**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 error message and warnings:

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) {

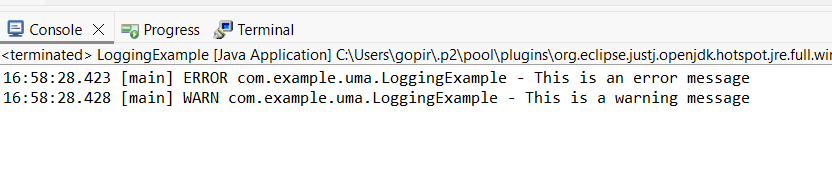
logger.error("This is an error message");

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

}

}

**Output:**

****