**Week – 02**

**PL/SQL programming - PLSQL\_Exercises**

**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.

**Explanation:**

* Implemented SQL blocks to simulate banking operations like loan discounting, VIP promotion, and due reminders.
* Structured each scenario using temporary tables and T-SQL control structures for clear logic flow.
* Used conditional updates to apply a 1% interest discount for customers aged above 60.
* Used balance-based filtering to set the IsVIP flag for high-value customers.
* Generated reminder messages for loans due within the next 30 days using date functions and joins.

**Code:**

***-- Scenario 1: Discount for seniors***

BEGIN

DECLARE @Customers TABLE (

CustomerID INT,

Name VARCHAR(100),

DOB DATE,

Balance DECIMAL(10,2)

);

DECLARE @Loans TABLE (

LoanID INT,

CustomerID INT,

InterestRate DECIMAL(5,2)

);

INSERT INTO @Customers VALUES

(1, 'Ravi Kumar', '1950-03-25', 9200.00),

(2, 'Sneha Reddy', '1985-11-10', 13000.00);

INSERT INTO @Loans VALUES

(1, 1, 7.25),

(2, 2, 6.50);

UPDATE @Loans

SET InterestRate = InterestRate - 1

WHERE CustomerID IN (

SELECT CustomerID FROM @Customers

WHERE DATEDIFF(YEAR, DOB, GETDATE()) > 60

);

SELECT 'Applied 1% discount to seniors' AS Result;

SELECT \* FROM @Loans;

END;

GO

***-- Scenario 2: VIP promotion***

BEGIN

DECLARE @Customers TABLE (

CustomerID INT,

Name VARCHAR(100),

Balance DECIMAL(10,2),

IsVIP BIT DEFAULT 0

);

INSERT INTO @Customers VALUES

(1, 'Ravi Kumar', 9200.00, 0),

(2, 'Sneha Reddy', 13000.00, 0),

(3, 'Amit Joshi', 15050.50, 0);

UPDATE @Customers

SET IsVIP = 1

WHERE Balance > 10000.00;

SELECT 'VIP promotions completed' AS Result;

SELECT \* FROM @Customers;

END;

GO

***-- Scenario 3: Loan due reminders***

BEGIN

DECLARE @Customers TABLE (

CustomerID INT,

Name VARCHAR(100)

);

DECLARE @Loans TABLE (

LoanID INT,

CustomerID INT,

DueDate DATE

);

INSERT INTO @Customers VALUES

(1, 'Ravi Kumar'),

(2, 'Sneha Reddy'),

(3, 'Amit Joshi');

INSERT INTO @Loans VALUES

(1, 1, DATEADD(DAY, 5, GETDATE())),

(2, 2, DATEADD(DAY, 35, GETDATE())),

(3, 3, GETDATE());

SELECT 'Loan due reminders:' AS Result;

SELECT CONCAT('Reminder for ', C.Name, ': Loan ', L.LoanID, ' due in ', DATEDIFF(DAY, GETDATE(), L.DueDate), ' days') AS Message

FROM @Loans L

JOIN @Customers C ON L.CustomerID = C.CustomerID

WHERE L.DueDate BETWEEN GETDATE() AND DATEADD(DAY, 30, GETDATE());

END;

GO

**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.

**Explanation:**

* Created ProcessMonthlyInterest to update savings account balances by 1% monthly using an UPDATE statement with interest logic.
* Created UpdateEmployeeBonus, which takes department and bonus percentage as input and updates employee salaries accordingly.
* Developed TransferFunds, a procedure to safely transfer funds between accounts after validating sufficient balance.
* Used conditional checks and print statements for validation feedback in procedures like TransferFunds.
* Tested all procedures successfully using sample data for accounts and employees in OneCompiler SQL Server.

**Code:**

***-- Create Tables and Insert Data***

DROP TABLE IF EXISTS Accounts;

DROP TABLE IF EXISTS Employees;

CREATE TABLE Accounts (

AccountID INT PRIMARY KEY,

CustomerName VARCHAR(100),

AccountType VARCHAR(20),

Balance DECIMAL(10,2)

);

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

Name VARCHAR(100),

Department VARCHAR(50),

Salary DECIMAL(10,2)

);

INSERT INTO Accounts VALUES

(1, 'Alice', 'Savings', 10000.00),

(2, 'Bob', 'Current', 5000.00),

(3, 'Carol', 'Savings', 20000.00);

INSERT INTO Employees VALUES

(1, 'David', 'Sales', 50000.00),

(2, 'Eva', 'Sales', 55000.00),

(3, 'Frank', 'HR', 60000.00);

GO

***-- Scenario 1 : Interest***

CREATE PROCEDURE ProcessMonthlyInterest

AS

BEGIN

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'Savings';

END;

GO

***-- Scenario 2 : Bonus***

CREATE PROCEDURE UpdateEmployeeBonus

@Dept VARCHAR(50),

@BonusPercent DECIMAL(5,2)

AS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* @BonusPercent / 100)

WHERE Department = @Dept;

END;

GO

***-- Scenario 3 : Transfer Funds***

CREATE PROCEDURE TransferFunds

@FromAccountID INT,

@ToAccountID INT,

@Amount DECIMAL(10,2)

AS

BEGIN

DECLARE @FromBalance DECIMAL(10,2);

SELECT @FromBalance = Balance FROM Accounts WHERE AccountID = @FromAccountID;

IF @FromBalance IS NULL OR @FromBalance < @Amount

BEGIN

PRINT 'Transfer failed: Insufficient funds or invalid account.';

RETURN;

END

UPDATE Accounts

SET Balance = Balance - @Amount

WHERE AccountID = @FromAccountID;

UPDATE Accounts

SET Balance = Balance + @Amount

WHERE AccountID = @ToAccountID;

PRINT 'Transfer successful.';

END;

GO

***-- Test Scenario 1: Interest***

EXEC ProcessMonthlyInterest;

SELECT \* FROM Accounts;

***-- Test Scenario 2: Bonus***

EXEC UpdateEmployeeBonus 'Sales', 10;

SELECT \* FROM Employees;

***-- Test Scenario 3: Transfer***

EXEC TransferFunds 1, 2, 3000;

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.**

**Explanation:**

* Defined a FibonacciGenerator class that generates the first N numbers of the Fibonacci sequence using iterative logic.
* The method generate(int count) handles edge cases like 0 or 1 input and returns a List<Integer> containing the sequence.
* JUnit 4 is used to create a separate test class FibonacciGeneratorTest with methods annotated using @Test.
* Test cases are written to check expected output for inputs like 0, 1, and 5 using assertEquals().
* A pom.xml file is configured with the JUnit dependency to enable unit testing through Maven.

**Code:**

**fibonacciGenerator.java:**

package com.exampledemo;

import java.util.List;

import java.util.ArrayList;

import java.util.List;

public class fibonacciGenerator {

public List<Integer> generate(int count) {

List<Integer> result = new ArrayList<>();

if (count <= 0) return result;

result.add(0);

if (count == 1) return result;

result.add(1);

for (int i = 2; i < count; i++) {

int next = result.get(i - 1) + result.get(i - 2);

result.add(next);

}

return result;

}

}

**fibonacciGeneratorTest.java:**

package com.exampledemo;

import org.junit.Test;

import static org.junit.Assert.\*;

import java.util.Arrays;

import java.util.List;

public class fibonacciGeneratorTest {

fibonacciGenerator fibGen = new fibonacciGenerator();

*@Test*

public void testFibonacci5() {

List<Integer> expected = Arrays.*asList*(0, 1, 1, 2, 3);

List<Integer> actual = fibGen.generate(5);

System.***out***.println("Generated: " + actual);

*assertEquals*(expected, actual);

}

*@Test*

public void testFibonacci0() {

List<Integer> expected = Arrays.*asList*();

*assertEquals*(expected, fibGen.generate(0));

}

*@Test*

public void testFibonacci1() {

List<Integer> expected = Arrays.*asList*(0);

*assertEquals*(expected, fibGen.generate(1));

}

}

**Main.java:**

package com.exampledemo;

import java.util.List;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.***in***);

System.***out***.print("Enter the number of Fibonacci terms: ");

int n = sc.nextInt();

fibonacciGenerator generator = new fibonacciGenerator();

List<Integer> series = generator.generate(n);

System.***out***.println("Fibonacci Series:");

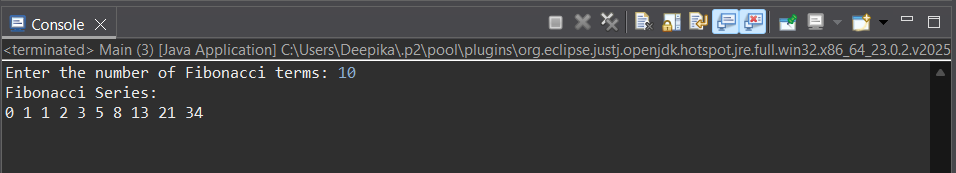
for (int num : series) {

System.***out***.print(num + " ");

}

}

}

**Output:  
**

**Exercise 3: Assertions in JUnit**

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

**Explanation:**

* A test class AssertionsTest was created to demonstrate various **JUnit assertion methods.**
* The @Test method includes assertions such as assertEquals, assertTrue, assertFalse, assertNull, and assertNotNull.
* These assertions help verify expected conditions in code and throw errors when conditions fail.
* The use of different assertion types ensures robust and comprehensive test coverage for logic verification.
* Successful test execution with a green bar confirms that all conditions and expectations were correctly validated.

**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);

*assertTrue*(5 > 3);

*assertFalse*(5 < 3);

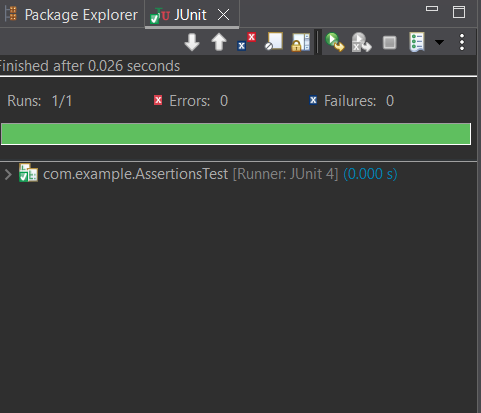
*assertNull*(null);

*assertNotNull*(new Object());

}

}

**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.**

**Explanation:**

* Implemented a BankAccount class with deposit(), withdraw(), and getBalance() methods.
* Structured JUnit tests using the **Arrange-Act-Assert (AAA)** pattern for clarity and readability.
* Used @Before method to set up a fresh account object with default balance before each test.
* Used @After method to log a message after each test run (can also be used for cleanup).
* Verified correct behavior of deposit and withdrawal using assertions like assertEquals().

**Code:**

**BankAccount.java:**

package com.example;

public class BankAccount {

private int balance;

public BankAccount(int initialBalance) {

this.balance = initialBalance;

}

public void deposit(int amount) {

balance += amount;

}

public void withdraw(int amount) {

balance -= amount;

}

public int getBalance() {

return balance;

}

}

**BankAccountTest.java:**

package com.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankAccountTest {

private BankAccount account;

*@Before*

public void setUp() {

account = new BankAccount(1000);

System.***out***.println("Before Test: Balance is reset to 1000");

}

*@After*

public void tearDown() {

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

}

*@Test*

public void testDeposit() {

account.deposit(500);

*assertEquals*(1500, account.getBalance());

}

*@Test*

public void testWithdraw() {

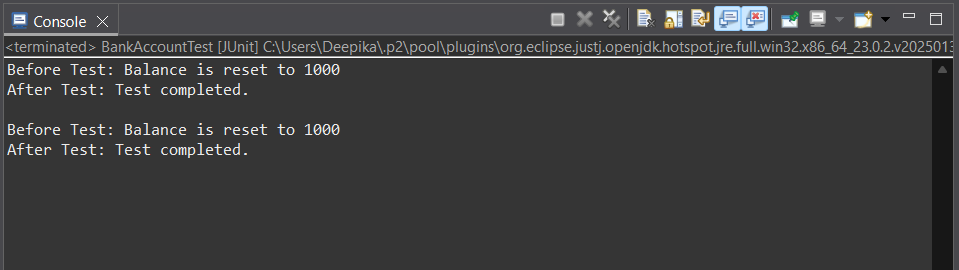
account.withdraw(300);

*assertEquals*(700, account.getBalance());

}

}

**Output:**

****

**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.**

**Explanation:**

* Created an interface ExternalApi to simulate an external service.
* Used Mockito to mock the external API and stub its getData() method.
* Injected the mocked API into MyService for isolated unit testing.
* Wrote a JUnit test that verifies service behaviour using predefined mock data.
* Ensured the test passed using assertEquals, confirming expected output.

**Code:**

**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 (Interface):**

package com.examplemock;

public interface ExternalApi {

String getData();

}

**MyService.java:**

package com.examplemock;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**MyServiceTest.java:**

package com.examplemock;

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() {

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

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

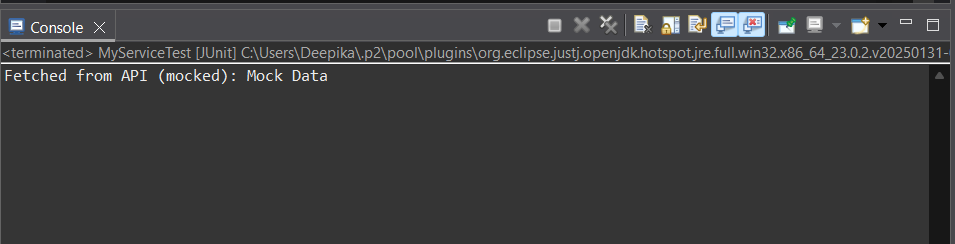
System.***out***.println("Fetched from API (mocked): " + result);

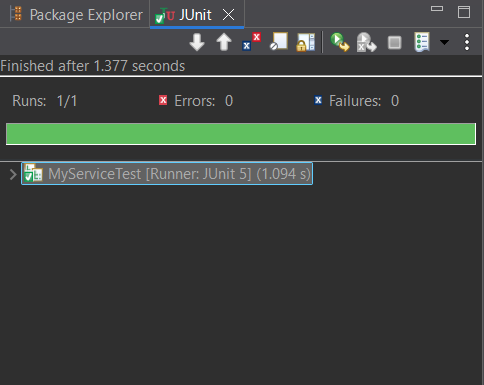
*assertEquals*("Mock Data", result);

}

}

**Output:**

****

****

**Exercise 2: Verifying Interactions**

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

**Explanation:**

* Created a mock of ExternalApi to simulate external behavior.
* Injected the mock into MyService to isolate it from real API calls.
* Called the fetchData() method that internally uses the mocked getData().
* Used Mockito.verify() to confirm that getData() was called exactly once.
* Test passed successfully, proving that the interaction occurred as expected.

**Code:**

**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 (Interface):**

package mockverify;

public interface ExternalApi {

String getData();

}

**MyServiceMock.java:**

package mockverify;

public class MyServiceMock {

private ExternalApi api;

public MyServiceMock(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

**MyServiceMockTest.java:**

package mockverify;

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

public class MyServiceTestMock {

*@Test*

public void testVerifyInteraction() {

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

MyServiceMock service = new MyServiceMock(mockApi);

service.fetchData();

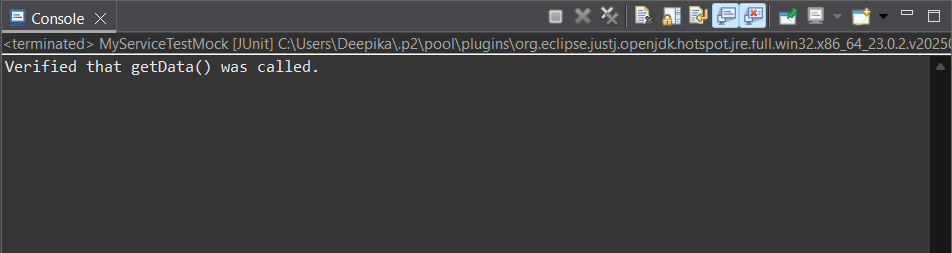
*verify*(mockApi).getData();

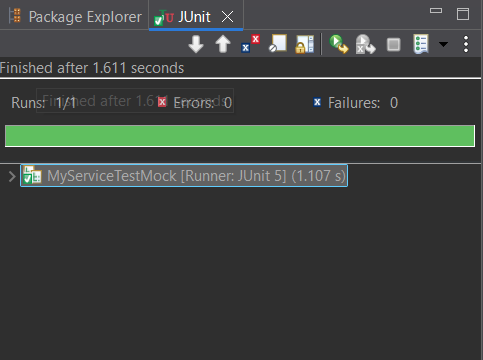
System.***out***.println("Verified that getData() was called.");

}

}

**Output:**

****

****

**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.**

**Explanation:**

* Created a Maven-based Java app to demonstrate SLF4J logging.
* Added dependencies for slf4j-api and logback-classic to pom.xml.
* Used LoggerFactory.getLogger() to initialize the logger.
* Logged error and warning messages using logger.error() and logger.warn().
* Output was printed to console with log level, timestamp, and message.

**Code:**

**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 com.examplelogging;

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) {

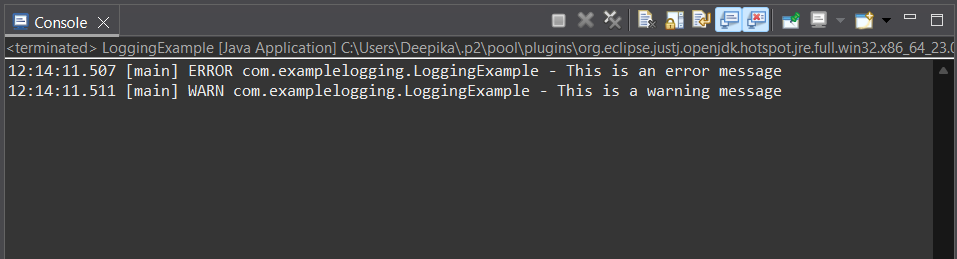
***logger***.error("This is an error message");

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

}

}

**Output:**

****