**Week two digital nurture:**

**PL-sql, mockito, junit, sl4j**

**Plsql:**

**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.

**Solution**:

Create the required tables:

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

age NUMBER,

balance NUMBER(10,2),

is\_vip CHAR(1) DEFAULT 'N'

);

CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

interest\_rate NUMBER(5,2),

due\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

Insert the values:

INSERT INTO customers (customer\_id, name, age, balance) VALUES (1, 'John Doe', 65, 15000);

INSERT INTO customers (customer\_id, name, age, balance) VALUES (2, 'Jane Smith', 45, 8000);

INSERT INTO customers (customer\_id, name, age, balance) VALUES (3, 'Mark Lee', 70, 12000);

INSERT INTO customers (customer\_id, name, age, balance) VALUES (4, 'Emily Clark', 30, 5000);

INSERT INTO loans (loan\_id, customer\_id, interest\_rate, due\_date) VALUES (101, 1, 7.5, SYSDATE + 20);

INSERT INTO loans (loan\_id, customer\_id, interest\_rate, due\_date) VALUES (102, 2, 6.5, SYSDATE + 40);

INSERT INTO loans (loan\_id, customer\_id, interest\_rate, due\_date) VALUES (103, 3, 8.0, SYSDATE + 10);

INSERT INTO loans (loan\_id, customer\_id, interest\_rate, due\_date) VALUES (104, 4, 7.0, SYSDATE - 5);

1. 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.

Sol.

BEGIN

FOR people IN (

SELECT l.loan\_id

from loans l

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

WHERE c.age > 60

) LOOP

update loans

SET interest\_rate = interest\_rate - 1

WHERE loan\_id = people.loan\_id;

END LOOP;

COMMIT;

END;

2. 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.

Sol.

BEGIN

FOR c IN (

SELECT customer\_id

FROM customers

where balance > 10000

) LOOP

UPDATE customers

set is\_vip = 'Y'

WHERE customer\_id = c.customer\_id;

END LOOP;

COMMIT;

END;

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

Sol.

BEGIN

FOR r IN (

SELECT c.name, l.due\_date

FROM loans l

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

WHERE l.due\_date between sysdate AND sysdate + 30

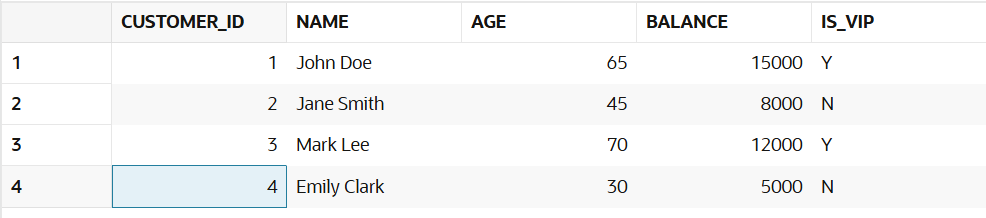
) LOOP

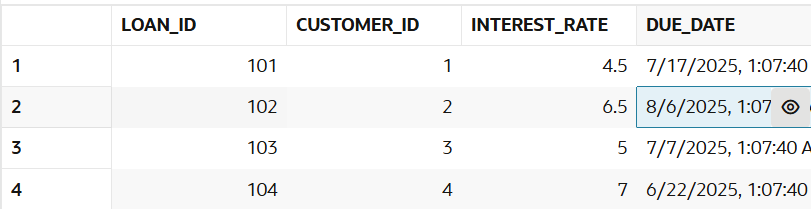
DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan is due on ' || TO\_CHAR(r.due\_date, 'YYYY-MM-DD') || ' for customer: ' || r.name);

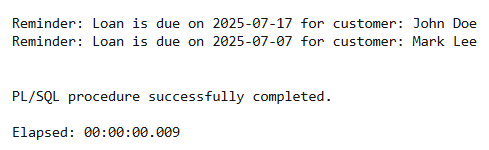
END LOOP;

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.

**Solution**:

Creating table:

CREATE TABLE savings\_accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

balance NUMBER(12,2)

);

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

name VARCHAR2(100),

department\_id NUMBER,

salary NUMBER(10,2)

);

CREATE TABLE bank\_accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

balance NUMBER(12,2)

);

Inserting values:

INSERT INTO savings\_accounts (account\_id, customer\_id, balance) VALUES (101, 1, 5000.00);

INSERT INTO savings\_accounts (account\_id, customer\_id, balance) VALUES (102, 2, 10000.00);

INSERT INTO savings\_accounts (account\_id, customer\_id, balance) VALUES (103, 3, 7500.00);

INSERT INTO employees (employee\_id, name, department\_id, salary) VALUES (201, 'Alice Johnson', 1, 5000.00);

INSERT INTO employees (employee\_id, name, department\_id, salary) VALUES (202, 'Bob Smith', 1, 5500.00);

INSERT INTO employees (employee\_id, name, department\_id, salary) VALUES (203, 'Charlie Lee', 2, 6000.00);

INSERT INTO employees (employee\_id, name, department\_id, salary) VALUES (204, 'Diana Adams', 2, 6200.00);

INSERT INTO bank\_accounts (account\_id, customer\_id, balance) VALUES (301, 1, 1000.00); -- Source

INSERT INTO bank\_accounts (account\_id, customer\_id, balance) VALUES (302, 2, 2000.00); -- Target

INSERT INTO bank\_accounts (account\_id, customer\_id, balance) VALUES (303, 3, 3000.00);

1. 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.

Sol.

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest as

BEGIN

UPDATE savings\_accounts

SET balance = balance \* 1.01;

END;

BEGIN

    ProcessMonthlyInterest;

end;

2. Write a stored procedure **UpdateEmployeeBonus** that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

Sol.

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN NUMBER,

p\_bonus\_percent IN NUMBER

) AS

BEGIN

UPDATE employees

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

WHERE department\_id = p\_department\_id;

END;

BEGIN

UpdateEmployeeBonus(2, 5);

END;

3. 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.

Sol.

CREATE OR REPLACE PROCEDURE TransferFunds (

p\_from\_account IN NUMBER,

p\_to\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_from\_balance NUMBER;

BEGIN

SELECT balance INTO v\_from\_balance

FROM bank\_accounts

WHERE account\_id = p\_from\_account

FOR UPDATE

IF v\_from\_balance < p\_amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in source account.');

END IF;

UPDATE bank\_accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account;

UPDATE bank\_accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account;

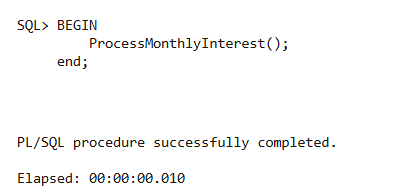
END;

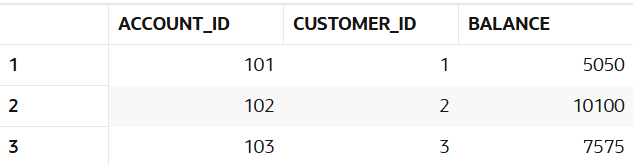
BEGIN

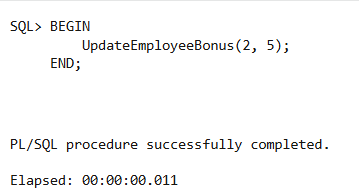
TransferFunds(1001, 1002, 250.00);

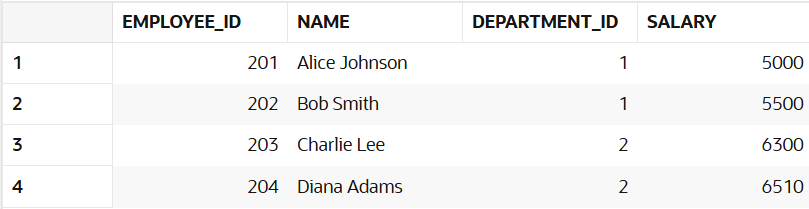
END;

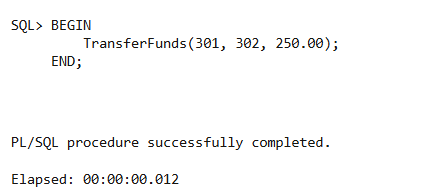
Output:

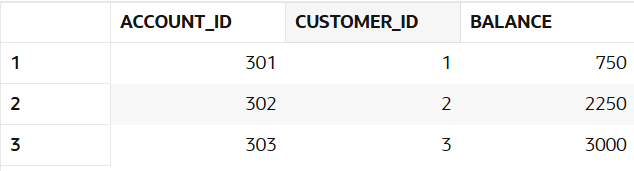












Junit:

**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>

1. **Create a new test class in your project.**

Solution:

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;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) throw new ArithmeticException("Division by zero");

return a / b;

}

}

CalculatorTest.java:

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAddition() {

assertEquals(8, calc.add(5, 3));

}

@Test

public void testSubtraction() {

assertEquals(2, calc.subtract(5, 3));

}

@Test

public void testMultiplication() {

assertEquals(15, calc.multiply(5, 3));

}

@Test

public void testDivision() {

assertEquals(2, calc.divide(6, 3));

}

@Test(expected = ArithmeticException.class)

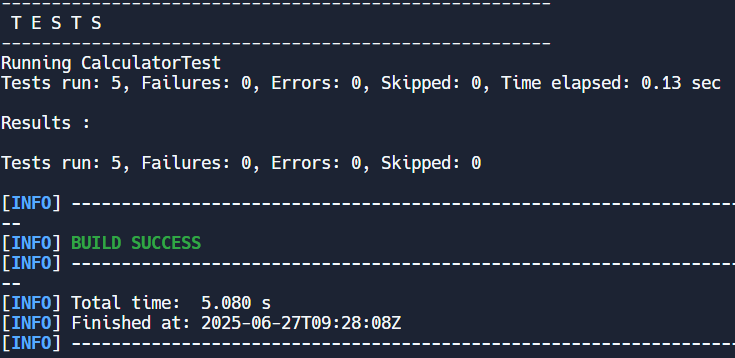
public void testDivisionByZero() {

calc.divide(5, 0);

}

}

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:

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());

}

}

Solution:

import org.junit.Test;

import static org.junit.Assert.\*;

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);

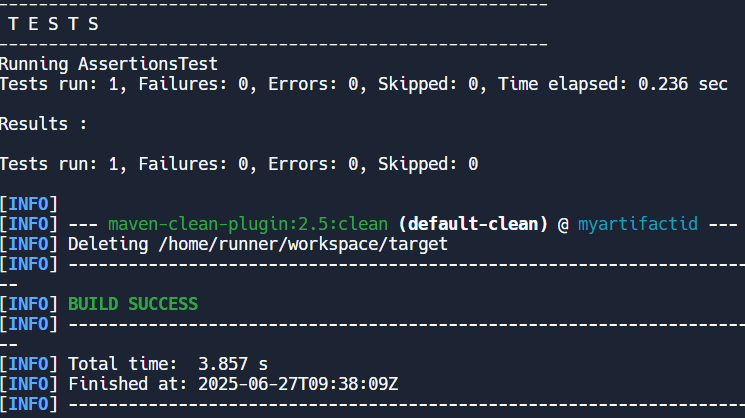
// Assert not null

assertNotNull(new Object());

}

}

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:

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;

}

public int multiply(int a, int b) {

return a \* b;

}

}

CalculatorTest.java

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("setup started");

}

@After

public void tearDown() {

System.out.println("all tests finished");

}

@Test

public void testAddition() {

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

assertEquals(5, result);

}

@Test

public void testMultiplication() {

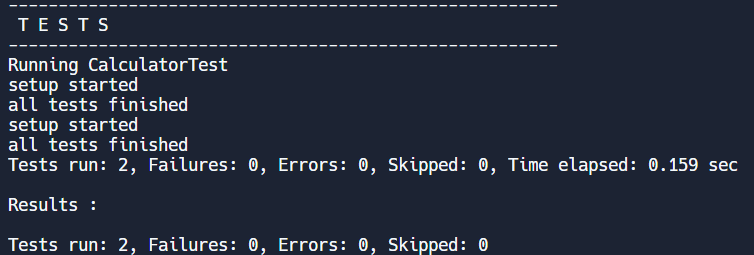
int res = calculator.multiply(4, 3);

assertEquals(12, res);

}

}

Output:



**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:

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);

}

}

Solution:

Exteranal.java

public interface ExteranalApi{

String getData();

}

MyService.java

public class MyService{

private ExteranalApi exteranalAPI;

public MyService(ExteranalApi exteranalAPI){

this.exteranalAPI = exteranalAPI;

}

public String fetchData(){

return exteranalAPI.getData();

}

}

MyServiceTest.java

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

import org.mockito.Mockito;

public class MyServiceTest {

@Test

public void testExternalApi() {

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

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

MyService service = new MyService(mockApi);

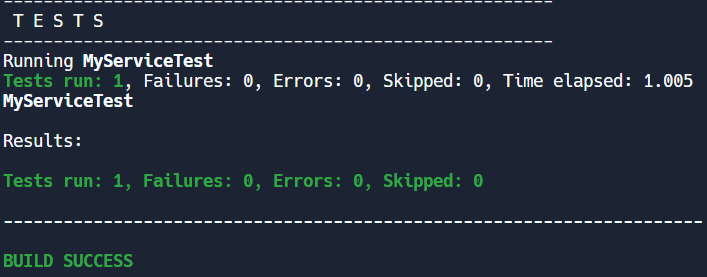
String result = service.fetchData();

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:

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();

}

}

Solution:

Exteranal.java

public interface ExteranalApi{

String getData();

}

MyService.java

public class MyService{

private ExteranalApi exteranalAPI;

public MyService(ExteranalApi exteranalAPI){

this.exteranalAPI = exteranalAPI;

}

public String fetchData(){

return exteranalAPI.getData();

}

}

MyServiceTest.java

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExteranalApi mockApi = mock(ExteranalApi.class);

MyService service = new MyService(mockApi);

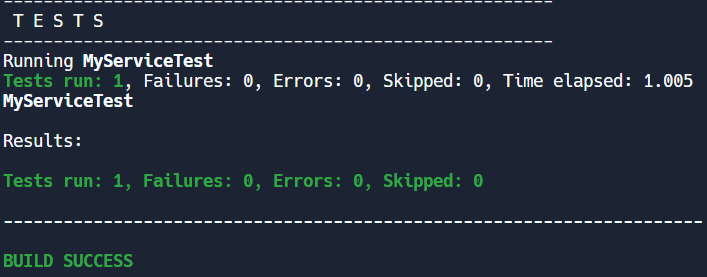
service.fetchData();

verify(mockApi).getData();

}

}

Output:



SL4J:

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>

1. 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");

} }

Solution:

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:

