**WEEK2:**

**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.**

**CODE:**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100),

age INT

);

CREATE TABLE loans (

loan\_id INT PRIMARY KEY,

customer\_id INT,

interest\_rate DECIMAL(5,2),

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

);

INSERT INTO customers (customer\_id, name, age) VALUES

(1, 'Alice', 65),

(2, 'Bob', 55);

INSERT INTO loans (loan\_id, customer\_id, interest\_rate) VALUES

(101, 1, 5.00),

(102, 2, 4.50);

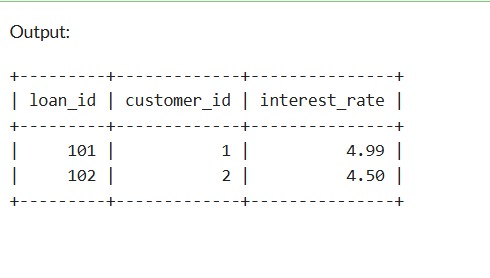
UPDATE loans l

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

SET l.interest\_rate = l.interest\_rate - 0.01

WHERE c.age > 60;

SELECT \* FROM loans;

**OUTPUT:** 

**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.**

**CODE:**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100),

balance DECIMAL(10,2),

IsVIP VARCHAR(5) DEFAULT 'FALSE'

);

INSERT INTO customers (customer\_id, name, balance) VALUES

(1, 'Alice', 15000),

(2, 'Bob', 8000),

(3, 'Charlie', 12000),

(4, 'Diana', 5000);

DELIMITER $$

CREATE PROCEDURE PromoteVIP()

BEGIN

UPDATE customers

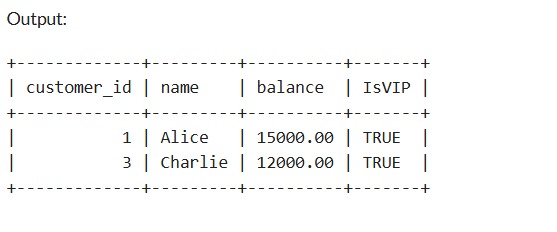
SET IsVIP = 'TRUE'

WHERE balance > 10000;

END$$

DELIMITER ;

CALL PromoteVIP(); SELECT \* FROM customers WHERE IsVIP = 'TRUE';



**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.**

**CODE:**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE loans (

loan\_id INT PRIMARY KEY,

customer\_id INT,

due\_date DATE,

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

);

INSERT INTO customers (customer\_id, name) VALUES

(1, 'Alice'),

(2, 'Bob'),

(3, 'Charlie');

INSERT INTO loans (loan\_id, customer\_id, due\_date) VALUES

(101, 1, CURDATE() + INTERVAL 10 DAY),

(102, 2, CURDATE() + INTERVAL 40 DAY),

(103, 3, CURDATE() + INTERVAL 25 DAY);

DELIMITER $$

CREATE PROCEDURE SendLoanReminders()

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE v\_customer\_name VARCHAR(100);

DECLARE v\_due\_date DATE;

DECLARE cur CURSOR FOR

SELECT c.name, l.due\_date

FROM loans l

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

WHERE l.due\_date BETWEEN CURDATE() AND CURDATE() + INTERVAL 30 DAY;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN cur;

read\_loop: LOOP

FETCH cur INTO v\_customer\_name, v\_due\_date;

IF done THEN

LEAVE read\_loop;

END IF;

SELECT CONCAT('Reminder: Loan due for ', v\_customer\_name, ' on ', DATE\_FORMAT(v\_due\_date, '%Y-%m-%d')) AS Reminder\_Message;

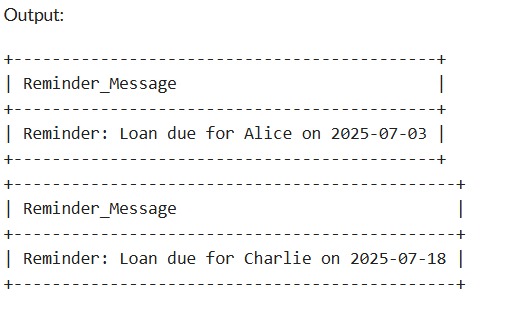
END LOOP;

CLOSE cur;

END$$

DELIMITER ;

CALL SendLoanReminders();



**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.

CODE:

CREATE TABLE savings\_accounts (

account\_id INT PRIMARY KEY,

balance DECIMAL(10,2));

INSERT INTO savings\_accounts (account\_id, balance) VALUES

(1, 1000.00),

(2, 2000.00),

(3, 1500.00);

DELIMITER $$

CREATE PROCEDURE ProcessMonthlyInterest()

BEGIN

UPDATE savings\_accounts

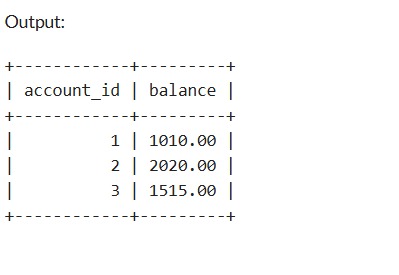
SET balance = balance \* 1.01;

END$$

DELIMITER ;

CALL ProcessMonthlyInterest();

SELECT \* FROM savings\_accounts;



**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.

CODE:

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

department\_id INT,

salary DECIMAL(10,2));

INSERT INTO employees (employee\_id, department\_id, salary) VALUES

(1, 101, 50000.00),

(2, 102, 60000.00),

(3, 101, 55000.00),

(4, 103, 70000.00);

DELIMITER $$

CREATE PROCEDURE UpdateEmployeeBonus(IN dept\_id INT, IN bonus\_percent DECIMAL(5,2))

BEGIN

UPDATE employees

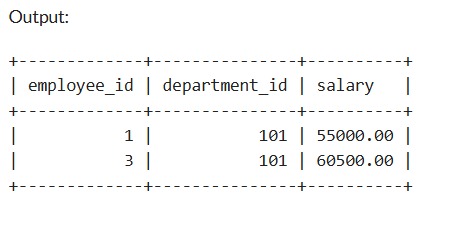
SET salary = salary + (salary \* bonus\_percent / 100)

WHERE department\_id = dept\_id;

END$$

DELIMITER ;

CALL UpdateEmployeeBonus(101, 10);

SELECT \* FROM employees WHERE department\_id = 101;

**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.

CODE:

CREATE TABLE savings\_accounts (

account\_id INT PRIMARY KEY,

balance DECIMAL(10,2));

INSERT INTO savings\_accounts (account\_id, balance) VALUES

(1, 2000.00),

(2, 1500.00),

(3, 3000.00);

DELIMITER $$

CREATE PROCEDURE TransferFunds(

IN src\_account\_id INT,

IN dest\_account\_id INT,

IN transfer\_amount DECIMAL(10,2))

BEGIN

DECLARE src\_balance DECIMAL(10,2);

SELECT balance INTO src\_balance

FROM savings\_accounts

WHERE account\_id = src\_account\_id;

IF src\_balance IS NULL THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Source account does not exist';

ELSEIF src\_balance < transfer\_amount THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Insufficient balance in source account';

ELSE

UPDATE savings\_accounts

SET balance = balance - transfer\_amount

WHERE account\_id = src\_account\_id;

UPDATE savings\_accounts

SET balance = balance + transfer\_amount

WHERE account\_id = dest\_account\_id;

IF ROW\_COUNT() = 0 THEN

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = 'Destination account does not exist';

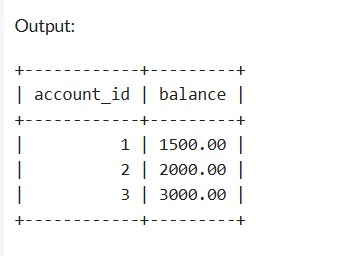
END IF;

END IF;

END$$

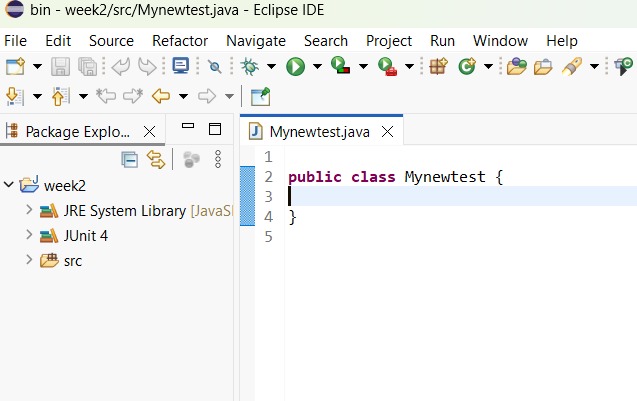
DELIMITER ;

CALL TransferFunds(1, 2, 500.00);

SELECT \* FROM savings\_accounts; 

**Exercise 1: Setting Up Junit**

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.



**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:

import org.junit.Test;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertTrue;

import static org.junit.Assert.assertFalse;

import static org.junit.Assert.assertNull;

import static org.junit.Assert.assertNotNull;

public class AssertionsTest {

@Test

public void testAssertions() {

System.out.println("Running testAssertions...");

System.out.println("Asserting: 2 + 3 equals 5");

assertEquals(5, 2 + 3);

System.out.println("Asserting: 5 is greater than 3");

assertTrue(5 > 3);

System.out.println("Asserting: 5 is NOT less than 3");

assertFalse(5 < 3);

System.out.println("Asserting: a null reference is null");

assertNull(null);

System.out.println("Asserting: a new Object() is not null");

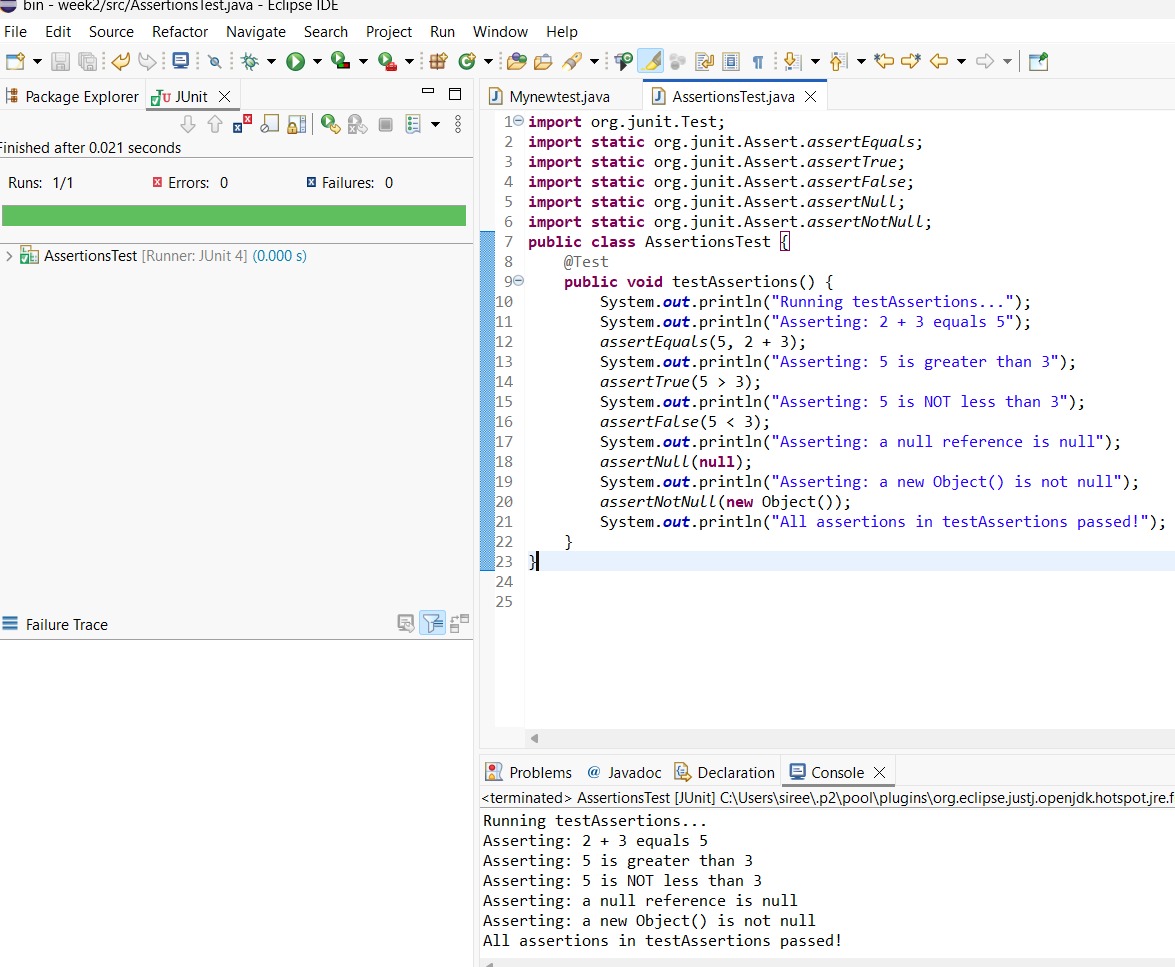
assertNotNull(new Object());

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

    }

}

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.

**Calculator.java:**

package com.example;

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:**

package com.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

import static org.junit.Assert.assertNotNull;

import static org.junit.Assert.assertTrue;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("Setup: Calculator initialized for a new test.");

}

@After

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator instance cleared.");

}

@Test

public void testAddPositiveNumbers() {

System.out.println("--- Running testAddPositiveNumbers ---");

int num1 = 5;

int num2 = 3;

int expectedSum = 8;

int actualSum = calculator.add(num1, num2);

System.out.println("Act: Performed addition " + num1 + " + " + num2);

assertEquals("The sum of positive numbers should be correct", expectedSum, actualSum);

assertNotNull("Calculator object should not be null", calculator);

System.out.println("Assert: Verified sum is " + actualSum);

}

@Test

public void testSubtractNumbers() {

System.out.println("--- Running testSubtractNumbers ---");

int num1 = 10;

int num2 = 4;

int expectedDifference = 6;

int actualDifference = calculator.subtract(num1, num2);

System.out.println("Act: Performed subtraction " + num1 + " - " + num2);

assertEquals("The difference should be correct", expectedDifference, actualDifference);

assertTrue("Difference should be positive", actualDifference > 0);

System.out.println("Assert: Verified difference is " + actualDifference);

    }

}

OUTPUT: