**PL/SQL**

**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:**

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, age, loan\_interest\_rate

FROM customers

FOR UPDATE;

v\_customer\_id customers.customer\_id%TYPE;

v\_age customers.age%TYPE;

v\_loan\_interest\_rate customers.loan\_interest\_rate%TYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO v\_customer\_id, v\_age, v\_loan\_interest\_rate;

EXIT WHEN cust\_cursor%NOTFOUND;

IF v\_age > 60 THEN

UPDATE customers

SET loan\_interest\_rate = loan\_interest\_rate - 1

WHERE CURRENT OF cust\_cursor;

END IF;

END LOOP;

CLOSE cust\_cursor;

DBMS\_OUTPUT.PUT\_LINE('Interest rate discount applied for eligible customers.');

END;

/

**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:**

DECLARE

CURSOR cust\_cursor IS

SELECT customer\_id, balance

FROM customers

FOR UPDATE;

v\_customer\_id customers.customer\_id%TYPE;

v\_balance customers.balance%TYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO v\_customer\_id, v\_balance;

EXIT WHEN cust\_cursor%NOTFOUND;

IF v\_balance > 10000 THEN

UPDATE customers

SET IsVIP = 'TRUE'

WHERE CURRENT OF cust\_cursor;

END IF;

END LOOP;

CLOSE cust\_cursor;

DBMS\_OUTPUT.PUT\_LINE('VIP status updated for eligible customers.');

END;

/

**Output:**



**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:**

DECLARE

CURSOR loan\_cursor IS

SELECT customer\_id, loan\_id, due\_date

FROM loans

WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30;

v\_customer\_id loans.customer\_id%TYPE;

v\_loan\_id loans.loan\_id%TYPE;

v\_due\_date loans.due\_date%TYPE;

BEGIN

OPEN loan\_cursor;

LOOP

FETCH loan\_cursor INTO v\_customer\_id, v\_loan\_id, v\_due\_date;

EXIT WHEN loan\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ID ' || v\_loan\_id ||

' for Customer ID ' || v\_customer\_id ||

' is due on ' || TO\_CHAR(v\_due\_date, 'DD-MON-YYYY'));

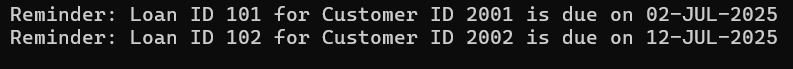
END LOOP;

CLOSE loan\_cursor;

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**.

**Code:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

UPDATE accounts

SET balance = balance + (balance \* 0.01)

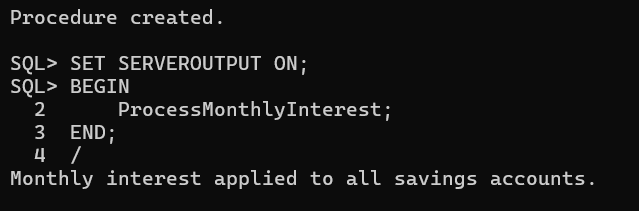
WHERE account\_type = 'SAVINGS';

DBMS\_OUTPUT.PUT\_LINE('Monthly interest applied to all savings accounts.');

END;

/

**Output:**



**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 OR REPLACE PROCEDURE UpdateEmployeeBonus (

p\_department\_id IN employees.department\_id%TYPE,

p\_bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE employees

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

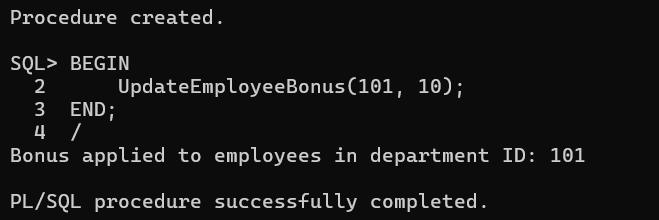
WHERE department\_id = p\_department\_id;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied to employees in department ID: ' || p\_department\_id);

END;

/

**Output:**



**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 OR REPLACE PROCEDURE TransferFunds (

p\_from\_account\_id IN accounts.account\_id%TYPE,

p\_to\_account\_id IN accounts.account\_id%TYPE,

p\_amount IN NUMBER

) IS

v\_balance accounts.balance%TYPE;

BEGIN

SELECT balance INTO v\_balance

FROM accounts

WHERE account\_id = p\_from\_account\_id

FOR UPDATE;

IF v\_balance < p\_amount THEN

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

END IF

UPDATE accounts

SET balance = balance - p\_amount

WHERE account\_id = p\_from\_account\_id;

UPDATE accounts

SET balance = balance + p\_amount

WHERE account\_id = p\_to\_account\_id;

DBMS\_OUTPUT.PUT\_LINE('Transfer of ' || p\_amount || ' completed from Account ' ||

p\_from\_account\_id || ' to Account ' || p\_to\_account\_id);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20002, 'One or both account IDs are invalid.');

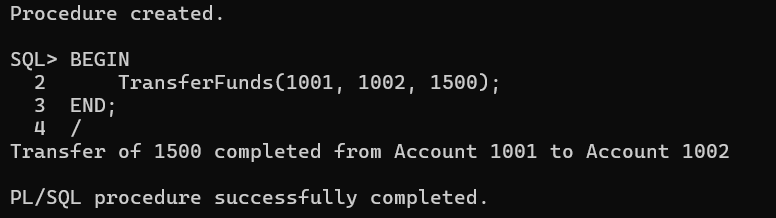
WHEN OTHERS THEN

RAISE\_APPLICATION\_ERROR(-20003, 'An unexpected error occurred: ' || SQLERRM);

END;

/

**Output:**



**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: junit junit 4.13.2 test 3. Create a new test class in your project.**

**Code:**

//Calculator.java

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

//CalculatorTest.java

package com.example;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

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

assertEquals(5, result);

}

}

**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()); } }**

**Code:**

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. Steps: 1. Write tests using the AAA pattern. 2. Use @Before and @After annotations for setup and teardown methods.**

**Code:**

//Calculator.java

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

//CalculatorTest.java

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

}

@After

public void tearDown() {

calculator = null;

}

@Test

public void testAdd() {

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

assertEquals(5, result);

}

@Test

public void testMultiply() {

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

assertEquals(12, result);

}

}

**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:**

//ExternalApi.java

public interface ExternalApi {

String getData();

}

//MyService.java

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.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() {

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

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

MyService service = new MyService(mockApi);

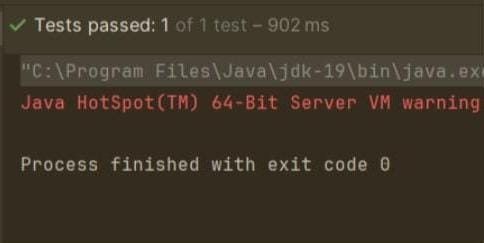
String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

**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(); } }**

**Code:**

//ExternalApi.java

public interface ExternalApi {

String getData();

}

//MyService.java

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.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 testVerifyInteraction() {

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

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

MyService service = new MyService(mockApi);

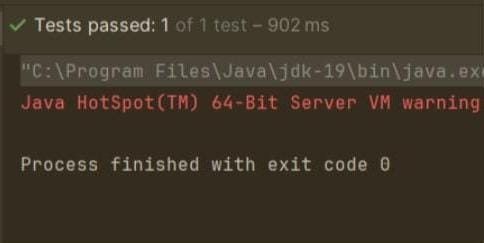
service.fetchData();

verify(mockApi).getData();

}

}

**Output:**



**LOGGING USING SLF4J**

**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: org.slf4j slf4j-api 1.7.30 ch.qos.logback logback-classic 1.2.3 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:**

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:**

