Week - 2

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.

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: Apply 1% discount for customers above 60

CODE :

```
BEGIN
    FOR cust_rec IN (
        SELECT c.CustomerID, TRUNC(MONTHS_BETWEEN(SYSDATE,
c.DOB) / 12) AS Age
        FROM Customers c
    ) LOOP
        IF cust_rec.Age > 60 THEN
            UPDATE Loans
            SET InterestRate = InterestRate - 1
            WHERE CustomerID = cust_rec.CustomerID;
        END IF;
    END LOOP;

    COMMIT;
END;
```

🔍 **Data:**

- John Doe: DOB = 1985-05-15 → age = 40 (as of 2025-06-26)
- Jane Smith: DOB = 1990-07-20 → age = 34

 **Output:**

No customers are over 60, so **no update occurs**.
There is **no visible output**, but the script runs successfully.

Week - 2

Scenario 2: Promote customers to VIP based on balance

Since your original `Customers` table doesn't have an `IsVIP` column, first **alter the table** to add it:

```
ALTER TABLE Customers ADD (IsVIP CHAR(1)); -- 'Y' for VIP,
NULL otherwise
```

NOW THE PL/SQL BLOCK:

```
BEGIN
    FOR cust IN (SELECT CustomerID, Balance FROM Customers)
LOOP
        IF cust.Balance > 10000 THEN
            UPDATE Customers
            SET IsVIP = 'Y'
            WHERE CustomerID = cust.CustomerID;
        END IF;
    END LOOP;

    COMMIT;
END;
```

YOU ADDED:

```
INSERT INTO Customers (...) VALUES (1, 'John Doe', ..., 1000,
...);
INSERT INTO Customers (...) VALUES (2, 'Jane Smith', ...,
1500, …);
```

🔍 **Data:**

- John Doe: Balance = 1000
- Jane Smith: Balance = 1500

Both balances are **below $10,000**, so again, **no update occurs**.

 **Output:**

Again, no visible output. You can verify with:

```
SELECT CustomerID, Name, IsVIP FROM Customers;
```

Week - 2

Result:

```
CUSTOMERID | NAME         | ISVIP
-----------|--------------|-------
1          | John Doe     | NULL
2          | Jane Smith   | NULL
```

Scenario 3: Remind customers with loans due in next 30 days

CODE :

```
DECLARE
    v_name Customers.Name%TYPE;
BEGIN
    FOR loan_rec IN (
        SELECT l.LoanID, l.CustomerID, l.EndDate
        FROM Loans l
        WHERE l.EndDate BETWEEN SYSDATE AND SYSDATE + 30
    ) LOOP
        SELECT Name INTO v_name FROM Customers WHERE
CustomerID = loan_rec.CustomerID;
        DBMS_OUTPUT.PUT_LINE('Reminder: Dear ' || v_name ||
                             ', your loan ID ' ||
loan_rec.LoanID ||
                             ' is due on ' ||
TO_CHAR(loan_rec.EndDate, 'YYYY-MM-DD'));
    END LOOP;
END;
```

MAKE SURE YOU ENABLE OUTPUT IN YOUR SQL ENVIRONMENT (LIKE SQL DEVELOPER OR SQL*PLUS):

```
SET SERVEROUTPUT ON;
```

YOU INSERTED :

```
INSERT INTO Loans (...) VALUES (1, 1, 5000, 5, SYSDATE,
ADD_MONTHS(SYSDATE, 60));
```

That loan ends in **60 months = 5 years**, so due date is **~2030-06-26**, not within 30 days.

Output:

No matching loans due in next 30 days → **no output from `DBMS_OUTPUT.PUT_LINE`**.

Week - 2

To test it, try inserting a loan that ends within 30 days:

```
INSERT INTO Loans (LoanID, CustomerID, LoanAmount,
InterestRate, StartDate, EndDate)
VALUES (2, 2, 2000, 4, SYSDATE, SYSDATE + 15); -- ends in 15
days
```

Then re-run the block and output will be:

```
Reminder: Dear Jane Smith, your loan ID 2 is due on
2025-07-11
```

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: ProcessMonthlyInterest

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS
BEGIN
    UPDATE Accounts
    SET Balance = Balance * 1.01,  -- Apply 1% interest
        LastModified = SYSDATE
    WHERE AccountType = 'Savings';

    COMMIT;
END;
/
```

Week - 2

OUTPUT :

- This procedure updates **all savings accounts** by adding 1% interest to their current balance.

Before running:

AccountID	CustomerID	AccountType	Balance
1	1	Savings	1000
2	2	Checking	1500

After running:

```
EXEC ProcessMonthlyInterest;
```

AccountID	CustomerID	AccountType	Balance
1	1	Savings	1010
2	2	Checking	1500

No visible output is printed, but balances are updated in the table.

Scenario 2: UpdateEmployeeBonus

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (
    p_Department IN VARCHAR2,
    p_BonusPercent IN NUMBER -- e.g., 5 means 5%
) IS
BEGIN
    UPDATE Employees
    SET Salary = Salary + (Salary * p_BonusPercent / 100)
    WHERE Department = p_Department;

    COMMIT;
END;
/
```

Week - 2

Usage example:

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

OUTPUT :

- This procedure adds a bonus percentage to the salary of employees in a given department.

Before running:

EmployeeID	Name	Department	Salary
1	Alice Johnson	HR	70000
2	Bob Brown	IT	60000

RUN:

```
EXEC UpdateEmployeeBonus('IT', 10);
```

After running:

EmployeeID	Name	Department	Salary
1	Alice Johnson	HR	70000
2	Bob Brown	IT	66000

No output printed; the salaries in the table are updated.

Scenario 3: TransferFunds

```
CREATE OR REPLACE PROCEDURE TransferFunds (
    p_FromAccountID IN NUMBER,
    p_ToAccountID IN NUMBER,
    p_Amount IN NUMBER
) IS
    v_FromBalance NUMBER;
BEGIN
    -- Check balance of source account
    SELECT Balance INTO v_FromBalance FROM Accounts WHERE
AccountID = p_FromAccountID;
    IF v_FromBalance < p_Amount THEN
        RAISE_APPLICATION_ERROR(-20001, 'Insufficient balance
in source account.');
```

Week - 2

```
    ELSE
        -- Deduct from source
        UPDATE Accounts
        SET Balance = Balance - p_Amount,
            LastModified = SYSDATE
        WHERE AccountID = p_FromAccountID;

        -- Add to destination
        UPDATE Accounts
        SET Balance = Balance + p_Amount,
            LastModified = SYSDATE
        WHERE AccountID = p_ToAccountID;

        COMMIT;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR(-20002, 'Account not
found.');
END;
/
```

Usage example:

```
EXEC TransferFunds(1, 2, 500); -- transfer $500 from account
1 to account 2
```

OUTPUT :

- Transfers money between accounts if the source account has enough balance.
- Raises error if not enough balance or account not found.

Example call:

```
EXEC TransferFunds(1, 2, 500);
```

Before:

AccountID	Balance
1	1000
2	1500

Week - 2

After:

AccountID	Balance
1	500
2	2000

If insufficient funds:

```
EXEC TransferFunds(1, 2, 2000);
```

Output:

```
ORA-20001: Insufficient balance in source account.
```

If invalid account ID:

```
EXEC TransferFunds(999, 2, 100);
```

Output:

```
ORA-20002: Account not found.
```

Week - 2

JUnit Testing Exercises

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.

Step 1: Create a new Java project

- Open your IDE (IntelliJ IDEA, Eclipse, etc.)
- Create a new Java project.

Step 2: Add JUnit dependency

- If you use **Maven**, add this to your `pom.xml` inside `<dependencies>`:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

- If you don't use Maven, download the JUnit 4.13.2 jar and add it to your project's classpath.

Step 3: Create a new test class

- In your `src/test/java` folder (or wherever your tests live), create a new Java class, e.g., `CalculatorTest.java`.

Week - 2

Example: Simple test class using JUnit 4

Create a simple class to test:

```
// src/main/java/Calculator.java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Create a test class:

```
// src/test/java/CalculatorTest.java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(5, 7);
        assertEquals(12, result);
    }
}
```

How to run tests

- In IntelliJ, right-click the test class → Run 'CalculatorTest'.
- In Eclipse, right-click the test → Run As → JUnit Test.
- Maven users can run:

```
mvn test
```

Week - 2

Output when running `CalculatorTest`

1. If the test passes:

```
JUnit version 4.13.2
.
Time: 0.005

OK (1 test)
```

- The dot `.` means the test ran successfully.
- `OK (1 test)` means one test passed.

2. If the test fails (for example, you change the expected value to 13 instead of 12):

```
assertEquals(13, result);
```

THE OUTPUT WILL LOOK LIKE:

```
mathematica
JUnit version 4.13.2
.E
Time: 0.005
There was 1 failure:
1) testAdd(CalculatorTest)
java.lang.AssertionError:
Expected :13
Actual   :12
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.Assert.failNotEquals(Assert.java:834)
    at org.junit.Assert.assertEquals(Assert.java:645)
    at org.junit.Assert.assertEquals(Assert.java:631)
    at CalculatorTest.testAdd(CalculatorTest.java:10)

FAILURES!!!
Tests run: 1,  Failures: 1
```

Week - 2

- The `E` or `F` indicates errors or failures.
- It tells you the expected and actual values, along with the line number.

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 Code with explanations:

```
import static org.junit.Assert.*;  // Import all assertion
methods
import org.junit.Test;

public class AssertionsTest {

    @Test
    public void testAssertions() {
        // Assert that two values are equal
        assertEquals("Sum of 2 + 3 should be 5", 5, 2 + 3);

        // Assert that a condition is true
```

Week - 2

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

        // Assert that a condition is false
        assertFalse("5 should NOT be less than 3", 5 < 3);

        // Assert that an object is null
        assertNull("Object should be null", null);

        // Assert that an object is not null
        assertNotNull("New Object should NOT be null", new
Object());
    }
}
```

How it works:

- `assertEquals(expected, actual)`
  Checks if the expected value matches the actual value.
- `assertTrue(condition)`
  Passes if the condition is `true`.
- `assertFalse(condition)`
  Passes if the condition is `false`.
- `assertNull(object)`
  Passes if the object is `null`.
- `assertNotNull(object)`
  Passes if the object is **not** `null`.

Running the test

When you run this test, you should see output indicating all assertions passed:

```
JUnit version 4.13.2
.
Time: 0.004

OK (1 test)
```

Week - 2

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.

What is AAA pattern?

- **Arrange**: Prepare objects and set up the test data.
- **Act**: Execute the method or functionality you want to test.
- **Assert**: Verify the result matches your expectations.

Using `@Before` and `@After`

- `@Before` — runs **before** each test method (setup).
- `@After` — runs **after** each test method (cleanup or teardown).

Example: BankAccount test with AAA, setup, teardown

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.After;
import org.junit.Test;

public class BankAccountTest {

    private BankAccount account;

    // Setup method runs before each test
    @Before
    public void setUp() {
        account = new BankAccount();
        account.deposit(100);  // Arrange initial balance
```

Week - 2

```
        System.out.println("Setup: Created account with
$100");
    }

    // Teardown method runs after each test
    @After
    public void tearDown() {
        account = null;
        System.out.println("Teardown: Account cleaned up");
    }

    @Test
    public void testWithdraw() {
        // Arrange: already done in setUp()

        // Act
        account.withdraw(30);

        // Assert
        assertEquals(70, account.getBalance(), 0.001);
    }

    @Test
    public void testDeposit() {
        // Arrange: already done in setUp()

        // Act
        account.deposit(50);

        // Assert
        assertEquals(150, account.getBalance(), 0.001);
    }
}

// BankAccount class for reference
class BankAccount {
    private double balance = 0;

    public void deposit(double amount) {
        if(amount > 0) {
            balance += amount;
        }
    }
```

Week - 2

```
    public void withdraw(double amount) {
        if(amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

What happens when you run this?

- `setUp()` runs before each test, creating a fresh `BankAccount` with $100.
- Each test performs its own actions (`withdraw`, `deposit`) and asserts results.
- `tearDown()` runs after each test, cleaning up.

Output :

```
Setup: Created account with $100
Teardown: Account cleaned up
Setup: Created account with $100
Teardown: Account cleaned up

JUnit version 4.13.2
..
Time: 0.005

OK (2 tests)
```

Summary:

- Use **AAA** to clearly separate setup, action, and verification in your tests.
- Use `@Before` to avoid repeating setup code in every test.
- Use `@After` for cleanup after tests if needed.

Week - 2

Mockito Hands-On 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 :

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

// Assuming these are your external API and service classes
interface ExternalApi {
    String getData();
}

class MyService {
    private final ExternalApi api;

    public MyService(ExternalApi api) {
        this.api = api;
    }
```

Week - 2

```
    public String fetchData() {
        return api.getData();
    }
}

public class MyServiceTest {

    @Test
    public void testExternalApi() {
        // Step 1: Create a mock object for the external API
        ExternalApi mockApi = Mockito.mock(ExternalApi.class);

        // Step 2: Stub the getData() method to return a predefined value
        when(mockApi.getData()).thenReturn("Mock Data");

        // Use the mock in the service
        MyService service = new MyService(mockApi);

        // Step 3: Call the service method and verify the stubbed response
        String result = service.fetchData();

        // Assert that the mocked response is returned
        assertEquals("Mock Data", result);
    }
}
```

OUTPUT :

If you run the `MyServiceTest.testExternalApi()` test in your IDE or build tool (like Maven or Gradle), the output would be:

- The test **passes** if the stubbed method returns the expected `"Mock Data"`.
- If something goes wrong (e.g., the result is different), the test **fails** with an assertion error.

Example test run output in console:

```
[INFO]
-------------------------------------------------------
[INFO]  T E S T S
[INFO]
-------------------------------------------------------
[INFO] Running MyServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.02 s - in MyServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Week - 2

```
[INFO]
[INFO]
------------------------------------------------------------------
-----------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------
-----------
```

If you want to **print** the result inside the test for demonstration, you could add:

```
System.out.println("Result from fetchData(): " + result);
```

and then the output would be:

```
Result from fetchData(): Mock Data
```

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();
}
}
```

Week - 2

CODE :

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

// External API interface
interface ExternalApi {
    String getData();
}

// Service class that uses the ExternalApi
class MyService {
    private final ExternalApi api;

    public MyService(ExternalApi api) {
        this.api = api;
    }

    public String fetchData() {
        return api.getData();
    }
}

public class MyServiceTest {

    @Test
    public void testVerifyInteraction() {
        // Step 1: Create mock object
        ExternalApi mockApi = Mockito.mock(ExternalApi.class);

        // Create service with the mock
        MyService service = new MyService(mockApi);

        // Step 2: Call the method
        service.fetchData();

        // Step 3: Verify the interaction (that getData() was called exactly once)
        verify(mockApi).getData();
    }
}
```

Explanation:

- `verify(mockApi).getData();` checks that `getData()` was called on the mock object.
- If `getData()` was not called, or called with unexpected arguments, the test fails.

Week - 2

What happens when you run your test `testVerifyInteraction()`:

- The test will **pass** if the `getData()` method on the mock `ExternalApi` was called exactly once.
- The test will **fail** if `getData()` was never called or called a different number of times.

Sample test run output in the console (successful run):

```
[INFO]
-------------------------------------------------------
[INFO]  T E S T S
[INFO]
-------------------------------------------------------
[INFO] Running MyServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.01 s - in MyServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
------------------------------------------------------------
-----------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------
-----------
```

What if the interaction never happens?

If you comment out or remove `service.fetchData();`, then `getData()` is never called and the test will fail with a message like:

```
org.mockito.exceptions.verification.junit.ArgumentsAreDiffere
nt:
Wanted but not invoked:
mockApi.getData();
-> at
MyServiceTest.testVerifyInteraction(MyServiceTest.java:XX)
However, there were zero interactions with this mock.
```

Week - 2

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:

<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:

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");
}
}

1. `pom.xml` dependencies

Make sure your `pom.xml` includes these:

```
<dependencies>
  <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>
```

Week - 2

```
    <version>1.2.3</version>
  </dependency>
</dependencies>
```

2. Java class for logging

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class LoggingExample {

    // Create logger instance for this class
    private static final Logger logger =
LoggerFactory.getLogger(LoggingExample.class);

    public static void main(String[] args) {
        // Log an error message
        logger.error("This is an error message");

        // Log a warning message
        logger.warn("This is a warning message");
    }
}
```

3. Running the app

When you run `LoggingExample`, you should see output similar to:

```
15:45:30.123 [main] ERROR LoggingExample - This is an error
message
15:45:30.125 [main] WARN  LoggingExample - This is a warning
message
```