PL/SQL programming
Exercise 1: Control Structures

This exercise involved using loops and IF conditions to apply logic over database records.
updated loan interest rates, VIP status, and printed reminders for upcoming loan dues.
Control structures like FOR loops and IF statements were used in anonymous PL/SQL blocks.

Table

```
-- Customers Table
CREATE TABLE Customers (
  CustomerID   NUMBER PRIMARY KEY,
  Name       VARCHAR2(50),
  Age        NUMBER,
  Balance     NUMBER(10, 2),
  IsVIP      VARCHAR2(5),
  LoanInterest NUMBER(5, 2),
  LoanDueDate  DATE
);
```

Scenario 1:

```
BEGIN
  FOR c IN (SELECT CustomerID, Age, LoanInterest FROM Customers) LOOP
    IF c.Age > 60 THEN
      UPDATE Customers
      SET LoanInterest = LoanInterest - (LoanInterest * 0.01)
      WHERE CustomerID = c.CustomerID;
    END IF;
  END LOOP;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('Loan discount applied for customers over 60.');
END;
```

Scenario 2:

```
BEGIN
  FOR c IN (SELECT CustomerID, Balance FROM Customers) LOOP
    IF c.Balance > 10000 THEN
```

```
        UPDATE Customers
        SET IsVIP = 'TRUE'
        WHERE CustomerID = c.CustomerID;
      END IF;
  END LOOP;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE('VIP status updated for eligible customers.');
END;
```

Scenario 3:

```
BEGIN
  FOR c IN (
    SELECT Name, LoanDueDate
    FROM Customers
    WHERE LoanDueDate BETWEEN SYSDATE AND SYSDATE + 30
  ) LOOP
    DBMS_OUTPUT.PUT_LINE('Reminder: Loan due for ' || c.Name || ' on ' ||
TO_CHAR(c.LoanDueDate, 'DD-Mon-YYYY'));
  END LOOP;
END;
```

Output:

```
INSERT INTO Customers VALUES (1, 'John', 65, 12000, 'FALSE', 7.5, SYSDATE +
10);
INSERT INTO Customers VALUES (2, 'Alice', 45, 8000, 'FALSE', 6.5, SYSDATE +
40);
INSERT INTO Customers VALUES (3, 'Mike', 70, 15000, 'FALSE', 8.0, SYSDATE +
5);
```

Loan discount applied for customers over 60.

VIP status updated for eligible customers.

Reminder: Loan due for John on 09-Jul-2025
Reminder: Loan due for Mike on 04-Jul-2025

Loan discount applied for customers over 60.
VIP status updated for eligible customers.
Reminder: Loan due for John on 09-Jul-2025
Reminder: Loan due for Mike on 04-Jul-2025

Exercise 3: Stored Procedures

reusable PL/SQL procedures to apply monthly interest, update bonuses, and transfer funds.
These procedures use parameters, transactions, and conditional logic.
They demonstrate modular, maintainable, and testable database operations.

Create tables :

```
-- 1. SavingsAccounts Table
CREATE TABLE SavingsAccounts (
  AccountID    NUMBER PRIMARY KEY,
  CustomerID   NUMBER,
  Balance      NUMBER(10, 2)
);

-- 2. Employees Table
CREATE TABLE Employees (
  EmpID        NUMBER PRIMARY KEY,
  Name         VARCHAR2(50),
  Department   VARCHAR2(30),
  Salary       NUMBER(10, 2)
);

-- 3. BankAccounts Table
CREATE TABLE BankAccounts (
  AccountID    NUMBER PRIMARY KEY,
  CustomerID   NUMBER,
  Balance      NUMBER(10, 2)
);
```

SCENARIO 1:

```
CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS
BEGIN
  FOR acc IN (SELECT AccountID, Balance FROM SavingsAccounts) LOOP
```

```
    UPDATE SavingsAccounts
    SET Balance = Balance + (acc.Balance * 0.01)
    WHERE AccountID = acc.AccountID;
  END LOOP;

  COMMIT;
END;
```

Scenario 2:

```
CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(
  deptName IN VARCHAR2,
  bonusPct IN NUMBER
) AS
BEGIN
  FOR emp IN (SELECT EmpID, Salary FROM Employees WHERE Department =
deptName) LOOP
    UPDATE Employees
    SET Salary = emp.Salary + (emp.Salary * bonusPct / 100)
    WHERE EmpID = emp.EmpID;
  END LOOP;

  COMMIT;
END;
```

Scenario 3:

```
CREATE OR REPLACE PROCEDURE TransferFunds(
  fromAcc IN NUMBER,
  toAcc   IN NUMBER,
  amount  IN NUMBER
) AS
  fromBal NUMBER;
BEGIN
  SELECT Balance INTO fromBal FROM BankAccounts WHERE AccountID =
fromAcc;

  IF fromBal < amount THEN
    RAISE_APPLICATION_ERROR(-20001, 'Insufficient balance in source
account.');
  ELSE
    UPDATE BankAccounts
```

```
    SET Balance = Balance - amount
    WHERE AccountID = fromAcc;

    UPDATE BankAccounts
    SET Balance = Balance + amount
    WHERE AccountID = toAcc;
  END IF;

  COMMIT;
END;
```

Output:

Interest applied successfully (1%).
10% bonus added to employees in Sales department.
₹500 successfully transferred from Account 201 to 202.

TDD using JUnit5 and Mockito
Exercise 1: Setting Up Junit

created a simple test class using JUnit 4 to test basic functionality like addition. This helped verify the setup of the testing environment using @Test and assertEquals.
JUnit was configured via Maven with test cases organized in a clean structure.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CalculatorTest {


   public int add(int a, int b) {
      return a + b;
   }

   @Test
   public void testAdd() {
      // Arrange
      CalculatorTest calc = new CalculatorTest();

      int result = calc.add(10, 5);

      assertEquals("10 + 5 should equal 15", 15, result);
   }
}
```

Output:

Tests run: 1, Failures: 0, Errors: 0

All tests passed.

Exercise 3: Assertions in Junit

explored JUnit's assertion methods such as assertEquals, assertTrue, and assertNull.
These assertions validate test results and ensure expected values during unit testing.
This exercise strengthens your understanding of how tests determine pass/fail logic.

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

public class AssertionsTest {

   @Test
   public void testVariousAssertions() {
      assertEquals("2 + 3 should be 5", 5, 2 + 3);

      assertTrue("5 is greater than 2", 5 > 2);

      assertFalse("2 is not greater than 5", 2 > 5);

      String name = null;
      assertNull("Name should be null", name);


      Object obj = new Object();
      assertNotNull("Object should not be null", obj);
   }
}
```

Output:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec

Results:

All assertions passed.

Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in Junit

structured test cases using the Arrange-Act-Assert (AAA) pattern.
also used @Before and @After annotations for initializing and cleaning up test resources.
This improves test readability, reusability, and ensures each test runs in a clean state**.**

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

public class CalculatorTest {

  private Calculator calculator;

  class Calculator {
    public int add(int a, int b) {
      return a + b;
    }

    public int subtract(int a, int b) {
      return a - b;
    }
  }
  @Before
  public void setUp() {
```

```
        calculator = new Calculator();
        System.out.println(" Setup complete");
    }
    @After
    public void tearDown() {
        calculator = null;
        System.out.println(" Teardown complete");
    }
    @Test
    public void testAddition() {
        int result = calculator.add(10, 5);
        assertEquals("10 + 5 should be 15", 15, result);
    }
    @Test
    public void testSubtraction() {
        // Act
        int result = calculator.subtract(8, 3);
        assertEquals("8 - 3 should be 5", 5, result);
    }
}
```

Output:

Setup complete

Teardown complete

Setup complete Teardown complete

Tests run: 2, Failures: 0, Errors: 0

All tests passed.

3. Mockito exercises
Exercise 1: Mocking and Stubbing

mocked an external API and stubbed its method to return fake data.
This decouples tests from real dependencies, ensuring reliability and speed.
used when(...).thenReturn(...) with Mockito.mock() to simulate API behavior.

```
import static org.mockito.Mockito.*;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
interface ExternalApi {
  String getData();
}

class MyService {
  private ExternalApi api;

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

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

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();

    assertEquals("Mock Data", result);
  }
```

}

Output:
Tests run: 1, Failures: 0, Errors: 0
All tests passed.

Exercise 2: Verifying Interactions

verified whether a mocked method was called, and optionally with specific arguments.
Using verify(...),confirmed interactions between your service and its dependencies.
This ensures the correct flow of logic and usage of collaborators in code**.**

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


interface ExternalApi {
  String getData();
}

class MyService {
  private ExternalApi api;

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

  public String fetchData() {
    return api.getData();  // Call to external API
  }
}

public class MyServiceTest {

  @Test
  public void testVerifyInteraction() {

    ExternalApi mockApi = mock(ExternalApi.class);

    MyService service = new MyService(mockApi);
    service.fetchData();

    verify(mockApi).getData();
```

```
    }
}
```

Output:

Tests run: 1, Failures: 0, Errors: 0
All method calls verified successfully.

SLF4J logging framework
Exercise 1: Logging Error Messages and Warning Levels

used SLF4J with Logback to log error and warning messages in a Java application. This helps track issues and monitor system health during development and production.
The logs are output to the console with timestamp, level, and message.

```
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:

```
12:00:00.123 [main] ERROR LoggingExample - This is an error message

12:00:00.124 [main] WARN  LoggingExample - This is a warning message
```