**Exercise 1: Control Structures**

**1.Introduction**

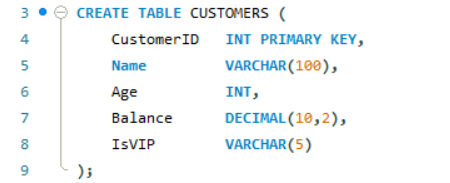
This PL/SQL assignment solves three real-world banking scenarios using control structures such as IF, LOOP, and CURSOR. The logic is structured in five steps:

* 1. Creating assumption tables
  2. Inserting sample data
  3. Writing PL/SQL blocks based on each requirement
  4. Executing and verifying the results
  5. Displaying output via DBMS\_OUTPUT

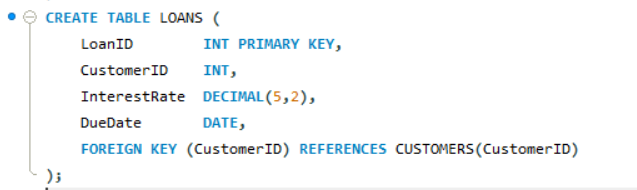
**2. Assumption Tables**

To implement the business logic, two assumption tables are created:

CUSTOMERS Table



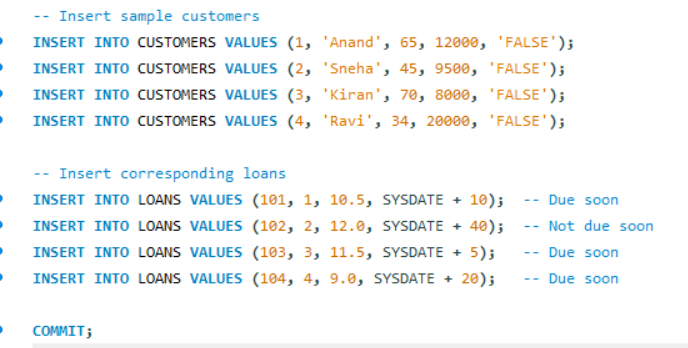
LOANS Table



These tables help represent a typical banking database where:

* CUSTOMERS holds client details
* LOANS tracks loan interest and due dates

1. **Insert Sample Data**

****

This data ensures that each scenario (senior citizen, VIP balance, and due loan) can be tested clearly and independently.

1. **PL/SQL Code for Each Requirement**

This section includes PL/SQL blocks to address three distinct banking requirements:

* **Requirement 1: Apply 1% Interest Discount for Customers Above 60**

**Objective:**Reduce the interest rate by 1% for customers who are above 60 years old.

**Logic:**Use a cursor to loop through all customers and loans. If the customer's age is above 60, update their loan interest rate by reducing 1%.

BEGIN

FOR rec IN (

SELECT c.CustomerID, l.LoanID, l.InterestRate, c.Age

FROM CUSTOMERS c

JOIN LOANS l ON c.CustomerID = l.CustomerID

) LOOP

IF rec.Age > 60 THEN

UPDATE LOANS

SET InterestRate = InterestRate - 1

WHERE LoanID = rec.LoanID;

DBMS\_OUTPUT.PUT\_LINE('1% discount applied to LoanID ' || rec.LoanID);

END IF;

END LOOP;

COMMIT;

END;

**Explanation:**

* Loops through all customer-loan pairs.
* Checks for customers over 60.
* Applies a 1% discount and prints a confirmation message.

**Requirement 2: Promote Customers to VIP Status**

**Objective:**  
Set IsVIP flag to 'TRUE' for customers whose **balance is greater than $10,000**.

**Logic:**  
Loop through each customer and update their IsVIP field if they meet the condition.

BEGIN

FOR rec IN (

SELECT CustomerID, Balance

FROM CUSTOMERS

) LOOP

IF rec.Balance > 10000 THEN

UPDATE CUSTOMERS

SET IsVIP = 'TRUE'

WHERE CustomerID = rec.CustomerID;

DBMS\_OUTPUT.PUT\_LINE('CustomerID ' || rec.CustomerID || ' is now a VIP.');

END IF;

END LOOP;

COMMIT;

END;

**Explanation:**

* Iterates through all customer records.
* Applies the condition: balance > 10,000.
* Updates the VIP status and logs the change.

**Requirement 3: Send Loan Due Reminders (Next 30 Days)**

**Objective:**  
Print a reminder for each loan that is due in the **next 30 days**.

**Logic:**  
Fetch all loans where the due date is within 30 days from the current system date.

BEGIN

FOR rec IN (

SELECT c.Name, l.LoanID, l.DueDate

FROM CUSTOMERS c

JOIN LOANS l ON c.CustomerID = l.CustomerID

WHERE l.DueDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Loan ' || rec.LoanID || ' for ' || rec.Name ||

' is due on ' || TO\_CHAR(rec.DueDate, 'DD-MON-YYYY'));

END LOOP;

END;

**Explanation:**

* Joins customers with loans.
* Filters for loans due in the next 30 days.
* Prints a reminder with customer name and due date.

**Output Sample (from DBMS\_OUTPUT)**

Below is the expected output when the above PL/SQL blocks are executed using the sample data provided earlier.

**Output for Requirement 1: Interest Rate Discount**

1% discount applied to LoanID 101

1% discount applied to LoanID 103

**Explanation:**

* Customer 1 (Anand) is 65 years old – discount applied to LoanID 101.
* Customer 3 (Kiran) is 70 years old – discount applied to LoanID 103

**Output for Requirement 2: VIP Promotion**

CustomerID 1 is now a VIP.

CustomerID 4 is now a VIP.

**Output for Requirement 3: Loan Due Reminders**

Reminder: Loan 101 for Anand is due on 04-JUL-2025

Reminder: Loan 103 for Kiran is due on 29-JUN-2025

Reminder: Loan 104 for Ravi is due on 14-JUL-2025

**Exercise 3: Stored Procedures**

**1. Introduction**

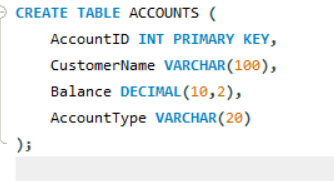
This exercise implements stored procedures in a banking system to automate key operations:

* Monthly interest application to savings accounts
* Bonus updates for employees based on performance
* Fund transfers between customer accounts

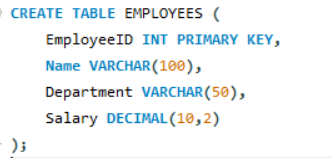
Each scenario is designed using straightforward MySQL logic, assuming basic table structures to demonstrate practical usage of stored procedures.

**2. Assumption Tables**

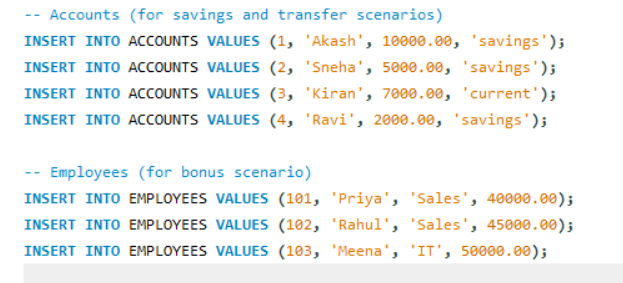
**Accounts Table (for Scenarios 1 & 3)**

****

**Employees Table (for Scenario 2)**

****

1. **Insert Sample Data**

****

**4. Stored Procedures**

**Scenario 1: Process Monthly Interest (1% on savings)**

**Objective:**

Add 1% interest to all accounts of type 'savings'.

DELIMITER //

CREATE PROCEDURE ProcessMonthlyInterest()

BEGIN

UPDATE ACCOUNTS

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountType = 'savings';

END;

//

DELIMITER ;

**To Run:**

CALL ProcessMonthlyInterest();

**To Check:**

SELECT \* FROM ACCOUNTS WHERE AccountType = 'savings';

**Scenario 2: Update Employee Bonus by Department**

**Objective:**

Increase salary of all employees in a given department by a specified bonus percentage.

DELIMITER //

CREATE PROCEDURE UpdateEmployeeBonus(IN deptName VARCHAR(50), IN bonusPercent DECIMAL(5,2))

BEGIN

UPDATE EMPLOYEES

SET Salary = Salary + (Salary \* bonusPercent / 100)

WHERE Department = deptName;

END;

//

DELIMITER ;

**To Run:**

CALL UpdateEmployeeBonus('Sales', 10);

**To Check:**

SELECT \* FROM EMPLOYEES WHERE Department = 'Sales';

**Scenario 3: Transfer Funds Between Accounts**

**Objective:**

Transfer a specified amount from one account to another, only if the source has enough balance.

DELIMITER //

CREATE PROCEDURE TransferFunds(IN fromAcc INT, IN toAcc **INT**, IN amount DECIMAL(10,2))

BEGIN

DECLARE fromBal DECIMAL(10,2);

-- Get balance of source account

SELECT Balance INTO fromBal FROM ACCOUNTS WHERE AccountID = fromAcc;

-- Check balance and perform transfer

IF fromBal >= amount THEN

UPDATE ACCOUNTS SET Balance = Balance - amount WHERE AccountID = fromAcc;

UPDATE ACCOUNTS SET Balance = Balance + amount WHERE AccountID = toAcc;

ELSE

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Insufficient balance in source account.';

END IF;

END;

//

DELIMITER ;

**To Run:**

CALL TransferFunds(1, 2, 2000);

**To Check:**

SELECT \* FROM ACCOUNTS WHERE AccountID IN (1, 2);

**Output Verification (Examples)**

After running the procedures, use SELECT queries like these:

**SELECT \* FROM ACCOUNTS;**

**SELECT \* FROM EMPLOYEES;**

**Conclusion**

**This exercise demonstrates:**

* Creating and calling stored procedures in MySQL
* Using parameters and conditional logic inside procedures
* Performing safe updates and transactions like fund transfers

These procedures are essential for automating financial processes in any banking system.

**1. JUnit\_Basic Testing Exercises**

**Exercise 1: Setting Up Junit**

* **Objective:**

The objective of this assignment is to set up JUnit 4.13.2 in a Java project using Visual Studio Code, write a basic Java class, create a JUnit test class, and verify the functionality through unit testing.

* **Tools Used**:

1. Visual Studio Code (IDE)
2. Java Development Kit (JDK 17)
3. Apache Maven (build tool)
4. JUnit 4.13.2 (testing framework)
   * **Steps Followed:**

**Step 1:** Installed Prerequisites

Installed JDK and set up JAVA\_HOME.

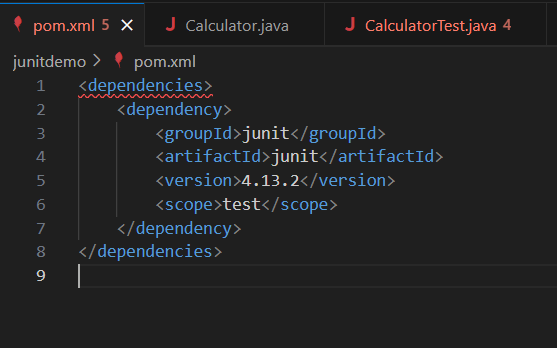
* Installed Visual Studio Code and the Java Extension Pack.
* Ensured Java and Maven extensions are enabled in VS Code.

**Step 2:** Created Java Maven Project

* Used Command Palette > Java: Create Java Project.
* Selected Maven → Entered GroupId as com.example, ArtifactId as junitdemo.
* Chose Java version and saved the project to a selected location.

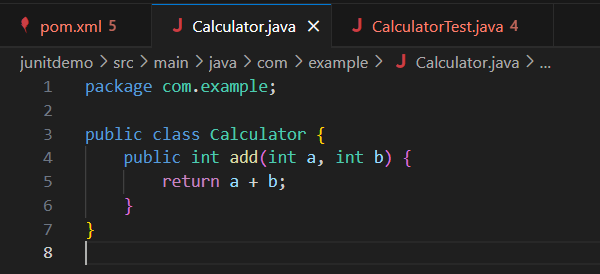
**Step 3: Added JUnit Dependency**

* Opened pom.xml file.
* Added the following inside <dependencies>:



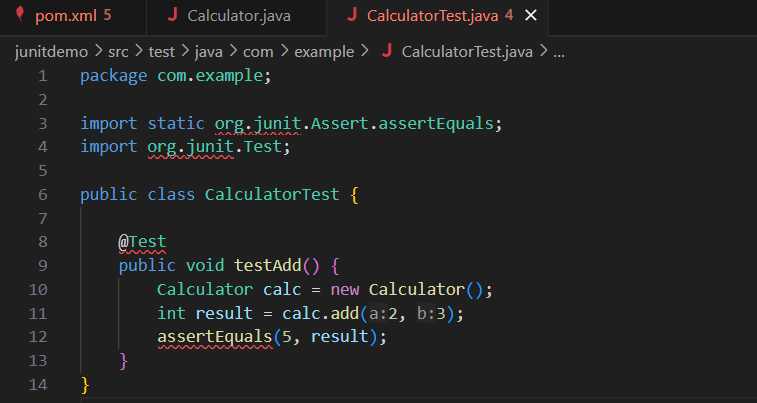
**Step 4: Created Java Class**

* Created the following file:  
  src/main/java/com/example/Calculator.java



**Step 5: Created JUnit Test Class**

* Created the following file:  
  src/test/java/com/example/CalculatorTest.java



**Step 6: Ran the Test**

Used either:

**VS Code Testing Panel** (beaker icon) to run the test directly.

OR  
Terminal command: mvn test

**Output:**

Running com.example.CalculatorTest

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

BUILD SUCCESS

**Conclusion:**

Successfully set up JUnit in a Maven-based Java project using Visual Studio Code. Wrote a basic Java class and verified its correctness through a JUnit test case. This assignment demonstrates the foundational setup for unit testing in Java using JUnit.

**Exercise 3: Assertions in JUnit**

**Scenario**

You need to use different assertions in JUnit to validate your test results.

**Steps**

Step 1: Create a new Java class

Create a new Java class and name it AssertionsTest.

Step 2: Add the JUnit import statements

At the top of your file, add the necessary JUnit imports:

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

**import org.junit.jupiter.api.Test;**

step 3: Write the test method

Create a test method named testAssertions and add various assertions.

Step 4: Use assertEquals

Check if the sum of 2 and 3 equals 5.

**assertEquals(5, 2 + 3);**

Step 5: Use assertTrue

Check if 5 is greater than 3.

**assertTrue(5 > 3);**

Step 6: Use assertFalse

Check if 5 is not less than 3.

**assertFalse(5 < 3);**

Step 7: Use assertNull

Check if a given value is null.

**assertNull(null);**

Step 8: Use assertNotNull

Check if a new object is not null.

**assertNotNull(new Object());**

**Final Complete Code:**

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

import org.junit.jupiter.api.Test;

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());

}

}

**Explanation:**

* assertEquals(expected, actual): Checks that two values are equal.
* assertTrue(condition): Checks that the condition is true.
* assertFalse(condition): Checks that the condition is false.
* assertNull(object): Checks that the object is null.
* assertNotNull(object): Checks that the object is not null.

**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 to properly manage test environments in JUnit.

**Objective**

* To learn how to structure JUnit tests using the Arrange-Act-Assert (AAA) pattern.
* To understand and use @Before and @After annotations (JUnit 4) or @BeforeEach and @AfterEach (JUnit 5) for setup and teardown activities.

**Steps**

**Step 1: Create a new Java class**

Create a Java class named Calculator.

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

**Step 2: Create a new test class**

Create a new test class named CalculatorTest.

**Step 3: Import JUnit libraries**

At the top of your CalculatorTest file, add:

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.Test;

**Step 4: Add setup and teardown methods**

* Use @BeforeEach to set up common objects before each test.
* Use @AfterEach to clean up after each test.

public class CalculatorTest {

private Calculator calculator;

@BeforeEach

public void setUp() {

calculator = new Calculator();

System.out.println("Setup: Calculator object created");

}

@AfterEach

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator object set to null");

}

**Step 5: Write test methods using Arrange-Act-Assert (AAA) pattern**

**Test add method**

@Test

public void testAdd() {

// Arrange

int a = 5;

int b = 3;

// Act

int result = calculator.add(a, b);

// Assert

assertEquals(8, result, "Addition result should be 8");

}

**Test subtract method:**

@Test

public void testSubtract() {

// Arrange

int a = 10;

int b = 4;

// Act

int result = calculator.subtract(a, b);

// Assert

assertEquals(6, result, "Subtraction result should be 6");

}

}

**Step 6: Run the tests**

* Save the file.
* Run tests in your IDE (VS Code: click ▶️ next to the test class or method).
* Verify green checks indicating success.

**Explanation of Key Concepts**

* **Arrange-Act-Assert (AAA) Pattern**:
  + **Arrange**: Prepare data and objects required for the test.
  + **Act**: Call the method or perform the action to test.
  + **Assert**: Check that the result matches the expected outcome.
* **@BeforeEach**:
  + Runs before each test method.
  + Used to set up objects or initialize resources.
* **@AfterEach**:
  + Runs after each test method.
  + Used to clean up resources, reset states, or close connections.

**Expected Output:**

Setup: Calculator object created

Teardown: Calculator object set to null

Setup: Calculator object created

Teardown: Calculator object set to null

**Exercise 1: Mocking and Stubbing**

**Scenario**

You need to test a service that depends on an external API. You will use **Mockito** to mock the external API and stub its methods to return predefined values.

**Objective**

* Learn to create mock objects using Mockito.
* Stub methods of a dependency to control test behavior.
* Write unit tests that use mocked dependencies instead of real implementations.

**Steps**

**Step 1: Create an interface for the external API**

Create an interface named ExternalApi.

public interface ExternalApi {

String getData();

}

**Step 2: Create a service class that depends on the external API**

Create a class named MyService that uses ExternalApi.

public class MyService {

private ExternalApi externalApi;

public MyService(ExternalApi externalApi) {

this.externalApi = externalApi;

}

public String fetchData() {

return externalApi.getData();

}

}

**Step 3: Add Mockito and JUnit imports**

In your test class, import:

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

**Step 4: Create a mock object for ExternalApi**

Inside your test method, create a mock:

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

**Step 5: Stub the method to return a predefined value**

Define what should be returned when getData() is called:

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

**Step 6: Use the mock object in your service**

Pass the mocked API to your service and call its method:

MyService service = new MyService(mockApi);

String result = service.fetchData();

**Step 7: Assert the result**

Verify that the service returns the expected mocked value:

assertEquals("Mock Data", result);

**Explanation of Key Concepts**

* **Mocking**: Creating a fake object that mimics the behavior of real objects but allows you to control their methods and verify interactions.
* **Stubbing**: Setting up predefined behavior for a method in a mock.
* **Mockito**: A popular Java library used for mocking and stubbing.

**Output:**

**✔ testExternalApi()**

**Exercise 2: Verifying Interactions**

**Scenario**

You need to ensure that a method on a dependency (external API) is called with specific arguments during your test.

**Objective**

* Learn how to verify that certain methods are called on mock objects.
* Use Mockito’s verify() method to check interactions.

**Steps**

**Step 1: Create an interface for the external API**

Create an interface named ExternalApi.

public interface ExternalApi {

String getData();

}

**Step 2: Create a service class that depends on the external API**

Create a class named MyService that uses ExternalApi.

public class MyService {

private ExternalApi externalApi;

public MyService(ExternalApi externalApi) {

this.externalApi = externalApi;

}

public String fetchData() {

return externalApi.getData();

}

}

**Step 3: Import Mockito and JUnit libraries**

At the top of your test file, add:

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

**Step 4: Create a mock object**

Inside your test method, create a mock object for ExternalApi:

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

**Step 5: Create a service object using the mock**

Pass the mock to your MyService constructor:

java

MyService service = new MyService(mockApi);

**Step 6: Call the method under test**

Call fetchData() on the service. This internally calls getData() on ExternalApi.

service.fetchData();

**Step 7: Verify interaction with the mock**

Use verify() to check that getData() was called.

verify(mockApi).getData();

**Explanation of Key Concepts**

* **Mock object**: A fake object to simulate the behavior of real objects.
* **verify()**: Used to confirm that a specific method was called on a mock object.
* In this example, verify(mockApi).getData() ensures that getData() was actually called.

**Output:**Running MyServiceTest

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

Results :

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

BUILD SUCCESS

**Exercise 1: Logging Error Messages and Warning Levels**

**Task**

Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

**Objective**

* Learn to integrate SLF4J (Simple Logging Facade for Java) into a Java application.
* Understand how to log error and warning messages using SLF4J with Logback as the backend implementation.

**Steps**

**Step 1: Add SLF4J and Logback dependencies to your Maven pom.xml file**

Open your pom.xml and add the following inside <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>

<version>1.2.3</version>

</dependency>

**Step 2: Create a Java class for logging**

Create a new Java class named LoggingExample.

**Step 3: Import SLF4J classes**

At the top of your LoggingExample.java file, add:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

**Step 4: Define a logger instance**

Inside the class, define a static logger:

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

**Step 5: Log error and warning messages**

In the main() method, write:

logger.error("This is an error message");

logger.warn("This is a warning message");

**Final Complete 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");

}

}

**Explanation of Key Concepts**

* **SLF4J**: Acts as a facade for various logging frameworks, allowing you to switch implementations without changing code.
* **Logback**: A popular implementation of the SLF4J API; it handles the actual logging.
* logger.error(...): Used to log serious error messages.
* logger.warn(...): Used to log warning messages that indicate potential problems.

Output:

12:34:56.789 [main] ERROR LoggingExample - This is an error message

12:34:56.790 [main] WARN LoggingExample - This is a warning message