**Deep Skilling Program**

**Hands-On Solutions[Week-2]**

**Week-2 :** 1. PL/SQL Programming

2. Unit Testing(Junit)

**Submitted by:** Bhavya Sree Pappanaboyina

**Email Id :** 229x1a2823@gmail.com

**GitHubRepo :**  https://github.com/Bhavyasree1606/deep-skilling-

solutions-6421608

PL/SQL Programming

**Exercise 1:** Control Structures

**1.Introduction:**

This document demonstrates **PL/SQL control structures** (loops, cursors, conditional logic) to automate banking operations

**Scenario 1:**

The bank wants to apply a discount to loan interest rates for customers above 60 years old.

**2.Code Implementation:**

**Step 1: Create Tables**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

Name VARCHAR2(100),

DOB DATE

);

CREATE TABLE Loans (

LoanID NUMBER PRIMARY KEY,

CustomerID NUMBER,

InterestRate NUMBER(5,2),

LastModified DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

**Step 2: Insert Sample Data**

-- Insert Customers

INSERT INTO Customers VALUES (101, 'Sundar Rao', TO\_DATE('1950-03-15', 'YYYY-MM-DD')); -- Age 74

INSERT INTO Customers VALUES (102, 'Leela Patel', TO\_DATE('1955-08-20', 'YYYY-MM-DD')); -- Age 68

INSERT INTO Customers VALUES (103, 'Arjun Nair', TO\_DATE('1961-01-10', 'YYYY-MM-DD')); -- Age 63

INSERT INTO Customers VALUES (104, 'Priya Shah', TO\_DATE('1988-06-05', 'YYYY-MM-DD')); -- Age 36

INSERT INTO Customers VALUES (105, 'Rajiv Menon', TO\_DATE('1995-11-23', 'YYYY-MM-DD')); -- Age 29

-- Insert Loans

INSERT INTO Loans VALUES (201, 101, 9.5, NULL); -- Eligible

INSERT INTO Loans VALUES (202, 102, 8.0, NULL); -- Eligible

INSERT INTO Loans VALUES (203, 103, 10.25, NULL);-- Eligible

INSERT INTO Loans VALUES (204, 104, 7.5, NULL); -- Not eligible (young)

INSERT INTO Loans VALUES (205, 105, NULL, NULL); -- Not eligible (NULL interest)

**Scenario 1:** The bank wants to apply a discount to loan interest rates for customers above 60 years old.

**Step 3: Run Your PL/SQL Block**

SET SERVEROUTPUT ON; -- Enable output to see messages from DBMS\_OUTPUT.PUT\_LINE.

DECLARE

    -- Cursor definition: Selects loan details for potential discount application.

    -- Includes CustomerName and calculated CustomerAge for logging clarity.

    -- The WHERE clause filters for customers over the senior age threshold and ensures a valid interest rate.

    CURSOR c\_senior\_eligible\_loans IS

        SELECT

            l.LoanID,

            l.CustomerID,

            c.Name AS CustomerName,

            l.InterestRate AS CurrentInterestRate,

            TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) AS CustomerAge -- Calculates age in full years

        FROM Loans l

        JOIN Customers c ON l.CustomerID = c.CustomerID

        WHERE TRUNC(MONTHS\_BETWEEN(SYSDATE, c.DOB) / 12) > 60 -- Only consider customers strictly older than 60

          AND l.InterestRate IS NOT NULL; -- Ensure interest rate exists

    -- Constant for the discount amount, allowing easy modification of the policy.

    c\_discount\_percentage CONSTANT NUMBER := 1.00; -- Represents 1% interest rate reduction.

    -- Constant for the minimum age to qualify for the senior discount.

    c\_senior\_age\_threshold CONSTANT NUMBER := 60;

    -- Counter to track the number of loans successfully updated during the process.

    v\_loans\_discounted\_count NUMBER := 0;

BEGIN

    -- Informative header to clearly state the purpose and policy of the block.

    DBMS\_OUTPUT.PUT\_LINE('--- Initiating Senior Loan Interest Rate Discount Process ---');

    DBMS\_OUTPUT.PUT\_LINE('Policy: Applying ' || c\_discount\_percentage || '% discount to loans for customers strictly older than ' || c\_senior\_age\_threshold || ' years.');

    DBMS\_OUTPUT.PUT\_LINE('------------------------------------------------------------');

    -- Iterate through each loan record that meets the senior customer criteria using a FOR loop.

    -- This method is concise and automatically handles cursor opening, fetching, and closing.

    FOR loan\_rec IN c\_senior\_eligible\_loans LOOP

        -- Begin a nested block to handle potential errors for individual loan updates.

        -- This ensures that if one loan update fails, the overall process continues for other loans.

        BEGIN

            -- Update the loan's interest rate.

            -- GREATEST(0, ...) is used to ensure the interest rate never falls below 0% after the discount.

            -- LastModified column is updated to reflect the time of this change, crucial for auditing.

            UPDATE Loans

            SET InterestRate = GREATEST(0, loan\_rec.CurrentInterestRate - c\_discount\_percentage),

                LastModified = SYSDATE

            WHERE LoanID = loan\_rec.LoanID;

            -- Log the details of the updated loan for verification.

            DBMS\_OUTPUT.PUT\_LINE('  Updated Loan ID: ' || loan\_rec.LoanID ||

                                 ' for Customer: ' || loan\_rec.CustomerName ||

                                 ' (Age: ' || loan\_rec.CustomerAge || ') | ' ||

                                 'Rate Change: ' || loan\_rec.CurrentInterestRate || '% -> ' ||

                                 GREATEST(0, loan\_rec.CurrentInterestRate - c\_discount\_percentage) || '%');

            v\_loans\_discounted\_count := v\_loans\_discounted\_count + 1; -- Increment counter for successful updates.

        EXCEPTION

            -- Catch specific errors that occur during the UPDATE of a single loan.

            -- This error message helps in identifying problematic loans without stopping the entire batch.

            WHEN OTHERS THEN

                DBMS\_OUTPUT.PUT\_LINE('  ERROR: Failed to update Loan ID ' || loan\_rec.LoanID ||

                                     ' for ' || loan\_rec.CustomerName || ': ' || SQLERRM);

                -- No ROLLBACK here, as we want to commit other successful updates.

                -- If strict all-or-nothing is required, remove the inner exception block.

        END; -- End of inner block

    END LOOP; -- End of cursor loop

    -- Commit all successful changes to the database.

    COMMIT;

    -- Final summary message indicating the total number of loans processed.

    DBMS\_OUTPUT.PUT\_LINE('------------------------------------------------------------');

    DBMS\_OUTPUT.PUT\_LINE('Process complete. ' || v\_loans\_discounted\_count || ' loans updated for senior customers.');

EXCEPTION

    -- Outer exception handler: catches any errors not handled by the inner blocks.

    -- If an error occurs here (e.g., cursor opening issue), all changes will be rolled back.

    WHEN OTHERS THEN

        ROLLBACK; -- Rollback all changes if a global error occurred.

        DBMS\_OUTPUT.PUT\_LINE('!!! CRITICAL PROCESS FAILURE !!!');

        DBMS\_OUTPUT.PUT\_LINE('An unhandled error occurred: ' || SQLERRM);

        DBMS\_OUTPUT.PUT\_LINE('All changes have been rolled back to maintain data consistency.');

END;

/

**Sample Output:**

--- Initiating Senior Loan Interest Rate Discount Process ---

Policy: Applying 1.0% discount to loans for customers strictly older than 60 years.

------------------------------------------------------------

**Updated Loan ID: 201 for Customer: Sundar Rao (Age: 74) | Rate Change: 9.5% -> 8.5%**

**Updated Loan ID: 202 for Customer: Leela Patel (Age: 68) | Rate Change: 8.0% -> 7.0%**

**Updated Loan ID: 203 for Customer: Arjun Nair (Age: 63) | Rate Change: 10.25% -> 9.25%**

------------------------------------------------------------

Process complete. 3 loans updated for senior customers.

**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 implementation:**

**PL/SQL Code:**

SET SERVEROUTPUT ON; -- Essential to display output messages from DBMS\_OUTPUT.PUT\_LINE.

DECLARE

    -- Cursor definition: Selects CustomerID, Name, and Balance for customers who are

    -- eligible for VIP status but are not currently marked as VIP.

    -- This specific WHERE clause (Balance > threshold AND IsVIP = 'N') is efficient

    -- as it only processes records that actually need an update.

    CURSOR c\_eligible\_vip\_candidates IS

        SELECT

            CustomerID,

            Name,

            Balance

        FROM Customers

        WHERE Balance > 10000 -- Hardcoded value, will be replaced by constant below

          AND NVL(IsVIP, 'N') = 'N'; -- Ensures only non-VIPs are considered, handles NULL IsVIP as 'N'.

    -- Constant for the VIP balance threshold. Makes the code more maintainable

    -- and business-rule driven.

    c\_vip\_balance\_threshold CONSTANT NUMBER := 10000.00;

    -- Counter to track the number of customers newly promoted to VIP status.

    v\_newly\_promoted\_count NUMBER := 0;

BEGIN

    -- Informative header to clearly state the purpose and policy of the block.

    DBMS\_OUTPUT.PUT\_LINE('--- Starting VIP Customer Promotion Process ---');

    DBMS\_OUTPUT.PUT\_LINE('Policy: Customers with a total balance strictly greater than $' || TO\_CHAR(c\_vip\_balance\_threshold, 'FM999,999.00') || ' will be promoted to VIP.');

    DBMS\_OUTPUT.PUT\_LINE('--------------------------------------------------');

    -- Re-open the cursor using the constant for the threshold

    -- (This approach is slightly less direct than embedding the constant directly,

    -- but shown to illustrate the point. For conciseness, the constant could be

    -- directly in the cursor WHERE clause if the cursor is defined inside the block after constant).

    -- For clarity and typical usage, the constant should be directly in the cursor.

    -- Let's redefine the cursor to use the constant directly for cleaner code:

    NULL; -- Placeholder. The actual loop below will use the constant directly.

    -- Iterate through each eligible customer using a FOR loop.

    -- This handles cursor opening, fetching, and closing automatically.

    FOR cust\_rec IN (

        SELECT

            CustomerID,

            Name,

            Balance

        FROM Customers

        WHERE Balance > c\_vip\_balance\_threshold -- Using the named constant here

          AND NVL(IsVIP, 'N') = 'N'

    ) LOOP

        -- Update the customer's IsVIP flag to 'Y' and update the LastModified timestamp.

        -- Using the exact column name 'LastModified' from DDL for consistency.

        UPDATE Customers

        SET IsVIP = 'Y',

            LastModified = SYSDATE

        WHERE CustomerID = cust\_rec.CustomerID;

        -- Log the successful promotion with relevant customer details.

        DBMS\_OUTPUT.PUT\_LINE('  PROMOTED: ' || cust\_rec.Name ||

                             ' (ID: ' || cust\_rec.CustomerID ||

                             ') | Balance: $' || TO\_CHAR(cust\_rec.Balance, 'FM999,999.00') || '. Successfully set to VIP.');

        v\_newly\_promoted\_count := v\_newly\_promoted\_count + 1; -- Increment the promotion counter.

    END LOOP; -- End of FOR loop

    -- Commit all successful promotions to the database.

    COMMIT;

    -- Final summary message indicating the total number of promotions.

    DBMS\_OUTPUT.PUT\_LINE('--------------------------------------------------');

    DBMS\_OUTPUT.PUT\_LINE('VIP Customer Promotion Process Completed. Total new VIPs: ' || v\_newly\_promoted\_count || '.');

    -- Optional: Inform if no customers were promoted (e.g., all already VIP or none met criteria).

    IF v\_newly\_promoted\_count = 0 THEN

        DBMS\_OUTPUT.PUT\_LINE('No customers were newly promoted to VIP status in this run.');

    END IF;

EXCEPTION

    -- General exception handler: catches any unhandled errors during the block's execution.

    WHEN OTHERS THEN

        ROLLBACK; -- Crucial: Rollback all changes if an error occurs to prevent partial updates and maintain data integrity.

        DBMS\_OUTPUT.PUT\_LINE('!!! CRITICAL PROCESS FAILURE !!!');

        DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred during VIP promotion: ' || SQLERRM);

        DBMS\_OUTPUT.PUT\_LINE('All changes have been rolled back to ensure data consistency.');

END;

/

**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.**

**PL/SQL Code:**

SET SERVEROUTPUT ON; -- Essential to display output messages from DBMS\_OUTPUT.PUT\_LINE.

DECLARE

    -- Cursor definition: Selects loan and customer details for loans with an EndDate

    -- falling within the next 30 days (inclusive of today).

    -- Joins Loans and Customers tables to retrieve customer names.

    -- Calculates DaysRemaining directly in the query for efficiency.

    -- Orders results by EndDate for chronological reminders.

    CURSOR c\_upcoming\_loan\_dues IS

        SELECT

            l.LoanID,

            c.Name AS CustomerName,

            l.LoanAmount,

            l.InterestRate,

            l.EndDate,

            TRUNC(l.EndDate) - TRUNC(SYSDATE) AS DaysRemaining -- Calculates full days until due

        FROM Loans l

        JOIN Customers c ON l.CustomerID = c.CustomerID

        WHERE l.EndDate BETWEEN TRUNC(SYSDATE) AND (TRUNC(SYSDATE) + 30)

        ORDER BY l.EndDate ASC;

    -- Counter for the number of reminder messages generated.

    v\_reminders\_generated\_count NUMBER := 0;

    -- Variable to store the calculated approximate payment.

    -- This calculation provides a simple total amount due, useful for a reminder.

    -- NOTE: This is a simplified approximation (LoanAmount + simple interest).

    -- For real-world scenarios, a more complex amortization or installment calculation might be needed.

    v\_approximate\_payment NUMBER(12, 2);

BEGIN

    -- Comprehensive header for the reminder process.

    DBMS\_OUTPUT.PUT\_LINE('=== LOAN PAYMENT REMINDER GENERATOR ===');

    DBMS\_OUTPUT.PUT\_LINE('Scanning for loans due between ' || TO\_CHAR(TRUNC(SYSDATE), 'DD-MON-YYYY') ||

                         ' and ' || TO\_CHAR(TRUNC(SYSDATE) + 30, 'DD-MON-YYYY') || '.');

    DBMS\_OUTPUT.PUT\_LINE('---------------------------------------');

    -- Iterate through each eligible loan using a FOR loop.

    -- This simplifies cursor management (no explicit OPEN, FETCH, CLOSE).

    FOR loan\_rec IN c\_upcoming\_loan\_dues LOOP

        -- Calculate the approximate total payment due for this reminder.

        -- Assumes simple interest applied to the principal for the purpose of a quick reminder.

        -- A real system would have a dedicated payment schedule.

        v\_approximate\_payment := loan\_rec.LoanAmount \* (1 + loan\_rec.InterestRate / 100);

        -- Print a detailed reminder message for the current loan.

        DBMS\_OUTPUT.PUT\_LINE('\*\* REMINDER for: ' || loan\_rec.CustomerName || ' \*\*');

        DBMS\_OUTPUT.PUT\_LINE('  Loan ID: ' || loan\_rec.LoanID);

        DBMS\_OUTPUT.PUT\_LINE('  Original Amount: $' || TO\_CHAR(loan\_rec.LoanAmount, 'FM999G999D00'));

        DBMS\_OUTPUT.PUT\_LINE('  Interest Rate: ' || TO\_CHAR(loan\_rec.InterestRate, 'FM90.00') || '%');

        DBMS\_OUTPUT.PUT\_LINE('  Final Due Date: ' || TO\_CHAR(loan\_rec.EndDate, 'DD-MON-YYYY') ||

                             ' (in ' || loan\_rec.DaysRemaining || ' days)');

        DBMS\_OUTPUT.PUT\_LINE('  Approximate Total Payment Due: $' ||

                             TO\_CHAR(v\_approximate\_payment, 'FM999G999D00'));

        DBMS\_OUTPUT.PUT\_LINE('  Please ensure timely payment. For details, contact your loan officer.');

        DBMS\_OUTPUT.PUT\_LINE('---------------------------------------');

        v\_reminders\_generated\_count := v\_reminders\_generated\_count + 1; -- Increment the reminder counter.

    END LOOP; -- End of FOR loop

    -- Provide a summary based on whether any reminders were generated.

    IF v\_reminders\_generated\_count = 0 THEN

        DBMS\_OUTPUT.PUT\_LINE('No loans are due within the specified 30-day window. No reminders were generated.');

    ELSE

        DBMS\_OUTPUT.PUT\_LINE('--- Loan Reminder Process Completed. Total reminders sent: ' || v\_reminders\_generated\_count || ' ---');

    END IF;

EXCEPTION

    -- General exception handler. This block only reads and prints, so no DML (UPDATE/INSERT/DELETE)

    -- is performed; therefore, no COMMIT or ROLLBACK is necessary.

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('!!! ERROR: An unexpected error occurred during loan reminder generation. !!!');

        DBMS\_OUTPUT.PUT\_LINE('Error Details: ' || SQLERRM);

        DBMS\_OUTPUT.PUT\_LINE('Please check data integrity and database connectivity.');

END;

/

**Output:**

=== LOAN PAYMENT REMINDER GENERATOR ===

Scanning for loans due between 27-JUN-2025 and 27-JUL-2025.

---------------------------------------

\*\* REMINDER for: Bob \*\*

Loan ID: 101

Original Amount: $3,000.00

Interest Rate: 6.00%

Final Due Date: 27-JUN-2025 (in 0 days)

Approximate Total Payment Due: $3,180.00

Please ensure timely payment. For details, contact your loan officer.

**Exercise 3: Stored Procedures**

**Scenario 1: Process Monthly Interest for Savings Accounts**

**CODE:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

v\_interest\_rate CONSTANT NUMBER := 0.01; -- 1% monthly interest

v\_updated\_count NUMBER := 0;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Starting monthly interest processing...');

DBMS\_OUTPUT.PUT\_LINE('--------------------------------------');

FOR acc IN (SELECT AccountID, Balance FROM Accounts WHERE AccountType = 'Savings' FOR UPDATE)

LOOP

UPDATE Accounts

SET Balance = ROUND(acc.Balance \* (1 + v\_interest\_rate), 2),

LastModified = SYSDATE

WHERE AccountID = acc.AccountID;

DBMS\_OUTPUT.PUT\_LINE('Account ' || acc.AccountID ||

': Added $' || ROUND(acc.Balance \* v\_interest\_rate, 2) ||

' interest. New balance: $' || ROUND(acc.Balance \* (1 + v\_interest\_rate), 2));

v\_updated\_count := v\_updated\_count + 1;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--------------------------------------');

DBMS\_OUTPUT.PUT\_LINE('Processed ' || v\_updated\_count || ' savings accounts');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

/

**SAMPLE OUTPUT:**

Starting monthly interest processing...

--------------------------------------

Account 101: Added $10.50 interest. New balance: $1060.50

Account 103: Added $25.25 interest. New balance: $2550.25

--------------------------------------

Processed 2 savings accounts

**Scenario 2: Employee Bonus by Department**

**Code:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

p\_department IN VARCHAR2,

p\_bonus\_percent IN NUMBER

) AS

v\_updated\_count NUMBER := 0;

BEGIN

IF p\_bonus\_percent <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Bonus percentage must be positive');

END IF;

DBMS\_OUTPUT.PUT\_LINE('Applying ' || p\_bonus\_percent || '% bonus to ' || p\_department || ' department');

DBMS\_OUTPUT.PUT\_LINE('--------------------------------------');

FOR emp IN (SELECT EmployeeID, Name, Salary FROM Employees

WHERE Department = p\_department FOR UPDATE)

LOOP

UPDATE Employees

SET Salary = Salary \* (1 + p\_bonus\_percent/100)

WHERE EmployeeID = emp.EmployeeID;

DBMS\_OUTPUT.PUT\_LINE(emp.Name || ': New salary $' ||

ROUND(emp.Salary \* (1 + p\_bonus\_percent/100), 2));

v\_updated\_count := v\_updated\_count + 1;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('--------------------------------------');

DBMS\_OUTPUT.PUT\_LINE('Updated ' || v\_updated\_count || ' employees');

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

/

**SAMPLE OUTPUT:**

Applying 10% bonus to IT department

--------------------------------------

John Smith: New salary $66000.00

Sarah Lee: New salary $77000.00

--------------------------------------

Updated 2 employees

**Scenario 3: Transfer Funds Between Accounts**

**CODE:**

CREATE OR REPLACE PROCEDURE TransferFunds(

p\_source\_account IN NUMBER,

p\_dest\_account IN NUMBER,

p\_amount IN NUMBER

) AS

v\_source\_balance NUMBER;

v\_dest\_balance NUMBER;

e\_insufficient\_funds EXCEPTION;

BEGIN

-- Input validation

IF p\_amount <= 0 THEN

RAISE\_APPLICATION\_ERROR(-20002, 'Amount must be positive');

END IF;

IF p\_source\_account = p\_dest\_account THEN

RAISE\_APPLICATION\_ERROR(-20003, 'Cannot transfer to same account');

END IF;

-- Lock and check accounts

BEGIN

SELECT Balance INTO v\_source\_balance

FROM Accounts

WHERE AccountID = p\_source\_account

FOR UPDATE;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20004, 'Source account not found');

END;

BEGIN

SELECT Balance INTO v\_dest\_balance

FROM Accounts

WHERE AccountID = p\_dest\_account

FOR UPDATE;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

RAISE\_APPLICATION\_ERROR(-20005, 'Destination account not found');

END;

-- Check sufficient funds

IF v\_source\_balance < p\_amount THEN

RAISE e\_insufficient\_funds;

END IF;

-- Perform transfer

UPDATE Accounts SET Balance = Balance - p\_amount

WHERE AccountID = p\_source\_account;

UPDATE Accounts SET Balance = Balance + p\_amount

WHERE AccountID = p\_dest\_account;

-- Record transactions

INSERT INTO Transactions VALUES (

TransactionID\_seq.NEXTVAL, p\_source\_account, SYSDATE, p\_amount, 'Transfer Out'

);

INSERT INTO Transactions VALUES (

TransactionID\_seq.NEXTVAL, p\_dest\_account, SYSDATE, p\_amount, 'Transfer In'

);

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transfer successful');

DBMS\_OUTPUT.PUT\_LINE('Source account new balance: $' || (v\_source\_balance - p\_amount));

DBMS\_OUTPUT.PUT\_LINE('Destination account new balance: $' || (v\_dest\_balance + p\_amount));

EXCEPTION

WHEN e\_insufficient\_funds THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: Insufficient funds. Current balance: $' || v\_source\_balance);

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

/

**SAMPLE OUTPUT:**

Transfer successful

Source account new balance: $800.00

Destination account new balance: $1200.00

**TDD using JUnit5 and Mockito**

**Exercise 1: Setting Up Junit**

**Objective:**

To set up JUnit 5 in a Java project and write unit tests to validate a simple utility class using Eclipse IDE.

**Introduction:**

This exercise demonstrates how to configure JUnit 5 in Eclipse and create basic unit tests for a string utility class.

**1.Project Setup:**

- Created Java project named `JUnitSetupExample` in Eclipse

- Added JUnit 5 library via Build Path > Add Libraries > JUnit 5

**2. Implementation:**

Created ‘**StringUtility.java’** with two methods:

package com.example;

public class StringUtility {

public String reverse(String input) {

return new StringBuilder(input).reverse().toString();

}

public String toUpperCase(String input) {

return input.toUpperCase();

}

}

**3. Test Creation:**

Created**StringUtilityTest.java**with two test cases:

package com.example;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class StringUtilityTest {

@Test

void testReverse() {

StringUtility util = new StringUtility();

assertEquals("olleh", util.reverse("hello"));

}

@Test

void testToUpperCase() {

StringUtility util = new StringUtility();

assertEquals("HELLO", util.toUpperCase("hello"));

}

}

**4. Test Execution:**

* Ran tests in Eclipse
* Verified all tests passed with green bar
* Console output showed

**Output:**

Runs: 2/2 Failures: 0 Errors: 0

JUnit Test Runner

Running StringUtilityTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 sec

Results :

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

BUILD SUCCESSFUL

**5.Conclusion:**

This project demonstrates how to:

* Set up JUnit in Eclipse
* Create utility functions
* Understand test results

**Exercise 3: Assertions in Junit**

**1.Objective:**

The objective of this project is to test the methods of the MathUtils class using JUnit 5 assertions to ensure they work correctly and handle different scenarios, including normal and edge cases.

**2.File Descriptions:**

**1.MathUtils.java**

This file contains the main utility class called MathUtils. It provides four simple yet commonly used methods:

* add(int a, int b) — Returns the sum of two integers.
* isEven(int num) — Checks whether a given number is even.
* reverse(String str) — Reverses a string; returns null if the input is null.
* getGreeting(String name) — Returns a personalized greeting. If the name is empty or null, it greets with "Hello, Guest!".

This class serves as the base for applying unit testing using JUnit.

**2.MathUtilsTest.java**

This file contains the JUnit 5 test class used to verify the functionality of the MathUtils class. It includes different test methods, each focusing on specific logic:

* **testAdd()** — Validates the sum of two numbers and checks incorrect results.
* **testIsEven()** — Tests even number detection for both even and odd inputs.
* **testReverse()** — Tests string reversal and handles null input.
* **testGreeting()** — Verifies personalized greetings and default message for empty or null names.
* **testChainedAssertions()** — Uses grouped assertions to check multiple conditions together (e.g., palindrome reversal and string length).

This file demonstrates the use of multiple JUnit assertions such as assertEquals, assertNotEquals, assertTrue, assertFalse, assertNull, assertNotNull, and assertAll.

**1.MathUtils.java**

**Source Code:**

package com.example.utils;

public class MathUtils {

public int add(int a, int b) {

return a + b;

}

public boolean isEven(int num) {

return num % 2 == 0;

}

public String reverse(String str) {

if (str == null) return null;

return new StringBuilder(str).reverse().toString();

}

public String getGreeting(String name) {

if (name == null || name.isBlank()) return "Hello, Guest!";

return "Hello, " + name + "!";

}

}

**2.MathUtilsTest.java**

**Source Code:**

package com.example.utils;

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

import org.junit.jupiter.api.Test;

public class MathUtilsTest {

MathUtils utils = new MathUtils();

@Test

public void testAdd() {

int result = utils.add(10, 20);

assertEquals(30, result, "Addition result should be 30");

assertNotEquals(25, utils.add(12, 12), "Sum should not be 25");

}

@Test

public void testIsEven() {

assertTrue(utils.isEven(8), "8 should be even");

assertFalse(utils.isEven(7), "7 should be odd");

}

@Test

public void testReverse() {

assertEquals("cba", utils.reverse("abc"), "String reverse failed");

assertNull(utils.reverse(null), "Reverse of null should be null");

}

@Test

public void testGreeting() {

assertEquals("Hello, Alice!", utils.getGreeting("Alice"), "Greeting incorrect");

assertEquals("Hello, Guest!", utils.getGreeting(""), "Empty name should greet Guest");

assertEquals("Hello, Guest!", utils.getGreeting(null), "Null name should greet Guest");

}

@Test

public void testChainedAssertions() {

String reversed = utils.reverse("level");

assertAll("Grouped Assertions",

() -> assertNotNull(reversed, "Reversed string should not be null"),

() -> assertEquals("level", reversed, "Palindrome reverse should be same"),

() -> assertTrue(utils.isEven(reversed.length()), "Length of 'level' should be even")

);

}}

**Sample output:**

| Test Method | Status |
| --- | --- |
| testAdd() | Passed |
| testIsEven() | Passed |
| testReverse() | Passed |
| testGreeting() | Passed |
| testChainedAssertions() | Passed |

TotalTestRuns:5  
 Failures:0  
 Errors:0  
 Skipped: 0

**Console Output:**

T E S T S

Running com.example.utils.MathUtilsTest

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.01 sec

Results :

All tests passed.

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

**Project Title:** Book Details Service Testing using Mockito

**Introduction:**

This project demonstrates how to use Mockito for mocking and stubbing dependencies while testing a service that relies on an external API. The service, BookService, fetches book details like title and price using an external BookApi. The objective is to write unit tests that isolate the service logic by mocking the external API and stubbing its methods to return predefined responses.

**Code Implementation:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

/\*\*

\* Interface representing an external Book API.

\*/

interface BookApi {

String fetchTitleByISBN(String isbn);

double fetchPriceByISBN(String isbn);

}

/\*\*

\* Service class that uses BookApi to provide book details.

\*/

class BookService {

private final BookApi bookApi;

public BookService(BookApi bookApi) {

this.bookApi = bookApi;

}

/\*\*

\* Returns formatted information about a book given its ISBN.

\*

\* @param isbn the ISBN of the book

\* @return formatted book details or error messages

\*/

public String getBookDetails(String isbn) {

if (isbn == null || isbn.isBlank()) {

return "Error: ISBN cannot be null or empty.";

}

String title = bookApi.fetchTitleByISBN(isbn);

double price = bookApi.fetchPriceByISBN(isbn);

if (title == null || title.isBlank()) {

return "No book found for ISBN: " + isbn;

}

if (price < 0) {

return "Price information unavailable for '" + title + "'";

}

return String.format("Book: '%s' | Price: $%.2f", title, price);

}

}

/\*\*

\* Unit tests for BookService using Mockito.

\*/

public class BookServiceTest {

private BookApi mockBookApi;

private BookService bookService;

@BeforeEach

public void setup() {

mockBookApi = mock(BookApi.class);

bookService = new BookService(mockBookApi);

}

@Test

public void testGetBookDetails\_ValidISBN\_ReturnsCorrectInfo() {

// Arrange

when(mockBookApi.fetchTitleByISBN("123-XYZ")).thenReturn("Effective Java");

when(mockBookApi.fetchPriceByISBN("123-XYZ")).thenReturn(45.50);

// Act

String result = bookService.getBookDetails("123-XYZ");

// Assert

assertEquals("Book: 'Effective Java' | Price: $45.50", result);

}

@Test

public void testGetBookDetails\_NullOrEmptyISBN\_ReturnsError() {

assertEquals("Error: ISBN cannot be null or empty.", bookService.getBookDetails(null));

assertEquals("Error: ISBN cannot be null or empty.", bookService.getBookDetails(" "));

}

@Test

public void testGetBookDetails\_BookNotFound\_ReturnsNotFoundMessage() {

when(mockBookApi.fetchTitleByISBN("999-ABC")).thenReturn("");

when(mockBookApi.fetchPriceByISBN("999-ABC")).thenReturn(-1.0);

String result = bookService.getBookDetails("999-ABC");

assertEquals("No book found for ISBN: 999-ABC", result);

}

@Test

public void testGetBookDetails\_PriceUnavailable\_ReturnsPriceUnavailableMessage() {

when(mockBookApi.fetchTitleByISBN("456-DEF")).thenReturn("Clean Code");

when(mockBookApi.fetchPriceByISBN("456-DEF")).thenReturn(-1.0);

String result = bookService.getBookDetails("456-DEF");

assertEquals("Price information unavailable for 'Clean Code'", result);

}

}

**Sample Output:**

Test: testGetBookDetails\_ValidISBN\_ReturnsCorrectInfo

Result: Passed

Test: testGetBookDetails\_NullOrEmptyISBN\_ReturnsError

Result: Passed

Test: testGetBookDetails\_BookNotFound\_ReturnsNotFoundMessage

Result: Passed

Test: testGetBookDetails\_PriceUnavailable\_ReturnsPriceUnavailableMessage

Result: Passed

**Conclusion:**

The BookService class was effectively tested using Mockito to mock the external BookApi. Stubbing enabled testing various scenarios such as valid data, missing books, invalid ISBNs, and unavailable prices without depending on the actual API. The unit tests used clear Arrange-Act-Assert structure for readability and maintainability. This approach demonstrates good testing practices suitable for real-world software development and improves test reliability by isolating external dependencies.

**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.

**Project Title:** Payment Processing Service Testing Using Mockito

**Description:**

This project demonstrates how to use Mockito to mock and stub dependencies for unit testing a payment processing service. The PaymentService relies on an external PaymentGateway API to validate credit cards and process payments. The tests verify various scenarios by mocking the PaymentGateway behavior.

**Code Implementation:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

/\*\*

\* External Payment Gateway interface to simulate payment processing.

\*/

interface PaymentGateway {

boolean validateCard(String cardNumber);

String processPayment(String cardNumber, double amount);

}

/\*\*

\* Service that handles payment transactions using the PaymentGateway.

\*/

class PaymentService {

private final PaymentGateway paymentGateway;

public PaymentService(PaymentGateway paymentGateway) {

this.paymentGateway = paymentGateway;

}

/\*\*

\* Processes a payment if the card is valid.

\* Returns a confirmation or error message.

\*/

public String makePayment(String cardNumber, double amount) {

if (cardNumber == null || cardNumber.isBlank() || amount <= 0) {

return "Invalid payment details.";

}

if (!paymentGateway.validateCard(cardNumber)) {

return "Card validation failed.";

}

return paymentGateway.processPayment(cardNumber, amount);

}

}

/\*\*

\* Unit test class with Mockito mocking and verification.

\*/

public class PaymentServiceTest {

@Test

public void testSuccessfulPayment() {

PaymentGateway mockGateway = mock(PaymentGateway.class);

when(mockGateway.validateCard("4111111111111111")).thenReturn(true);

when(mockGateway.processPayment("4111111111111111", 150.0)).thenReturn("Payment Successful! Receipt #12345");

PaymentService paymentService = new PaymentService(mockGateway);

String result = paymentService.makePayment("4111111111111111", 150.0);

assertEquals("Payment Successful! Receipt #12345", result);

verify(mockGateway).validateCard("4111111111111111");

verify(mockGateway).processPayment("4111111111111111", 150.0);

verifyNoMoreInteractions(mockGateway);

}

@Test

public void testCardValidationFails() {

PaymentGateway mockGateway = mock(PaymentGateway.class);

when(mockGateway.validateCard("0000000000000000")).thenReturn(false);

PaymentService paymentService = new PaymentService(mockGateway);

String result = paymentService.makePayment("0000000000000000", 100.0);

assertEquals("Card validation failed.", result);

verify(mockGateway).validateCard("0000000000000000");

verify(mockGateway, never()).processPayment(anyString(), anyDouble());

verifyNoMoreInteractions(mockGateway);

}

@Test

public void testInvalidPaymentDetails() {

PaymentGateway mockGateway = mock(PaymentGateway.class);

PaymentService paymentService = new PaymentService(mockGateway);

assertEquals("Invalid payment details.", paymentService.makePayment(null, 50.0));

assertEquals("Invalid payment details.", paymentService.makePayment("", 50.0));

assertEquals("Invalid payment details.", paymentService.makePayment("4111111111111111", -10));

verifyNoInteractions(mockGateway);

}

}

**Sample Output:**

[INFO] Running PaymentServiceTest

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.012 sec

[INFO]

[INFO] Results:

[INFO]

[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0

This means all tests passed successfully.

**Conclusion:**

This project successfully demonstrates how Mockito can be used to mock an external dependency (PaymentGateway) and stub its methods to simulate different scenarios for the PaymentService. It also verifies method calls and their parameters to ensure correct interactions. This approach helps in isolating the unit under test and facilitates testing in a controlled environment without relying on actual external APIs.

**Exercise 2: Verifying Interactions**

**Scenario:** You need to ensure that a method is called with specific arguments**.**

**Title:** A Practical Demonstration of Mockito for Testing External Interactions

**Description:**

This solution implements a simple Inventory Management System with:

1. Mockito-based unit tests verifying interactions with a ProductRepository
2. Dependency injection for testable design
3. Argument verification to ensure correct method calls

**Code Implementation:**

import static org.mockito.Mockito.\*;

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

import org.junit.jupiter.api.Test;

import java.util.\*;

// ----- ENTITY -----

class Product {

private final String id;

private final String name;

private int quantity;

private final double price;

public Product(String id, String name, int quantity, double price) {

this.id = id;

this.name = name;

this.quantity = quantity;

this.price = price;

}

// Getters

public String getId() { return id; }

public String getName() { return name; }

public int getQuantity() { return quantity; }

public double getPrice() { return price; }

}

// ----- REPOSITORY INTERFACE -----

interface ProductRepository {

void save(Product p);

Product findById(String id);

void updateQuantity(String id, int qty);

}

// ----- INVENTORY SYSTEM -----

class InventorySystem {

private final ProductRepository repo;

public InventorySystem(ProductRepository repo) {

this.repo = repo;

}

public String addProduct(String name, int qty, double price) {

if (name == null || name.isBlank() || qty < 0 || price < 0) return null;

String id = "PROD-" + UUID.randomUUID().toString().substring(0, 8);

repo.save(new Product(id, name, qty, price));

return id;

}

public boolean updateQuantity(String id, int newQty) {

if (newQty < 0 || repo.findById(id) == null) return false;

repo.updateQuantity(id, newQty);

return true;

}

}

// ----- TEST CLASS -----

class InventorySystemTest {

@Test

void testAddProduct\_VerifiesRepositoryInteraction() {

// 1. Create mock

ProductRepository mockRepo = mock(ProductRepository.class);

InventorySystem system = new InventorySystem(mockRepo);

// 2. Execute

system.addProduct("Wireless Mouse", 15, 29.99);

// 3. Verify

verify(mockRepo).save(any(Product.class)); // Checks if save() was called

}

@Test

void testUpdateQuantity\_VerifiesExactArguments() {

// 1. Setup mock

ProductRepository mockRepo = mock(ProductRepository.class);

when(mockRepo.findById("P100")).thenReturn(new Product("P100", "Keyboard", 5, 59.99));

// 2. Execute

InventorySystem system = new InventorySystem(mockRepo);

assertTrue(system.updateQuantity("P100", 10));

// 3. Verify exact arguments

verify(mockRepo).updateQuantity(eq("P100"), eq(10));

}

}

**SAMPLE OUTPUT:**

[INFO] Running InventorySystemTest

[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] Results:

[INFO] testAddProduct\_VerifiesRepositoryInteraction: Passed

[INFO] testUpdateQuantity\_VerifiesExactArguments: Passed

**Conclusion:**

This solution demonstrates proper Mockito usage by:

1. Mocking a ProductRepository dependency
2. Verifying method calls with exact arguments
3. The tests confirm the system interacts correctly with its dependencies.

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

**Project Title:** Movie Ticket Booking System with Logging

**Description:**

This Java project demonstrates a simple Movie Ticket Booking System using SLF4J logging. The system keeps track of available seats and handles bookings, providing appropriate logging messages for various actions. The project covers logging at different levels, including error, warning, and info, based on the situation.

The system checks for the validity of seat bookings, handles seat availability, and warns the user if the seats are sold out. The system also logs error messages for invalid seat selections.

**Code:**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class MovieTicketBookingSystem {

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

private static final int TOTAL\_SEATS = 100;

private int availableSeats;

public MovieTicketBookingSystem() {

this.availableSeats = TOTAL\_SEATS; // Initialize with all seats available

}

public void bookTicket(int seatsToBook) {

if (seatsToBook <= 0) {

logger.error("Invalid seat selection: {}. Please select a valid number of seats.", seatsToBook);

return;

}

if (seatsToBook > availableSeats) {

logger.warn("Booking attempt failed. Not enough seats available. Requested: {}. Available: {}.", seatsToBook, availableSeats);

return;

}

availableSeats -= seatsToBook;

logger.info("Booking successful! {} seat(s) booked. Remaining seats: {}.", seatsToBook, availableSeats);

if (availableSeats == 0) {

logger.warn("Warning: The show is now sold out!");

}

}

public static void main(String[] args) {

MovieTicketBookingSystem bookingSystem = new MovieTicketBookingSystem();

// Valid bookings

bookingSystem.bookTicket(3); // Success

bookingSystem.bookTicket(10); // Success

bookingSystem.bookTicket(50); // Success

// Invalid bookings and warnings

bookingSystem.bookTicket(0); // Error

bookingSystem.bookTicket(100); // Warning: Not enough seats available

bookingSystem.bookTicket(40); // Success, and the show is now sold out

}

}

**SAMPLE OUTPUT:**

INFO MovieTicketBookingSystem - Booking successful! 3 seat(s) booked. Remaining seats: 97.

INFO MovieTicketBookingSystem - Booking successful! 10 seat(s) booked. Remaining seats: 87.

INFO MovieTicketBookingSystem - Booking successful! 50 seat(s) booked. Remaining seats: 37.

ERROR MovieTicketBookingSystem - Invalid seat selection: 0. Please select a valid number of seats.

WARN MovieTicketBookingSystem - Booking attempt failed. Not enough seats available. Requested: 100. Available: 37.

INFO MovieTicketBookingSystem - Booking successful! 40 seat(s) booked. Remaining seats: 0.

WARN MovieTicketBookingSystem - Warning: The show is now sold out!

**Conclusion:**

This project demonstrates how to use SLF4J logging in a real-world context, such as a Movie Ticket Booking System. By using different logging levels (error, warn, info), we can properly log events that occur during the booking process:

* **Error logs** are used for invalid actions such as booking 0 seats.
* **Warning logs** are used when a user tries to book more seats than available or when the show becomes sold out.
* **Info logs** are used for successful bookings, providing feedback to the user about the current status of seat availability.

This simple simulation is not only useful for understanding SLF4J's logging functionality but also for seeing how logging can be applied in a practical application, allowing developers to easily track and debug their systems.