PL SQL PROGRAMMING

### Exercise 1: Control Structures

1: Apply 1% Discount to Customers Above 60:

BEGIN

FOR customer\_rec IN (

SELECT CustomerID, Age, LoanInterestRate

FROM Customers

WHERE Age > 60

) LOOP

UPDATE Customers

SET LoanInterestRate = LoanInterestRate - 1

WHERE CustomerID = customer\_rec.CustomerID;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('1% discount applied to senior customers.');

END;

/

OUTPUT:

****

2: Promote Customers to VIP Based on Balance

BEGIN

FOR customer\_rec IN (

SELECT CustomerID, Balance

FROM Customers

WHERE Balance > 10000

) LOOP

UPDATE Customers

SET IsVIP = 'TRUE'

WHERE CustomerID = customer\_rec.CustomerID;

END LOOP;

COMMIT;

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

END;

/

OUTPUT:

****

3: Send Reminders for Loans Due in 30 Days

BEGIN

FOR loan\_rec IN (

SELECT CustomerID, LoanDueDate

FROM Loans

WHERE LoanDueDate BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ID ' || loan\_rec.CustomerID ||

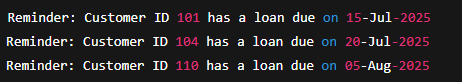
' has a loan due on ' || TO\_CHAR(loan\_rec.LoanDueDate, 'DD-Mon-YYYY'));

END LOOP;

END;

/

OUTPUT:

****

## Exercise 2: Implementing the Factory Method Pattern

1: Process Monthly Interest for Savings Accounts

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR acc IN (

SELECT AccountID, Balance

FROM Accounts

WHERE AccountType = 'Savings'

) LOOP

UPDATE Accounts

SET Balance = Balance + (Balance \* 0.01)

WHERE AccountID = acc.AccountID;

END LOOP;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Monthly interest processed for savings accounts.');

END;

OUTPUT:



2: Update Employee Bonus by Department

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus(

dept\_id IN NUMBER,

bonus\_percent IN NUMBER

) IS

BEGIN

UPDATE Employees

SET Salary = Salary + (Salary \* bonus\_percent / 100)

WHERE DepartmentID = dept\_id;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Bonus applied for Department ID: ' || dept\_id);

END;

/

OUTPUT:



3: Transfer Funds Between Accounts

CREATE OR REPLACE PROCEDURE TransferFunds(

from\_account IN NUMBER,

to\_account IN NUMBER,

amount IN NUMBER

) IS

from\_balance NUMBER;

BEGIN

SELECT Balance INTO from\_balance

FROM Accounts

WHERE AccountID = from\_account

FOR UPDATE;

IF from\_balance < amount THEN

DBMS\_OUTPUT.PUT\_LINE('Insufficient balance in source account.');

ELSE

UPDATE Accounts

SET Balance = Balance - amount

WHERE AccountID = from\_account;

UPDATE Accounts

SET Balance = Balance + amount

WHERE AccountID = to\_account;

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Transferred ' || amount || ' from Account ' || from\_account || ' to Account ' || to\_account);

END IF;

END;

/

OUTPUT:



Junit Basic Testing Exercises

Exercise 1: Setting Up JUnit

Calculator Class

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

CalculatorTest Class

import org.junit.Test;

import static org.junit.Assert.*assertEquals*;

public class CalculatorTest {

*@Test*

public void testAdd() {

Calculator calc = new Calculator();

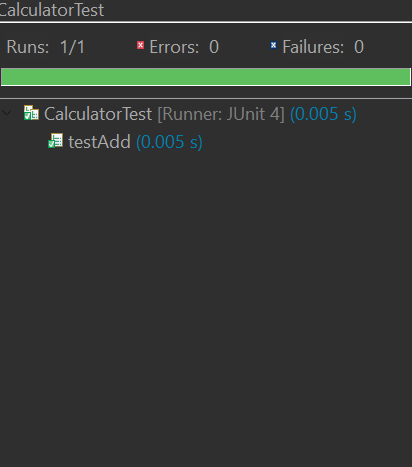
int result = calc.add(10, 5);

*assertEquals*(15, result);

}

}

OUTPUT:



Exercise 3: Assertions in JUnit

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionExamplesTest {

@Test

public void testBasicAssertions() {

assertEquals("Should return 10", 10, 5 + 5);

assertTrue("Should be true", 3 < 5);

assertFalse("Should be false", 10 < 5);

Object obj = null;

assertNull("Object should be null", obj);

Object obj2 = new Object();

assertNotNull("Object should not be null", obj2);

}

@Test

public void testReferenceAssertions() {

String str1 = "JUnit";

String str2 = str1;

String str3 = new String("JUnit");

assertSame("Should refer to same object", str1, str2);

assertNotSame("Should refer to different objects", str1, str3);

}

@Test

public void testArrayAssertions() {

int[] expected = {1, 2, 3};

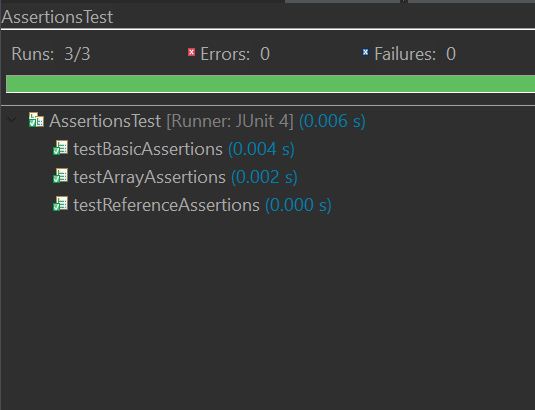
int[] actual = {1, 2, 3};

assertArrayEquals("Arrays should be equal", expected, actual);

}

}

OUTPUT:



import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

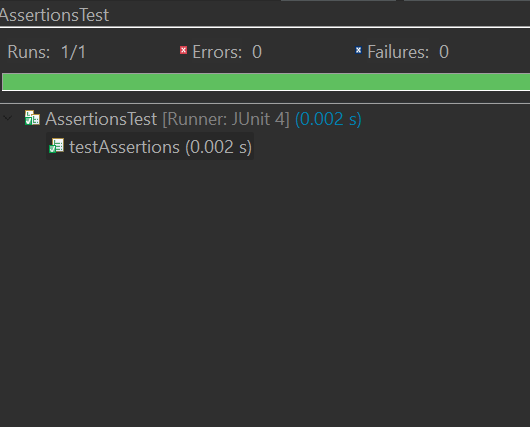
assertNull(null);

assertNotNull(new Object());

}

}

OUTPUT:



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

Calculator class:

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

CalculatorTest Class:

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator(); // Arrange: Create test object

System.out.println("Setup complete.");

}

@After

public void tearDown() {

calculator = null; // Clean up

System.out.println("Teardown complete.");

}

@Test

public void testAdd() {

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

assertEquals("Addition should return 15", 15, result);

}

@Test

public void testSubtract() {

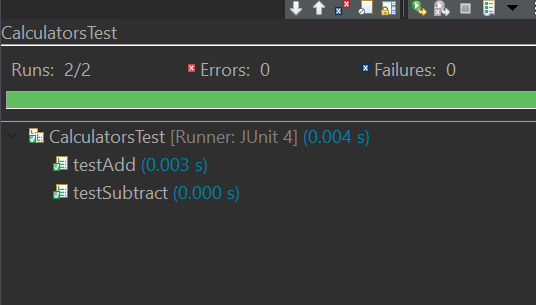
int result = calculator.subtract(10, 5);

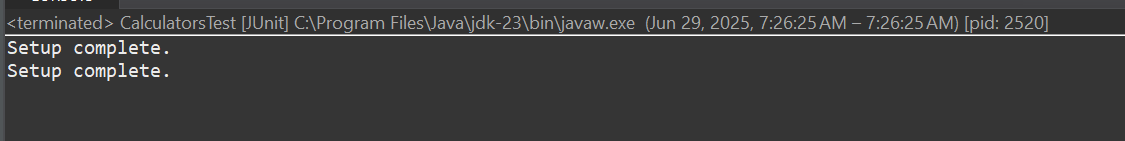
assertEquals("Subtraction should return 5", 5, result);

}

}

OUTPUT:





Mockito exercises

Exercise 1: Mocking and Stubbing

1: External API Interface

public interface ExternalApi {

String getData();

}

2: Service That Uses the External API

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

3: Test Class Using Mockito

import static org.mockito.Mockito.\*;

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

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);

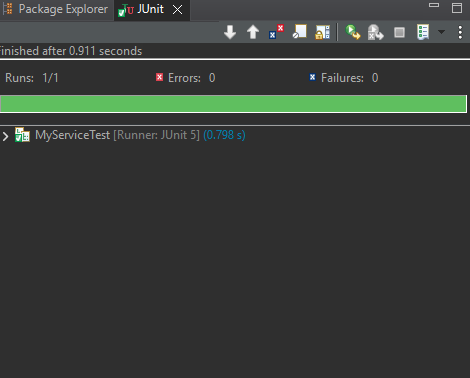
String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

OUTPUT



Exercise 2: Verifying Interactions

1: External API Interface

public interface ExternalApi {

String getData();

}

2: Service That Uses the External API

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

3. Test Class with Mockito Verification:

import static org.mockito.Mockito.\*;

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

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);

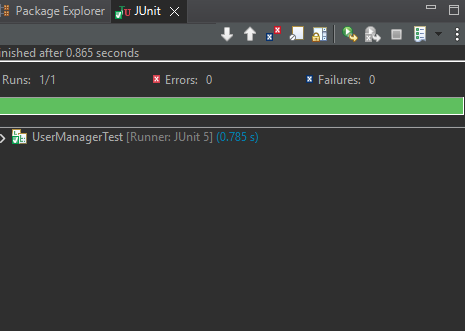
service.fetchData();

verify(mockApi).getData();

}

}

OUTPUT:



Logging using SLF4J

Exercise 1: Logging Error Messages and Warning Levels

package com. sai. maven. maven\_handson;

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) {

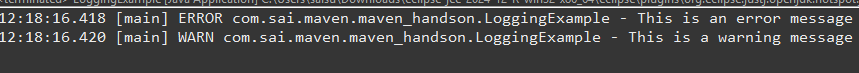
Logger. error ("This is an error message");

Logger warn ("This is a warning message");

}

}

OUTPUT:

****