WEEK – 2

PL/SQL PROGRAMMING

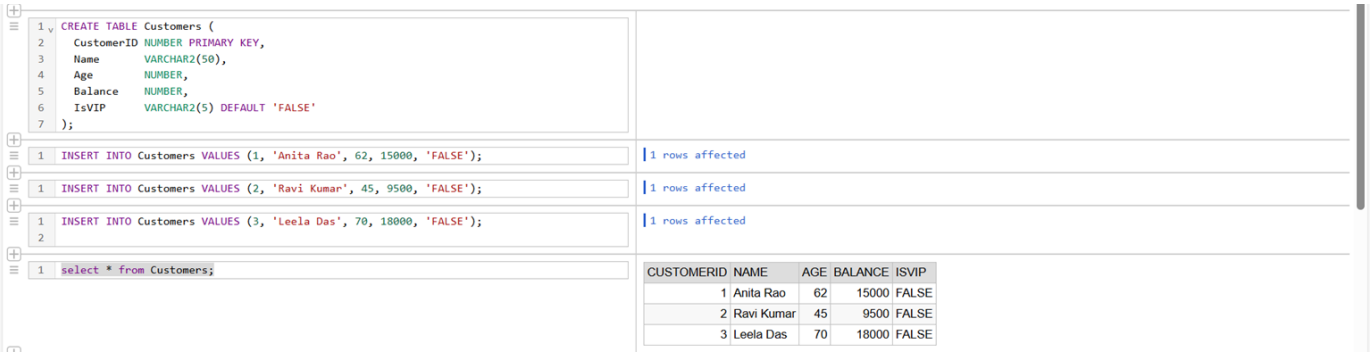
Exercise 1: Control Structures

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.

CODE :

```
CREATE TABLE Customers (
  CustomerID NUMBER PRIMARY KEY,
  Name       VARCHAR2(50),
  Age        NUMBER,
  Balance    NUMBER,
  IsVIP      VARCHAR2(5) DEFAULT 'FALSE'
);

INSERT INTO Customers VALUES (1, 'Anita Rao', 62, 15000, 'FALSE');
INSERT INTO Customers VALUES (2, 'Ravi Kumar', 45, 9500, 'FALSE');
INSERT INTO Customers VALUES (3, 'Leela Das', 70, 18000, 'FALSE');
COMMIT;

SELECT * FROM Customers;
```




```
CREATE TABLE Loans (
  LoanID       NUMBER PRIMARY KEY,
  CustomerID   NUMBER,
  InterestRate NUMBER,
  DueDate      DATE,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

INSERT INTO Loans VALUES (101, 1, 9.5, SYSDATE + 10);
INSERT INTO Loans VALUES (102, 2, 10.5, SYSDATE + 40);
INSERT INTO Loans VALUES (103, 3, 11.0, SYSDATE + 20);
COMMIT;
```

SELECT * FROM Loans;

```
CREATE TABLE Loans (
  LoanID        NUMBER PRIMARY KEY,
  CustomerID    NUMBER,
  InterestRate  NUMBER,
  DueDate       DATE,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

```
INSERT INTO Loans VALUES (101, 1, 9.5, SYSDATE + 10);
```

1 rows affected

```
INSERT INTO Loans VALUES (102, 3, 18.0, SYSDATE + 20);
```

1 rows affected

```
INSERT INTO Loans VALUES (103, 3, 11.0, SYSDATE + 20);
```

1 rows affected

```
select * from Loans;
```

LOANID	CUSTOMERID	INTERESTRATE	DUEDATE
101	1	9.5	07-JUL-25
102	3	18	17-JUL-25
103	3	11	17-JUL-25

```
COMMIT;
```

```
BEGIN
  FOR rec IN (
    SELECT LoanID, l.CustomerID, c.Name, c.Age, l.InterestRate
    FROM Loans l
    JOIN Customers c ON l.CustomerID = c.CustomerID
  ) LOOP
    IF rec.Age > 60 THEN
      UPDATE Loans
      SET InterestRate = rec.InterestRate - 1
      WHERE LoanID = rec.LoanID;

      DBMS_OUTPUT.PUT_LINE(' Interest rate discounted for ' || rec.Name);
    END IF;
  END LOOP;
  COMMIT;
END;
/
```

```
BEGIN
  FOR rec IN (
    SELECT LoanID, l.CustomerID, c.Name, c.Age, l.InterestRate
    FROM Loans l
    JOIN Customers c ON l.CustomerID = c.CustomerID
  ) LOOP
    IF rec.Age > 60 THEN
      UPDATE Loans
      SET InterestRate = rec.InterestRate - 1
      WHERE LoanID = rec.LoanID;

      DBMS_OUTPUT.PUT_LINE(' Interest rate discounted for ' || rec.Name);
    END IF;
  END LOOP;
  COMMIT;
END;
/
```

1 rows affected

```
dbms_output:
 Interest rate discounted for Anita Rao
 Interest rate discounted for Leela Das
 Interest rate discounted for Leela Das
```

Scenario 2: A customer can be promoted to VIP status based on their balance.

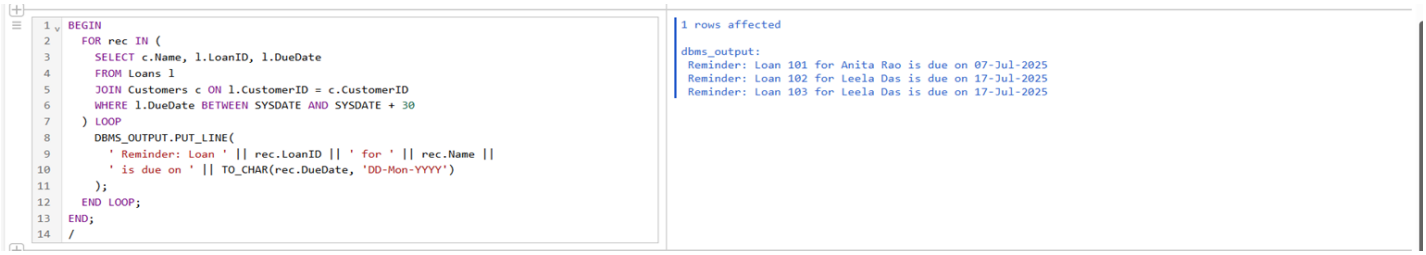
Question: Write a PL/SQL block that iterates through all customers and sets a flag IsVIP to TRUE for those with a balance over $10,000

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

```
    UPDATE Customers
    SET IsVIP = 'TRUE'
    WHERE CustomerID = rec.CustomerID;

    DBMS_OUTPUT.PUT_LINE(' ' || rec.Name || ' has been promoted to VIP status.');
  END IF;
 END LOOP;
 COMMIT;
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.

```
BEGIN
 FOR rec IN (
  SELECT c.Name, l.LoanID, l.DueDate
  FROM Loans l
  JOIN Customers c ON l.CustomerID = c.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;
/
```



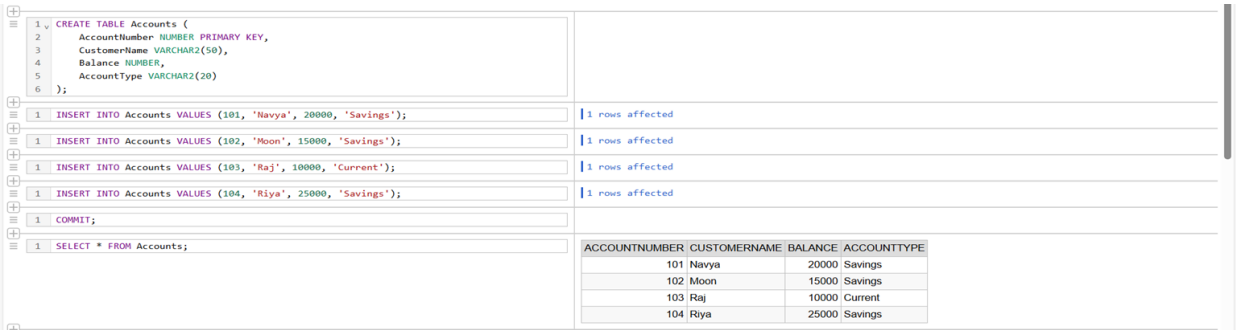
Exercise 3: Stored Procedures

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.

CODE :

```
CREATE TABLE Accounts (
   AccountNumber NUMBER PRIMARY KEY,
   CustomerName VARCHAR2(50),
   Balance NUMBER,
   AccountType VARCHAR2(20)
);
INSERT INTO Accounts VALUES (101, 'Navya', 20000, 'Savings');
INSERT INTO Accounts VALUES (102, 'Moon', 15000, 'Savings');
INSERT INTO Accounts VALUES (103, 'Raj', 10000, 'Current');
INSERT INTO Accounts VALUES (104, 'Riya', 25000, 'Savings');
COMMIT;
SELECT * FROM Accounts;
```

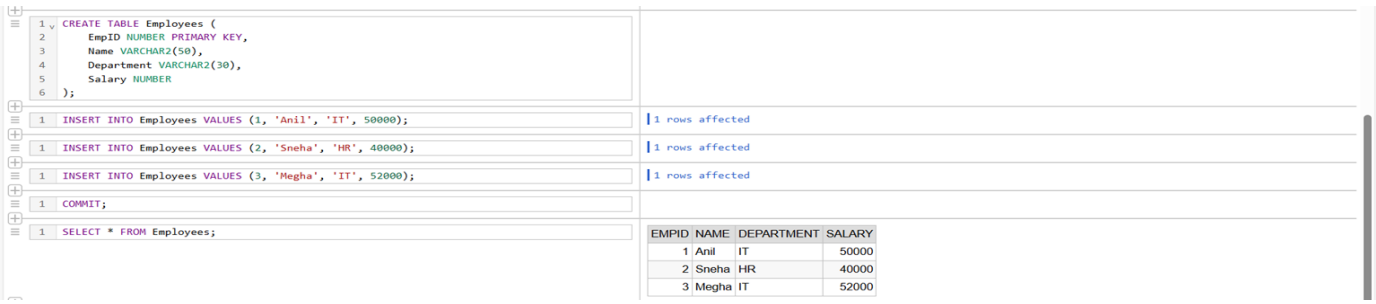


```
CREATE TABLE Accounts (
    AccountNumber NUMBER PRIMARY KEY,
    CustomerName VARCHAR2(50),
    Balance NUMBER,
    AccountType VARCHAR2(20)
);
INSERT INTO Accounts VALUES (101, 'Navya', 20000, 'Savings');
INSERT INTO Accounts VALUES (102, 'Moon', 15000, 'Savings');
INSERT INTO Accounts VALUES (103, 'Raj', 10000, 'Current');
INSERT INTO Accounts VALUES (104, 'Riya', 25000, 'Savings');
COMMIT;
SELECT * FROM Accounts;
```

1 rows affected
1 rows affected
1 rows affected
1 rows affected

ACCOUNTNUMBER	CUSTOMERNAME	BALANCE	ACCOUNTTYPE
101	Navya	20000	Savings
102	Moon	15000	Savings
103	Raj	10000	Current
104	Riya	25000	Savings

```
CREATE TABLE Employees (
   EmpID NUMBER PRIMARY KEY,
   Name VARCHAR2(50),
   Department VARCHAR2(30),
   Salary NUMBER
);
INSERT INTO Employees VALUES (1, 'Anil', 'IT', 50000);
INSERT INTO Employees VALUES (2, 'Sneha', 'HR', 40000);
INSERT INTO Employees VALUES (3, 'Megha', 'IT', 52000);
COMMIT;
SELECT * FROM Employees;
```



```
CREATE TABLE Employees (
    EmpID NUMBER PRIMARY KEY,
    Name VARCHAR2(50),
    Department VARCHAR2(30),
    Salary NUMBER
);
INSERT INTO Employees VALUES (1, 'Anil', 'IT', 50000);
INSERT INTO Employees VALUES (2, 'Sneha', 'HR', 40000);
INSERT INTO Employees VALUES (3, 'Megha', 'IT', 52000);
COMMIT;
SELECT * FROM Employees;
```

1 rows affected
1 rows affected
1 rows affected

EMPID	NAME	DEPARTMENT	SALARY
1	Anil	IT	50000
2	Sneha	HR	40000
3	Megha	IT	52000

```
BEGIN
  FOR acc IN (
    SELECT AccountNumber, Balance FROM Accounts WHERE AccountType = 'Savings'
```

```
  ) LOOP
    UPDATE Accounts
    SET Balance = acc.Balance + (acc.Balance * 0.01)
    WHERE AccountNumber = acc.AccountNumber;

    DBMS_OUTPUT.PUT_LINE('Interest applied to Account ' || acc.AccountNumber);
  END LOOP;
  COMMIT;
END;
/
```

```
BEGIN
  FOR acc IN (
    SELECT AccountNumber, Balance FROM Accounts WHERE AccountType = 'Savings'
  ) LOOP
    UPDATE Accounts
    SET Balance = acc.Balance + (acc.Balance * 0.01)
    WHERE AccountNumber = acc.AccountNumber;

    DBMS_OUTPUT.PUT_LINE('Interest applied to Account ' || acc.AccountNumber);
  END LOOP;
  COMMIT;
END;
/
```

```
1 rows affected

dbms_output:
Interest applied to Account 101
Interest applied to Account 102
Interest applied to Account 104
```

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.

```
BEGIN
  FOR emp IN (
    SELECT EmpID, Name, Salary FROM Employees WHERE Department = 'IT'
  ) LOOP
    UPDATE Employees
    SET Salary = emp.Salary + (emp.Salary * 10 / 100)
    WHERE EmpID = emp.EmpID;

    DBMS_OUTPUT.PUT_LINE(emp.Name || ' received bonus. New Salary: ' ||
                         TO_CHAR(emp.Salary + (emp.Salary * 10 / 100)));
  END LOOP;
  COMMIT;
END;
/
```

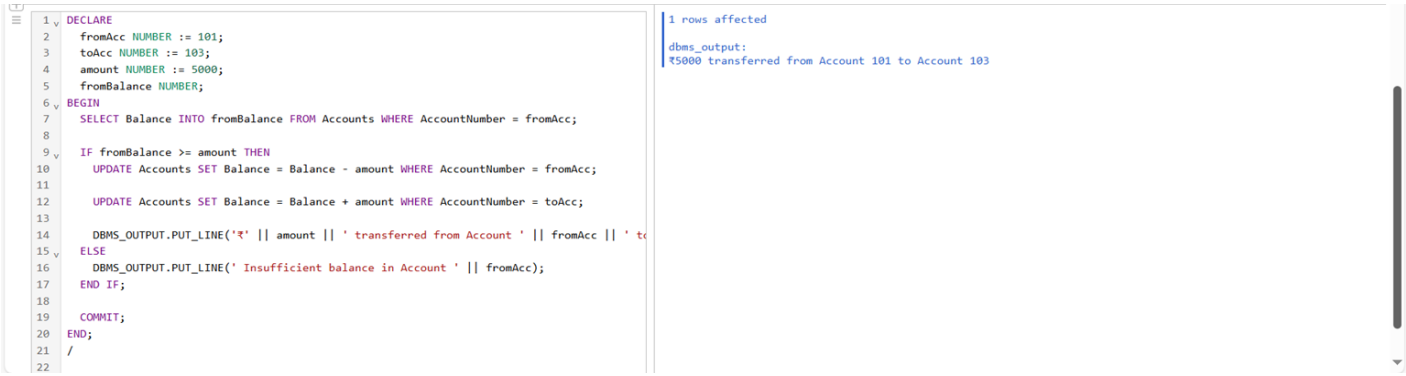
```
BEGIN
  FOR emp IN (
    SELECT EmpID, Name, Salary FROM Employees WHERE Department = 'IT'
  ) LOOP
    UPDATE Employees
    SET Salary = emp.Salary + (emp.Salary * 10 / 100)
    WHERE EmpID = emp.EmpID;

    DBMS_OUTPUT.PUT_LINE(emp.Name || ' received bonus. New Salary: ' ||
                         TO_CHAR(emp.Salary + (emp.Salary * 10 / 100)));
  END LOOP;
  COMMIT;
END;
/
```

```
1 rows affected

dbms_output:
Anil received bonus. New Salary: 55000
Megha received bonus. New Salary: 57200
```

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.

```
DECLARE
  fromAcc NUMBER := 101;
  toAcc NUMBER := 103;
  amount NUMBER := 5000;
  fromBalance NUMBER;
BEGIN
  SELECT Balance INTO fromBalance FROM Accounts WHERE AccountNumber = fromAcc;

  IF fromBalance >= amount THEN
    UPDATE Accounts SET Balance = Balance - amount WHERE AccountNumber = fromAcc;

    UPDATE Accounts SET Balance = Balance + amount WHERE AccountNumber = toAcc;

    DBMS_OUTPUT.PUT_LINE('₹' || amount || ' transferred from Account ' || fromAcc || ' to
Account ' || toAcc);
  ELSE
    DBMS_OUTPUT.PUT_LINE(' Insufficient balance in Account ' || fromAcc);
  END IF;

  COMMIT;
END;
/
```



TDD USING JUNIT – 5 AND MOCKITO

Junit_Basic 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: junit junit 4.13.2 test 3. Create a new test class in your project.

CODE :

pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>Navya</groupId>
  <artifactId>myartifactid</artifactId>
  <version>0.0-SNAPSHOT</version>

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

Calculator.java

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

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

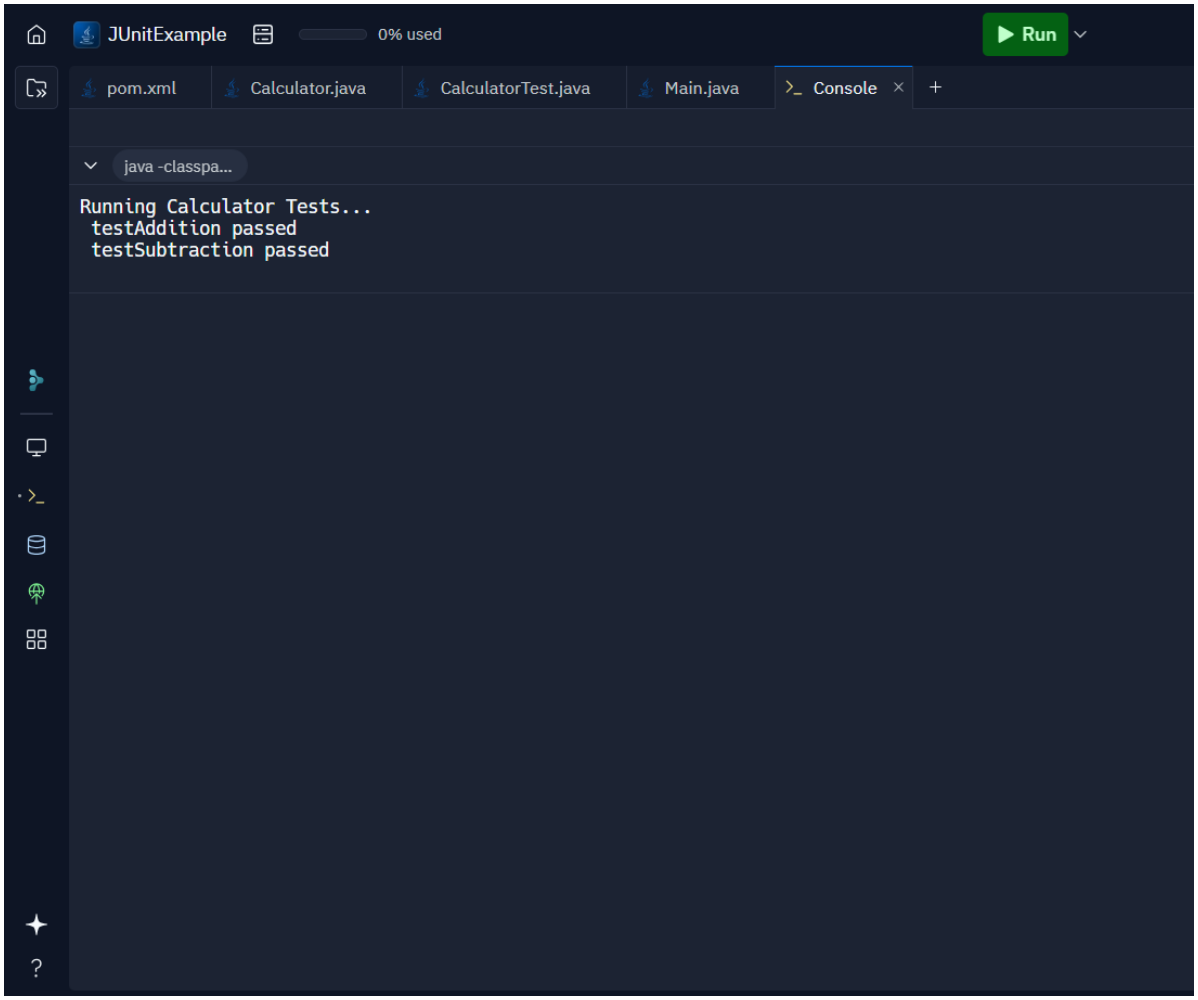
CalculatorTest.java

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

public class CalculatorTest {

    @Test
    public void testAddition() {
        Calculator calc = new Calculator();
        int result = calc.add(10, 5);
        assertEquals(15, result);
    }

    @Test
    public void testSubtraction() {
        Calculator calc = new Calculator();
        int result = calc.subtract(10, 5);
        assertEquals(5, result);
    }
}
```

Main.java

```
public class Main {
   public static void main(String[] args) {
      Calculator calc = new Calculator();

      int add = calc.add(10, 5);
      int sub = calc.subtract(10, 5);
      System.out.println("Running Calculator Tests...");
      if (add == 15) {
         System.out.println(" testAddition passed");
      } else {
         System.out.println(" testAddition failed. Expected 15 but got " + add);
      }

      if (sub == 5) {
         System.out.println(" testSubtraction passed");
      } else {
         System.out.println(" testSubtraction failed. Expected 5 but got " + sub);
      }
   }
}
```

OUTPUT :



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());
}
}
```

CODE :

pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>Navya</groupId>
  <artifactId>myartifactid</artifactId>
  <version>0.0-SNAPSHOT</version>

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

AssertionsTest.java

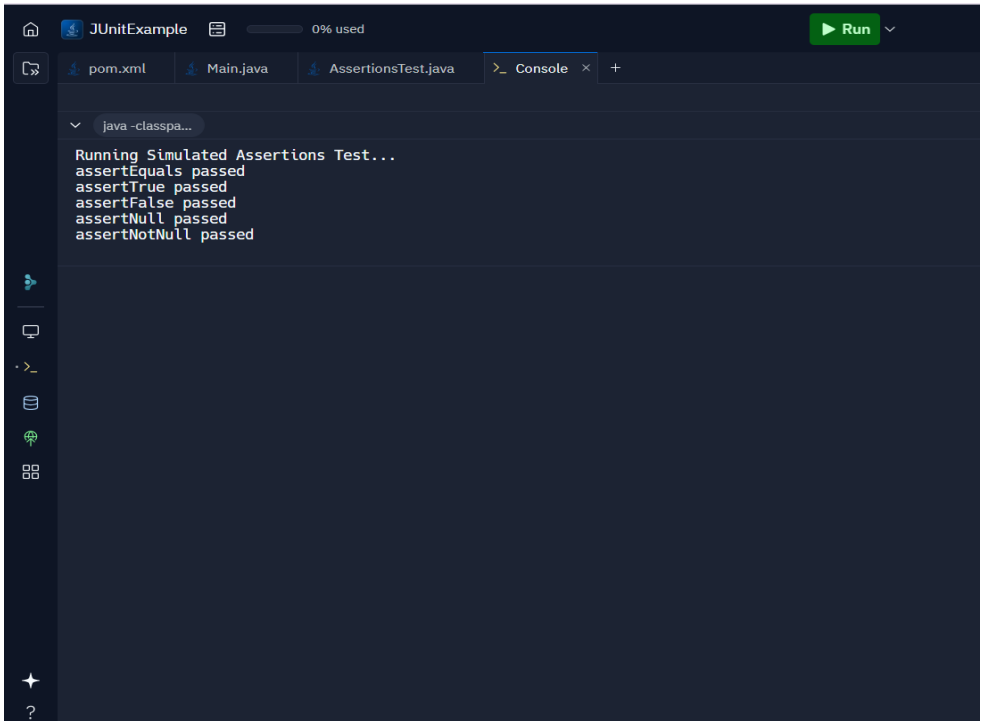
```
import org.junit.Test;
import static org.junit.Assert.*;

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());
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        System.out.println(" Running Simulated Assertions Test...");

        if (2 + 3 == 5) System.out.println(" assertEquals passed");
        else System.out.println(" assertEquals failed");

        if (5 > 3) System.out.println(" assertTrue passed");
        else System.out.println(" assertTrue failed");

        if (!(5 < 3)) System.out.println(" assertFalse passed");
        else System.out.println(" assertFalse failed");

        Object obj = null;
        if (obj == null) System.out.println(" assertNull passed");
        else System.out.println(" assertNull failed");

        Object obj2 = new Object();
        if (obj2 != null) System.out.println(" assertNotNull passed");
        else System.out.println(" assertNotNull failed");
    }
}
```

OUTPUT :



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.

Calculator.java

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

   public int multiply(int a, int b) {
      return a * b;
   }
}
```

CalculatorTest.java

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

public class CalculatorTest {
```

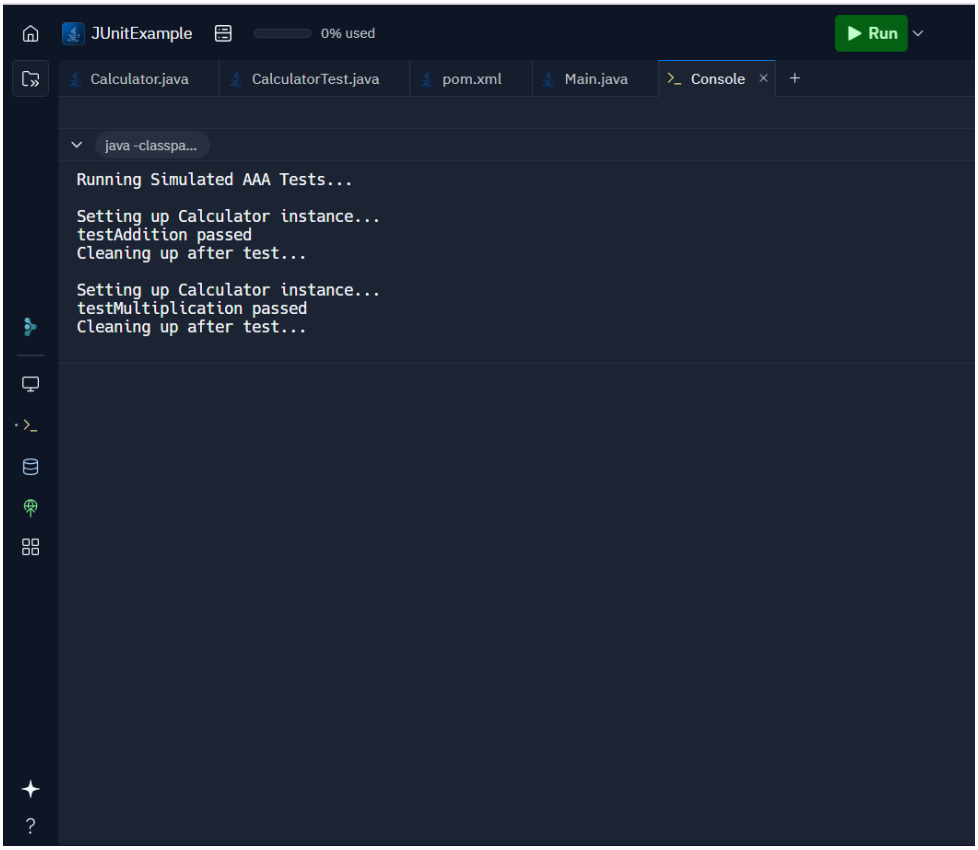
```
    private Calculator calc;

    @Before
    public void setUp() {
        System.out.println("Setting up Calculator instance...");
        calc = new Calculator();
    }

    @After
    public void tearDown() {
        System.out.println("Cleaning up after test...\n");
        calc = null;
    }

    @Test
    public void testAddition() {

        int result = calc.add(4, 6);

        assertEquals(10, result);
        System.out.println(" testAddition passed");
    }

    @Test
    public void testMultiplication() {

        int result = calc.multiply(3, 5);

        assertEquals(15, result);
        System.out.println(" testMultiplication passed");
    }
}
```

Main.java

```
public class Main {
    static Calculator calc;

    public static void setUp() {
        System.out.println(" Setting up Calculator instance...");
        calc = new Calculator();
    }

    public static void tearDown() {
        System.out.println(" Cleaning up after test...\n");
        calc = null;
    }

    public static void testAddition() {
        setUp();

        int result = calc.add(4, 6);
```

```
        if (result == 10) {
            System.out.println(" testAddition passed");
        } else {
            System.out.println(" testAddition failed");
        }

        tearDown();
    }

    public static void testMultiplication() {
        setUp();


        int result = calc.multiply(3, 5);


        if (result == 15) {
            System.out.println(" testMultiplication passed");
        } else {
            System.out.println(" testMultiplication failed");
        }

        tearDown();
    }

    public static void main(String[] args) {
        System.out.println(" Running Simulated AAA Tests...\n");
        testAddition();
        testMultiplication();
    }
}
```

OUTPUT :



Mockito 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 :

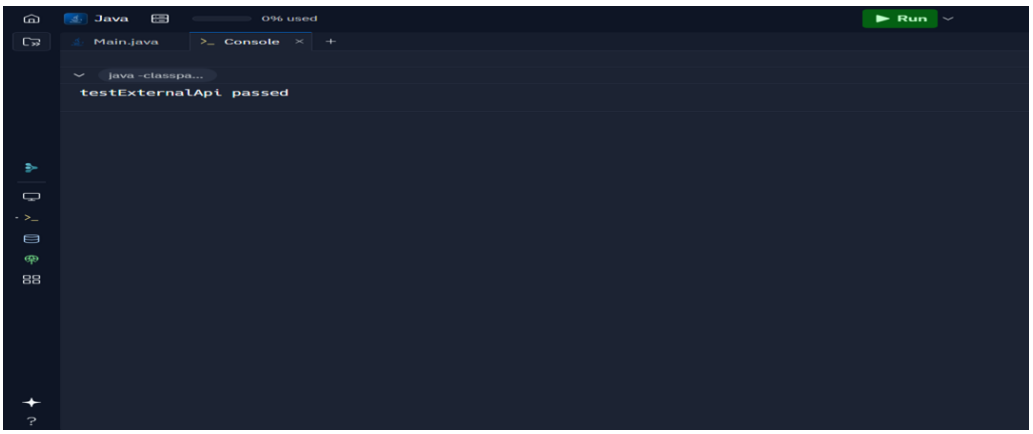
1. Create a mock object for the external API.
2. Stub the methods to return predefined values.
3. Write a test case that uses the mock object.

Main.java

```
// Step 1: Define ExternalApi interface
interface ExternalApi {
   String getData();
}

// Step 2: MyService depends on ExternalApi
class MyService {
   private ExternalApi api;

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

   public String fetchData() {
```

```
        return api.getData();
    }
}

// Step 3: Simulated Test Case (Manual mocking & stubbing)
public class Main {
    public static void main(String[] args) {
        // Create mock object (manual)
        ExternalApi mockApi = new ExternalApi() {
            @Override
            public String getData() {
                // Stubbed method: return predefined value
                return "Mock Data";
            }
        };

        // Pass mock into MyService
        MyService service = new MyService(mockApi);

        // Simulate assertion
        String result = service.fetchData();
        if ("Mock Data".equals(result)) {
            System.out.println(" testExternalApi passed");
        } else {
            System.out.println(" testExternalApi failed");
        }
    }
}
```

OUTPUT :



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.

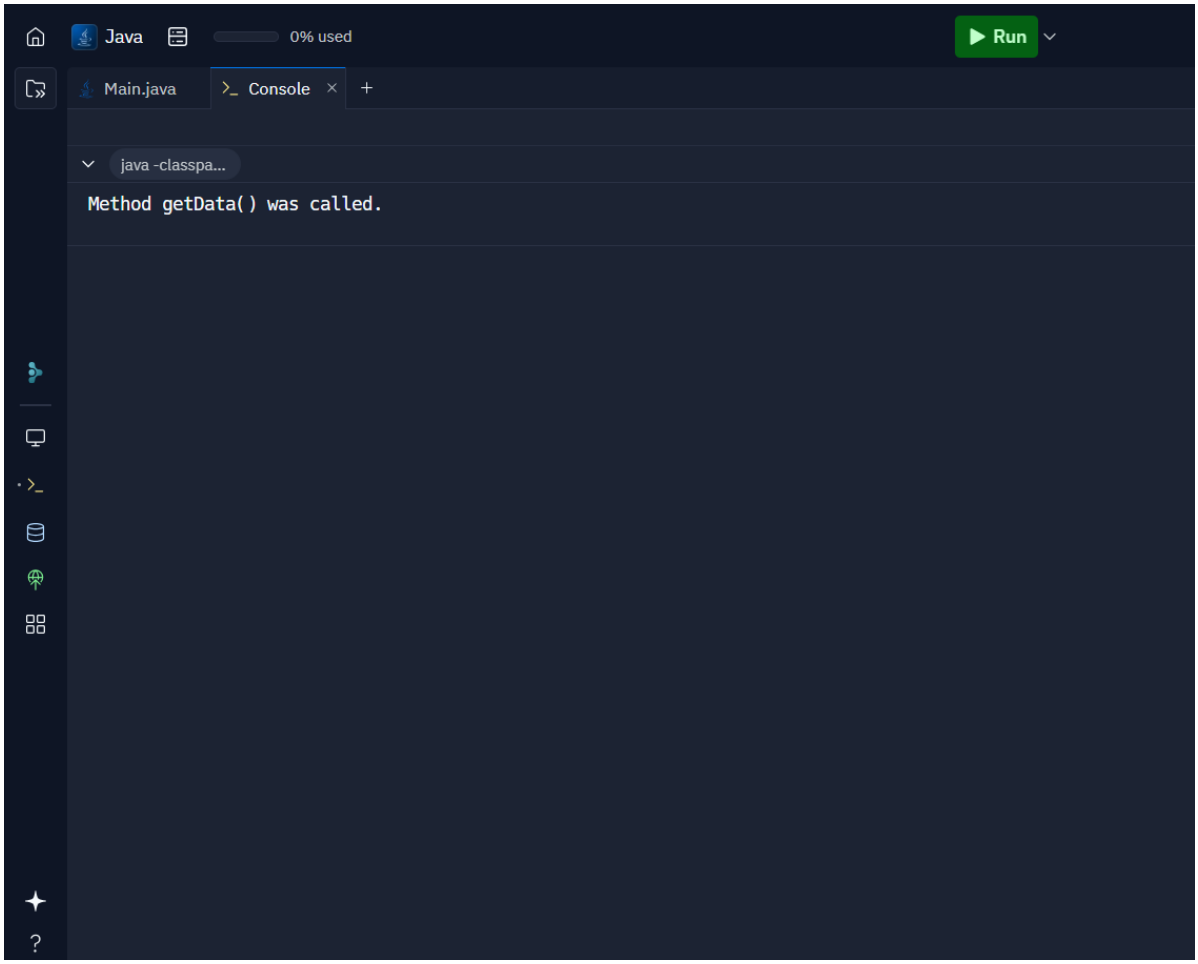
Main.java

```
// Step 1: ExternalApi interface with tracking variable
interface ExternalApi {
   String getData();
}

// Step 2: Manual Mock that tracks interaction
class MockExternalApi implements ExternalApi {
   boolean wasCalled = false;

   @Override
   public String getData() {
      wasCalled = true;  // Track call
      return "Mock Data";
   }

   public boolean verifyCalled() {
      return wasCalled;
   }
}

// Step 3: Service class that uses the API
class MyService {
   private ExternalApi api;

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

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

// Step 4: Simulated Test Case
public class Main {
   public static void main(String[] args) {
      // Create mock
      MockExternalApi mockApi = new MockExternalApi();

      // Call method
      MyService service = new MyService(mockApi);
      service.fetchData();

      // Simulate verify(mockApi).getData()
      if (mockApi.verifyCalled()) {
         System.out.println(" Method getData() was called.");
      } else {
         System.out.println(" Method getData() was NOT called.");
      }
   }
}
```

OUTPUT :

SL4J LOGGING FRAMEWORK

Exercise 1: Logging Error Messages and Warning Levels



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

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.navya.logging</groupId>
    <artifactId>LoggingExample</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>24</maven.compiler.source>
        <maven.compiler.target>24</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
```

```
    <dependencies>
        <!-- SLF4J API -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.30</version>
        </dependency>

        <!-- Logback (SLF4J Implementation) -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
        </dependency>
    </dependencies>


</project>
```

LoggerExample.java

```
package com.navya.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");
    }
}
```

OUTPUT :