Candidate Name:- ROHITHGOWDA V

Superset ID:-6430364

Mail ID:- [rohith2005v@gmail.com](mailto:rohith2005v@gmail.com)

**WEEK – 2 HANDS ON EXERCISE (JAVA FSE DEEPSKILLING)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**SLF4J LOGGING FRAMEWORK**

**(Logging using SLF4J)**

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

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

**Step-by-Step Solution**:

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

<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>

1. Create a Java class that uses SLF4J for 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");

}

}

**Code for the above question:-**

**Java Class:- LoggingExample.java**

package com.example.riohith;  
  
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:**

C:\Users\rohit\.jdks\ms-17.0.15\bin\java.exe

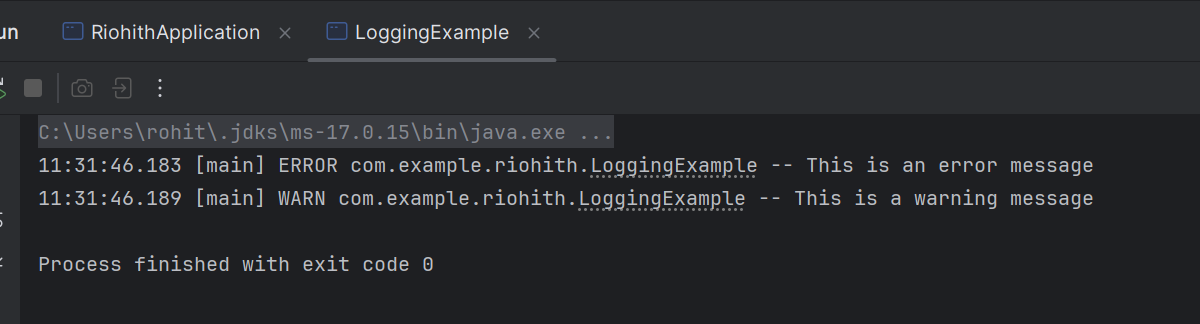
11:31:46.183 [main] ERROR com.example.riohith. LoggingExample

--This is an error message

11:31:46.189 [main] WARN com.example.riohith. Logging Example

--This is a warning message

Process finished with exit code 0

**Output Image:**

**TDD USING JUNIT 5 AND MOCKITO**

**(Mockito Hands-On 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.

Solution Code:

import static org.mockito.Mockito.\*;

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);

}

}

**Code for the above question:**

**Java class: ExternalApi.java**

package com.example.riohith;  
  
public interface ExternalApi {  
 String getData();  
}

**Java class: MyService.java**

package com.example.riohith;  
  
public class MyService {  
 private final ExternalApi api;  
  
 public MyService(ExternalApi api) {  
 this.api = api;  
 }  
  
 public String fetchData() {  
 return api.getData();  
 }  
}

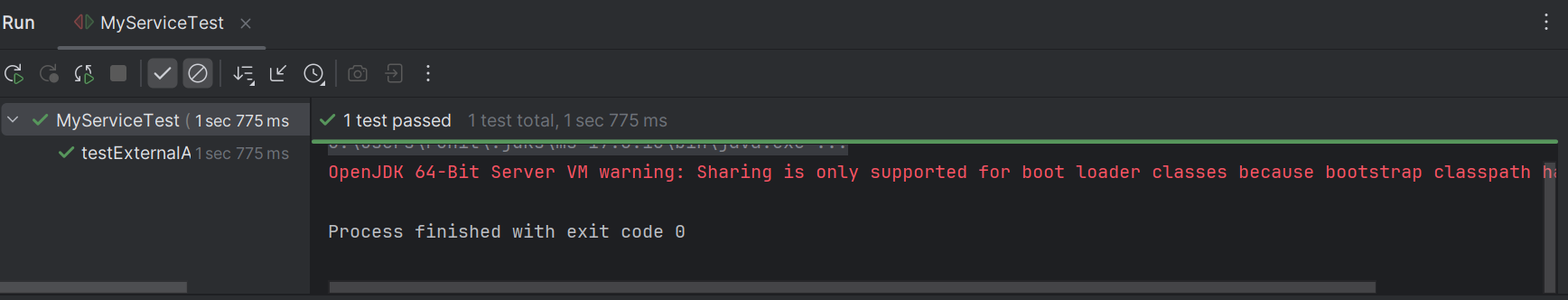
**Java class : MyServiceTest.java**

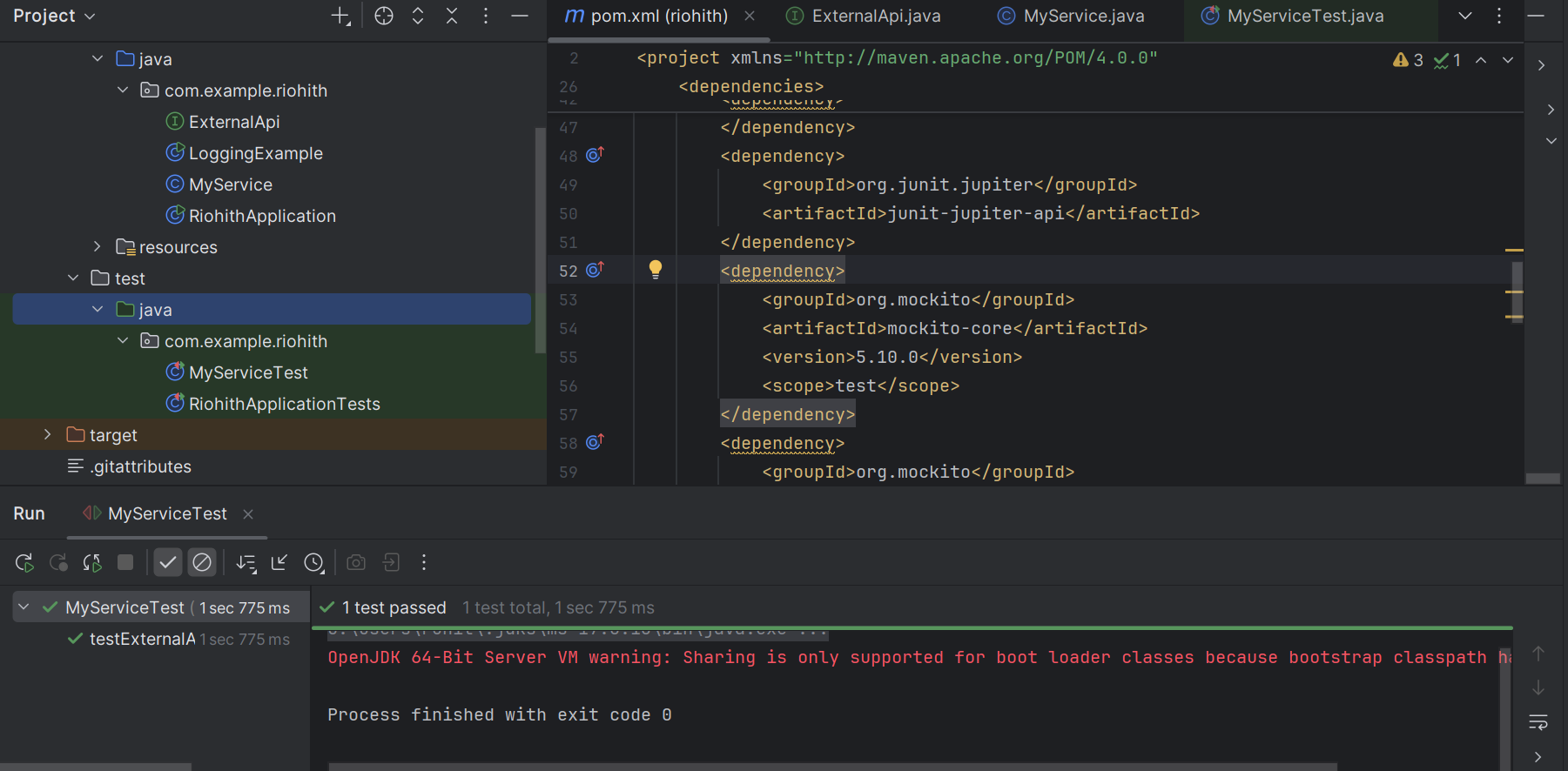
package com.example.riohith;  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.mockito.Mockito.\*;  
  
public class MyServiceTest {  
  
 @Test  
 public void testExternalApi() {  
 ExternalApi mockApi = *mock*(ExternalApi.class);  
 *when*(mockApi.getData()).thenReturn("Mock Data");  
  
 MyService service = new MyService(mockApi);  
 String result = service.fetchData();  
  
 *assertEquals*("Mock Data", result);  
 }  
}

**Output:**

Process finished with exit code 0

**Output Image:**

****



**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.

Solution Code:

import static org.mockito.Mockito.\*;

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();

}

}

**Code for above question:**

**Java class: ExternalApi.java**

package com.example.riohith;  
  
public interface ExternalApi {  
 String getData();  
}

**Java class: MyService.java**

package com.example.riohith;  
  
public class MyService {  
 private final ExternalApi api;  
  
 public MyService(ExternalApi api) {  
 this.api = api;  
 }  
  
 public String fetchData() {  
 return api.getData();  
 }  
}

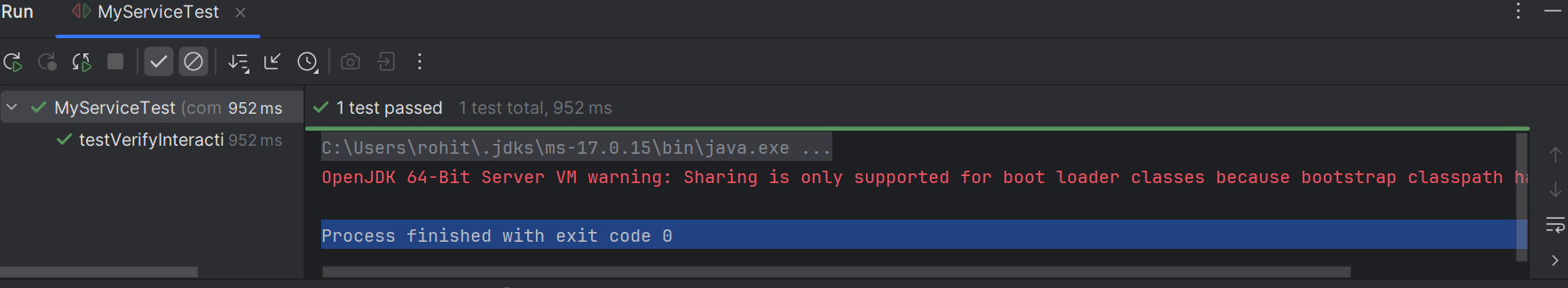
**Java class:MyServiceTest.java**

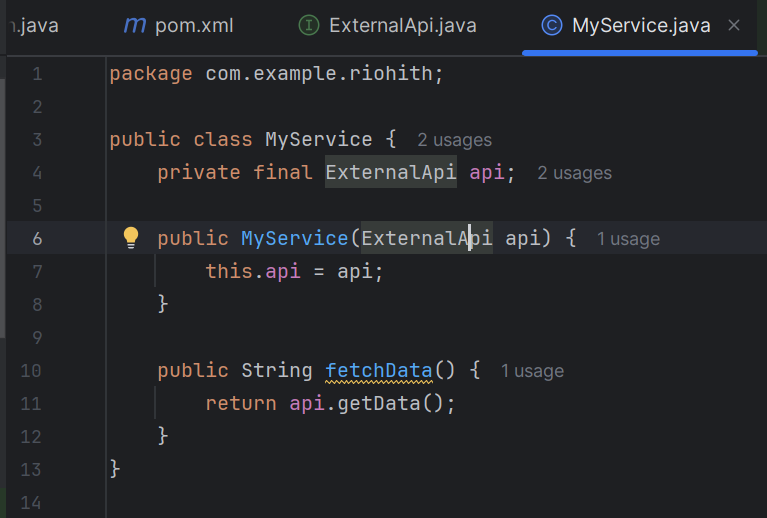
package com.example.riohith;  
import org.junit.jupiter.api.Test;  
import static org.mockito.Mockito.\*;  
  
public class MyServiceTest {  
  
 @Test  
 public void testVerifyInteraction() {  
 ExternalApi mockApi = *mock*(ExternalApi.class);  
 MyService service = new MyService(mockApi);  
 service.fetchData();  
 *verify*(mockApi).getData();  
 }  
}

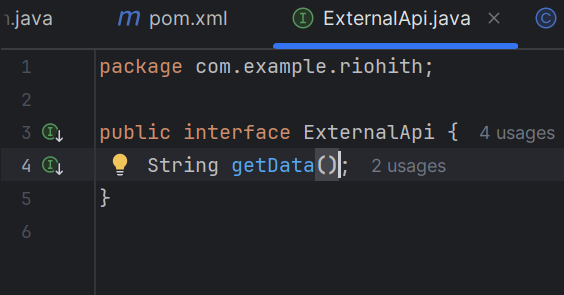
**Output:**

Process finished with exit code 0

**Output Image:**

****

****

****

**TDD USING JUNIT 5 AND MOCKITO**

**(JUnit 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:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

1. Create a new test class in your project.

**Code for above question:**

<dependencies>

<!-- JUnit 4 dependency -->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

**Java Class: Calculator.java**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

**Java Class: CalculatorTest.java**

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

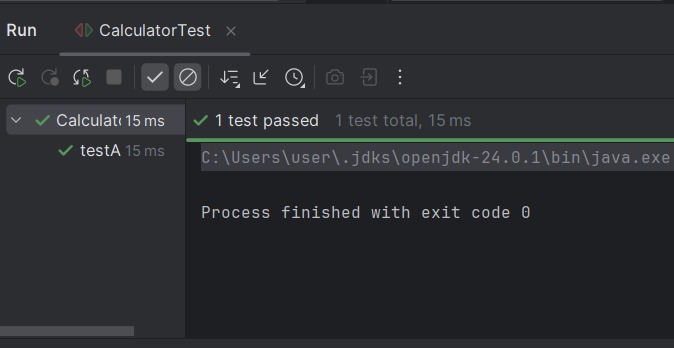
assertEquals(5, calc.add(2, 3));

}

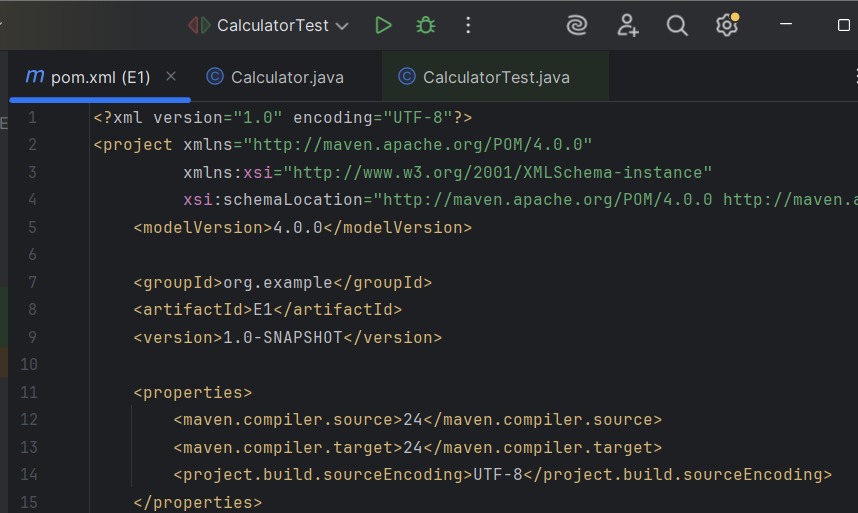
}

**Output:**

Process finished with exit code 0

**Output Image:**

****

****

**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 for Above question:**

**Java class: AssertionsTest.java**

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());

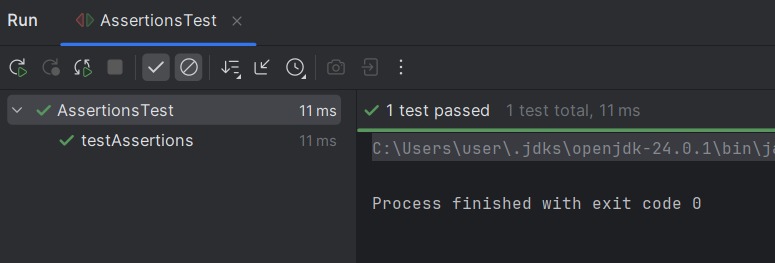
}

}

**Output:**

Process finished with exit code 0

**Output Image:**

****

****

**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.

**Code for the above question**:

**Java class: 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;

}

}

**Java Class: CalculatorTest.java**

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();

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

}

@After

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator instance cleared.");

}

@Test

public void testAdd() {

int a = 5, b = 3;

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

assertEquals(8, result);

}

@Test

public void testSubtract() {

int a = 10, b = 4;

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

assertEquals(6, result);

}

}

**Output:**

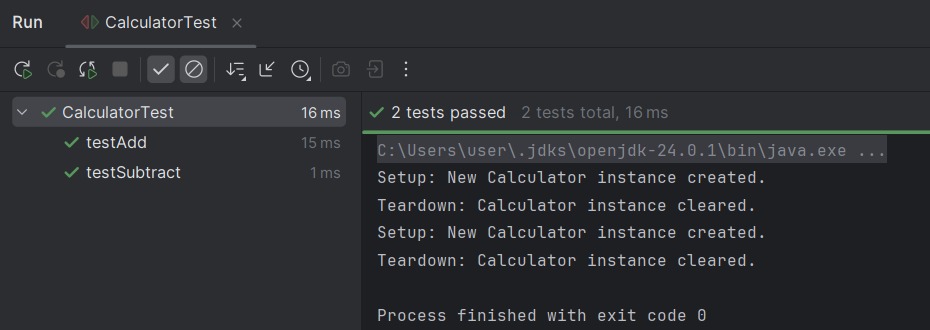
Setup: New Calculator instance created.

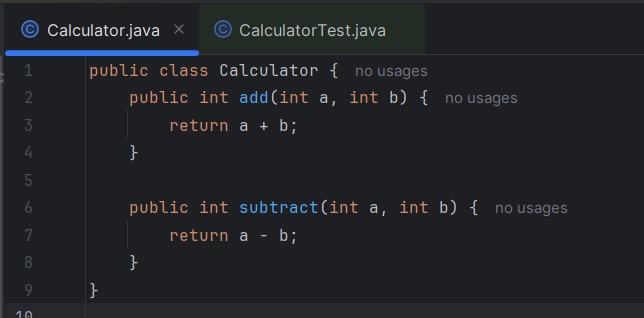
Teardown: Calculator instance cleared.

Setup: New Calculator instance created.

Teardown: Calculator instance cleared.

Tests passed: 2 of 2 tests

**Output Image:**



**PL/SQL PROGRAMMING**

**(PL\_SQL\_EXCERCISES)**

**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.

**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.

**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.

**Code for above question:**

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE customers';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE loan\_account';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE customers (

customer\_id NUMBER,

name VARCHAR2(50),

age NUMBER,

balance NUMBER,

interest\_rate NUMBER,

isvip CHAR(1)

);

CREATE TABLE loan\_account (

loan\_id NUMBER,

customer\_id NUMBER,

due\_date DATE

);

INSERT INTO customers VALUES (1, 'Alice', 65, 15000, 10.0, 'N');

INSERT INTO customers VALUES (2, 'Bob', 45, 9000, 11.5, 'N');

INSERT INTO customers VALUES (3, 'Charlie', 70, 12000, 12.0, 'N');

INSERT INTO customers VALUES (4, 'David', 50, 8000, 9.0, 'N');

INSERT INTO loan\_account VALUES (101, 1, SYSDATE + 10);

INSERT INTO loan\_account VALUES (102, 2, SYSDATE + 40);

INSERT INTO loan\_account VALUES (103, 3, SYSDATE + 5);

INSERT INTO loan\_account VALUES (104, 4, SYSDATE + 25);

COMMIT;

SET SERVEROUTPUT ON;

BEGIN

FOR c IN (SELECT \* FROM customers) LOOP

IF c.age > 60 THEN

UPDATE customers SET interest\_rate = interest\_rate - 1 WHERE customer\_id = c.customer\_id;

END IF;

IF c.balance > 10000 THEN

UPDATE customers SET isvip = 'Y' WHERE customer\_id = c.customer\_id;

END IF;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('--- Scenario 3: Loan Reminders ---');

FOR l IN (

SELECT l.loan\_id, c.name, l.due\_date

FROM loan\_account l

JOIN customers c ON c.customer\_id = l.customer\_id

WHERE l.due\_date BETWEEN SYSDATE AND SYSDATE + 30

) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: ' || l.name || ' has loan ID ' || l.loan\_id || ' due on ' || TO\_CHAR(l.due\_date, 'DD-MON-YYYY'));

END LOOP;

END;

/

SELECT \* FROM customers;

SELECT \* FROM loan\_account;

**Output:**

**Scenario 1: Interest rate discounted for customers above 60**

| customer\_id | name | age | interest\_rate (after 1% discount) |

| --------------- | ------- --| --- -| -------------------------------------- |

| 1 | Alice | 65 | 9.0 |

| 2 | Bob | 45 | 11.5 |

| 3 | Charlie | 70 | 11.0 |

| 4 | David | 50 | 9.0 |

**Scenario 2: VIP status set for customers with balance > 10,000**

| customer\_id | name | balance | is vip |

| --------------- | ---------| ---------- | ------ |

| 1 | Alice | 15000 | Y |

| 2 | Bob | 9000 | N |

| 3 | Charlie | 12000 | Y |

| 4 | David | 8000 | N |

**Scenario 3: Output from DBMS\_OUTPUT for loans due in next 30 days**

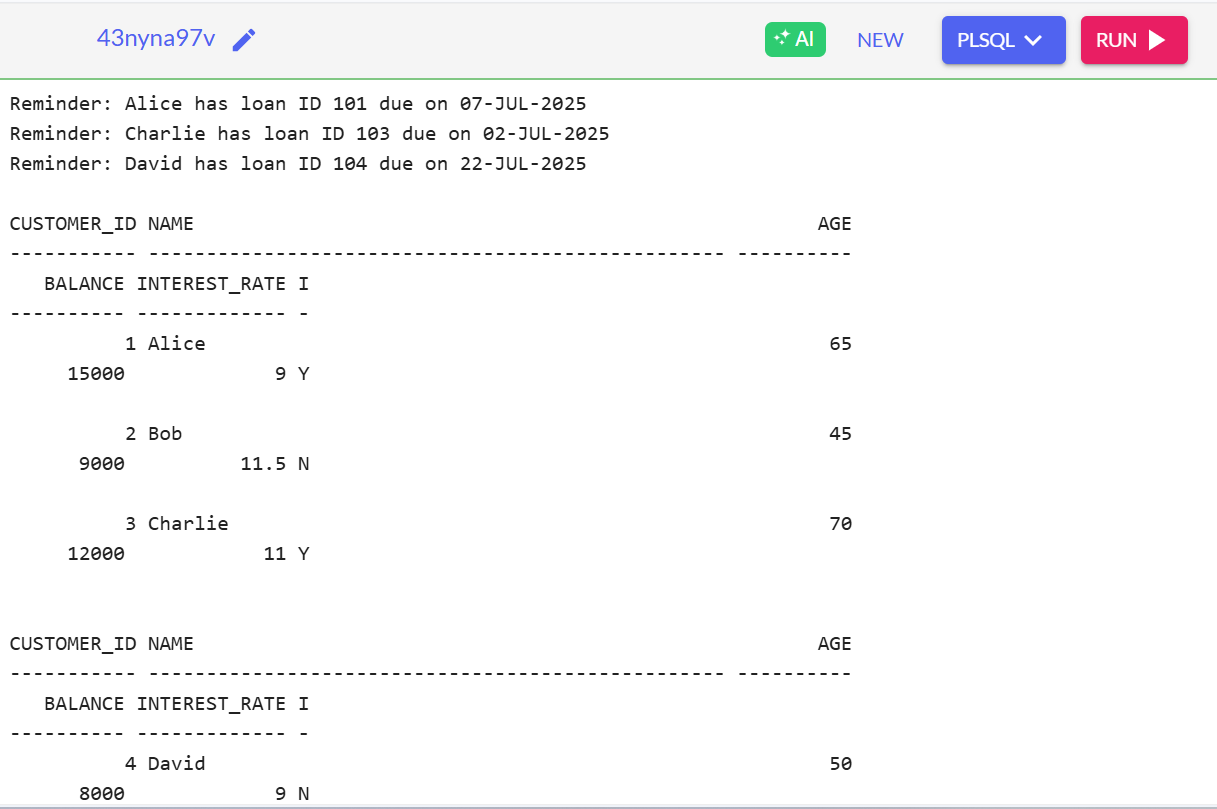
--- Scenario 3: Loan Reminders ---

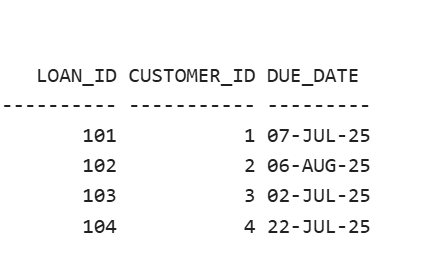
Reminder: Alice has loan ID 101 due on 06-JUL-2025

Reminder: Charlie has loan ID 103 due on 01-JUL-2025

Reminder: David has loan ID 104 due on 21-JUL-2025

**Output Image:**

****

****

**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.

**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.

**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.

**Code for above question:**

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE savings\_accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE employees';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

BEGIN

EXECUTE IMMEDIATE 'DROP TABLE bank\_accounts';

EXCEPTION

WHEN OTHERS THEN NULL;

END;

/

CREATE TABLE savings\_accounts (

account\_id NUMBER,

customer\_name VARCHAR2(50),

balance NUMBER

);

CREATE TABLE employees (

emp\_id NUMBER,

name VARCHAR2(50),

department\_id NUMBER,

salary NUMBER

);

CREATE TABLE bank\_accounts (

account\_id NUMBER,

customer\_name VARCHAR2(50),

balance NUMBER

);

INSERT INTO savings\_accounts VALUES (1, 'Alice', 10000);

INSERT INTO savings\_accounts VALUES (2, 'Bob', 5000);

INSERT INTO employees VALUES (101, 'John', 10, 30000);

INSERT INTO employees VALUES (102, 'Jane', 20, 35000);

INSERT INTO employees VALUES (103, 'Jack', 10, 28000);

INSERT INTO bank\_accounts VALUES (1, 'Alice', 8000);

INSERT INTO bank\_accounts VALUES (2, 'Bob', 4000);

COMMIT;

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS

BEGIN

FOR acc IN (SELECT \* FROM savings\_accounts) LOOP

UPDATE savings\_accounts

SET balance = balance + (balance \* 0.01)

WHERE account\_id = acc.account\_id;

END LOOP;

END;

/

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 department\_id = dept\_id;

END;

/

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 bank\_accounts WHERE account\_id = from\_account;

IF from\_balance < amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient balance in source account');

END IF;

UPDATE bank\_accounts SET balance = balance - amount WHERE account\_id = from\_account;

UPDATE bank\_accounts SET balance = balance + amount WHERE account\_id = to\_account;

END;

/

BEGIN

ProcessMonthlyInterest;

UpdateEmployeeBonus(10, 10);

TransferFunds(1, 2, 2000);

END;

/

SELECT \* FROM savings\_accounts;

SELECT \* FROM employees;

SELECT \* FROM bank\_accounts;

**Output:**

**Scenario 1:Savings Accounts (After 1% Interest)**

ACCOUNT\_ID CUSTOMER\_NAME BALANCE

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

1 Alice 10100

2 Bob 5050

**Scenario 2:Employees (Department 10, +10% Bonus)**

EMP\_ID NAME DEPARTMENT\_ID SALARY

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

101 John 10 33000

102 Jane 20 35000

103 Jack 10 30800

**Scenario 3:Bank Accounts (After Transfer 2000 from Alice to Bob)**

ACCOUNT\_ID CUSTOMER\_NAME BALANCE

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

1 Alice 6000

2 Bob 6000

**Output Image:**

