**CTS WEEK 2:**

**Module 3 - 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.

**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.

**SOLUTION:**

**TABLE CREATION:**

customers table creation:

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY, -- Unique ID for each customer

age NUMBER, -- Age of the customer

interest\_rate NUMBER, -- Current loan interest rate

balance NUMBER, -- Account balance

IsVIP BOOLEAN -- VIP status flag

);

loans table creation:

CREATE TABLE loans (

loan\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

due\_date DATE

);

**INSERTION OF DATA:**

Insert into customers table:

INSERT INTO customers VALUES (1, 65, 5.5, 12000, FALSE);

INSERT INTO customers VALUES (2, 45, 6.0, 8000, FALSE);

INSERT INTO customers VALUES (3, 70, 5.0, 15000, FALSE);

commit;

Insert into loans table:

INSERT INTO loans VALUES (101, 1, SYSDATE + 15); -- Loan due in 15 days

INSERT INTO loans VALUES (102, 2, SYSDATE + 35); -- Loan due in 35 days

INSERT INTO loans VALUES (103, 3, SYSDATE + 10); -- Loan due in 10 days

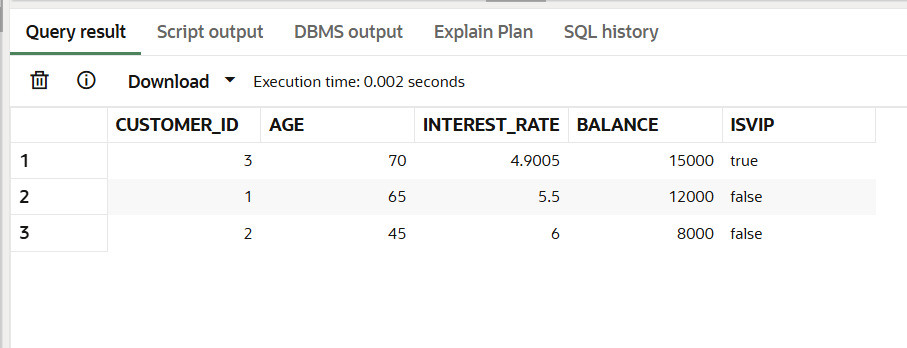
Commit;

SELECT \* FROM customers;

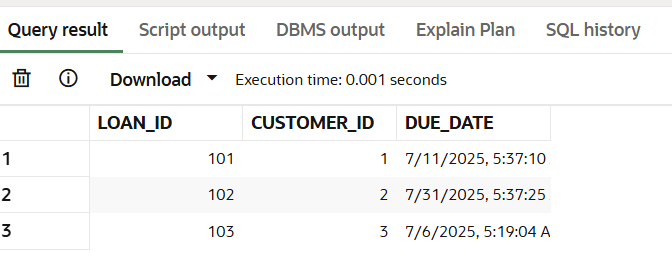
SELECT \* FROM loans;

**OUTPUT:**

Customers table:



Loans table:



**Scenario 1: Apply 1% Discount to Interest Rates**

BEGIN

FOR rec IN (SELECT customer\_id, interest\_rate FROM customers WHERE age > 60) LOOP

UPDATE customers

SET interest\_rate = rec.interest\_rate \* 0.99

WHERE customer\_id = rec.customer\_id;

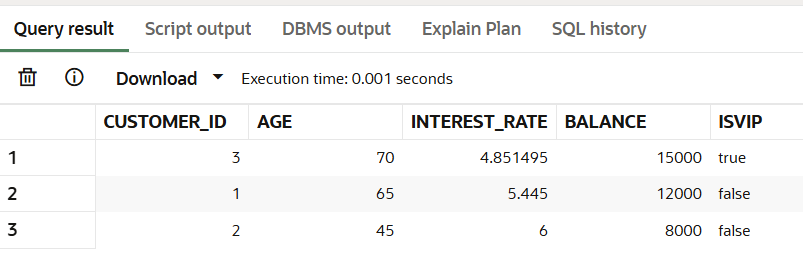
END LOOP;

COMMIT;

END;

/

**OUTPUT:**

****

**Scenario 2: Promote to VIP Based on Balance:**

BEGIN

FOR rec IN (SELECT customer\_id FROM customers WHERE balance > 10000) LOOP

UPDATE customers

SET IsVIP = TRUE

WHERE customer\_id = rec.customer\_id;

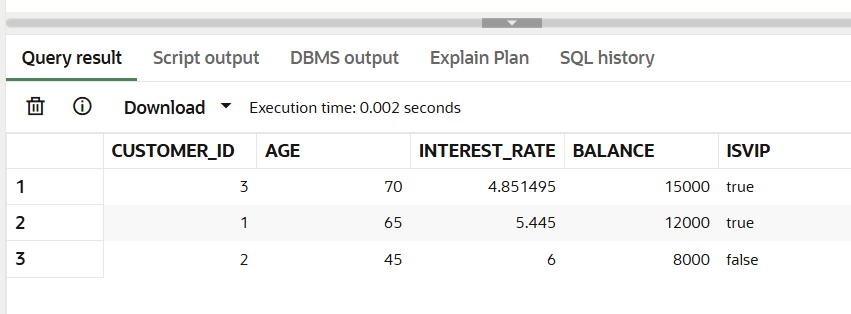
END LOOP;

COMMIT;

END;

/

**OUTPUT**:



**Scenario 3: Loan Due Reminders:**

BEGIN

FOR rec IN (SELECT loan\_id, customer\_id, due\_date FROM loans WHERE due\_date BETWEEN SYSDATE AND SYSDATE + 30) LOOP

DBMS\_OUTPUT.PUT\_LINE('Reminder: Customer ID ' || rec.customer\_id ||

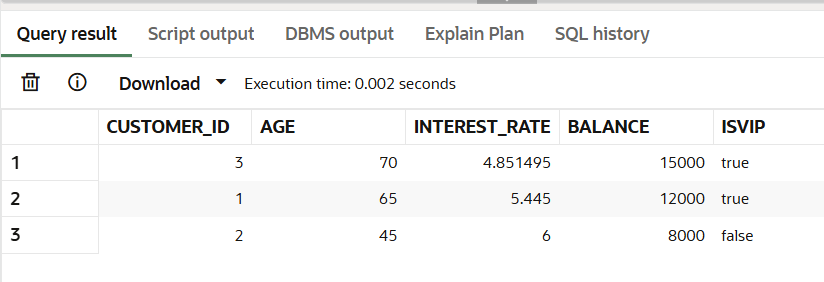
', your loan (ID: ' || rec.loan\_id || ') is due on ' || rec.due\_date || '.');

END LOOP;

END;

/

**OUTPUT:**

****

**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.

**SOLUTION:**

**Table creation:** savings\_accounts

CREATE TABLE savings\_accounts (

account\_id NUMBER PRIMARY KEY,

balance NUMBER,

interest\_rate NUMBER DEFAULT 1 -- Interest rate in percentage );

**Insertion of data:**

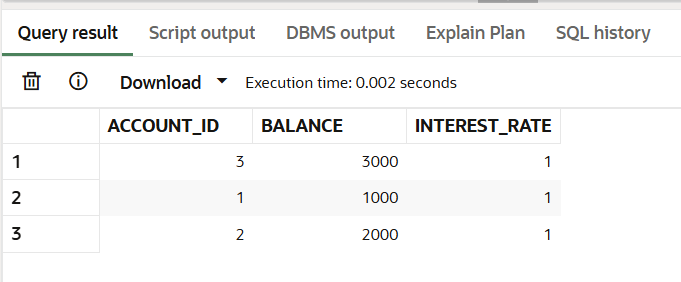
INSERT INTO savings\_accounts VALUES (1, 1000, 1); -- $1000 balance

INSERT INTO savings\_accounts VALUES (2, 2000, 1); -- $2000 balance

INSERT INTO savings\_accounts VALUES (3, 3000, 1); -- $3000 balance

COMMIT;

**Table output:**

****

**Create the Stored Procedure ProcessMonthlyInterest:**

CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest AS

BEGIN

FOR rec IN (SELECT account\_id, balance, interest\_rate FROM savings\_accounts) LOOP

UPDATE savings\_accounts

SET balance = balance + (balance \* interest\_rate / 100)

WHERE account\_id = rec.account\_id;

END LOOP;

COMMIT;

END;

/

**Execute the Procedure:**

BEGIN

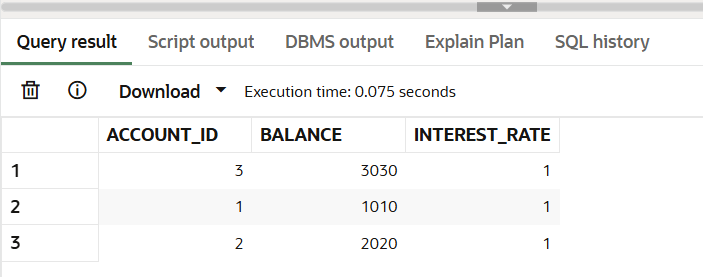
ProcessMonthlyInterest;

END;

/

**OUTPUT:**

SELECT \* FROM savings\_accounts;



**Scenario 2: Update Employee Bonus:**

**Table Creation:** employees

CREATE TABLE employees (

emp\_id NUMBER PRIMARY KEY,

department\_id NUMBER,

salary NUMBER,

performance\_rating NUMBER

);

**Insertion of data:**

INSERT INTO employees VALUES (101, 10, 50000, 5);

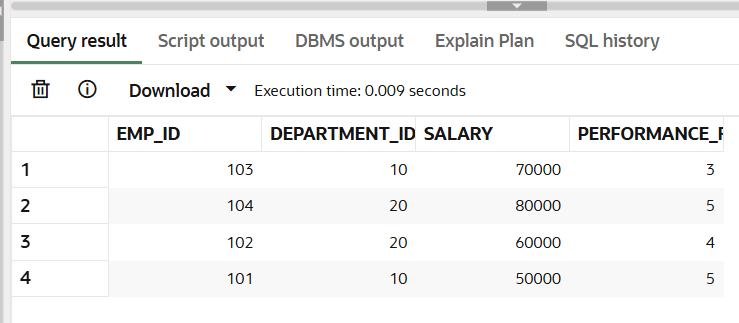
INSERT INTO employees VALUES (102, 20, 60000, 4);

INSERT INTO employees VALUES (103, 10, 70000, 3);

INSERT INTO employees VALUES (104, 20, 80000, 5);

COMMIT;

**OUTPUT:**

****

**Create the Stored Procedure UpdateEmployeeBonus:**

CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (

dept\_id IN NUMBER,

bonus\_percentage IN NUMBER

) AS

BEGIN

UPDATE employees

SET salary = salary + (salary \* bonus\_percentage / 100)

WHERE department\_id = dept\_id;

COMMIT;

END;

/

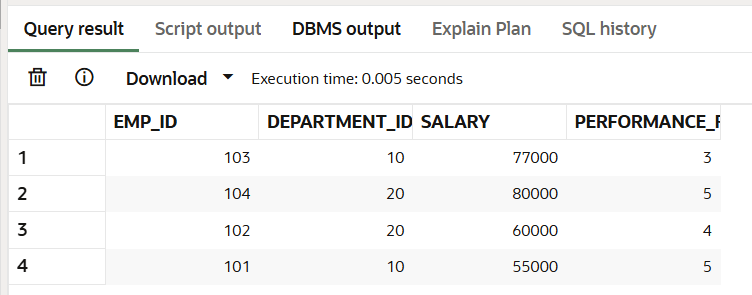
**Execute the Procedure:**

BEGIN

UpdateEmployeeBonus(10, 10);

END;

/

**OUTPUT:  
**

**Scenario 3: Transfer Funds:**

**Table creation: accounts**

CREATE TABLE accounts (

account\_id NUMBER PRIMARY KEY,

customer\_id NUMBER,

balance NUMBER

);

**Insertion of data:**

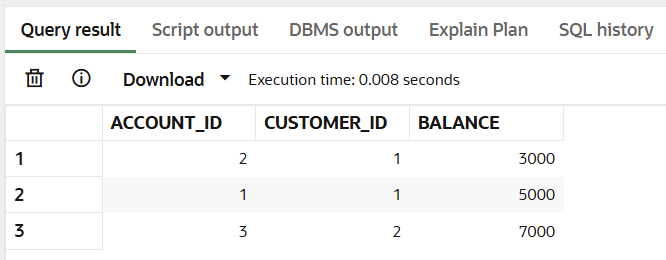
INSERT INTO accounts VALUES (1, 1, 5000);

INSERT INTO accounts VALUES (2, 1, 3000);

INSERT INTO accounts VALUES (3, 2, 7000); COMMIT;

**OUTPUT:**

Select \* from accounts;



**Create the Stored Procedure TransferFunds**

CREATE OR REPLACE PROCEDURE TransferFunds (

from\_account\_id IN NUMBER,

to\_account\_id IN NUMBER,

transfer\_amount IN NUMBER

) AS

insufficient\_funds EXCEPTION;

v\_balance NUMBER;

BEGIN

-- Check if the source account has sufficient balance

SELECT balance INTO v\_balance FROM accounts WHERE account\_id = from\_account\_id;

IF v\_balance < transfer\_amount THEN

RAISE insufficient\_funds;

END IF;

-- Deduct from the source account

UPDATE accounts

SET balance = balance - transfer\_amount

WHERE account\_id = from\_account\_id;

-- Add to the target account

UPDATE accounts

SET balance = balance + transfer\_amount

WHERE account\_id = to\_account\_id;

COMMIT;

EXCEPTION

WHEN insufficient\_funds THEN

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

END;

/

**Execute the Procedure:**

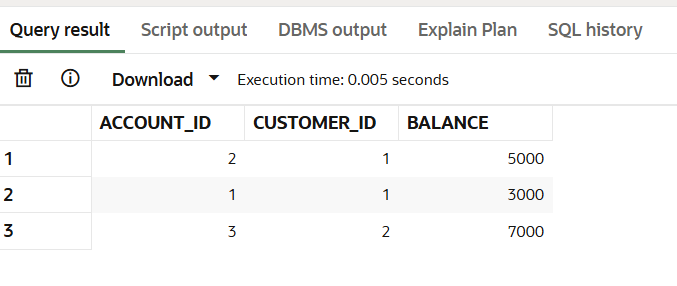
BEGIN

TransferFunds(1, 2, 2000); -- Transfer $2000 from account 1 to account 2

END;

/

**OUTPUT:**SELECT \* FROM accounts;



**Module 4 – Test driven development and Logging framework**

**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>

3. Create a new test class in your project.

**SOLUTION:**

**Calculator.java**

public class Calculator {

public int add(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 testAdd() {**

**Calculator calculator = new Calculator();**

**int result = calculator.add(2, 3);**

**assertEquals(5, result);**

**}**

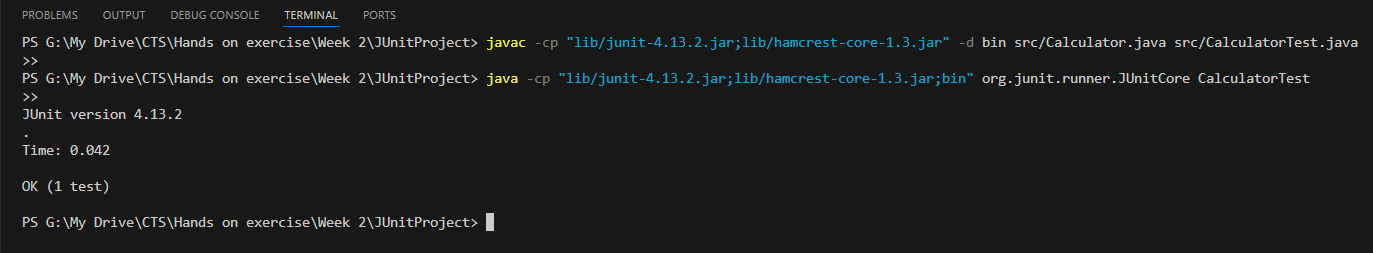
**}**

**Compilation:**

javac -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar" -d bin src/Calculator.java src/CalculatorTest.java

**Run the Test:**

java -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar;bin" org.junit.runner.JUnitCore CalculatorTest

**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());

}

}

**SOLUTION:**

**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());**

**}**

**}**

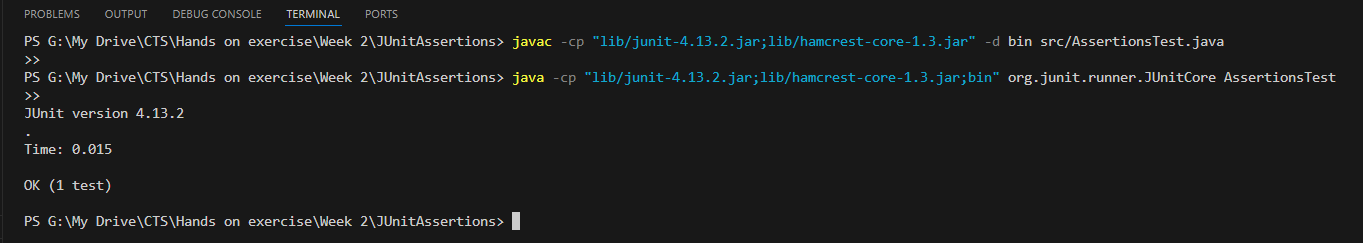
**Compilation:**

javac -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar" -d bin src/AssertionsTest.java

**Run the Test:**

java -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar;bin" org.junit.runner.JUnitCore AssertionsTest

**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.

**SOLUTION:**

**CalculatorTest.java**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

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

}

@After

public void tearDown() {

calculator = null;

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

}

@Test

public void testAddition() {

int a = 5;

int b = 3;

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

assertEquals(8, result);

}

@Test

public void testSubtraction() {

int a = 10;

int b = 4;

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

assertEquals(6, result);

}

}

class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

**Compilation:**

javac -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar" -d bin src/CalculatorTest.java

**Run the Test:**

java -cp "lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar;bin" org.junit.runner.JUnitCore CalculatorTest

**Output:**

****

**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);

}

}

**SOLUTION:**

**ExternalApi.java**

public interface ExternalApi {

String getData();

}

**MyService.java**

**public class MyService {**

**private ExternalApi externalApi;**

**public MyService(ExternalApi externalApi) {**

**this.externalApi = externalApi;**

**}**

**public String fetchData() {**

**return externalApi.getData();**

**}}**

**MyServiceTest.java**

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

**import org.junit.Test;**

**public class MyServiceTest {**

**@Test**

**public void testExternalApi() {**

**ExternalApi mockApi = org.mockito.Mockito.mock(ExternalApi.class);**

**org.mockito.Mockito.when(mockApi.getData()).thenReturn("Mock Data");**

**MyService service = new MyService(mockApi);**

**String result = service.fetchData();**

**assertEquals("Mock Data", result);**

**}**

**}**

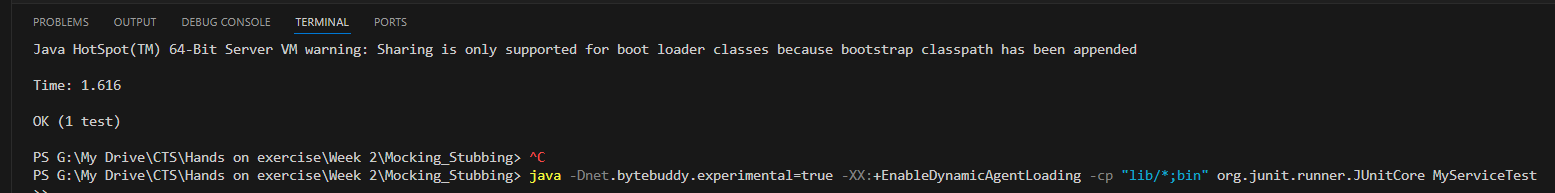
**Compilation:**

javac -cp "lib/\*" -d bin src/\*.java

**Run the Test:**

java -cp "lib/\*;bin" org.junit.runner.JUnitCore MyServiceTest

**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.

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();

}

}

**SOLUTION:**

**ExternalApi.java**

**public interface ExternalApi {**

**String getData();**

**}**

**MyService.java**

**public class MyService {**

**private ExternalApi externalApi;**

**public MyService(ExternalApi externalApi) {**

**this.externalApi = externalApi;**

**}**

**public String fetchData() {**

**return externalApi.getData();**

**}**

**}**

**MyServiceTest.java**

**import static org.mockito.Mockito.\*;**

**import org.junit.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();**

**}**

**}**

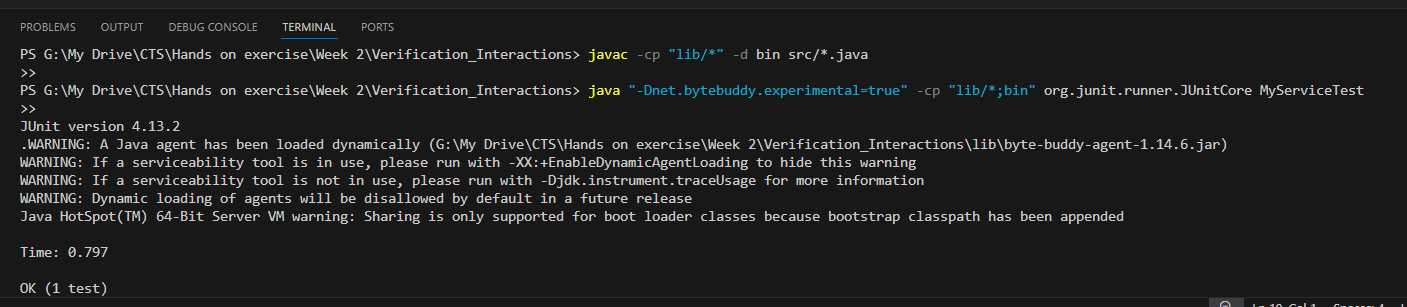
**Compilation:**

javac -cp "lib/\*" -d bin src/\*.java

**Run the Test:**

java "-Dnet.bytebuddy.experimental=true" -cp "lib/\*;bin" org.junit.runner.JUnitCore MyServiceTest

**OUTPUT:**

****

**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>

2. 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");

} }

**SOLUTION:**

**LoggingExample.java**

**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");**

**}**

**}**

**Launch.json**

{

"version": "0.2.0",

"configurations": [

{

"type": "java",

"name": "Debug (LoggingExample)",

"request": "launch",

"mainClass": "LoggingExample",

"classpath": [

"${workspaceFolder}/src",

"${workspaceFolder}/lib/slf4j-api-1.7.30.jar",

"${workspaceFolder}/lib/logback-classic-1.2.3.jar",

"${workspaceFolder}/lib/logback-core-1.2.3.jar"

]

}

]

}

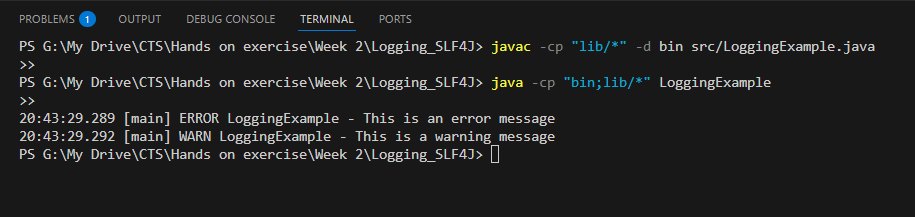
**Compilation:**

javac -cp "lib/\*" -d bin src/LoggingExample.java

**Test to Run:**

java -cp "bin;lib/\*" LoggingExample

**OUTPUT:**

****