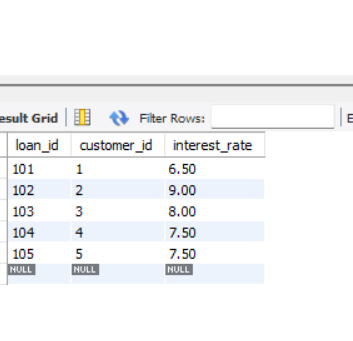
**Exercise 1: Control Structures**

Scenario:1  
DELIMITER //

CREATE PROCEDURE apply\_interest\_discount() BEGIN DECLARE done INT DEFAULT FALSE; DECLARE cust\_id INT;

DECLARE cur CURSOR FOR   
 SELECT DISTINCT c.customer\_id  
 FROM customers c  
 JOIN loans l ON c.customer\_id = l.customer\_id  
 WHERE c.age > 60;  
  
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  
  
OPEN cur;  
  
fetch\_loop: LOOP  
 FETCH cur INTO cust\_id;  
 IF done THEN  
 LEAVE fetch\_loop;  
 END IF;  
  
 UPDATE loans  
 SET interest\_rate = interest\_rate - 1  
 WHERE customer\_id = cust\_id;  
  
END LOOP;  
  
CLOSE cur;

END //

DELIMITER ;  
  
OUTPUT:  


Scenario:2  
  
DELIMITER //

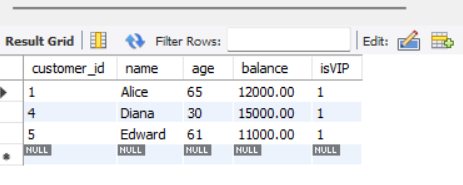
CREATE PROCEDURE promote\_to\_vip()

BEGIN

UPDATE customers

SET isVIP = TRUE

WHERE balance > 10000; END //

DELIMITER ;  
  
Output:  


Scenario:3

DELIMITER //

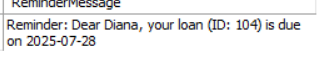
CREATE PROCEDURE send\_due\_reminders() BEGIN DECLARE done INT DEFAULT FALSE; DECLARE cust\_name VARCHAR(100); DECLARE due DATE; DECLARE loan\_id INT;

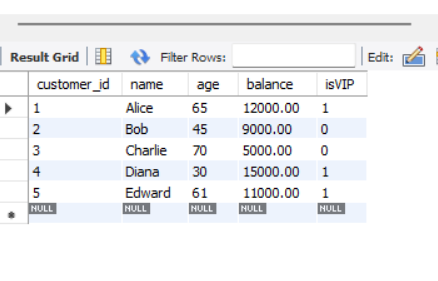
DECLARE cur CURSOR FOR   
 SELECT c.name, l.due\_date, l.loan\_id  
 FROM loans l  
 JOIN customers c ON l.customer\_id = c.customer\_id  
 WHERE l.due\_date BETWEEN CURDATE() AND DATE\_ADD(CURDATE(), INTERVAL 30 DAY);  
  
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  
  
OPEN cur;  
  
fetch\_loop: LOOP  
 FETCH cur INTO cust\_name, due, loan\_id;  
 IF done THEN  
 LEAVE fetch\_loop;  
 END IF;  
  
 SELECT CONCAT('Reminder: Dear ', cust\_name, ', your loan (ID: ', loan\_id, ') is due on ', due) AS ReminderMessage;  
  
END LOOP;  
  
CLOSE cur;

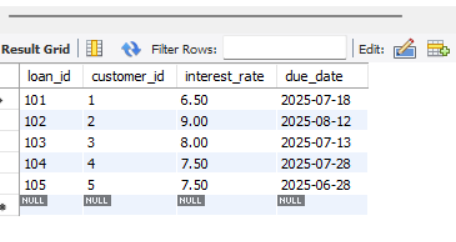
END //

DELIMITER ;

OUTPUT:  
CALL send\_due\_reminders();



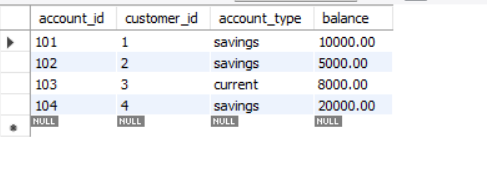
TABLES:  
  




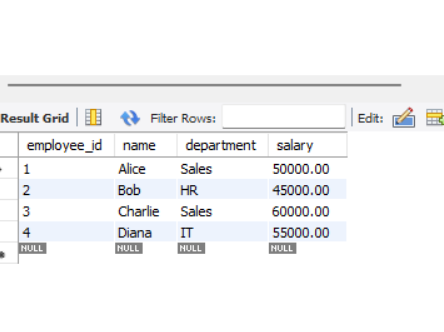
**Exercise 3: Stored Procedures**

Tables:

SELECT \* FROM accounts;



SELECT \* FROM employees;



**Scenario 1:**

DELIMITER //

CREATE PROCEDURE ProcessMonthlyInterest()

BEGIN

UPDATE accounts

SET balance = balance + (balance \* 0.01)

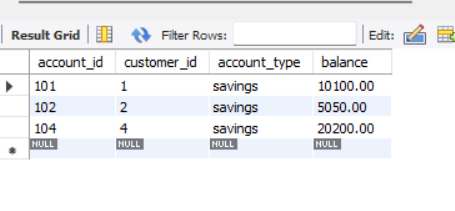
WHERE account\_type = 'savings';

END //

DELIMITER ;

OUTPUT:

SELECT \* FROM accounts WHERE account\_type = 'savings';



**Scenario 2:**  
**DELIMITER //**

**CREATE PROCEDURE UpdateEmployeeBonus(**

**IN dept\_name VARCHAR(50),**

**IN bonus\_pct DECIMAL(5,2)**

**)**

**BEGIN**

**UPDATE employees**

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

**WHERE department = dept\_name;**

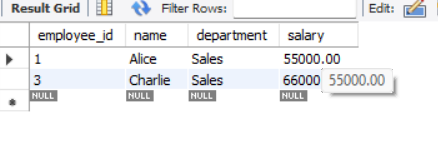
**END //**

**DELIMITER ;**

**OUTPUT:**

**CALL UpdateEmployeeBonus('Sales', 10);**

**SELECT \* FROM employees WHERE department = 'Sales';**



Scenario 3:

DELIMITER //

CREATE PROCEDURE TransferFunds( IN source\_id INT, IN dest\_id INT, IN amount DECIMAL(10,2) ) BEGIN DECLARE source\_balance DECIMAL(10,2);

-- Get the balance of source account  
SELECT balance INTO source\_balance  
FROM accounts  
WHERE account\_id = source\_id;  
  
-- Check if balance is enough  
IF source\_balance >= amount THEN  
 -- Deduct from source  
 UPDATE accounts  
 SET balance = balance - amount  
 WHERE account\_id = source\_id;  
  
 -- Add to destination  
 UPDATE accounts  
 SET balance = balance + amount  
 WHERE account\_id = dest\_id;  
ELSE  
 SIGNAL SQLSTATE '45000'  
 SET MESSAGE\_TEXT = 'Insufficient funds in source account';  
END IF;

END //

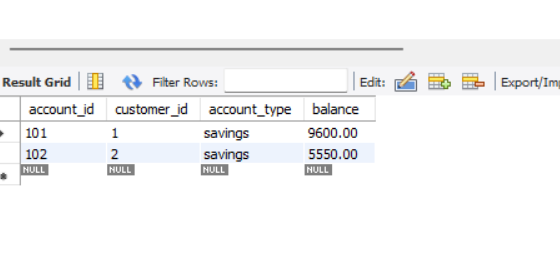
DELIMITER ;

OUTPUT:

CALL TransferFunds(101, 102, 500);

SELECT \* FROM accounts

WHERE account\_id IN (101, 102);



Exercise 1: Setting Up JUnit  
  
pom.xml

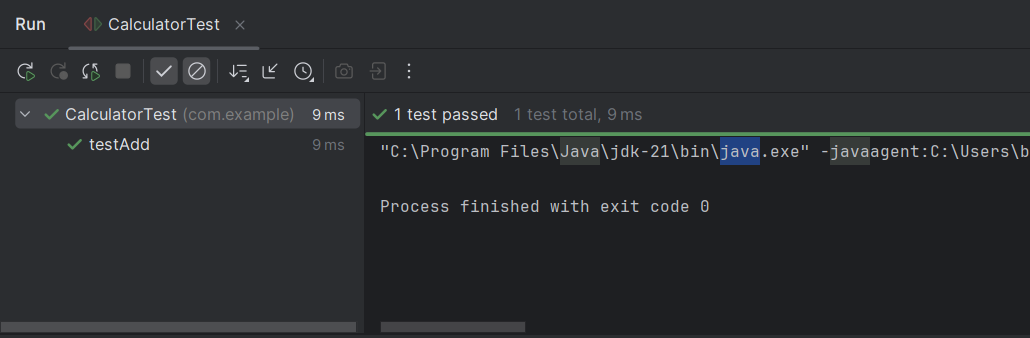
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 <http://maven.apache.org/xsd/maven-4.0.0.xsd>">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>org.example</groupId>  
 <artifactId>unittest</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>21</maven.compiler.source>  
 <maven.compiler.target>21</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 </properties>  
 <dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
</project>

src/main/java/ Calculator

package com.example;  
  
public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
}

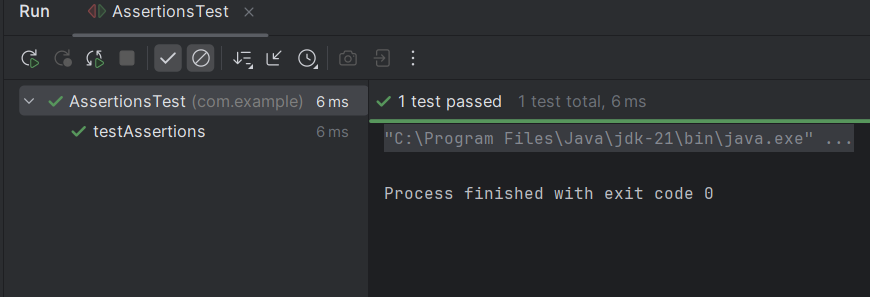
src/test/java/ CalculatorTest:

package com.example;  
  
import org.junit.Test;  
import static org.junit.Assert.*assertEquals*;  
  
public class CalculatorTest {  
  
 @Test  
 public void testAdd() {  
 Calculator c = new Calculator();  
 int result = c.add(2, 3);  
 *assertEquals*(5, result);  
 }  
}

OUTPUT:  
  


Exercise 3: Assertions in JUnit  
  
src/test/java/ AssertionsTest:

package com.example;  
  
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());  
 }  
}

OUTPUT:  


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

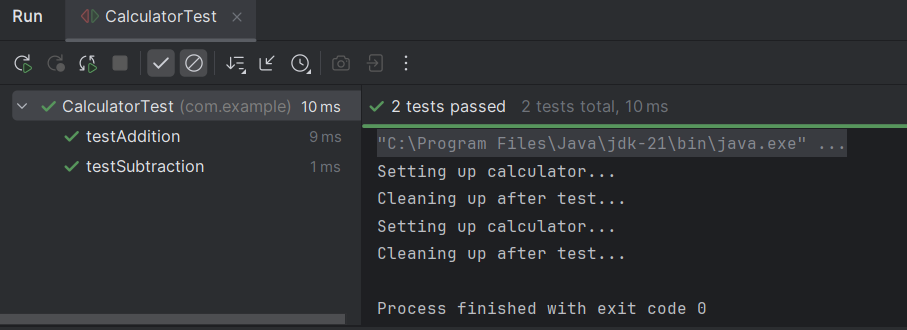
src/main/java/ Calculator

package com.example;  
  
public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int subtract(int a, int b) {  
 return a - b;  
 }  
}

src/test/java/ CalculatorTest:

package com.example;  
  
import org.junit.Before;  
import org.junit.After;  
import org.junit.Test;  
import static org.junit.Assert.\*;  
  
public class CalculatorTest {  
  
 private Calculator calculator;  
  
 // Setup method: runs before every test  
 @Before  
 public void setUp() {  
 System.*out*.println("Setting up calculator...");  
 calculator = new Calculator(); // Arrange: common object  
 }  
  
 // Teardown method: runs after every test  
 @After  
 public void tearDown() {  
 System.*out*.println("Cleaning up after test...");  
 calculator = null;  
 }  
  
 // Test 1: Addition  
 @Test  
 public void testAddition() {  
 // Act  
 int result = calculator.add(10, 5);  
  
 // Assert  
 *assertEquals*(15, result);  
 }  
  
 // Test 2: Subtraction  
 @Test  
 public void testSubtraction() {  
 // Act  
 int result = calculator.subtract(10, 4);  
  
 // Assert  
 *assertEquals*(6, result);  
 }  
}

OUTPUT:



Exercise 1: Mocking and Stubbing:

src/main/java/ ExternalApi

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

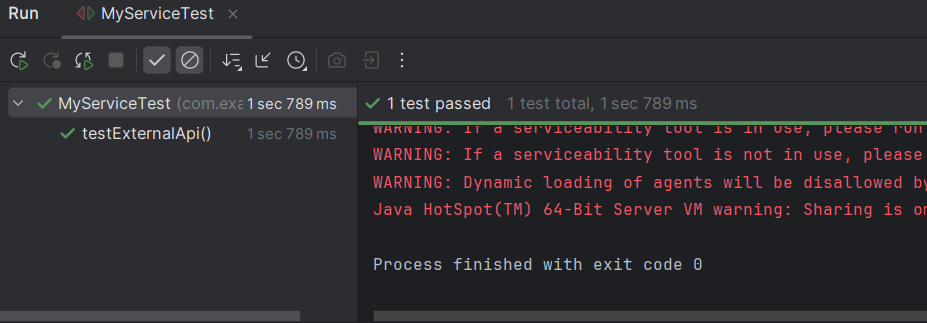
src/main/java/ MyService:

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

src/test/java/ MyServiceTest:

package com.example;  
  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
import static org.mockito.Mockito.\*;  
  
public class MyServiceTest {  
  
 @Test  
 public void testExternalApi() {  
 // Arrange: Create a mock of ExternalApi  
 ExternalApi mockApi = *mock*(ExternalApi.class);  
  
 // Stub the getData method  
 *when*(mockApi.getData()).thenReturn("Mock Data");  
  
 // Inject mock into MyService  
 MyService service = new MyService(mockApi);  
  
 // Act: Call method under test  
 String result = service.fetchData();  
  
 // Assert: Check the result  
 *assertEquals*("Mock Data", result);  
 }  
}

OUTPUT:



Exercise 2: Verifying Interactions

src/main/java/ ExternalApi:

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

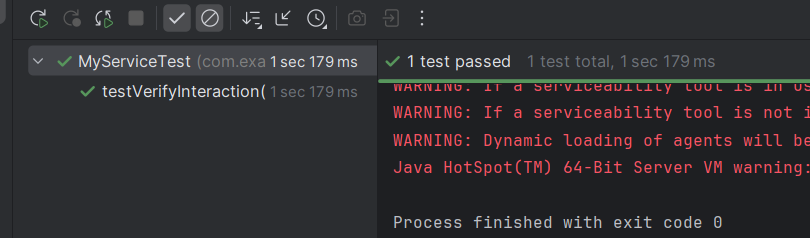
src/main/java/ MyService:

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

src/test/java/ MyServiceTest:

package com.example;  
  
import org.junit.jupiter.api.Test;  
import static org.mockito.Mockito.\*;  
  
public class MyServiceTest {  
  
 @Test  
 public void testVerifyInteraction() {  
 // Create a mock for the ExternalApi  
 ExternalApi mockApi = *mock*(ExternalApi.class);  
  
 // Create MyService using the mocked API  
 MyService service = new MyService(mockApi);  
  
 // Call the method we want to test  
 service.fetchData();  
  
 // Verify that mockApi.getData() was called exactly once  
 *verify*(mockApi).getData();  
 }  
}

OUTPUT:



Exercise 1: Logging Error Messages and Warning Levels

src/main/java/LoggingExample.java

package com.example;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
public class LoggingExample {  
 // Create a logger for this class  
 private static final Logger *logger* = LoggerFactory.*getLogger*(LoggingExample.class);  
  
 public static void main(String[] args) {  
 // Log an error message  
 *logger*.error("This is an error message");  
  
 // Log a warning message  
 *logger*.warn("This is a warning message");  
 }  
}

OUTPUT:

