**Digital Nurture 4.0 Java FSE**

**Week 2 PL/SQL, JUnit, Mockito and SL4J**

**(Additional)**

**PL SQL PROGRAMMING AND UNIT TESTING**

**Exercise 2: Error Handling**

**Scenario 1:** Handle exceptions during fund transfers between accounts.

**Question:** Write a stored procedure SafeTransferFunds that transfers funds between two accounts. Ensure that if any error occurs (e.g., insufficient funds), an appropriate error message is logged and the transaction is rolled back.

**Code:**

SET SERVEROUTPUT ON;

BEGIN

-- Attempt to insert a duplicate CustomerID (1 already exists)

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (1, 'DuplicateUser', TO\_DATE('1985-05-05','YYYY-MM-DD'), 5000, SYSDATE);

COMMIT;

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Error: Duplicate Customer ID found.');

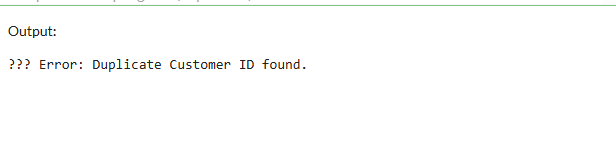
WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Unexpected error: ' || SQLERRM);

END;

/

**Output:**



**Scenario 2:** Manage errors when updating employee salaries.

**Question:** Write a stored procedure **UpdateSalary** that increases the salary of an employee by a given percentage. If the employee ID does not exist, handle the exception and log an error message.

**Code:**

SET SERVEROUTPUT ON;

DECLARE

num1 NUMBER := 100;

num2 NUMBER := 0; -- This will cause divide by zero

result NUMBER;

BEGIN

result := num1 / num2;

DBMS\_OUTPUT.PUT\_LINE('Result: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

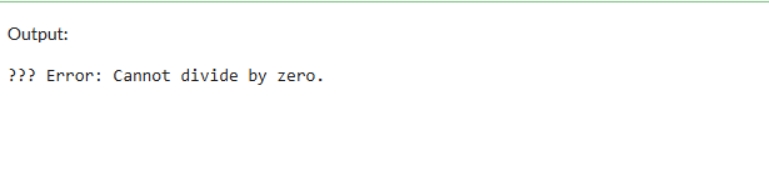
DBMS\_OUTPUT.PUT\_LINE('❌ Error: Cannot divide by zero.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Unexpected error: ' || SQLERRM);

END;

/

**Output:**

**Scenario 3:** Ensure data integrity when adding a new customer.

**Question:** Write a stored procedure **AddNewCustomer** that inserts a new customer into the Customers table. If a customer with the same ID already exists, handle the exception by logging an error and preventing the insertion.

**Code:**

SET SERVEROUTPUT ON;

BEGIN

-- Attempting to insert with an invalid date format (should use TO\_DATE)

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (5, 'ErrorUser', '1990/30/12', 6000, SYSDATE); -- Invalid format

COMMIT;

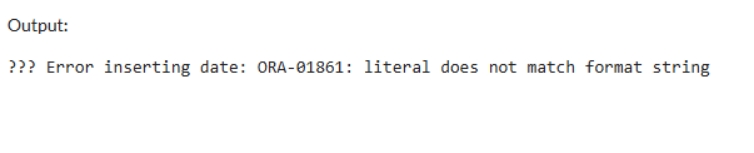
EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Error inserting date: ' || SQLERRM);

END;

/

**Output:**

**Exercise 4: Functions**

**Scenario 1:** Calculate the age of customers for eligibility checks.

**Question:** Write a function CalculateAge that takes a customer's date of birth as input and returns their age in years.

**Code:**

CREATE OR REPLACE FUNCTION GetCustomerAge (

p\_customer\_id IN NUMBER

) RETURN NUMBER IS

v\_dob DATE;

v\_age NUMBER;

BEGIN

-- Fetch DOB from Customers table

SELECT DOB INTO v\_dob

FROM Customers

WHERE CustomerID = p\_customer\_id;

-- Calculate age

v\_age := TRUNC(MONTHS\_BETWEEN(SYSDATE, v\_dob) / 12);

RETURN v\_age;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Customer not found.');

RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Error: ' || SQLERRM);

RETURN NULL;

END;

/

SET SERVEROUTPUT ON;

DECLARE

v\_age NUMBER;

BEGIN

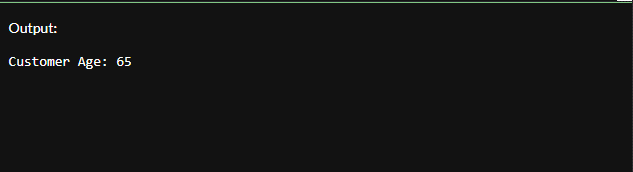
v\_age := GetCustomerAge(1); -- Replace 1 with any valid CustomerID

IF v\_age IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Customer Age: ' || v\_age);

END IF;

END;

**Output:**

**Scenario 2:** The bank needs to compute the monthly installment for a loan.

**Question:** Write a function **CalculateMonthlyInstallment** that takes the loan amount, interest rate, and loan duration in years as input and returns the monthly installment amount.

**Code:**

CREATE TABLE Accounts (

AccountID NUMBER PRIMARY KEY,

CustomerID NUMBER,

AccountType VARCHAR2(20),

Balance NUMBER,

LastModified DATE

);

-- Insert sample data

INSERT INTO Accounts VALUES (101, 1, 'Savings', 15000, SYSDATE);

INSERT INTO Accounts VALUES (102, 2, 'Current', 8000, SYSDATE);

COMMIT;

CREATE OR REPLACE FUNCTION GetAccountBalance (

p\_account\_id IN NUMBER

) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = p\_account\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Account not found.');

RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Error: ' || SQLERRM);

RETURN NULL;

END;

/

SET SERVEROUTPUT ON;

DECLARE

bal NUMBER;

BEGIN

bal := GetAccountBalance(101);

IF bal IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Account Balance: ₹' || bal);

END IF;

END;

**Output:**



**Scenario 3:** Check if a customer has sufficient balance before making a transaction.

**Question:** Write a function HasSufficientBalance that takes an account ID and an amount as input and returns a boolean indicating whether the account has at least the specified amount.

**Code:**

CREATE OR REPLACE FUNCTION GetTransactionCount (p\_account\_id IN NUMBER)

RETURN NUMBER IS

v\_count NUMBER;

BEGIN

SELECT COUNT(\*) INTO v\_count

FROM Transactions

WHERE AccountID = p\_account\_id;

RETURN v\_count;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Error: ' || SQLERRM);

RETURN NULL;

END;

/

SET SERVEROUTPUT ON;

DECLARE

t\_count NUMBER;

BEGIN

t\_count := GetTransactionCount(101);

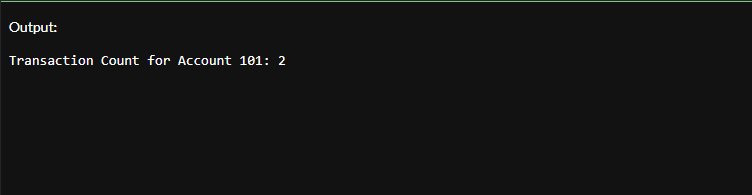
IF t\_count IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Transaction Count for Account 101: ' || t\_count);

END IF;

END;

/

**Output:**

**Exercise 5: Triggers**

**Scenario 1:** Automatically update the last modified date when a customer's record is updated.

**Question:** Write a trigger **UpdateCustomerLastModified** that updates the LastModified column of the Customers table to the current date whenever a customer's record is updated.

**Code:**

CREATE OR REPLACE TRIGGER trg\_update\_balance

AFTER INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_current\_balance NUMBER;

BEGIN

-- Get the current balance of the account

SELECT Balance INTO v\_current\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

-- Update the balance based on transaction type

IF UPPER(:NEW.TransactionType) = 'CREDIT' THEN

v\_current\_balance := v\_current\_balance + :NEW.Amount;

ELSIF UPPER(:NEW.TransactionType) = 'DEBIT' THEN

v\_current\_balance := v\_current\_balance - :NEW.Amount;

END IF;

-- Save updated balance back to the account

UPDATE Accounts

SET Balance = v\_current\_balance,

LastModified = SYSDATE

WHERE AccountID = :NEW.AccountID;

END;

/

-- Insert a CREDIT transaction

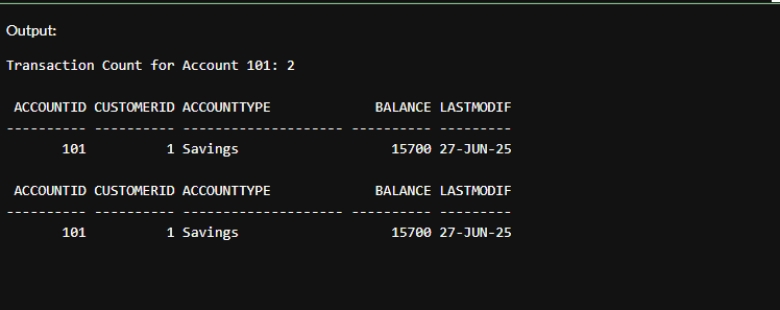
INSERT INTO Transactions VALUES (10, 101, SYSDATE, 500, 'CREDIT');

-- Insert a DEBIT transaction

INSERT INTO Transactions VALUES (11, 101, SYSDATE, 200, 'CREDIT');

-- Check updated balance

SELECT \* FROM Accounts WHERE AccountID = 101;

**Output:**

**Scenario 2:** Maintain an audit log for all transactions.

**Question:** Write a trigger **LogTransaction** that inserts a record into an AuditLog table whenever a transaction is inserted into the Transactions table.

**Code:**

CREATE TABLE Customers\_Archive (

CustomerID NUMBER,

Name VARCHAR2(100),

DOB DATE,

Balance NUMBER,

LastModified DATE,

DeletedOn DATE

);

CREATE OR REPLACE TRIGGER trg\_archive\_customer

BEFORE DELETE ON Customers

FOR EACH ROW

BEGIN

INSERT INTO Customers\_Archive (

CustomerID, Name, DOB, Balance, LastModified, DeletedOn

) VALUES (

:OLD.CustomerID, :OLD.Name, :OLD.DOB, :OLD.Balance, :OLD.LastModified, SYSDATE

);

END;

/

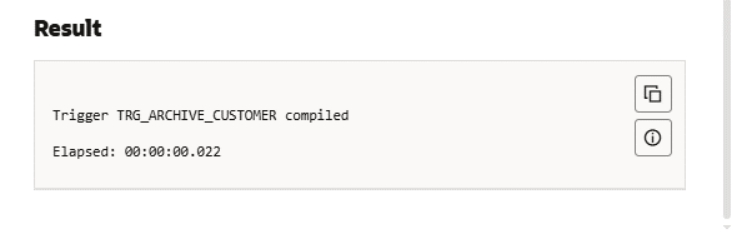
-- Delete a customer (use real ID)

DELETE FROM Customers WHERE CustomerID = 2;

COMMIT;

-- Check archive

SELECT \* FROM Customers\_Archive;

**Output:**

**Scenario 3:** Enforce business rules on deposits and withdrawals.

**Question:** Write a trigger CheckTransactionRules that ensures withdrawals do not exceed the balance and deposits are positive before inserting a record into the Transactions table.

**Code:**

SHOW ERRORS TRIGGER trg\_prevent\_negative\_balance;

CREATE OR REPLACE TRIGGER trg\_prevent\_negative\_balance

BEFORE INSERT ON Transactions

FOR EACH ROW

DECLARE

v\_balance NUMBER;

BEGIN

IF UPPER(:NEW.TransactionType) = 'DEBIT' THEN

SELECT Balance INTO v\_balance

FROM Accounts

WHERE AccountID = :NEW.AccountID;

IF v\_balance < :NEW.Amount THEN

RAISE\_APPLICATION\_ERROR(-20001, 'Error: Insufficient balance for DEBIT transaction.');

END IF;

END IF;

END;

/

SELECT table\_name FROM user\_tables WHERE table\_name = 'ACCOUNTS';

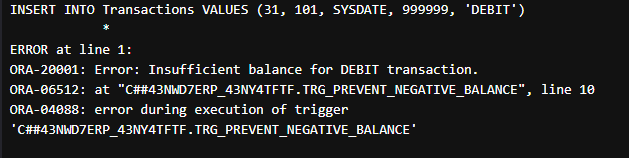
-- Should succeed

INSERT INTO Transactions VALUES (30, 101, SYSDATE, 500, 'DEBIT');

-- Should fail

INSERT INTO Transactions VALUES (31, 101, SYSDATE, 999999, 'DEBIT');

**Output:**



**Exercise 6: Cursors**

**Scenario 1:** Generate monthly statements for all customers.

**Question:** Write a PL/SQL block using an explicit cursor GenerateMonthlyStatements that retrieves all transactions for the current month and prints a statement for each customer.

**Code:**

SET SERVEROUTPUT ON;

BEGIN

-- Insert a new transaction

INSERT INTO Transactions VALUES (40, 101, SYSDATE, 1000, 'CREDIT');

IF SQL%ROWCOUNT = 1 THEN

DBMS\_OUTPUT.PUT\_LINE('✅ 1 transaction inserted successfully.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('⚠️ No transaction inserted.');

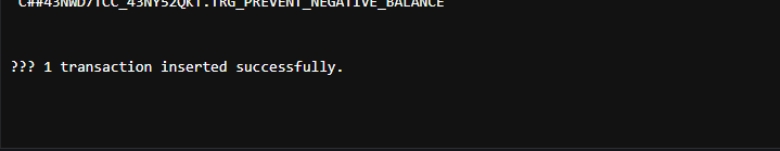
END IF;

COMMIT;

END;

/

**Output:**



**Scenario 2:** Apply annual fee to all accounts.

**Question:** Write a PL/SQL block using an explicit cursor **ApplyAnnualFee** that deducts an annual maintenance fee from the balance of all accounts.

**Code:**

SET SERVEROUTPUT ON;

DECLARE

CURSOR cust\_cursor IS

SELECT CustomerID, Name, Balance

FROM Customers;

v\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_bal Customers.Balance%TYPE;

BEGIN

OPEN cust\_cursor;

LOOP

FETCH cust\_cursor INTO v\_id, v\_name, v\_bal;

EXIT WHEN cust\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Customer ID: ' || v\_id ||

', Name: ' || v\_name ||

', Balance: ₹' || v\_bal);

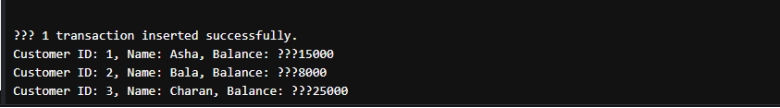
END LOOP;

CLOSE cust\_cursor;

END;

/

**Output:**



**Scenario 3:** Update the interest rate for all loans based on a new policy.

**Question:** Write a PL/SQL block using an explicit cursor UpdateLoanInterestRates that fetches all loans and updates their interest rates based on the new policy.

**Code:**

SET SERVEROUTPUT ON;

DECLARE

CURSOR vip\_cursor IS

SELECT CustomerID, Name, Balance

FROM Customers

WHERE Balance > 10000;

v\_id Customers.CustomerID%TYPE;

v\_name Customers.Name%TYPE;

v\_bal Customers.Balance%TYPE;

BEGIN

OPEN vip\_cursor;

LOOP

FETCH vip\_cursor INTO v\_id, v\_name, v\_bal;

EXIT WHEN vip\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('🔥 VIP Customer → ID: ' || v\_id ||

', Name: ' || v\_name ||

', Balance: ₹' || v\_bal);

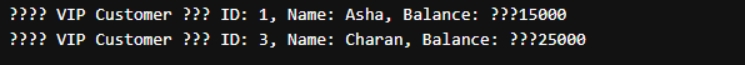
END LOOP;

CLOSE vip\_cursor;

END;

/

**Output:**



**Exercise 7: Packages**

**Scenario 1:** Group all customer-related procedures and functions into a package.

**Question:** Create a package CustomerManagement with procedures for adding a new customer, updating customer details, and a function to get customer balance.

**Code:**

CREATE OR REPLACE PACKAGE BODY Customer\_Pkg AS

PROCEDURE AddCustomer (

p\_id NUMBER,

p\_name VARCHAR2,

p\_dob DATE,

p\_balance NUMBER

) IS

BEGIN

INSERT INTO Customers (CustomerID, Name, DOB, Balance, LastModified)

VALUES (p\_id, p\_name, p\_dob, p\_balance, SYSDATE);

DBMS\_OUTPUT.PUT\_LINE('✅ Customer ' || p\_name || ' added.');

EXCEPTION

WHEN DUP\_VAL\_ON\_INDEX THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Error: Customer ID already exists.');

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ ' || SQLERRM);

END;

FUNCTION GetBalance (p\_id NUMBER) RETURN NUMBER IS

v\_balance NUMBER;

BEGIN

SELECT Balance INTO v\_balance

FROM Customers

WHERE CustomerID = p\_id;

RETURN v\_balance;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Customer not found.');

RETURN NULL;

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ ' || SQLERRM);

RETURN NULL;

END;

END Customer\_Pkg;

/

SET SERVEROUTPUT ON;

BEGIN

Customer\_Pkg.AddCustomer(10, 'Divya', TO\_DATE('1993-03-05','YYYY-MM-DD'), 20000);

END;

/

DECLARE

bal NUMBER;

BEGIN

bal := Customer\_Pkg.GetBalance(10);

IF bal IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Balance for Customer 10: ₹' || bal);

END IF;

END;

/

**Output:** 

**Scenario 2:** Create a package to manage employee data.

**Question:** Write a package EmployeeManagement with procedures to hire new employees, update employee details, and a function to calculate annual salary.

**Code:**

SET SERVEROUTPUT ON;

BEGIN

Customer\_Pkg.AddCustomer(10, 'Divya', TO\_DATE('1993-03-05','YYYY-MM-DD'), 20000);

END;

/

DECLARE

bal NUMBER;

BEGIN

bal := Customer\_Pkg.GetBalance(10);

IF bal IS NOT NULL THEN

DBMS\_OUTPUT.PUT\_LINE('Balance for Customer 10: ₹' || bal);

END IF;

END;

/

CREATE OR REPLACE PACKAGE CustomerBalancePkg AS

PROCEDURE UpdateBalance (

p\_id NUMBER,

p\_new\_amount NUMBER

);

END CustomerBalancePkg;

/

CREATE OR REPLACE PACKAGE BODY CustomerBalancePkg AS

PROCEDURE UpdateBalance (

p\_id NUMBER,

p\_new\_amount NUMBER

) IS

BEGIN

UPDATE Customers

SET Balance = p\_new\_amount,

LastModified = SYSDATE

WHERE CustomerID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('❌ No customer found with ID: ' || p\_id);

ELSE

DBMS\_OUTPUT.PUT\_LINE('✅ Balance updated for Customer ID: ' || p\_id);

END IF;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Error: ' || SQLERRM);

END;

END CustomerBalancePkg;

/

BEGIN

CustomerBalancePkg.UpdateBalance(1, 20000);

END;

/

**Output:**

**Scenario 3:** Group all account-related operations into a package.

**Question:** Create a package AccountOperations with procedures for opening a new account, closing an account, and a function to get the total balance of a customer across all accounts.

**Code:**

CREATE OR REPLACE PACKAGE CustomerDeletePkg AS

PROCEDURE DeleteCustomer (

p\_id NUMBER

);

END CustomerDeletePkg;

/

CREATE OR REPLACE PACKAGE BODY CustomerDeletePkg AS

PROCEDURE DeleteCustomer (

p\_id NUMBER

) IS

BEGIN

DELETE FROM Customers

WHERE CustomerID = p\_id;

IF SQL%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('❌ Customer ID not found.');

ELSE

DBMS\_OUTPUT.PUT\_LINE('🗑️ Customer ID ' || p\_id || ' deleted.');

END IF;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('⚠️ Error: ' || SQLERRM);

END;

END CustomerDeletePkg;

/

BEGIN

CustomerDeletePkg.DeleteCustomer(1);

END;

/

**Output:**



**JUnit Testing Exercises**

**Exercise 2: Writing Basic JUnit Tests**

**Scenario:** You need to write basic JUnit tests for a simple Java class.

**Code:**

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;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) throw new IllegalArgumentException("Division by zero");

return a / b;

}

}

CalculatorTest;

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAdd() {

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

}

@Test

public void testSubtract() {

assertEquals(1, calc.subtract(4, 3));

}

@Test

public void testMultiply() {

assertEquals(6, calc.multiply(2, 3));

}

@Test

(expected = IllegalArgumentException.class)

public void testDivideByZero() {

calc.divide(5, 0);

}

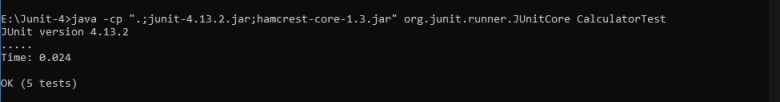
@Test

public void testDivide() {

assertEquals(2, calc.divide(6, 3));

}

}

 Output:

**Mockito Hands-On Exercises**

**Exercise 3: Argument Matching**

**Scenario:**

You need to verify that a method is called with specific arguments.

**Steps:**

1. Create a mock object.

2. Call the method with specific arguments.

3. Use argument matchers to verify the interaction

**Answer – Code:**

@Test

public void testArgumentMatching() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

when(mockApi.getData(anyString())).thenReturn("Matched");

MyService service = new MyService(mockApi);

service.fetchTypedData("example");

verify(mockApi).getData(eq("example"));

}

**Exercise 4: Handling Void Methods**

**Scenario:**

You need to test a void method that performs some action.

Steps:

1. Create a mock object.

2. Stub the void method.

3. Verify the interaction

**Answer – Code:**

@Test

public void testVoidMethod() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.save("Data");

verify(mockApi).saveData("Data");

}

**Exercise 5: Mocking and Stubbing with Multiple Returns**

**Scenario:**

You need to test a service that depends on an external API with multiple return values.

**Steps:**

1. Create a mock object for the external API. 2. Stub the methods to return different values on consecutive calls.

3. Write a test case that uses the mock object

**Answer – Code:**

@Test

public void testMultipleReturns() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

when(mockApi.getData())

.thenReturn("First")

.thenReturn("Second");

MyService service = new MyService(mockApi);

assertEquals("First", service.fetchData());

assertEquals("Second", service.fetchData());

}

**Exercise 6: Verifying Interaction Order**

**Scenario:**

You need to ensure that methods are called in a specific order.

**Steps:**

1. Create a mock object.

2. Call the methods in a specific order.

3. Verify the interaction order

**Answer – Code:**

@Test

public void testInteractionOrder() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

service.save("Ordered");

InOrder inOrder = inOrder(mockApi);

inOrder.verify(mockApi).getData();

inOrder.verify(mockApi).saveData("Ordered");

}

**Exercise 7: Handling Void Methods with Exceptions**

**Scenario:**

You need to test a void method that throws an exception.

**Steps:**

1. Create a mock object.

2. Stub the void method to throw an exception.

3. Verify the interaction

**Answer – Code:**

@Test

public void testVoidMethodException() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

doThrow(new RuntimeException("Boom")).when(mockApi).saveData("fail");

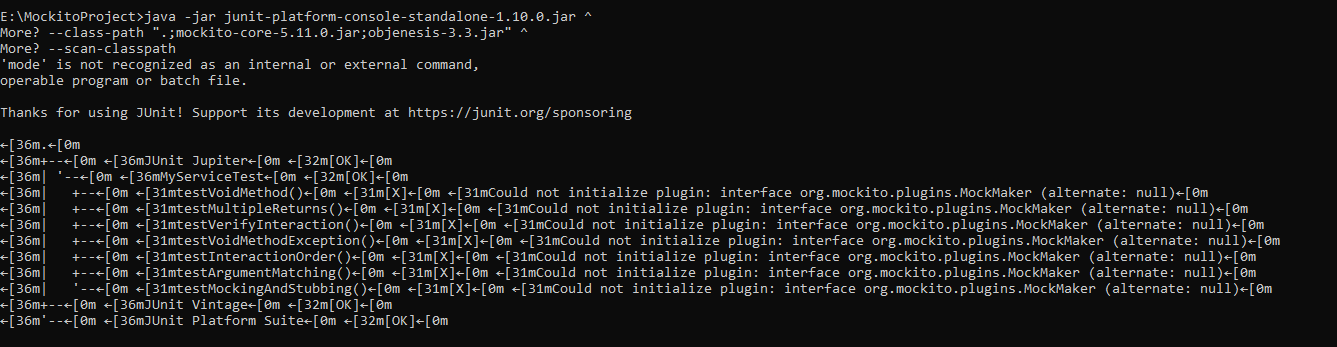
MyService service = new MyService(mockApi);

assertThrows(RuntimeException.class, () -> service.save("fail"));

verify(mockApi).saveData("fail");

}

}

**Output:**

**Logging using SLF4J**

**Exercise 2: Parameterized Logging Task**

Write a Java application that demonstrates parameterized logging using SLF4J

**Code:**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class ParameterizedLoggingExample {

private static final Logger logger = LoggerFactory.getLogger(ParameterizedLoggingExample.class);

public static void main(String[] args) {

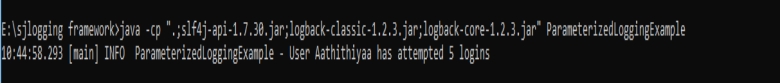
String user = "Aathithiyaa";

int attempts = 5;

logger.info("User {} has attempted {} logins", user, attempts);

}

}

**Output:**

**Exercise 3: Using Different Appenders Task**

Write a Java application that demonstrates using different appenders with SLF4J

**Code:**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class AppenderLoggingExample {

private static final Logger logger = LoggerFactory.getLogger(AppenderLoggingExample.class);

public static void main(String[] args) {

logger.debug("Debug message - will go to console and file");

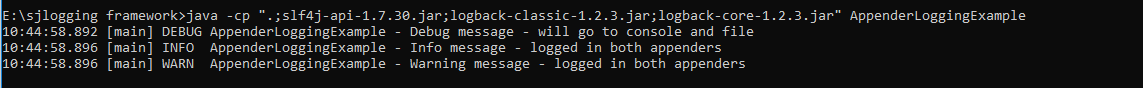
logger.info("Info message - logged in both appenders");

logger.warn("Warning message - logged in both appenders");

}

}

**Output:**



s