**Week 2 PL/SQL and Unit Testing**

**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.

**Solution**

CREATE TABLE customers (

loan\_interest NUMBER,

customer\_age NUMBER

);

INSERT INTO customers (loan\_interest, customer\_age)

VALUES (2000, 23),(1200,62),(2300,45),(1233,25),(2349,70);

SELECT \* FROM customers;

BEGIN

UPDATE customers

SET loan\_interest = loan\_interest - (loan\_interest \* 0.01)

WHERE customer\_age > 60;

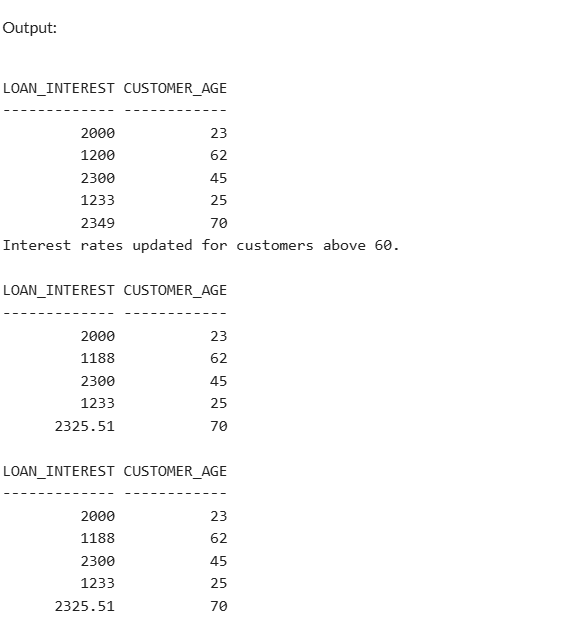
DBMS\_OUTPUT.PUT\_LINE('Interest rates updated for customers above 60.');

END;

/

select \* from customers;

**Output**

****

**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.

**CREATE TABLE customers (**

**customer\_id NUMBER,**

**customer\_name VARCHAR2(50),**

**balance NUMBER,**

**IsVIP CHAR(1) DEFAULT 'N'**

**);**

**INSERT INTO customers VALUES (1, 'Pavithra', 12000, 'N'),(2, 'Ravi', 8000, 'N'),(3,'Rohith',24000,'N'),(4,'Monisha',23400,'N');**

**select \* from customers;**

**DECLARE**

**v\_count NUMBER;**

**BEGIN**

**UPDATE customers**

**SET IsVIP = 'Y'**

**WHERE balance > 10000;**

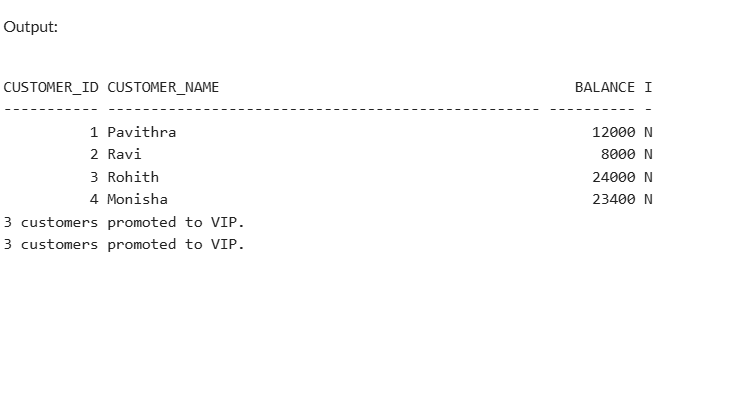
**SELECT COUNT(\*) INTO v\_count FROM customers WHERE balance > 10000;**

**DBMS\_OUTPUT.PUT\_LINE(v\_count || ' customers promoted to VIP.');**

**END;**

**/**

**select \* from customers;**

****

**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.

CREATE TABLE customers (

customer\_id NUMBER,

customer\_name VARCHAR2(100),

loan\_due\_date DATE

);

INSERT INTO customers VALUES (1, 'Pavithra', SYSDATE + 10);

INSERT INTO customers VALUES (2, 'Ravi', SYSDATE + 40);

INSERT INTO customers VALUES (3, 'Anjali', SYSDATE + 5);

INSERT INTO customers VALUES (4, 'Manoj', SYSDATE + 29);

INSERT INTO customers VALUES (5, 'Divya', SYSDATE - 2);

BEGIN

FOR customer\_rec IN (

SELECT customer\_name, loan\_due\_date

FROM customers

WHERE loan\_due\_date BETWEEN SYSDATE AND SYSDATE + 30

)

LOOP

DBMS\_OUTPUT.PUT\_LINE(

'Reminder: Dear ' || customer\_rec.customer\_name ||

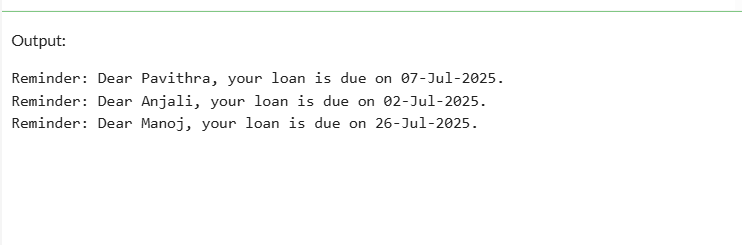
', your loan is due on ' || TO\_CHAR(customer\_rec.loan\_due\_date, 'DD-Mon-YYYY') || '.'

);

END LOOP;

END;

/



**SEC-3 STORED PROCEDURES  
scenario:1**

**CREATE TABLE accounts (**

**account\_id NUMBER PRIMARY KEY,**

**account\_holder VARCHAR2(100),**

**account\_type VARCHAR2(20), -- e.g., 'savings', 'current'**

**balance NUMBER(15,2),**

**created\_date DATE DEFAULT SYSDATE**

**);**

**INSERT INTO accounts (account\_id, account\_holder, account\_type, balance)**

**VALUES (101, 'John Doe', 'savings', 1000.00);**

**INSERT INTO accounts (account\_id, account\_holder, account\_type, balance)**

**VALUES (102, 'Jane Smith', 'current', 2000.00);**

**INSERT INTO accounts (account\_id, account\_holder, account\_type, balance)**

**VALUES (103, 'Alice Brown', 'savings', 1500.00);**

**CREATE OR REPLACE PROCEDURE ProcessMonthlyInterest IS**

**BEGIN**

**UPDATE accounts**

**SET balance = balance + (balance \* 0.01)**

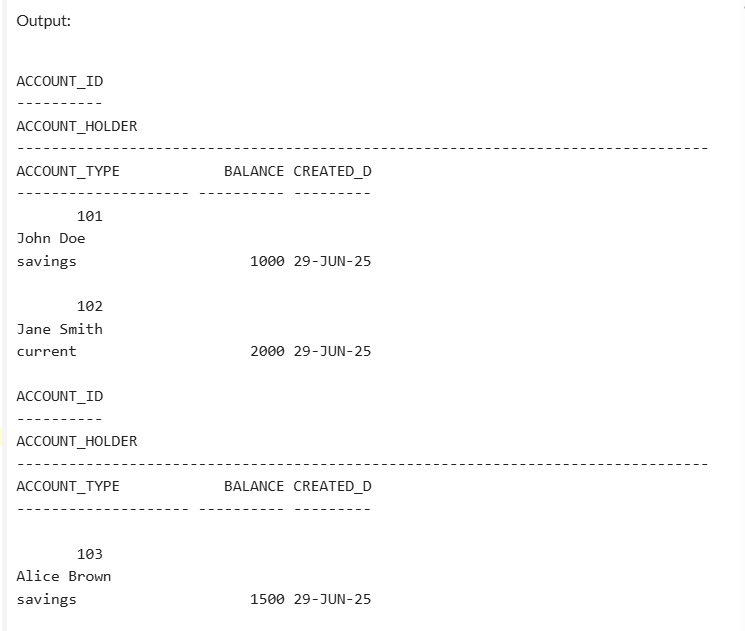
**WHERE account\_type = 'savings';**

**COMMIT;**

**END;**

**/**

**select \* from accounts;**

****

**Section 2:**

**CREATE TABLE employees (**

**employee\_id NUMBER PRIMARY KEY,**

**employee\_name VARCHAR2(100),**

**department\_id NUMBER,**

**salary NUMBER(10,2)**

**);**

**INSERT INTO employees VALUES (1, 'Alice', 101, 50000);**

**INSERT INTO employees VALUES (2, 'Bob', 102, 60000);**

**INSERT INTO employees VALUES (3, 'Charlie', 101, 55000);**

**INSERT INTO employees VALUES (4, 'David', 103, 70000);**

**COMMIT;**

**CREATE OR REPLACE PROCEDURE UpdateEmployeeBonus (**

**p\_dept\_id IN NUMBER,**

**p\_bonus\_percent IN NUMBER**

**) IS**

**BEGIN**

**UPDATE employees**

**SET salary = salary + (salary \* (p\_bonus\_percent / 100))**

**WHERE department\_id = p\_dept\_id;**

**COMMIT;**

**END;**

**/**

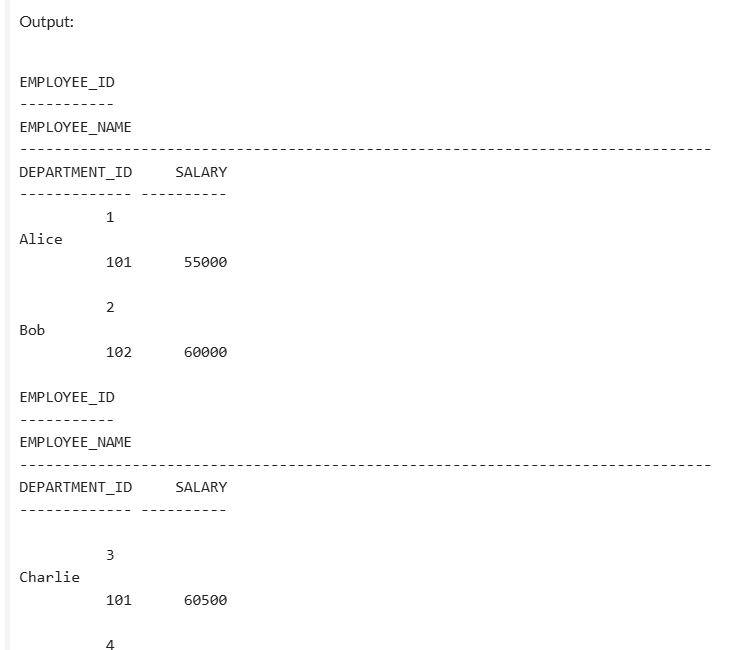
**BEGIN**

**-- Give a 10% bonus to employees in department 101**

**UpdateEmployeeBonus(101, 10);**

**END;**

**/**

**SELECT \* FROM employees;  
**

**Sec:3  
CREATE TABLE accounts (**

**account\_id NUMBER PRIMARY KEY,**

**account\_holder VARCHAR2(100),**

**balance NUMBER(15,2)**

**);**

**INSERT INTO accounts VALUES (1, 'Alice', 5000);**

**INSERT INTO accounts VALUES (2, 'Bob', 3000);**

**COMMIT;**

**CREATE OR REPLACE PROCEDURE TransferFunds (**

**p\_from\_account\_id IN NUMBER,**

**p\_to\_account\_id IN NUMBER,**

**p\_amount IN NUMBER**

**) IS**

**v\_balance NUMBER;**

**BEGIN**

**-- Check if source account exists and get its balance**

**SELECT balance INTO v\_balance**

**FROM accounts**

**WHERE account\_id = p\_from\_account\_id**

**FOR UPDATE;**

**-- Check if balance is sufficient**

**IF v\_balance < p\_amount THEN**

**RAISE\_APPLICATION\_ERROR(-20001, 'Insufficient funds in the source account');**

**END IF;**

**-- Deduct from source**

**UPDATE accounts**

**SET balance = balance - p\_amount**

**WHERE account\_id = p\_from\_account\_id;**

**-- Add to destination**

**UPDATE accounts**

**SET balance = balance + p\_amount**

**WHERE account\_id = p\_to\_account\_id;**

**COMMIT;**

**END;**

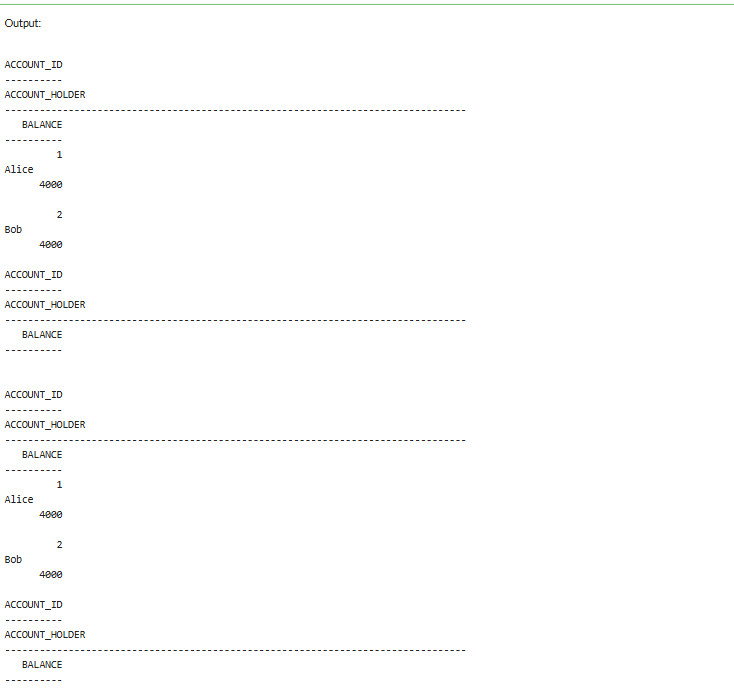
**/BEGIN**

**-- Transfer 1000 from account 1 to account 2**

**TransferFunds(1, 2, 1000);**

**END;**

**/**

****

**TDD using JUnit5 and Mockito**

**Exercise 1: Setting Up Junit**

**package com.example.junitdemo;**

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

**import org.junit.Test;**

**public class AppTest {**

**@Test**

**public void testAddition() {**

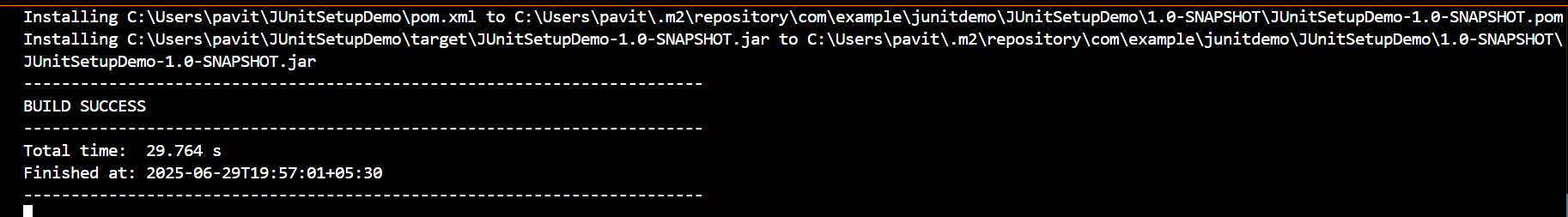
**int result = 2 + 3;**

**assertEquals(5, result);**

**}**

**}**

**Output:**

****

**Sec -3**

**package com.example.junitdemo;**

**public class UserUtils {**

**public boolean isAdult(int age) {**

**return age >= 18;**

**}**

**public String getUserRole(String username) {**

**if (username == null) return null;**

**if (username.equals("admin")) return "ADMIN";**

**return "USER";**

**}**

**}**

**package com.example.junitdemo;**

**import org.junit.Test;**

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

**public class UserUtilsTest {**

**UserUtils utils = new UserUtils();**

**@Test**

**public void testIsAdult() {**

**assertTrue(utils.isAdult(20)); // age >= 18**

**assertFalse(utils.isAdult(16)); // age < 18**

**}**

**@Test**

**public void testUserRole() {**

**assertEquals("ADMIN", utils.getUserRole("admin"));**

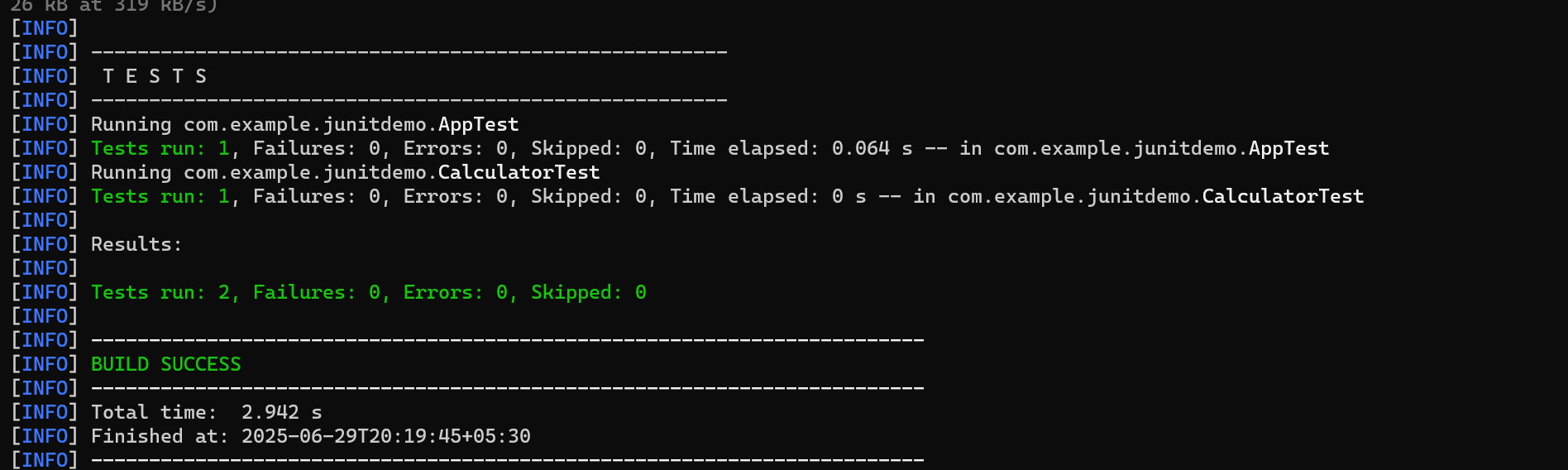
**assertEquals("USER", utils.getUserRole("pavithra"));**

**assertNull(utils.getUserRole(null)); // null case**

**assertNotNull(utils.getUserRole("someone"));**

**}**

**}**

**Output  
**

**Exercise 4: Exception Testing and Setup/Teardown**

BankAccount.java  
package com.example.junitdemo;

public class BankAccount {

private int balance;

public BankAccount(int openingBalance) {

this.balance = openingBalance;

}

public void deposit(int amount) {

if (amount <= 0) throw new IllegalArgumentException("Invalid deposit amount");

balance += amount;

}

public void withdraw(int amount) {

if (amount > balance) throw new IllegalArgumentException("Insufficient funds");

balance -= amount;

}

public int getBalance() {

return balance;

}

}  
BankAccounttest.java package com.example.junitdemo;

import org.junit.\*;

import static org.junit.Assert.\*;

public class BankAccountTest {

BankAccount account;

@Before

public void setUp() {

account = new BankAccount(1000); // every test starts with ₹1000

System.out.println("🔧 setUp: Account created with ₹1000");

}

@After

public void tearDown() {

System.out.println("🧹 tearDown: Test finished");

}

@Test

public void testDeposit() {

account.deposit(500);

assertEquals(1500, account.getBalance());

}

@Test(expected = IllegalArgumentException.class)

public void testInvalidDeposit() {

account.deposit(-100); // should throw exception

}

@Test

public void testWithdraw() {

account.withdraw(400);

assertEquals(600, account.getBalance());

}

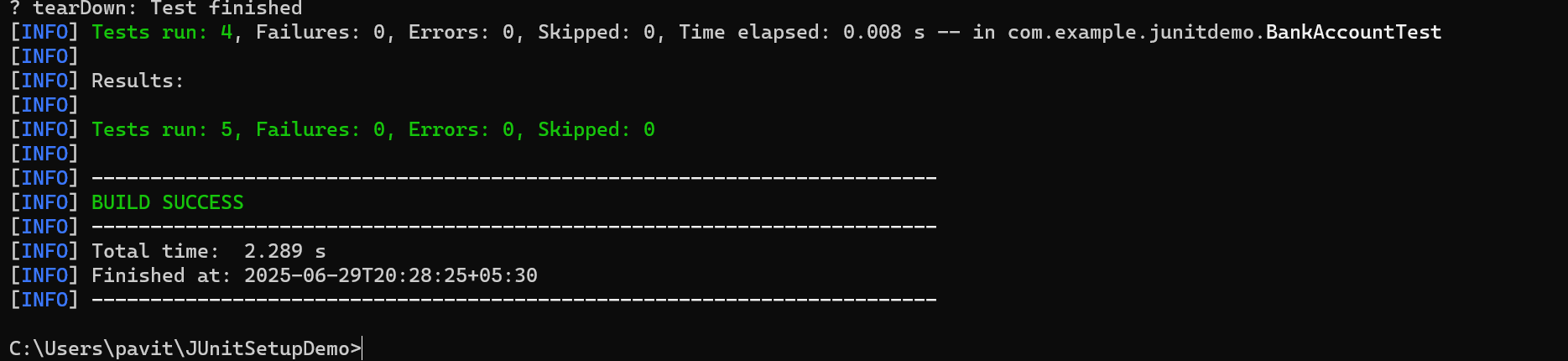
@Test(expected = IllegalArgumentException.class)

public void testOverWithdraw() {

account.withdraw(2000); // should throw exception

}

}

**Output  
**

**Exercise 6:**

MyService.java

package com.example.mockito;

public class MyService {

    private ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public String fetchData() {

        return api.getData();

    }

}

MyServiceTest.java

ackage com.example.mockito;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.AfterEach;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class MyServiceTest {

    @Test

    public void testExternalApi() {

        // 1. Create mock

        ExternalApi mockApi = mock(ExternalApi.class);

        // 2. Stub method

        when(mockApi.getData()).thenReturn("Mock Data");

        // 3. Inject mock

        MyService service = new MyService(mockApi);

        // 4. Call method and verify

        String result = service.fetchData();

        assertEquals("Mock Data", result);

    }

}

ExternalAPI.java

package com.example.mockito;

public interface ExternalApi {

    String getData();

}

**Exercise 7:**

package com.example.junitdemo;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

// Dummy interface

interface ExternalApi {

    String getData();

}

// Service class depending on ExternalApi

class MyService {

    private ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public void fetchData() {

        api.getData(); // this is what we're verifying

    }

}

// Actual test class

public class MyServiceTest {

    @Test

    public void testVerifyInteraction() {

        ExternalApi mockApi = mock(ExternalApi.class);

        MyService service = new MyService(mockApi);

        service.fetchData(); // should call getData()

        verify(mockApi).getData(); // verify interaction

    }

}

**Exercise 1: SLF4J Logging**

package com.example.junitdemo;

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) {

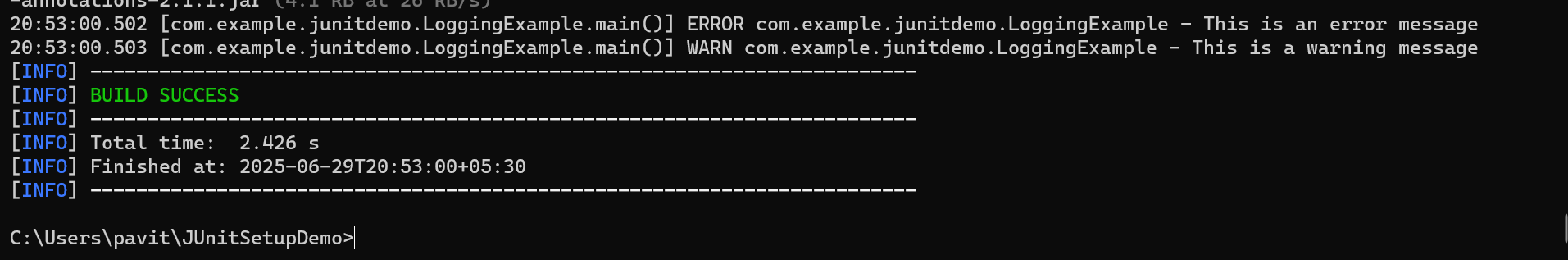
        logger.error("This is an error message");

        logger.warn("This is a warning message");

    }

}

**Output:**

****