**JUNT EXERCISE:**

**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. test**

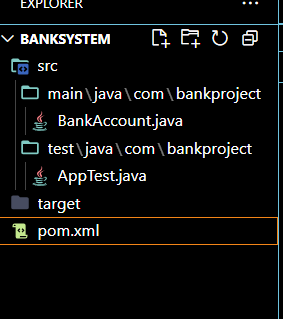
**3. Create a new test class in your project.**

**=**

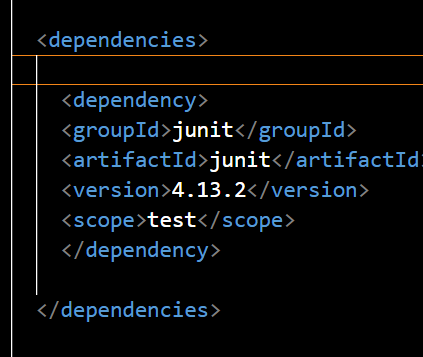
**STEP 1**: JAVA MAVEN PROJECT “BANKSYSTEM” CREATED.

**STEP 2:** I loaded 2 jar files in the project to create manual dependency. I have discussed below.

**STEP 3**: TEST CLASS “BankAccountTest.java” created.

****

**MAVEN DEPENDECY ADDED:**

****

**TEST CLASS ALSO CREATED.**

**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.**

**=**

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());

}

}

**FOR OUR PROJRCT:**

package com.bankproject;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankAssertionsTest {

@Test

public void testBalanceEquality() {

BankAccount account = new BankAccount("Arjun", 1000);

assertEquals(1000, account.getBalance(), 0.001); // Assert Equals

}

@Test

public void testSuccessfulDeposit() {

BankAccount account = new BankAccount("Meera", 500);

account.deposit(200);

assertTrue(account.getBalance() == 700); // Assert True

}

@Test

public void testFailedWithdrawal() {

BankAccount account = new BankAccount("Ravi", 300);

boolean result = account.withdraw(500); // More than balance

assertFalse(result); // Assert False

}

@Test

public void testAccountHolderNameNotNull() {

BankAccount account = new BankAccount("Priya", 900);

assertNotNull(account.getAccountHolder()); // Assert Not Null

}

@Test

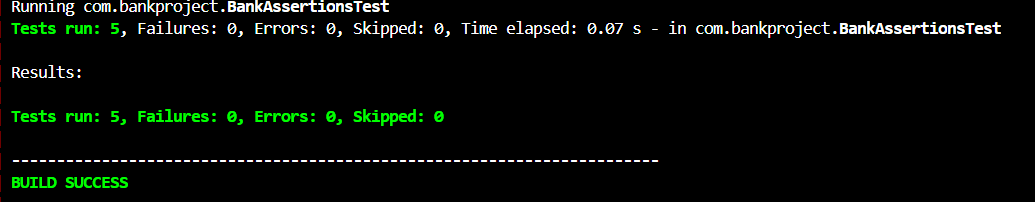
public void testNullAccountHolder() {

BankAccount account = new BankAccount(null, 0);

assertNull(account.getAccountHolder()); // Assert Null

}

}

****

**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.**

**=**

package com.bankproject;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankAccountTestFixture {

private BankAccount account;

// Setup method

@Before

public void setUp() {

account = new BankAccount("SWARNADRI", 1000);

System.out.println("Setup: New account IS created!!!");

}

// Teardown method – runs AFTER every test

@After

public void tearDown() {

account = null;

System.out.println("Teardown: Account IS reset !!!\n");

}

@Test

public void testDeposit() {

// Arrange done in setUp()

// Act

account.deposit(500);

// Assert

assertEquals(1500, account.getBalance(), 0.001);

}

@Test

public void testWithdrawSuccess() {

// Arrange done in setUp()

// Act

boolean success = account.withdraw(400);

// Assert

assertTrue(success);

assertEquals(600, account.getBalance(), 0.001);

}

@Test

public void testWithdrawFail() {

// Act

boolean success = account.withdraw(2000); // exceeds balance

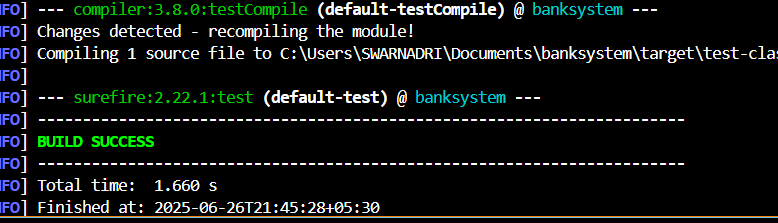
// Assert

assertFalse(success);

assertEquals(1000, account.getBalance(), 0.001); // no deduction

}

}

****

**MOCKITO EXERCISE**

**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);

}

}

**NOW IMPLEMENTING THIS STRUCTURE IN MY PROJECT:**

WE HAVE USED API THAT WORKS FOR SENDING EMAIL AND DEFINED THE CLASS FOR IT SO THAT MOKITO CAN MOCK IT.

**EmailService.java:**

package com.bankproject;

public class EmailService {

public String sendEmail(String recipient, String message) {

return "Email sent to " + recipient + ": " + message;

}

}

**BankService.java:**

package com.bankproject;

public class BankService {

private EmailService emailService;

public BankService(EmailService emailService) {

this.emailService = emailService;

}

public String notifyCustomer(String name, String message) {

return emailService.sendEmail(name, message);

}

}

**BankServiceTest.java:**

package com.bankproject;

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

public class BankServiceTest {

@Test

public void testNotifyCustomer\_UsesMockedEmailService() {

EmailService mockEmail = mock(EmailService.class);

when(mockEmail.sendEmail( "Welcome!"))

.thenReturn("Mocked: Email sent!");

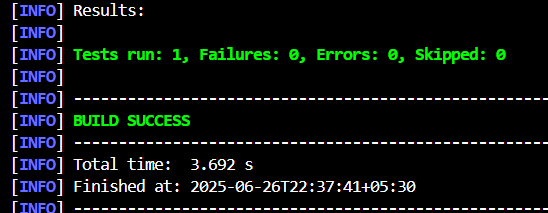
BankService bankService = new BankService(mockEmail);

String result = bankService.notifyCustomer("Welcome!");

assertEquals("Mocked: Email sent!”);

}

}



**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();

}

}

**IMPLEMENTING IN OUR BANKSYSTEM PROJECT:**

1. **WE HAVE REUSED THE “EmailService.java” FLE**
2. **WE HAVE REUSED THE “BankSystem.java” FILE**
3. **WE HAVE ISSUED A NEW TEST CLASS FOLLOWING THE GIVEN SYNTAX NAMED “BankServiceVerifyTest”**

**BankServiceVerifyTest.java :**

package com.bankproject;

import org.junit.Test;

import static org.mockito.Mockito.\*;

public class BankServiceVerifyTest {

@Test

public void testEmailSentToCorrectCustomer() {

EmailService mockEmail = mock(EmailService.class);

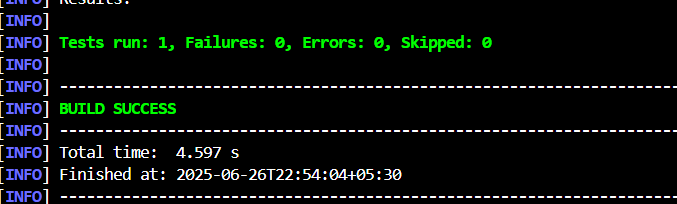
BankService bankService = new BankService(mockEmail);

bankService.notifyCustomer("Bob", "Account Approved");

verify(mockEmail).sendEmail("Bob", "Account Approved");

}

}



**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>

**STEP 2:**

**CREATE JAVA CLASS “LoggingExample”**

=

package com.bankproject;

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 - something failed badly!");

logger.warn(" This is a warning message - be cautious!");

System.out.println("Check your logs above ↑");

}

}

