**Week 2 – Unit Testing SupersetId-6431499**

**Name : Balaji V**

**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.

**Code:**

**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>JUnitTestProject</artifactId>

<version>1.0-SNAPSHOT</version>

<properties>

<maven.compiler.source>23</maven.compiler.source>

<maven.compiler.target>23</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>

**Calculator.java:**

package org.example;

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 double divide(int a,int b){

if(b==0){

throw new IllegalArgumentException("Cannot divide by zero");

}

return (double) a/b;

  }

**CalculatorTest.java:**

package org.example;

import org.junit.Test;

import org.junit.Before;

import org.junit.After;

import static org.junit.Assert.\*;

public class CalculatorTest{

private Calculator calculator;

@Before

public void setUp(){

calculator=new Calculator();

System.out.println("Setting up test...");

}

@After

public void tearDown(){

calculator=null;

System.out.println("Cleaning up after test...");

}

@Test

public void testAdd(){

int result=calculator.add(5, 3);

assertEquals(8, result);

}

@Test

public void testSubtract(){

int result=calculator.subtract(10, 4);

assertEquals(6, result);

}

@Test

public void testMultiply(){

int result=calculator.multiply(7, 6);

assertEquals(42, result);

}

@Test

public void testDivide(){

double result = calculator.divide(15, 3);

assertEquals(5.0, result, 0.001);

}

@Test(expected=IllegalArgumentException.class)

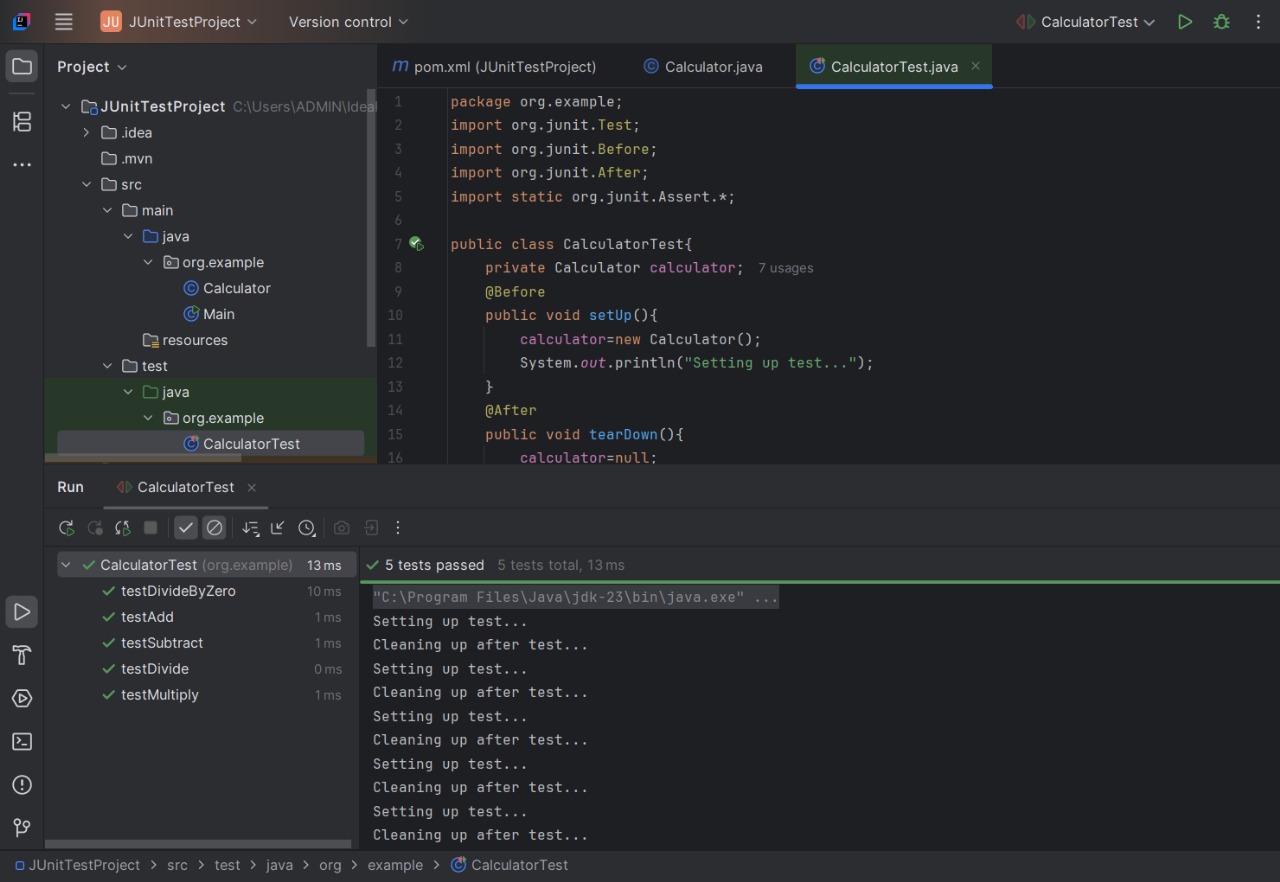
public void testDivideByZero(){

calculator.divide(10, 0);

}

}

**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());

}

}

**Code:**

**Assertion.java:**package org.example;

public class Assertion{

public int add(int a,int b){

return a+b;

}

public int subtract(int a,int b){

return a-b;

}

public String getFullName(String firstName,String lastName){

return firstName+" "+lastName;

}

public boolean isEmpty(String str){

return str==null || str.isEmpty();

}

public String getNullString(){

return null;

}

public String getWelcomeMessage(){

return "Welcome, Balaji!";

}

}

**AssertionTest.java:**

package org.example;

import org.junit.Test;

import org.junit.Before;

import static org.junit.Assert.\*;

public class AssertionTest{

private Assertion utils;

@Before

public void setUp(){

utils=new Assertion();

}

@Test

public void testAssertions(){

assertEquals("Addition should work correctly", 5, 2 + 3);

int larger=5;

int smaller=3;

assertTrue("Larger number should be greater than smaller", larger > smaller);

assertFalse("Smaller number should not be greater than larger", smaller > larger);

String nullString=null;

assertNull("Null string should be null", nullString);

Object testObject=new Object();

assertNotNull("Created object should not be null", testObject);

}

@Test

public void testAddition(){

assertEquals("Addition should return correct result", 7, utils.add(3, 4));

assertEquals("Addition with negative numbers", -1, utils.add(-3, 2));

assertEquals("Addition with zero", 5, utils.add(5, 0));

}

@Test

public void testSubtractionConditions(){

assertEquals("Subtraction result should be 5", 5, utils.subtract(10, 5));

assertTrue("Subtraction result should be 5", utils.subtract(10, 5) == 5);

assertFalse("Subtraction result should not be 3", utils.subtract(10, 5) == 3);

assertEquals("Subtraction with negative result", -2, utils.subtract(3, 5));

}

@Test

public void testFullName(){

assertEquals("Full name should be joined correctly", "Kavi Priya", utils.getFullName("Kavi", "Priya"));

assertEquals("Full name with single character", "A B", utils.getFullName("A", "B"));

}

@Test

public void testEmptyStringCheck(){

assertTrue("Empty string should return true", utils.isEmpty(""));

assertTrue("Null string should return true", utils.isEmpty(null));

assertFalse("Non-empty string should return false", utils.isEmpty("Hello"));

assertFalse("String with spaces should return false", utils.isEmpty(" "));

}

@Test

public void testNullChecks(){

assertNull("Should return null", utils.getNullString());

assertNotNull("Welcome message should not be null", utils.getWelcomeMessage());

assertEquals("Welcome message should match expected", "Welcome, Balaji!", utils.getWelcomeMessage());

}

@Test

public void testEdgeCases(){

assertEquals("Adding zero should not change value", 10, utils.add(10, 0));

assertEquals("Adding negative numbers", -8, utils.add(-3, -5));

assertEquals("Subtracting from itself should be zero", 0, utils.subtract(5, 5));

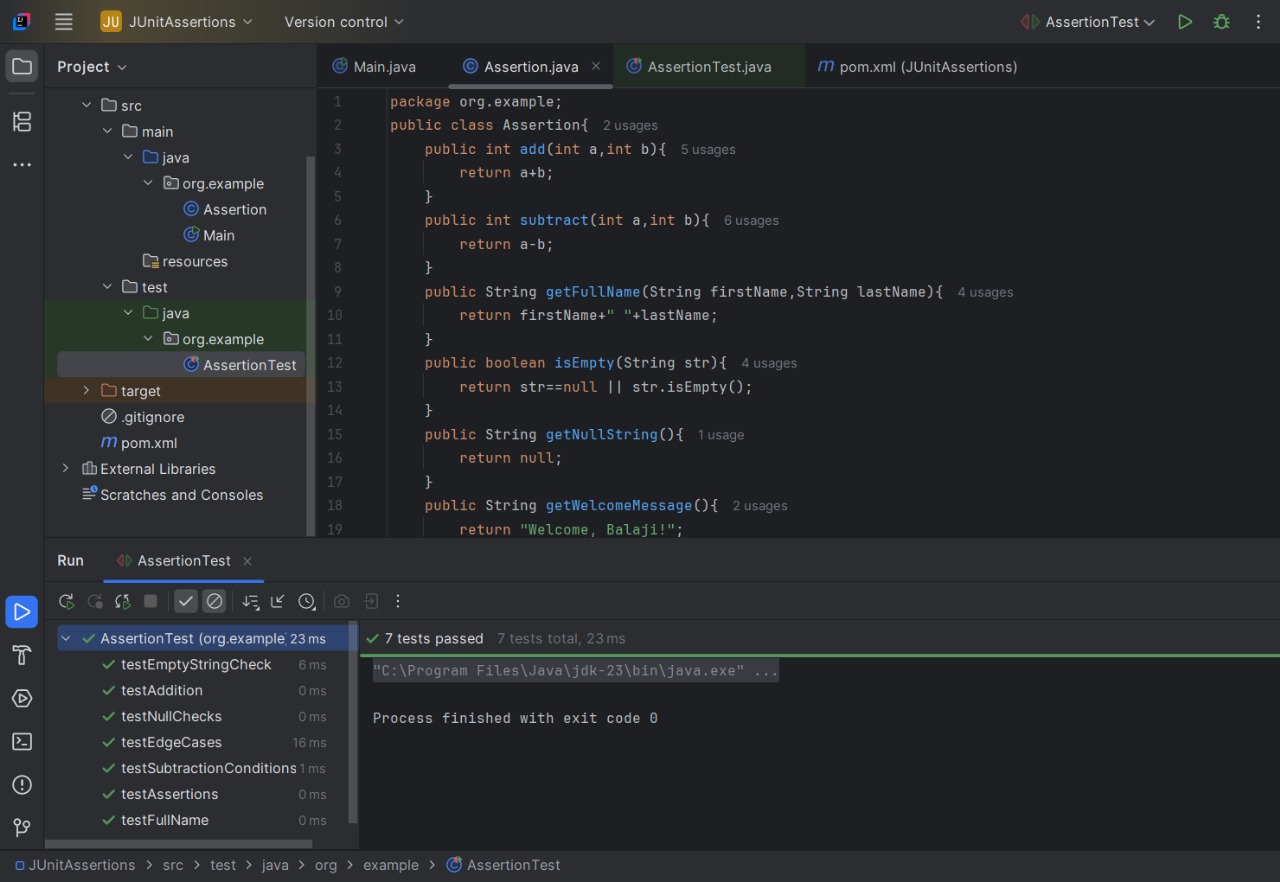
assertEquals("Subtracting zero should not change value", 10, utils.subtract(10, 0));

assertEquals("Single letter names", "X Y", utils.getFullName("X", "Y"));

}

}

**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.

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

**Code:**

**Calculator.java:**

package org.example.calculator;

import java.util.List;

import java.util.ArrayList;

public class Calculator{

private List<Double> list;

public Calculator(){

this.list=new ArrayList<>();

}

public double add(double a,double b){

double result=a + b;

list.add(result);

return result;

}

public double subtract(double a,double b){

double result=a-b;

list.add(result);

return result;

}

public double multiply(double a,double b){

double result=a\*b;

list.add(result);

return result;

}

public double divide(double a,double b){

if(b==0){

throw new IllegalArgumentException("Cannot divide by zero");

}

double result=a/b;

list.add(result);

return result;

}

public String getNullOperationStatus(){

return null;

}

}

**CalculatorTest.java:**

package org.example.calculator;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

import java.util.List;

public class CalculatorTest{

private Calculator calculator;

@Before

public void setUp(){

calculator=new Calculator();

System.out.println("Calculator created");

}

@After

public void tearDown(){

calculator=null;

System.out.println("Calculator cleaned up\n");

}

@Test

public void testAddition(){

double result=calculator.add(10.5, 5.5);

assertEquals("Addition result should be 16.0", 16.0, result, 0.001);

}

@Test

public void testSubtraction(){

double result=calculator.subtract(20.0, 7.5);

assertEquals("Subtraction result should be 12.5", 12.5, result, 0.001);

}

@Test

public void testMultiplication(){

double result=calculator.multiply(4.0, 3.5);

assertEquals("Multiplication should return 14.0", 14.0, result, 0.001);

}

@Test

public void testDivision(){

double result=calculator.divide(15.0, 3.0);

assertEquals("Division should return 5.0", 5.0, result, 0.001);

}

@Test(expected=IllegalArgumentException.class)

public void testDivisionByZeroThrowsException(){

calculator.divide(10.0, 0.0);

}

@Test

public void testListNotNull(){

calculator.add(2.0, 3.0);

List<Double> list=getInternalList(calculator);

assertNotNull("List should not be null after addition", list);

}

@Test

public void testListContainsResult(){

calculator.multiply(2.0, 2.0);

List<Double> list = getInternalList(calculator);

assertTrue("List should contain 4.0 after multiplication", list.contains(4.0));

assertFalse("List should not contain 10.0", list.contains(10.0));

}

@Test

public void testListAfterMultipleOperations(){

calculator.add(1.0, 1.0);

calculator.subtract(5.0, 3.0);

calculator.multiply(2.0, 2.0);

List<Double> list=getInternalList(calculator);

assertEquals("List should have 3 results", 3, list.size());

assertTrue("List should contain 4.0", list.contains(4.0));

}

@Test

public void testLastOperationStatusIsNull(){

String status = calculator.getNullOperationStatus();

assertNull("getLastOperationStatus() should return null", status);

}

private List<Double> getInternalList(Calculator calculator){

try{

java.lang.reflect.Field field = Calculator.class.getDeclaredField("list");

field.setAccessible(true);

return (List<Double>) field.get(calculator);

}catch(Exception e){

fail("Unable to access list: " + e.getMessage());

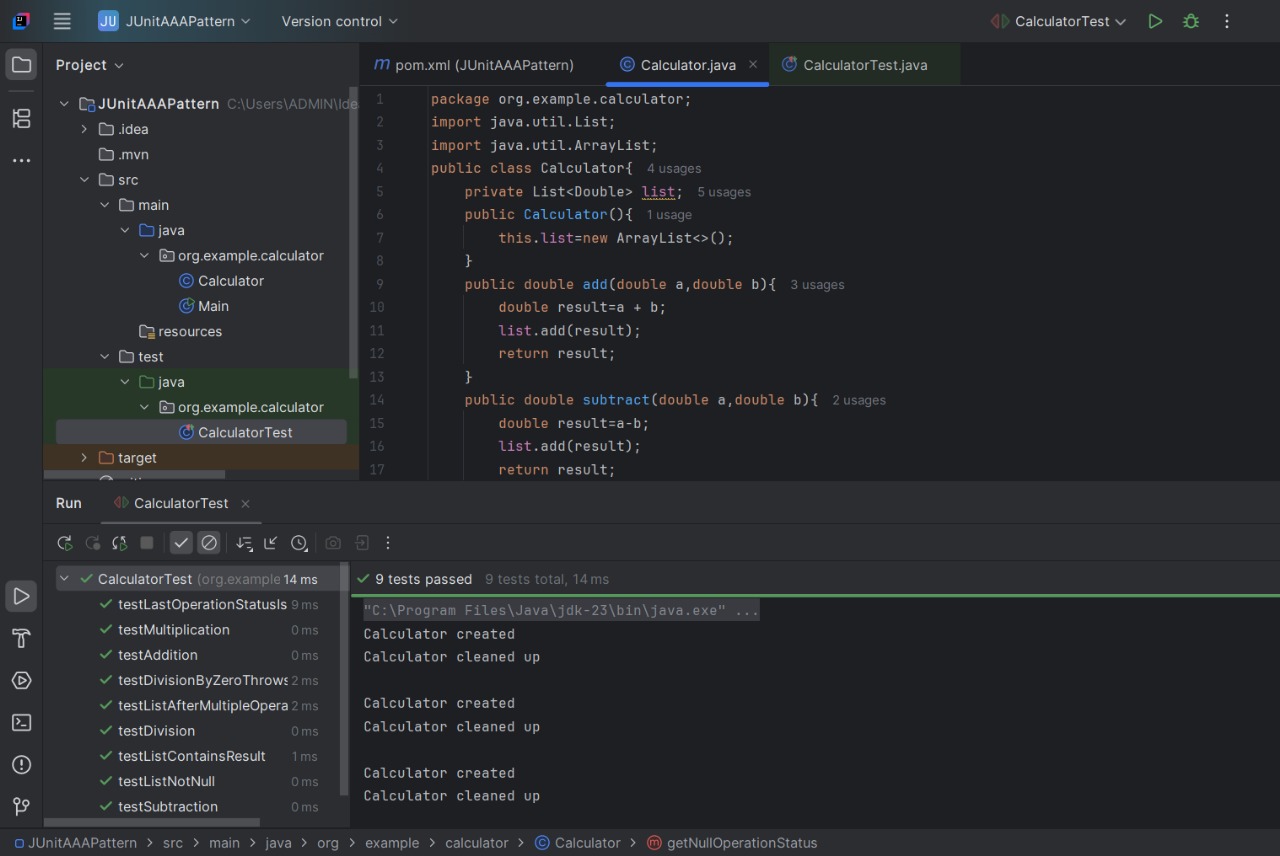
return null;

}

}

}

**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.

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);

} }

**Code:**

**BookApiClient.java:**

package org.example;

public interface BookApiClient {

String getBookTitle(String isbn);

String getAuthorName(String isbn);

String getAvailability(String isbn);

}

**BookService.java:**

package org.example;

public class BookService {

private final BookApiClient bookApiClient;

public BookService(BookApiClient bookApiClient){

this.bookApiClient=bookApiClient;

}

public String getBookInfo(String isbn){

String title=bookApiClient.getBookTitle(isbn);

String author=bookApiClient.getAuthorName(isbn);

return "Book: "+title+" by "+author;

}

public String checkBookStatus(String isbn){

String availability=bookApiClient.getAvailability(isbn);

return "Status: " + availability.toUpperCase();

}

public String getFullBookDetails(String isbn){

String title=bookApiClient.getBookTitle(isbn);

String author=bookApiClient.getAuthorName(isbn);

String availability=bookApiClient.getAvailability(isbn);

return String.format("Title: %s | Author: %s | Available: %s", title, author, availability);

}

}

**BookServiceTest.java:**

package org.example;

import static org.junit.jupiter.api.Assertions.\*;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class BookServiceTest{

@Test

public void testGetBookInfo(){

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

when(mockApi.getBookTitle("978-0134685991")).thenReturn("Effective Java");

when(mockApi.getAuthorName("978-0134685991")).thenReturn("Joshua Bloch");

BookService service=new BookService(mockApi);

String result=service.getBookInfo("978-0134685991");

assertEquals("Book: Effective Java by Joshua Bloch", result);

}

@Test

public void testCheckBookStatus(){

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

when(mockApi.getAvailability("978-0134685991")).thenReturn("in stock");

BookService service=new BookService(mockApi);

String result=service.checkBookStatus("978-0134685991");

assertEquals("Status: IN STOCK", result);

}

@Test

public void testGetFullBookDetails(){

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

when(mockApi.getBookTitle("978-0134685991")).thenReturn("Clean Code");

when(mockApi.getAuthorName("978-0134685991")).thenReturn("Robert C. Martin");

when(mockApi.getAvailability("978-0134685991")).thenReturn("available");

BookService service=new BookService(mockApi);

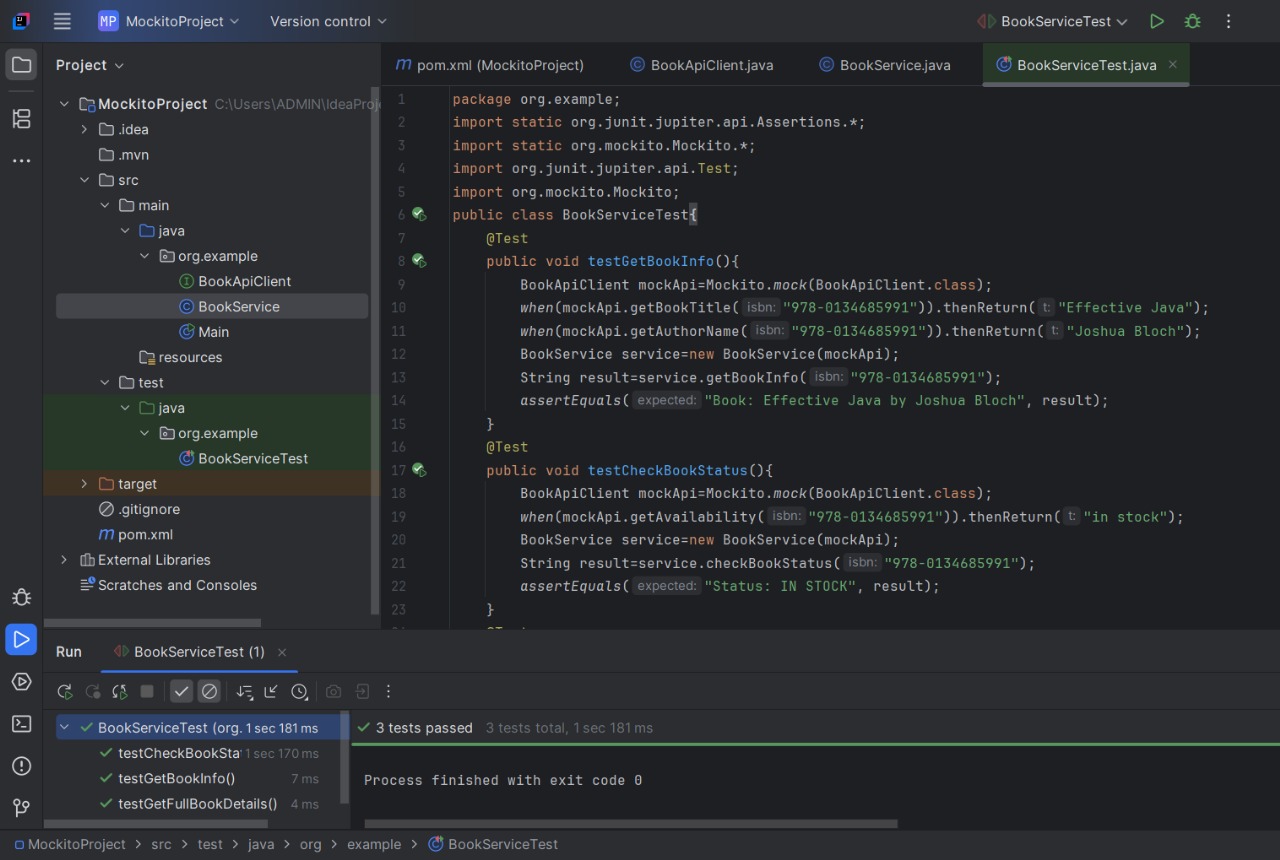
String result=service.getFullBookDetails("978-0134685991");

assertEquals("Title: Clean Code | Author: Robert C. Martin | Available: available", result);

}

}

**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();

}

}

**Code:**

**ExternalApi.java:**

package org.example;

public interface ExternalApi{

String getData();

String getData(String parameter);

void processData(String data);

}

**MyService.java:**

package org.example;

public class MyService{

private final ExternalApi externalApi;

public MyService(ExternalApi externalApi){

this.externalApi = externalApi;

}

public String fetchData(){

return externalApi.getData();

}

public String fetchDataWithParameter(String param){

return externalApi.getData(param);

}

public void processInformation(String info){

externalApi.processData(info);

}

}

**MyServiceTest.java:**  
package org.example;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.junit.jupiter.api.BeforeEach;

import org.mockito.Mock;

import org.mockito.MockitoAnnotations;

public class MyServiceTest{

@Mock

private ExternalApi mockApi;

private MyService service;

@BeforeEach

void setUp() {

MockitoAnnotations.openMocks(this);

service=new MyService(mockApi);

}

@Test

public void testVerifyInteraction(){

service.fetchData();

verify(mockApi).getData();

}

@Test

public void testVerifyInteractionWithArguments(){

String testParameter="test-data";

service.fetchDataWithParameter(testParameter);

verify(mockApi).getData(testParameter);

}

@Test

public void testVerifyVoidMethodInteraction(){

String testData="processing-data";

service.processInformation(testData);

verify(mockApi).processData(testData);

}

@Test

public void testVerifyNumberOfInteractions(){

service.fetchData();

service.fetchData();

verify(mockApi, times(2)).getData();

}

@Test

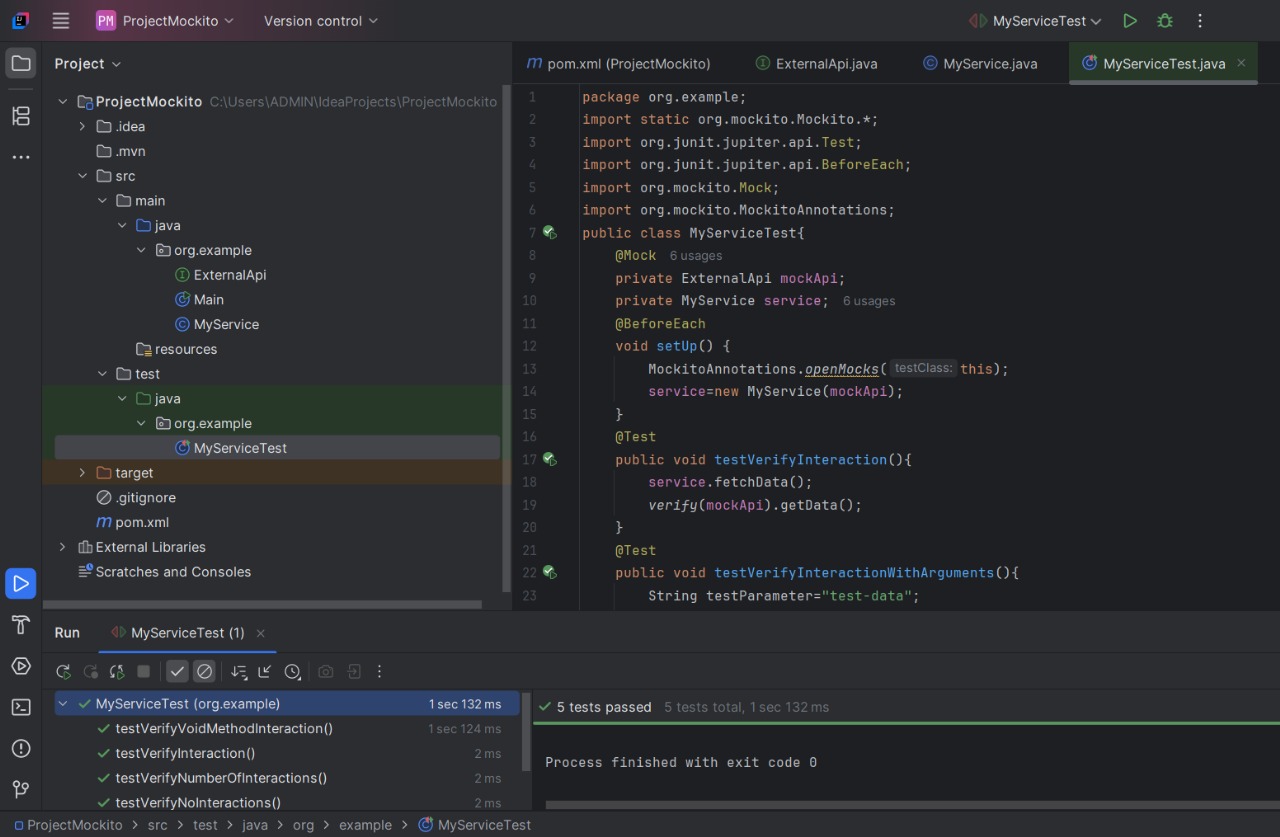
public void testVerifyNoInteractions(){

verifyNoInteractions(mockApi);

}

}

**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");

}

}

**Code:**

**LoggingDetails.java:**

package org.example;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingDetails{

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

public static void main(String[] args){

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

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

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

logger.debug("This is a debug message");

logger.trace("This is a trace message");

String username="Ashwanth\_Raj";

int attempts=3;

logger.error("User {} failed to login after {} attempts",username,attempts);

try{

int result=10/0;

}catch(ArithmeticException e){

logger.error("An arithmetic error occurred", e);

}

}

}

**logback.xml:**

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<appender name="FILE" class="ch.qos.logback.core.FileAppender">

<file>logs/application.log</file>

<encoder>

<pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="DEBUG">

<appender-ref ref="CONSOLE"/>

<appender-ref ref="FILE"/>

</root>

</configuration>

**Output:**

