JUnit5 hands on 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.

**Solution:**

**Step 1: Create a Java Class (Calculator.java)**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

Step 2: Add JUnit to Maven (pom.xml)

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

Step 3: Create a Test Class (CalculatorTest.java)

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

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

}

}

Output:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running CalculatorTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.01 sec

Results :

Tests run: 1, Failures: 0

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());

}

}

Solution code:

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

// Assert that 2 + 3 equals 5

assertEquals(5, 2 + 3);

// Assert that 5 is greater than 3

assertTrue(5 > 3);

// Assert that 5 is not less than 3

assertFalse(5 < 3);

// Assert that the object is null

Object obj1 = null;

assertNull(obj1);

// Assert that the object is not null

Object obj2 = new Object();

assertNotNull(obj2);

}

}

Output:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running AssertionsTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

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.

Solution:

Java Class to Test (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;

}

}

Test Class Using AAA, @Before, @After (CalculatorTest.java):

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

private Calculator calculator;

// Setup method – runs before each test

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("Setup complete.");

}

// Teardown method – runs after each test

@After

public void tearDown() {

System.out.println("Teardown complete.\n");

}

// Test 1: Addition using AAA pattern

@Test

public void testAdd() {

// Arrange

int a = 5;

int b = 3;

// Act

int result = calculator.add(a, b);

// Assert

assertEquals(8, result);

}

@Test

public void testSubtract() {

int a = 10;

int b = 4;

int result = calculator.subtract(a, b);

assertEquals(6, result);

}

}

Output:

Setup complete.

Teardown complete.

Setup complete.

Teardown complete.

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running CalculatorTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

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.

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);

}

}

Solution:

ExternalApi.java:

public interface ExternalApi {

String getData();

}

MyService.java:

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

MyServiceTest.java:

import org.junit.jupiter.api.Test;

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

import static org.mockito.Mockito.\*;

public class MyServiceTest {

@Test

public void testExternalApi() {

ExternalApi mockApi = mock(ExternalApi.class);

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

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

Dependencies:

<!-- JUnit 5 -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.10.0</version>

<scope>test</scope>

</dependency>

<!-- Mockito -->

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.11.0</version>

<scope>test</scope>

</dependency>

Output:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running MyServiceTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

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();

}

}

Solution:

ExternalApi.java:

public interface ExternalApi {

String getData();

}

MyService.java;

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

MyServiceTest.java:

import org.junit.jupiter.api.Test;

import static org.mockito.Mockito.\*;

public class MyServiceTest {

@Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

}

Output:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running MyServiceTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0