JUnit Testing Exercises

**Exercise 1:** Setting Up JUnit

**Scenario:** You need to set up JUnit in your Java project to start writing unit tests.

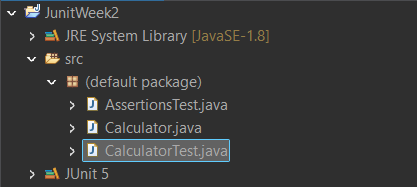
**Steps:**

1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).

2. Add JUnit dependency to your project.

3. Create a new test class in your project.

**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:

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

*@Test*

public void testAssertions() {

*assertEquals*(5, 2 + 3);

*assertTrue*(5 > 3);

*assertFalse*(5 < 3);

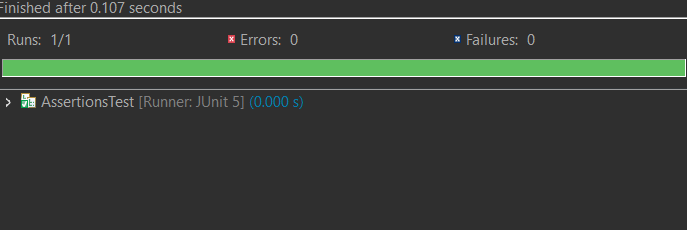
*assertNull*(null);

*assertNotNull*(new Object());

}

}

**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.

**Steps:**

1. Write tests using the AAA pattern.

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

**SOLUTION:**

//Create a java class file calculator.java  
public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int multiply(int a, int b) {

return a \* b;

}

}

//Create a test file

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

public class CalculatorTest {

private Calculator calculator;

*@Before*

public void setUp() {

calculator = new Calculator();

System.***out***.println("Setup completed.");

}

*@After*

public void tearDown() {

calculator = null;

System.***out***.println("Teardown completed.");

}

*@Test*

public void testAddition() {

int result = calculator.add(2, 3);

*assertEquals*(5, result);

}

*@Test*

public void testMultiplication() {

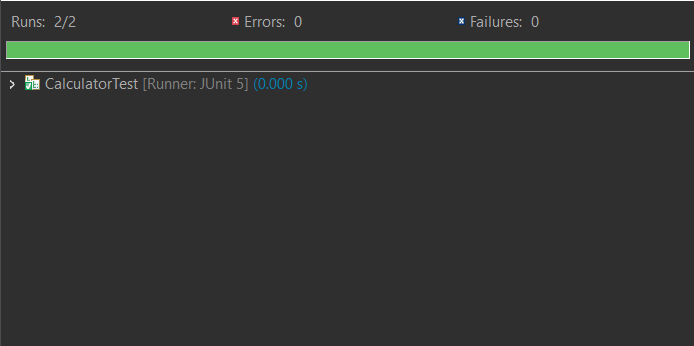
int result = calculator.multiply(4, 5);

*assertEquals*(20, result);

}

}

**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.

**SOLUTION:**

//Create interface ExternalApi.java

public interface ExternalApi {

String getData();

}

Create class MyService.java

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

}

Create MyServiceTest.java

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

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);

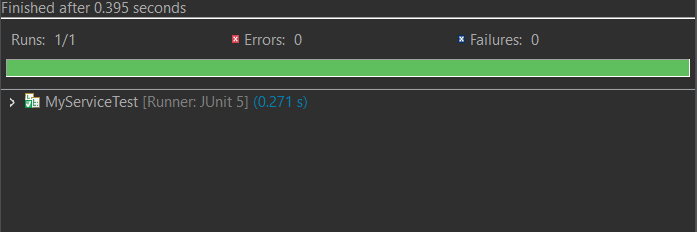
String result = service.fetchData();

*assertEquals*("Mock Data", result);

}

}

**OUTPUT:**



**Exercise 2:** Verifying Interactions

**Scenario:** You need to ensure that a method is called with specific arguments.

//Edit MyServiceTest.java

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

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();

}

}

**OUTPUT:**