**JUnit\_Basic 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>

1. Create a new test class in your project.

**Step 1:** 1. Creating a new java project in the Eclipse.

File > New > Maven project

* 1. Select Create a simple project then Group id and Artifact id.
  2. Click on Finish.

**Step 2:**  Add JUnit dependency to your project.

Open src then there will be pom.xml in that I need to install Junit the code is following, insert the code in pom.xml for installation of Junit.

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

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

For the we need create a class and a test the project to test Junit.

1. Creating a class > Go to src/main/java > right click >new>class>give class name then finish.

//this the class which I have created in src/main/java, class name is Calculator

package JUnitDemo; public class Calculator { public int add (int a,int b) {

return a+b;

}

}

1. For testing the class Calculator, we need to have another class.

Right click on src/test/java > new > class

Name that class as CalculatorTest to test the Calculator using variables, methods, objects and constructors. //to test the project import org.junit.Test; import JUnitDemo.Calculator; import static org.junit.Assert.\*; public class CalculatorTest {

@Test

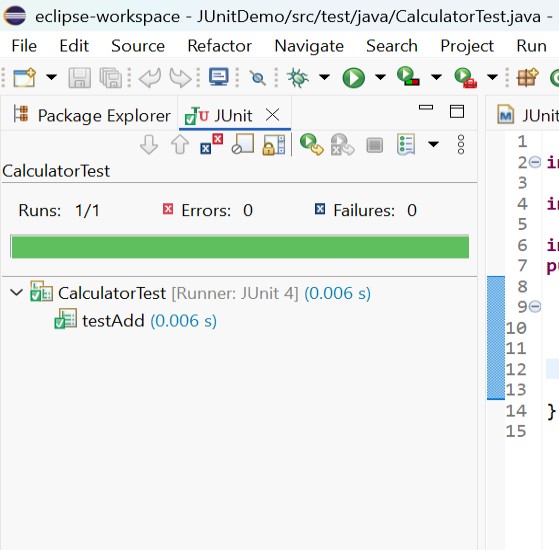
public void testAdd() {

Calculator cal=new Calculator(); int result=cal.add(3, 3); *assertEquals*(6,result);

}

}

**Output:**



The code has test successfully without errors using Junit Testing and this all about setting up Junit.

**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 ());

}

}

**Explanation:**

Here we are using different Asserts

The explanation of the Asserts that are using for the testing Junit.

1. assertEquals(5,2+3) Checks of 2+3 is equals to 5. So, the result will Pass.
2. assertTrue(5>3) >> Checks the condition is True. So, the result will Pass.
3. assertFalse(5<3) >> Checks the condition is False. So, the result will Pass.
4. assertNull(null) >> Checks whether the given condition is null. Then the result will Pass.
5. assertNotNull(new Object ()) >> Ensures the Object is Null. Then the result will Pass.

If we test the Solution code it will execute without errors because all the conditions satisfy so all the results of asserts will pass.

**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.

**Explanation:**

**AAA Pattern:** AAA pattern stands for **Arrange-Act-Assert** Patterns. It is very important and makes testing much easier as it separates testing objectives, actions, and the results of the test.

**Arrange:** Set up the test environment.

**Act:** Execute the code to test.

**Assert:** Verify the results.

**Setup and Teardown Methods:**

**@Before -** Runs before every @Test method – useful for initializing common test data or objects

@**After -** Runs after every @Test method – used for cleanup (like closing connections, resetting data, etc.).

**Example Code:**

**Calculator.java**

Public class calculator {

Public int add (int a, int b) { return a+b;

}

}

**CalculatorTest.java** import org,junit.Before; import org.junit.After; import org.junit.Test; import JUnitDemo.Calculator; import static org.junit.Assert.\*; public class CalculatorTest { private Calculator calculator;

@Before

public void setup () {

//arrange: initialize before eact test

Calculator=new Calculator ();

System.out.println(“Setup before test”);

}

@After public void teardown () {

//cleanup after each test

System.out.println(“Teardown after test”);

} @Test public void testAddition() {

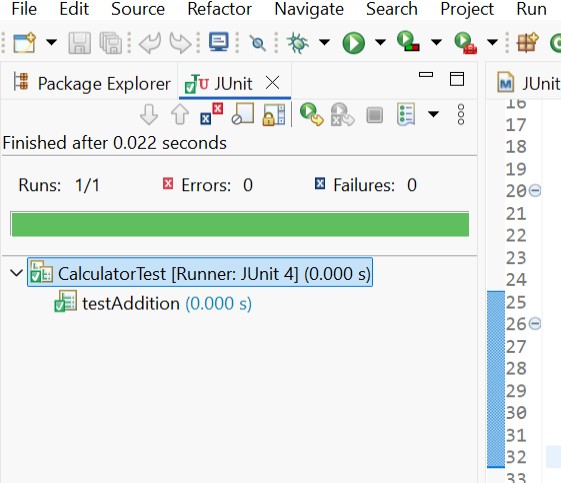
//act int result = calculator.add(10,5);

//assert assertEquals(15, result);

}

}

**Output:**



Setup before test

Teardown after test

# 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);

}

}

**Explanation:**

**Steps:**

1. **Create a mock object for the external API.**

Use Mockito.mock() to create a mock of the interface

Syntax: Externalapi mockApi = Mockito.mock(Externalapi.class);

For this I have created an interface for Externalapi.

//Externalapi package MockitoProject; public interface Externalapi {

String getData();

1. **Stub the methods to return predefined values.**

Tell the mock what to return when a method is called:

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

This means: whenever getData () is called on mockApi, it will return "Mock Data".

For another I have created a class named as Myservice:

//Myservice class for creating prefined values for both mock and stub methods which has already

given in test class. package MockitoProject; public class Myservice { private Externalapi externalapi; public Myservice(Externalapi externalapi) { this.externalapi = externalapi;

} public String fetchData() { return externalapi.getData();

}

}

**3. Write a test case that uses the mock object.**

The test case that has given is:

package MockitoProject; import static org.mockito.Mockito.\*; import static org.junit.jupiter.api.Assertions.\*; import org.junit.jupiter.api.Test; public class MyserviceTest {

@Test public void testExternalapi() {

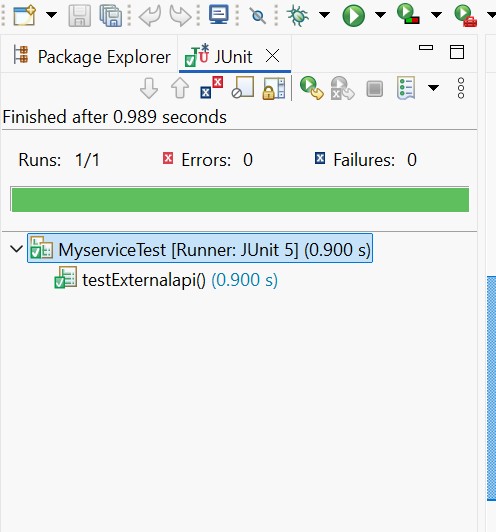
Externalapi mockApi = *mock* (Externalapi.class); *when* (mockApi.getData()).thenReturn("Mock Data");

Myservice service = new Myservice(mockApi); String result = service.fetchData(); *assertEquals*("Mock Data", result);

}

Output:

I have tested the code that has executed without errors, below is the reference.



**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();

}

**Steps:**

1. **Create a mock object:**

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

* + ExternalApi is mocked using Mockito.
  + This mock will simulate the behavior of a real ExternalApi without making actual external calls.

1. **Inject the mock into the service:**

MyService service = new MyService(mockApi);

* + The mock is passed to the MyService constructor.
  + This allows us to control and verify the interaction between MyService and ExternalApi.

1. **Call the method under test:**

service.fetchData();

* + This is the method we want to test.
  + Internally, it is expected to call getData() on the ExternalApi.

1. **Verify the interaction:**

verify(mockApi).getData();

* + This line verifies that the getData() method was called exactly once on the mock object.
  + If this call is missing or incorrect, the test will fail.
  + This pattern is useful when you want to test interactions between classes, not just return values.
  + Mockito's verify () is commonly used in unit tests to ensure the behavior of a method, especially when there’s no return value to assert.

1. **Purpose of Verification:**

* + Ensures that MyService is correctly using its dependency (ExternalApi).
  + Helps catch issues in logic, especially when dealing with side effects or external systems**.**

1. **Expected Output:**

* + The test will pass if getData() is called during the execution of fetchData().
  + The test will fail if getData() is not called or if it’s called with incorrect arguments.

# 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>

1. 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");

}

}

**Explanation:**

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<!-- Logback Implementation -->

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

After keeping this in the pom.xml the slf4j will be installed.

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");

} }

Creating a java class naming it as Loggingexample with the given solution.

And executing it with java application, below is the output of the given solution.

**Output:**

