Name : K. Taraka Lakshmi Prasanna

Superset ID : 6314755

Email : 22BQ1A4268@vvit.net

Week – 2

Topic : TDD Using Junit and Mockito and SLF4J

Junit Basic Testing :

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: junit junit 4.13.2 test

3. Create a new test class in your project

Solution:

To create a new Java Project in Intellij IDEA, Navigate to File > New > Project.And then choose Maven ,Give a name to ypur project and click finish.

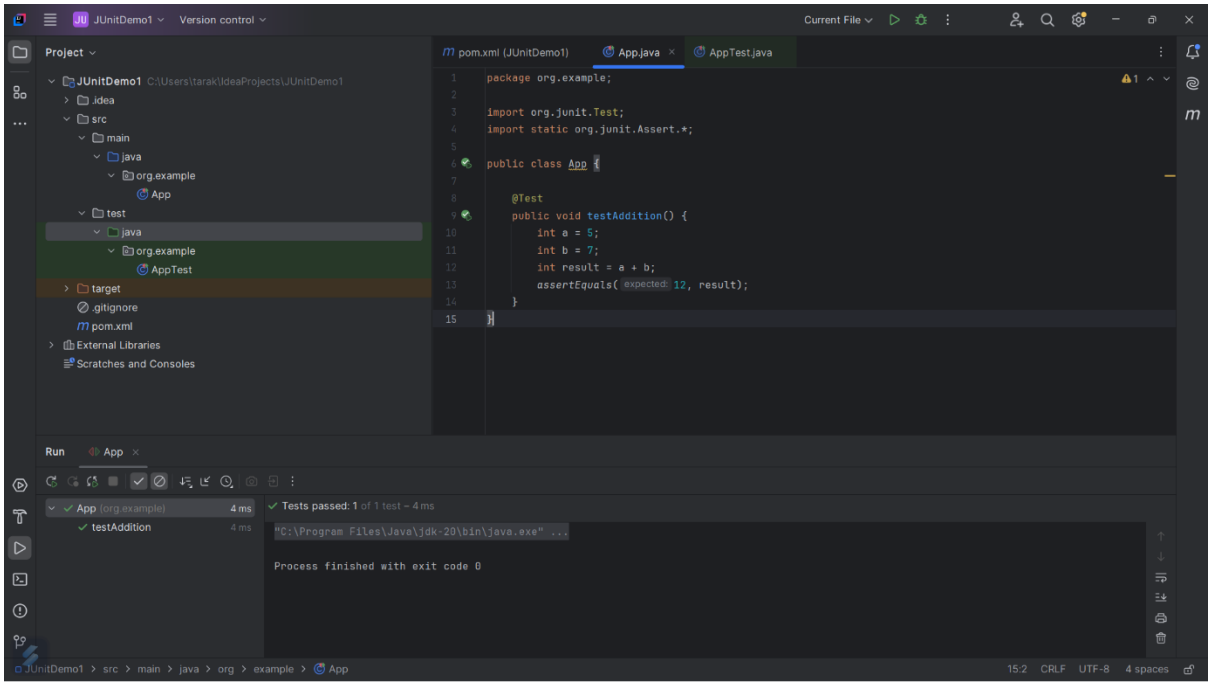
Add Junit dependency code in pom.xml file

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Save the file.Maven will automatically download Junit library.

Inside src/test/java/ we have AppTest file change the file according to your tests

Run the tests by clicking on the run button.



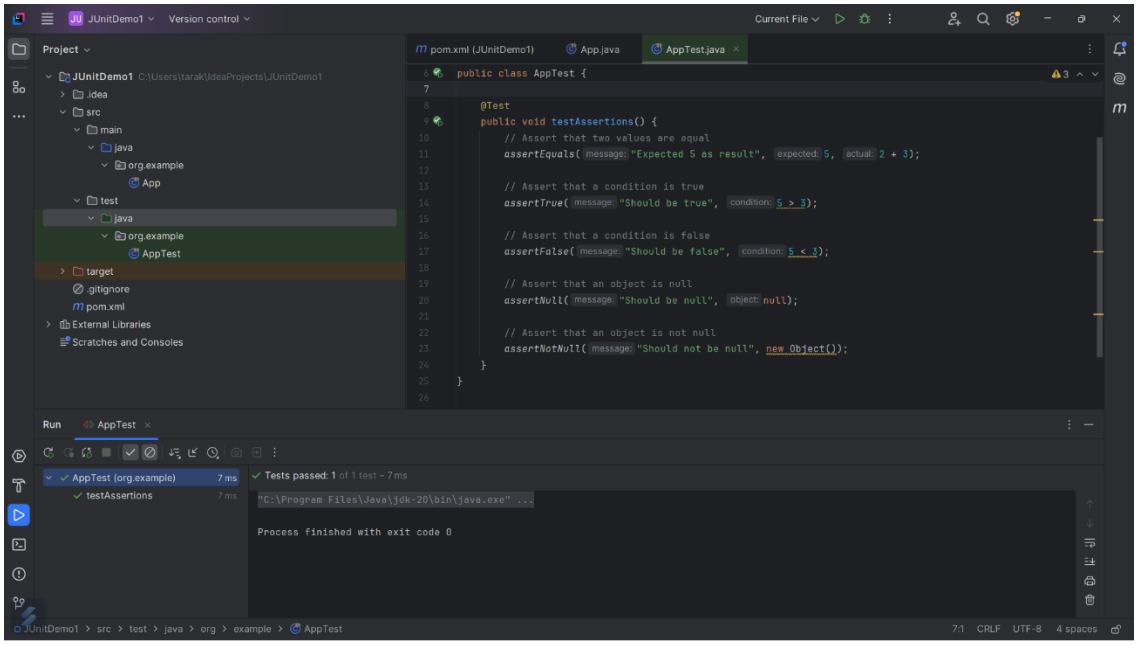
Exercise 2: Assertions in JUnit

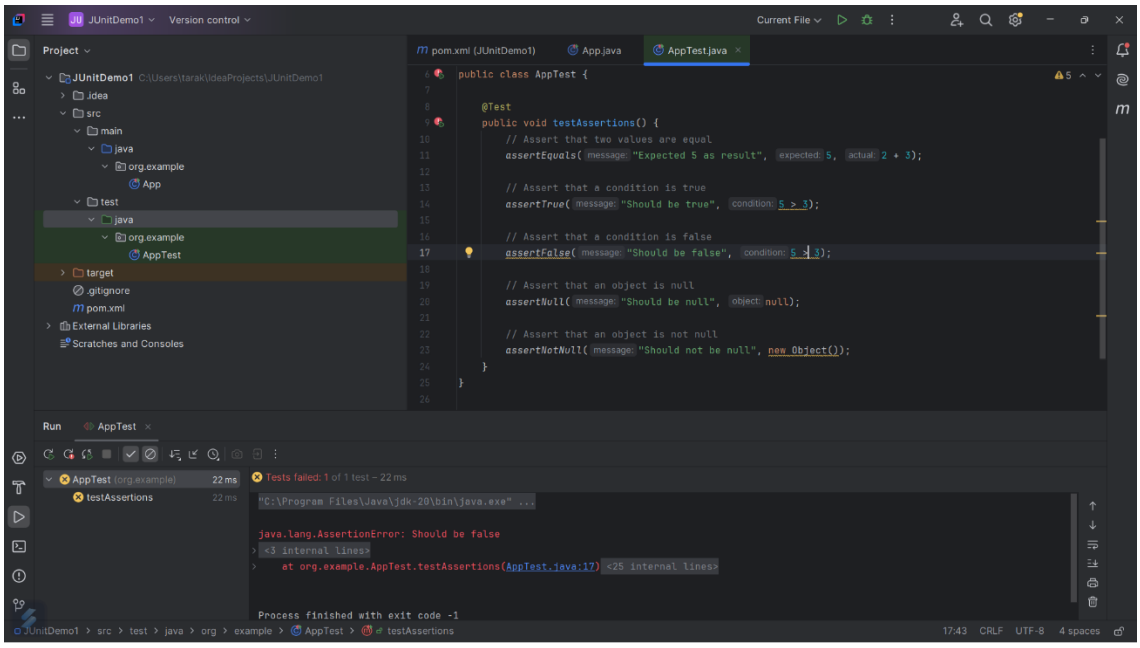
Scenario: You need to use different assertions in JUnit to validate your test results.

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());
    }
}
```

Assertion	Purpose
assertEquals()	Compares expected and actual values
assertTrue()	Passes if condition is true
assertFalse()	Passes if condition is false
assertNull()	Passes if object is null
assertNotNull()	Passes if object is **not** 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.

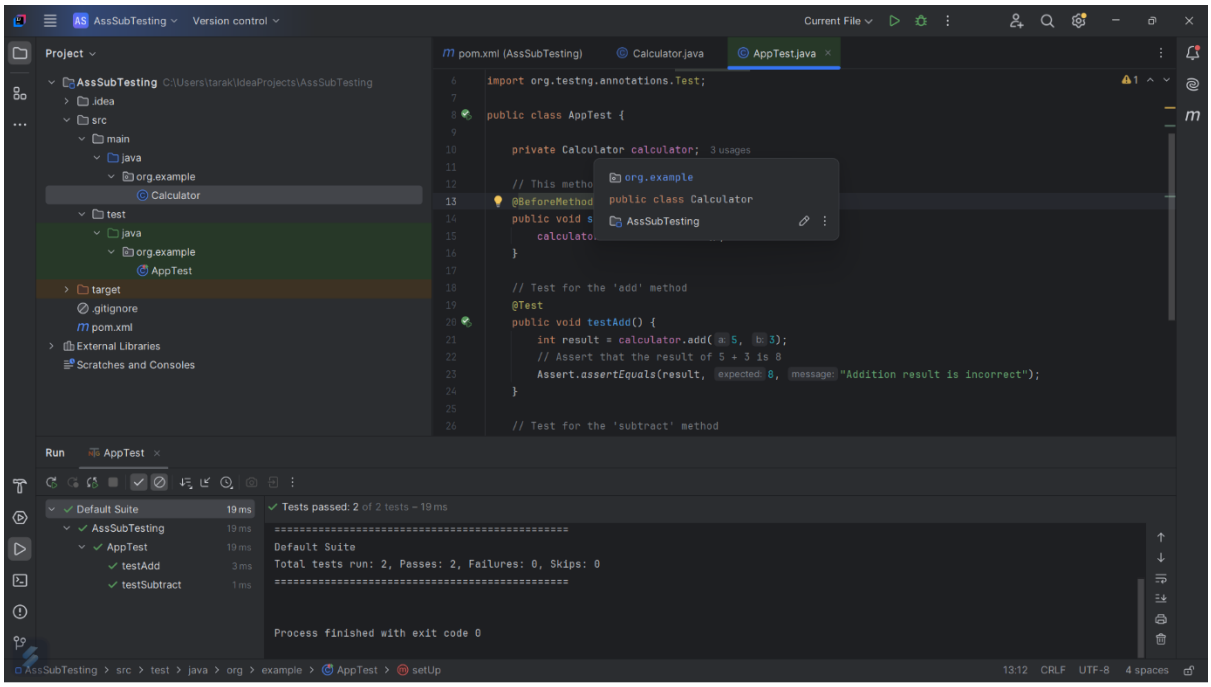
Solution :

The **Arrange-Act-Assert (AAA)** pattern structures test code into 3 clear steps:

1. **Arrange**: Set up the test objects and prepare the environment.
2. **Act**: Call the method being tested.
3. **Assert**: Verify that the result is as expected.

What are @Before and @After?

- @Before → runs **before each test**. Good for initializing objects.
- @After → runs **after each test**. Good for cleanup (e.g., closing resources).



Mockito:

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 :

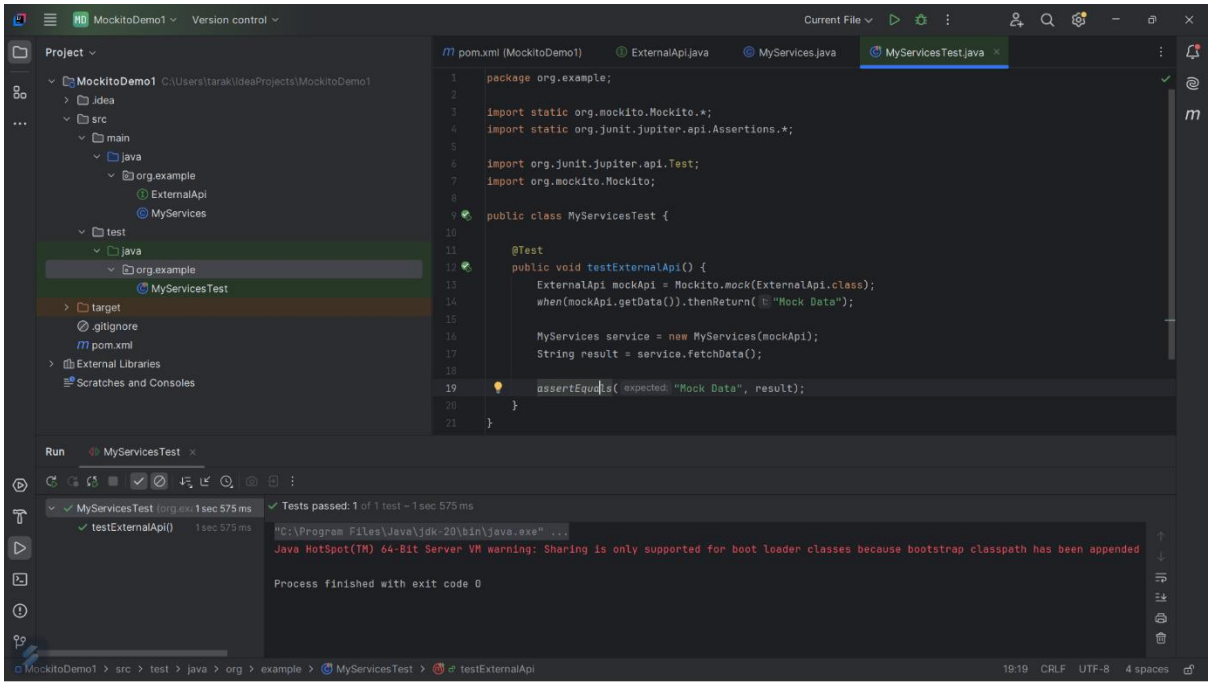
Mocking is the act of creating **fake (dummy) objects** that mimic the behavior of real objects.

You use mocks when:

- The real object is **external** (like a database, API, or file system).
- The real object is **slow**, **unpredictable**, or **expensive** to use.
- You want to **test logic in isolation** without depending on external systems.

Stubbing is telling a mock **what to return** when a specific method is called.

Concept	Definition	Example in Mockito
Mocking	Creating a fake object to simulate a real one	Mockito.mock(MyClass.class)
Stubbing	Defining what the mock should return when a method is called	when(mock.method()).thenReturn(value)



Exercise 2: Verifying Interactions

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

Steps:

1. Create a mock object.

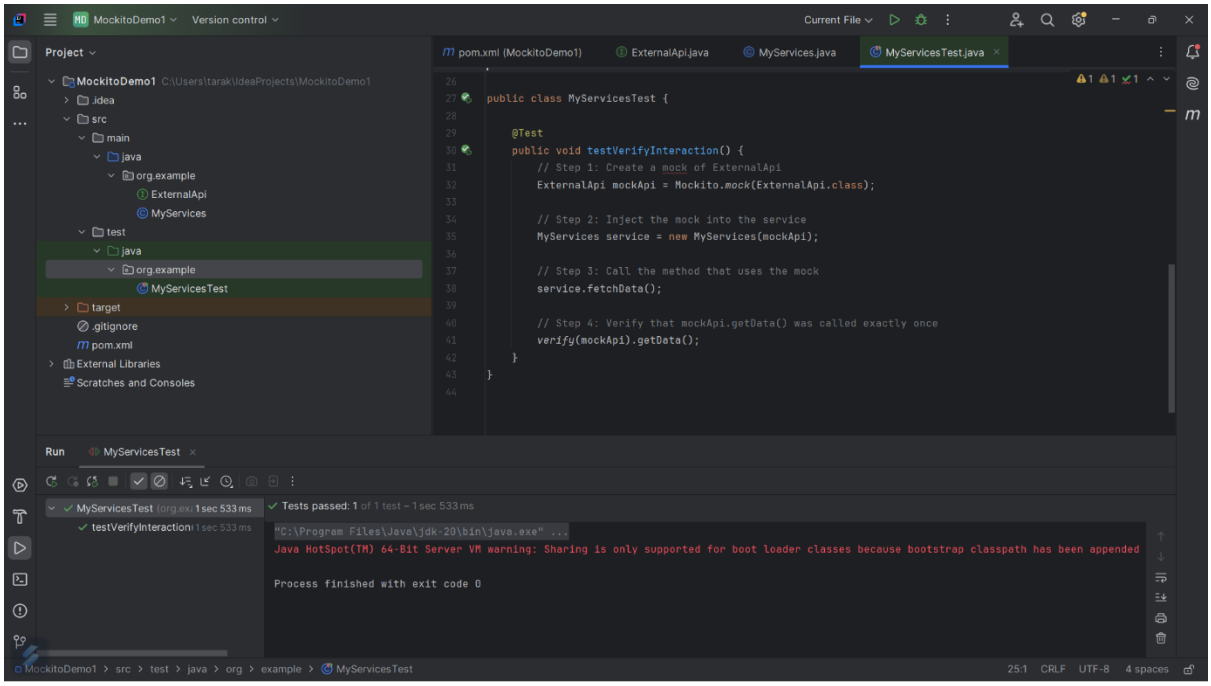
2. Call the method with specific arguments.

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();
    }
}
```



Logging Using SL4J:

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>

<!-- Logback Implementation -->
<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");

    }

}
```

SLF4J is a simple logging facade for Java that allows switching between different logging frameworks without changing application code.