**WEEK 2-TDD USING JUNIT5 AND MOCKITO 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.

# Solution Code Calculator.java java

Copy code

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

# CalculatorTest.java

java

Copy code

package com.example;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

@Test

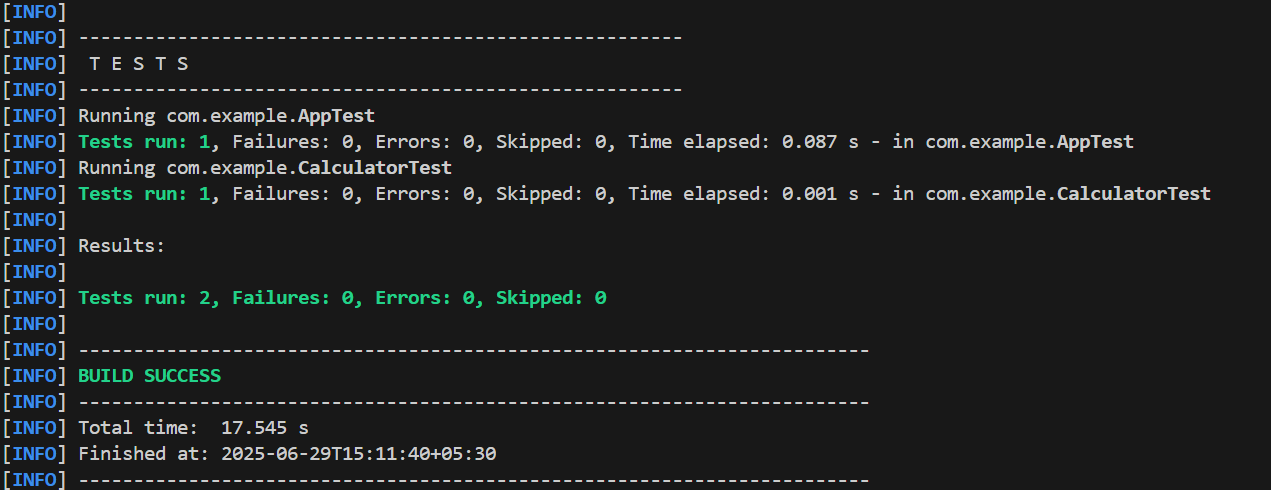
public void testAdd() {

Calculator calculator = new Calculator(); int result = calculator.add(3, 2); assertEquals(5, result);

}

}

**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:

**AssertionsTest.java**

public class AssertionsTest { @Test public void

testAssertions() {

assertEquals(5, 2 + 3);

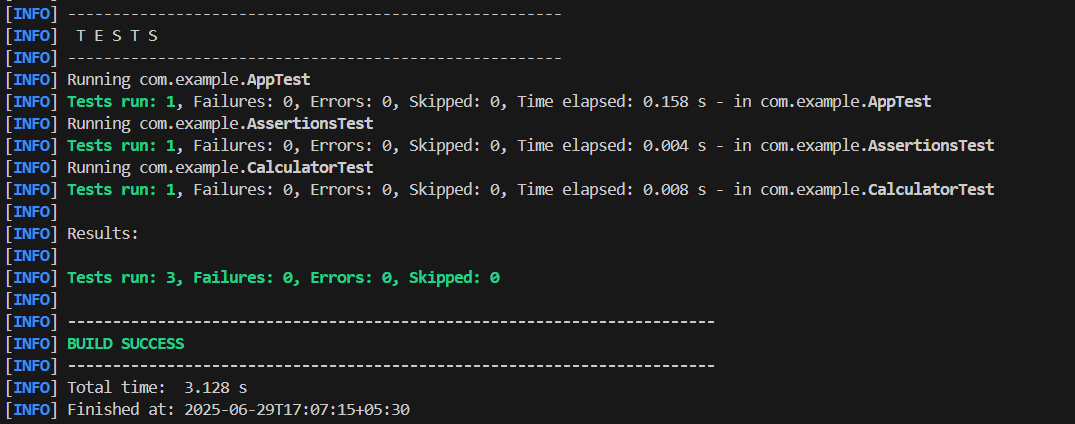
assertTrue(5 > 3);

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 CODE

# Calculator.java

package com.example; public class Calculator {

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

}

public int subtract(int a, int b) { return a - b;

}}

# CalculatorFixtureTest.Java

package com.example;

import org.junit.After; import org.junit.Before; import org.junit.Test;

import static org.junit.Assert.\*; public class CalculatorFixtureTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator(); System.out.println("Setup done.");

}

@After

public void tearDown() { calculator = null;

System.out.println("Teardown done.");

}

@Test

public void testAddition() {

int result = calculator.add(10, 5); assertEquals(15, result);

}

@Test

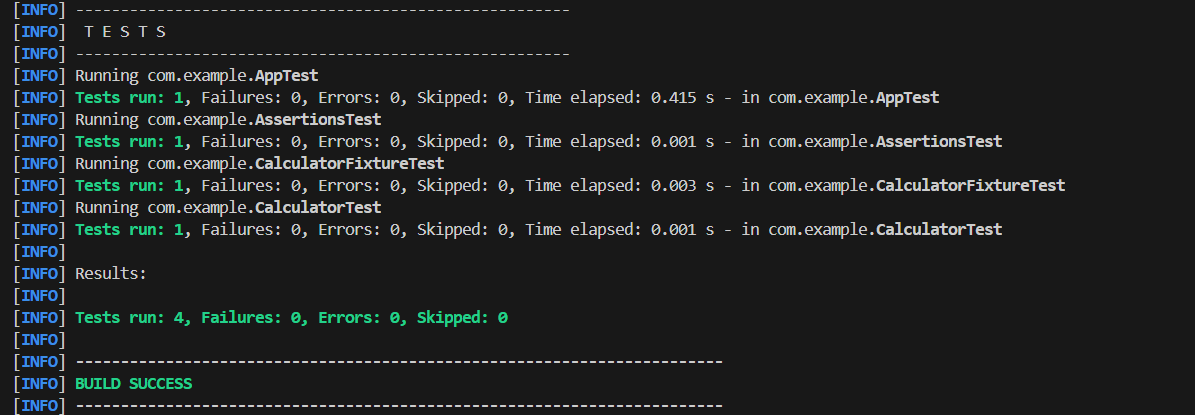
public void testSubtraction() {

int result = calculator.subtract(8, 3); assertEquals(5, 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.

# 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 ExternalApi.java

package com.example;

public interface ExternalApi { String getData();

}

# MyService.java

package com.example; public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) { this.api = api;

}

public String fetchData() { return api.getData();

}

}

# MyServiceTest.java

package com.example;

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

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

public class MyServiceTest { @Test

public void testExternalApi() {

ExternalApi mockApi = Mockito.mock(ExternalApi.class); when(mockApi.getData()).thenReturn("Mock Data");

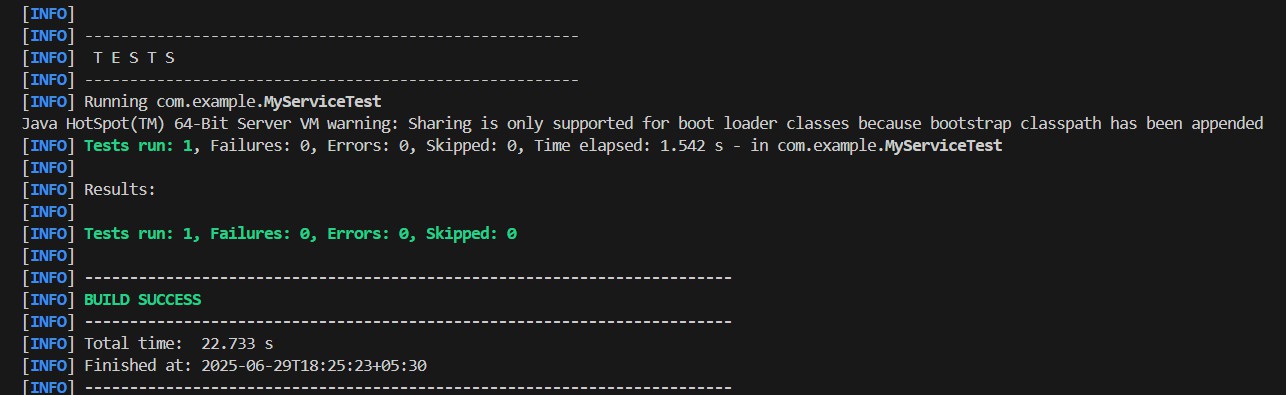
MyService service = new MyService(mockApi); String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

**OUTPUT**

****

**EXERCISE 2: VERIFYING INTERACTIONS**

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

# Steps:

1. Create a mock object.
2. Call the method with specific arguments.
3. Verify the interaction.

# Solution Code:

package com.example;

import static org.mockito.Mockito.\*; import org.junit.jupiter.api.Test;

public class MyServiceTest { @Test

public void testVerifyInteraction() {

ExternalApi mockApi = mock(ExternalApi.class);

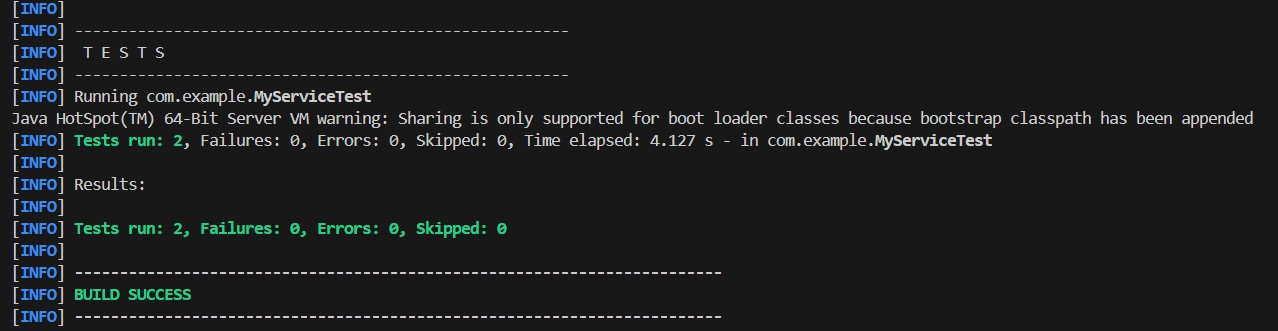
MyService service = new MyService(mockApi); service.fetchData();

verify(mockApi).getData();

}

}

**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>

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");

} }

**OUTPUT**

****