**TDD using JUnit5 and Mockito**

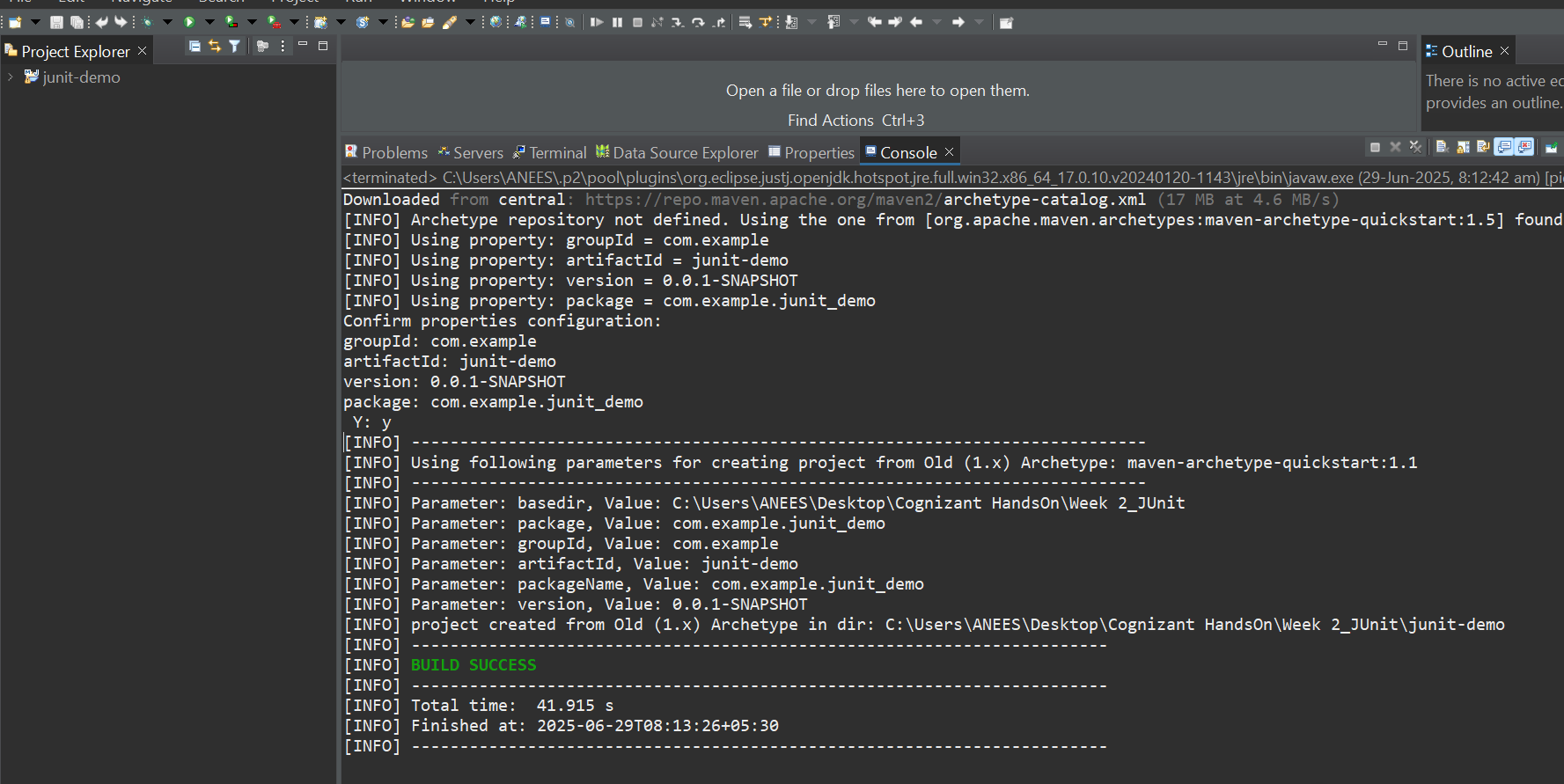
**Exercise 1: Setting Up JUnit**

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 in Eclipse.



1. Add JUnit dependency to your project. If you are using Maven, add the following dependency to your pom.xml.

<dependency>

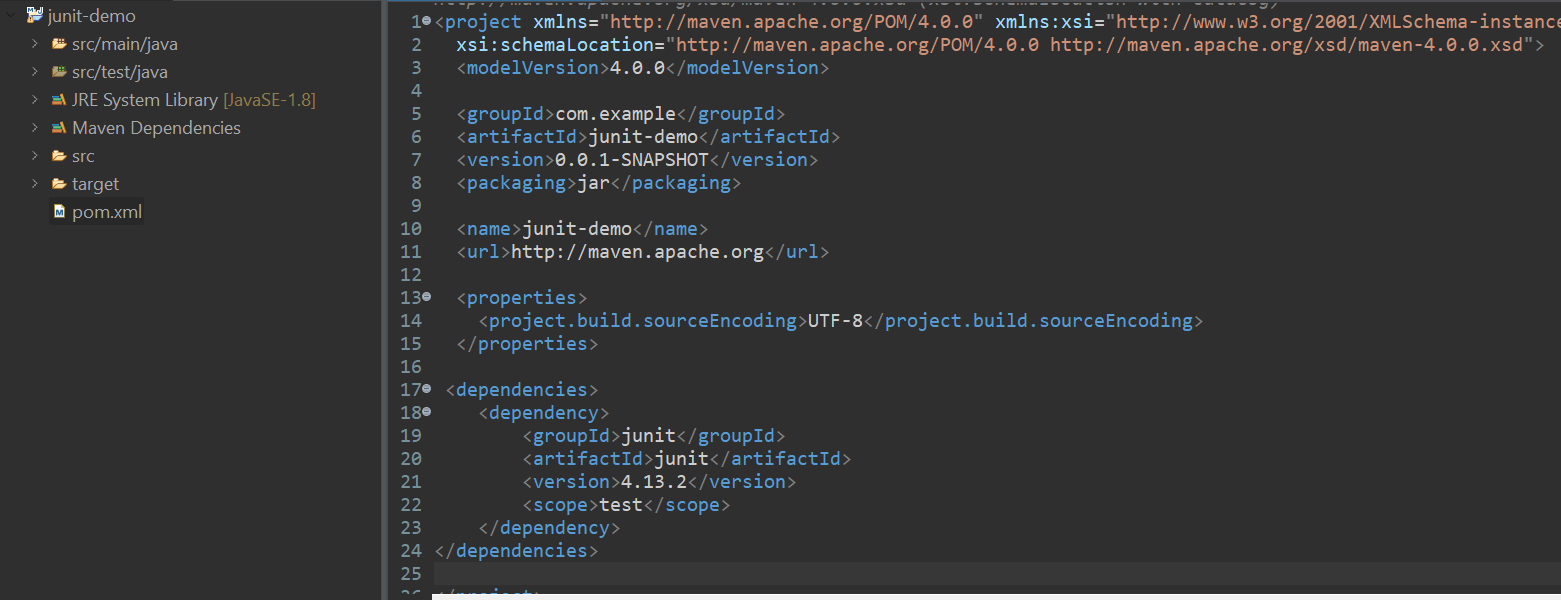
<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

 3.Create a new test class in your project.

package com.example.junit\_demo;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

int result = 5 + 3;

assertEquals(8, result);

}

@Test

public void testSubtraction() {

int result = 10 - 4;

assertEquals(6, result);

}

@Test

public void testMultiplication() {

int result = 6 \* 7;

assertEquals(42, result);

}

@Test

public void testDivision() {

int result = 20 / 5;

assertEquals(4, result);

}

@Test(expected = ArithmeticException.class)

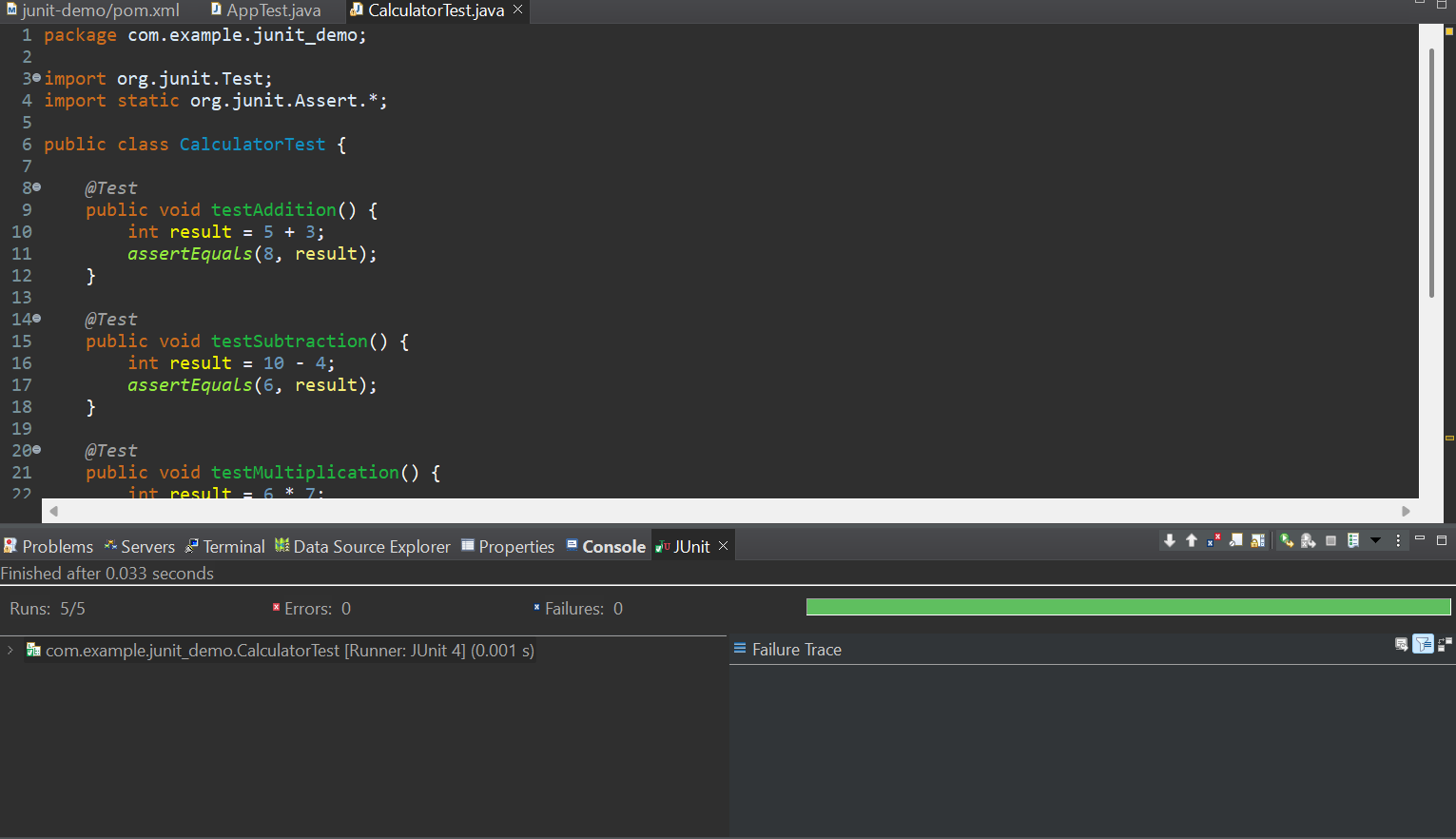
public void testDivideByZero() {

int result = 10 / 0; // This should throw ArithmeticException

}

}

**Output:**



**Exercise 2: Writing Basic JUnit Tests**

Scenario: You need to write basic JUnit tests for a simple Java class.

**Steps:**

1. Create a new Java class with some methods to test.

Calculator.java

package com.example.junit\_demo;

public class Calculator

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new ArithmeticException("Cannot divide by zero!");

}

return a / b;

}

}

2. Write JUnit tests for these methods.

CalculatorTest.java

package com.example.junit\_demo;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calculator = new Calculator();

*@Test*

public void testAdd() {

*assertEquals*(8, calculator.add(5, 3));

}

*@Test*

public void testSubtract() {

*assertEquals*(2, calculator.subtract(5, 3));

}

*@Test*

public void testMultiply() {

*assertEquals*(15, calculator.multiply(3, 5));

}

*@Test*

public void testDivide() {

*assertEquals*(4, calculator.divide(20, 5));

}

*@Test*(expected = ArithmeticException.class)

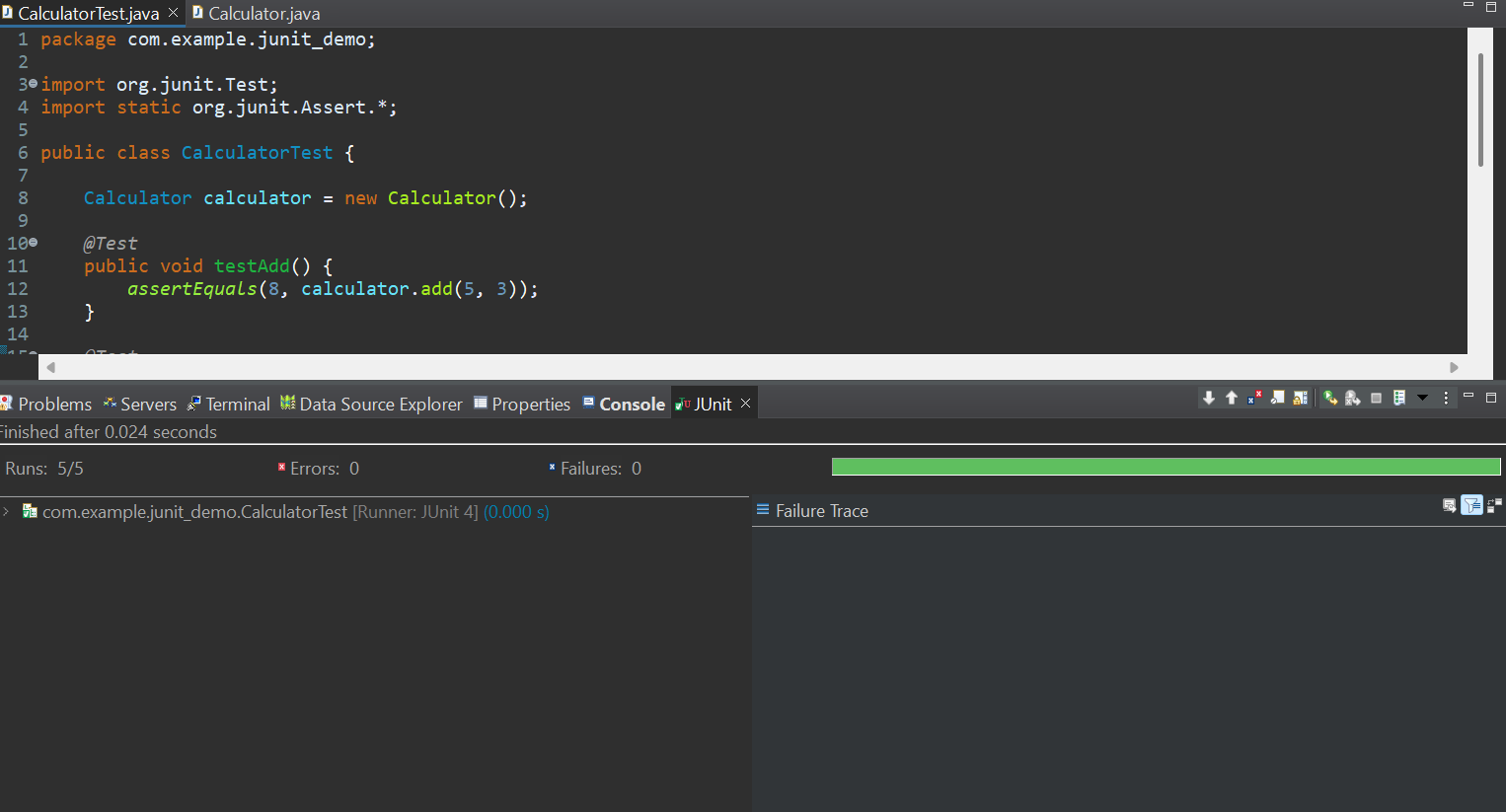
public void testDivideByZero() {

calculator.divide(10, 0);

}

}

**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.

AssertionsTest.java

package com.example.junit\_demo

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest{

@Test

public void testVariousAssertions() {

// assertEquals: Check string length

String word = "hello";

assertEquals("Length should be 5", 5, word.length());

// assertTrue: Check if string contains a substring

assertTrue("Should contain 'ell'", word.contains("ell"));

// assertFalse: Check if string is not empty

assertFalse("String should not be empty", word.isEmpty() == true);

// assertNull: Check null for uninitialized object

Integer number = null;

assertNull("Number should be null", number);

// assertNotNull: Check object instantiation

Integer age = Integer.valueOf(25);

assertNotNull("Age should not be null", age);

// assertArrayEquals: Compare string arrays

String[] expectedFruits = {"Apple", "Banana", "Cherry"};

String[] actualFruits = {"Apple", "Banana", "Cherry"};

assertArrayEquals("Fruit arrays should be equal", expectedFruits, actualFruits);

// assertSame: Same reference test for Integer cache

Integer x = 100;

Integer y = 100;

assertSame("x and y should refer to the same Integer object (Java Integer caching)", x, y);

// assertNotSame: Different object instances with same value

String city1 = new String("Delhi");

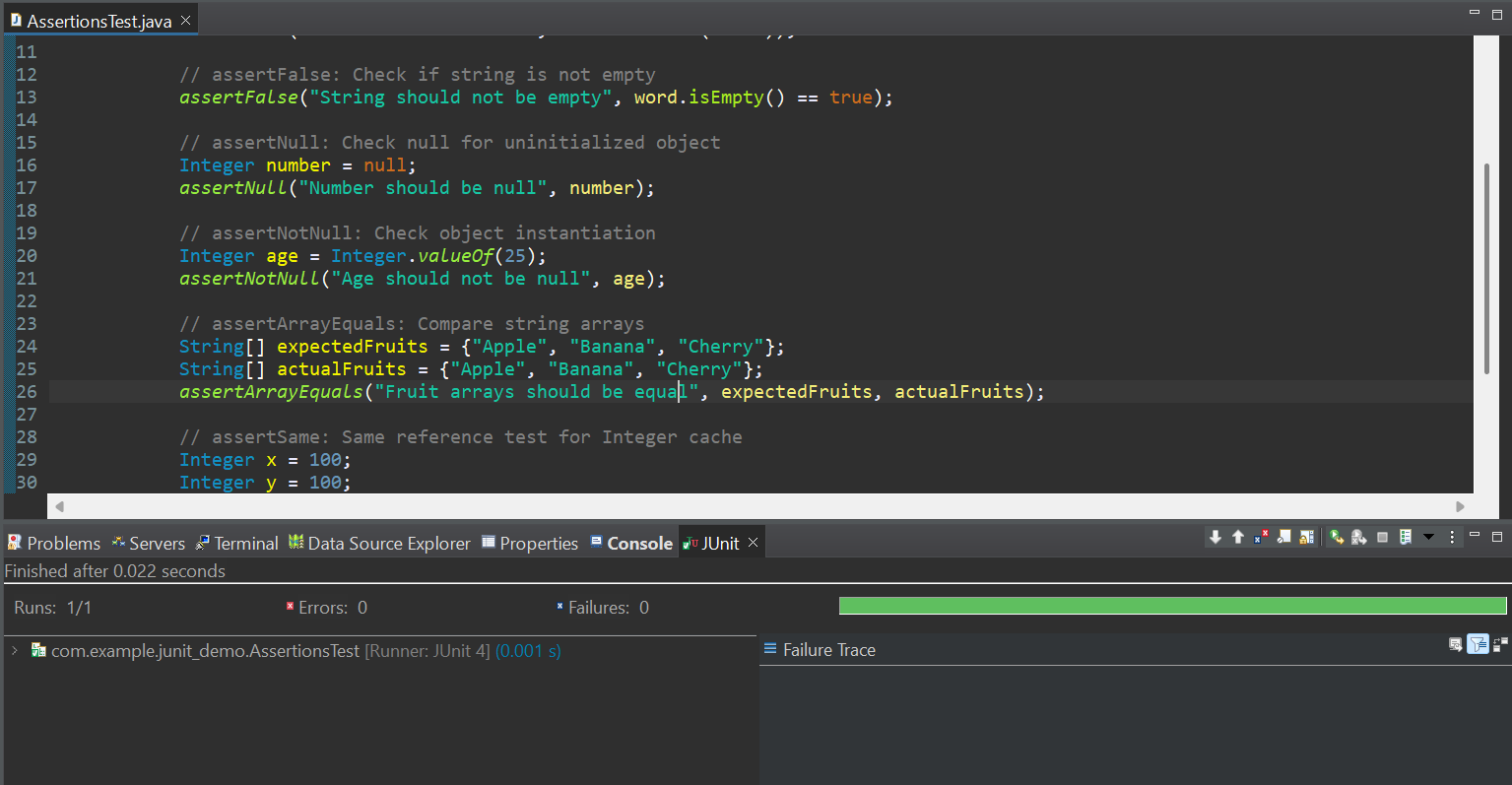
String city2 = new String("Delhi");

assertNotSame("city1 and city2 are different objects", city1, city2);

}

}

**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.

MessageService.java

package com.example.junit\_demo;

public class MessageService {

public String getWelcomeMessage(String name) {

return "Welcome, " + name + "!";

}

public String getGoodbyeMessage(String name) {

return "Goodbye, " + name + "!";

}

}

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

MessageServiceTest.java

package com.example.junit\_demo;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MessageServiceTest {

MessageService service;

@Before

public void setUp() {

service = new MessageService();

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

}

@After

public void tearDown() {

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

}

@Test

public void testWelcomeMessage() {

// Arrange

String name = "Anees";

// Act

String result = service.getWelcomeMessage(name);

// Assert

assertEquals("Welcome, Anees!", result);

}

@Test

public void testGoodbyeMessage() {

// Arrange

String name = "Anees";

// Act

String result = service.getGoodbyeMessage(name);

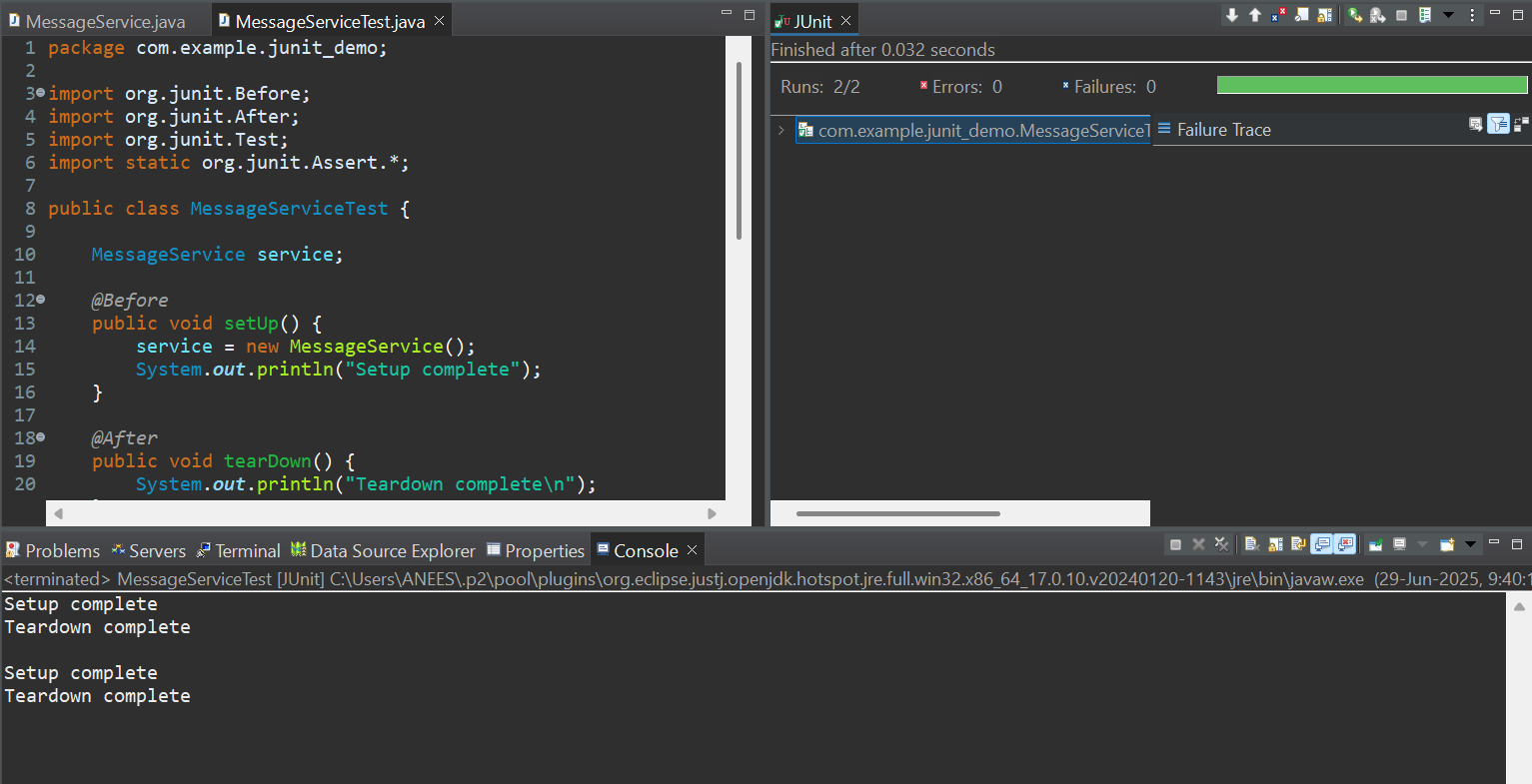
// Assert

assertEquals("Goodbye, Anees!", result);

}

}

**Output:**

****