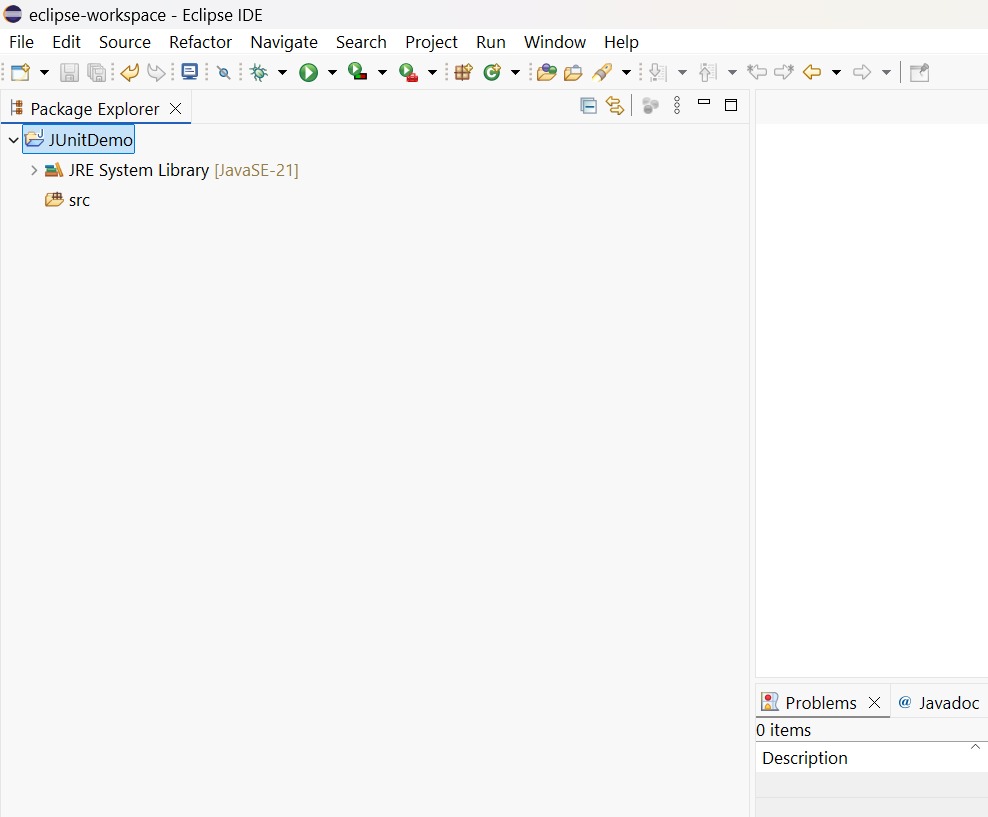
**JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit**

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

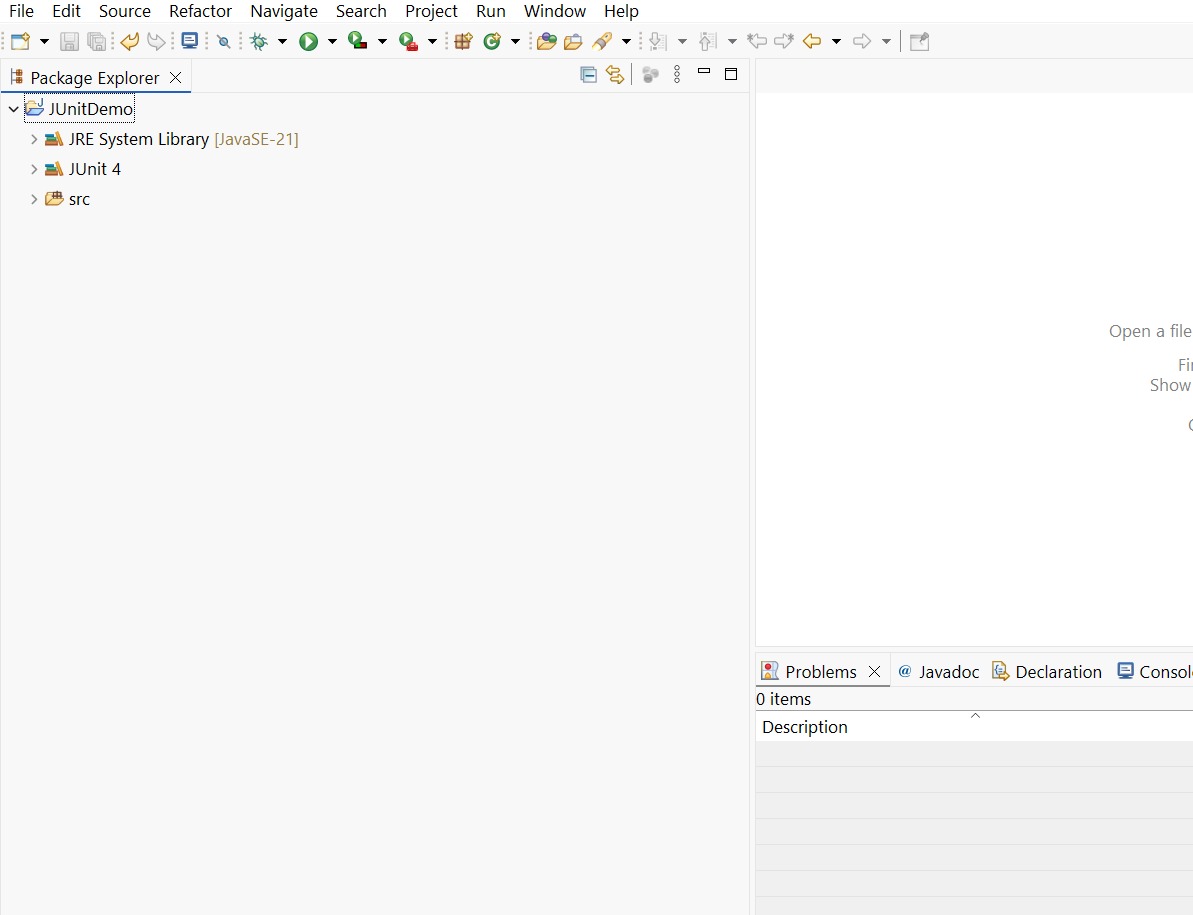
Steps:

**1. Create a new Java project**



**2. To set Junit in our java project**

1. Right-click the project → **Build Path → Configure Build Path**.
2. Click the **Libraries** tab → Click **Add Library…**
3. Select **JUnit** → Click **Next**.
4. Choose:
   * **JUnit 4** (recommended for beginners)
   * Or **JUnit 5** if you are familiar with newer syntax
5. Click **Finish**, then **Apply and Close**.

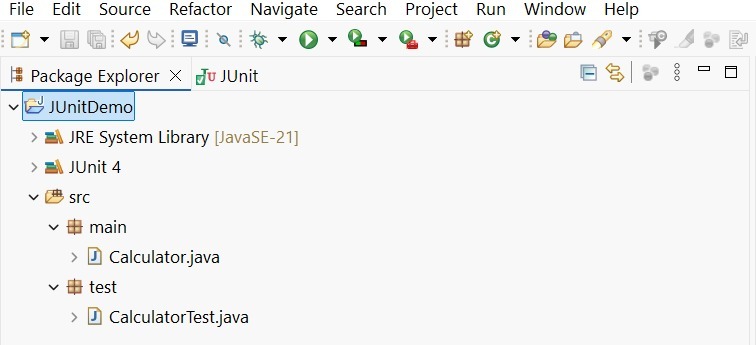


**Exercise 2: Writing Basic JUnit Tests**

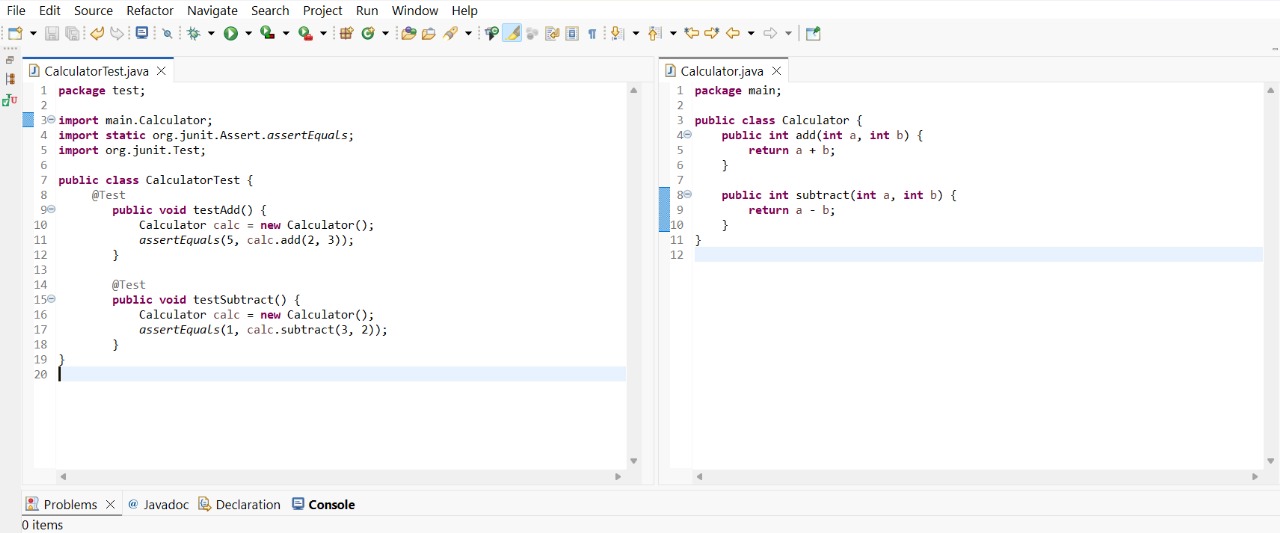
**Scenario**: You need to write basic JUnit tests for a simple Java class.

Steps:

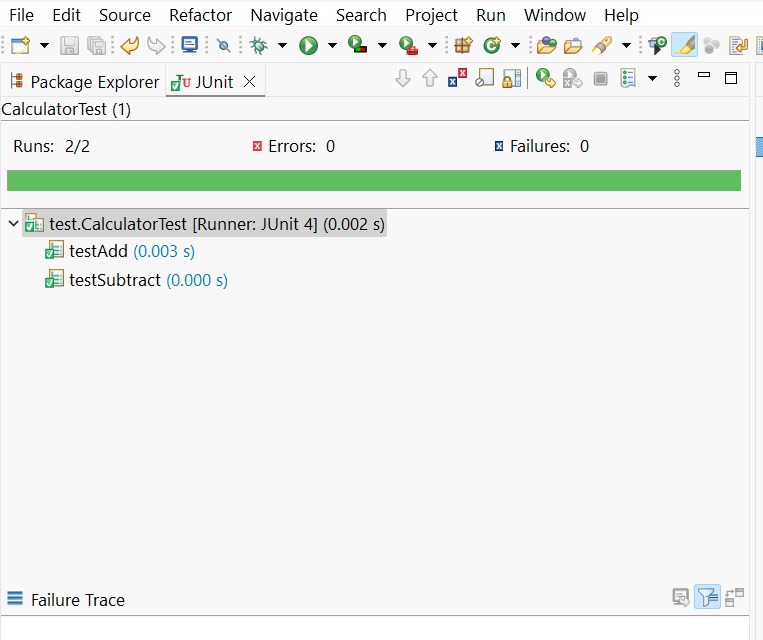
1. **Create a new Java class with some methods to test.**



Code



Output

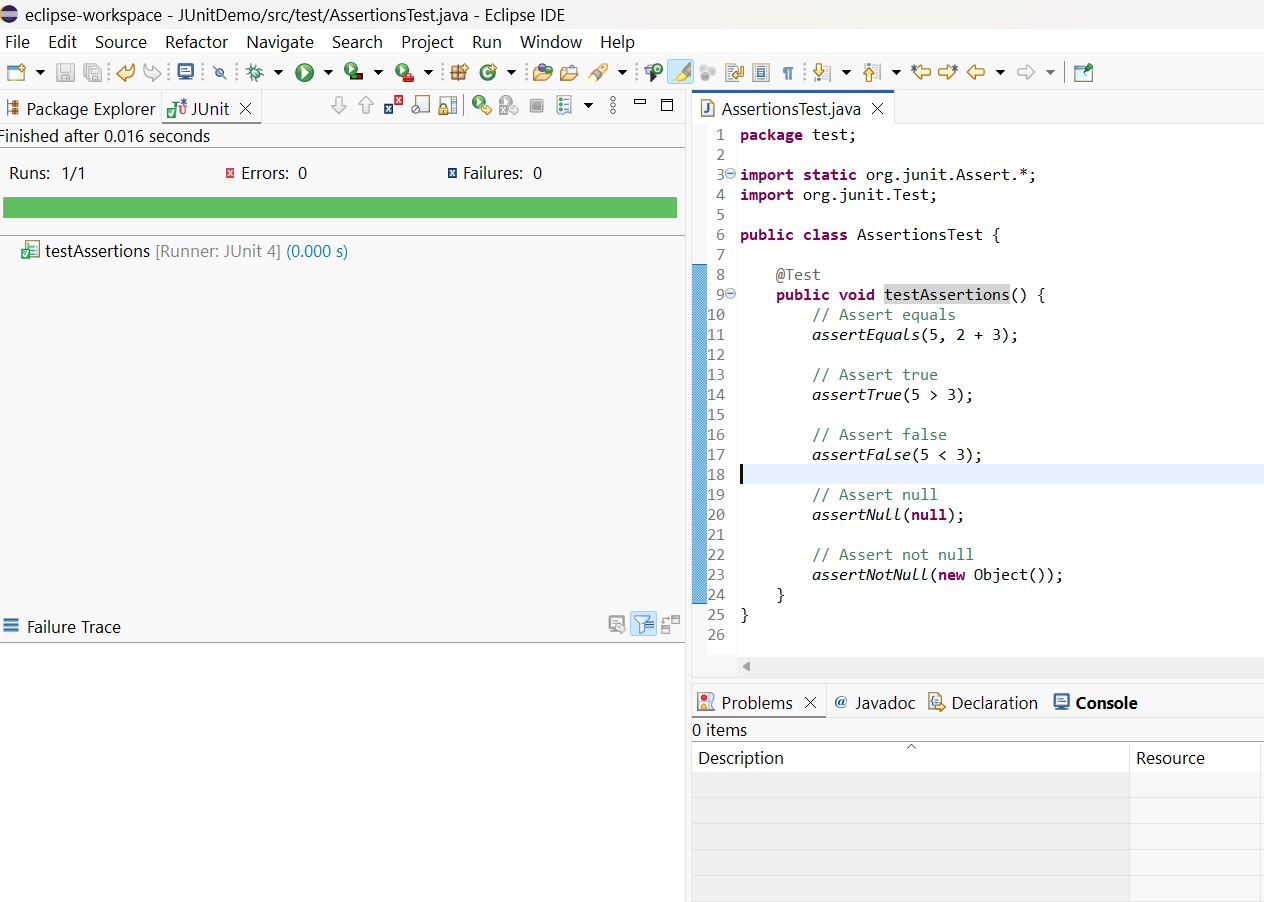


**Exercise 3: Assertions in JUnit**

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

Steps:

**1. Write tests using various JUnit assertions.**

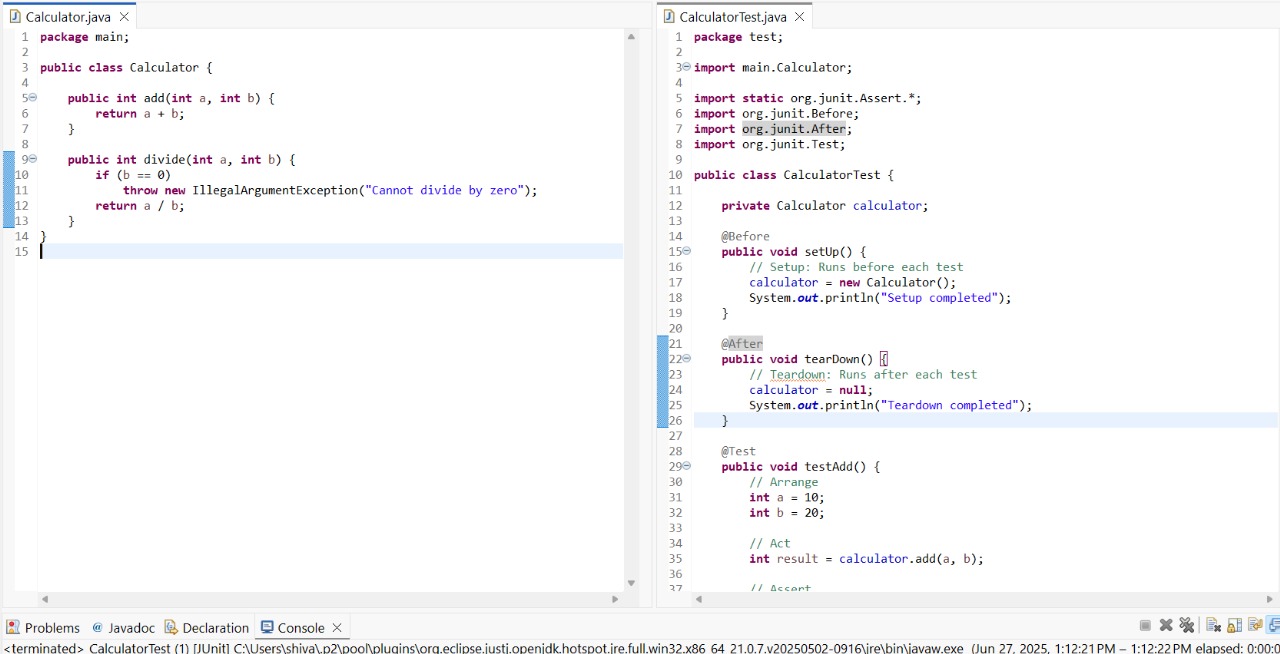


**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit**

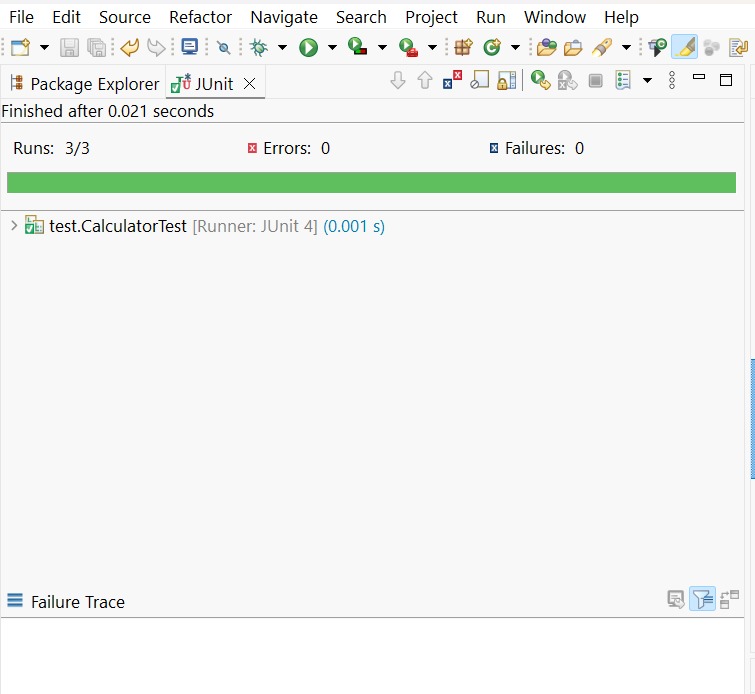
**Scenario**: You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.

Steps:

1. **Write tests using the AAA pattern.**



Output



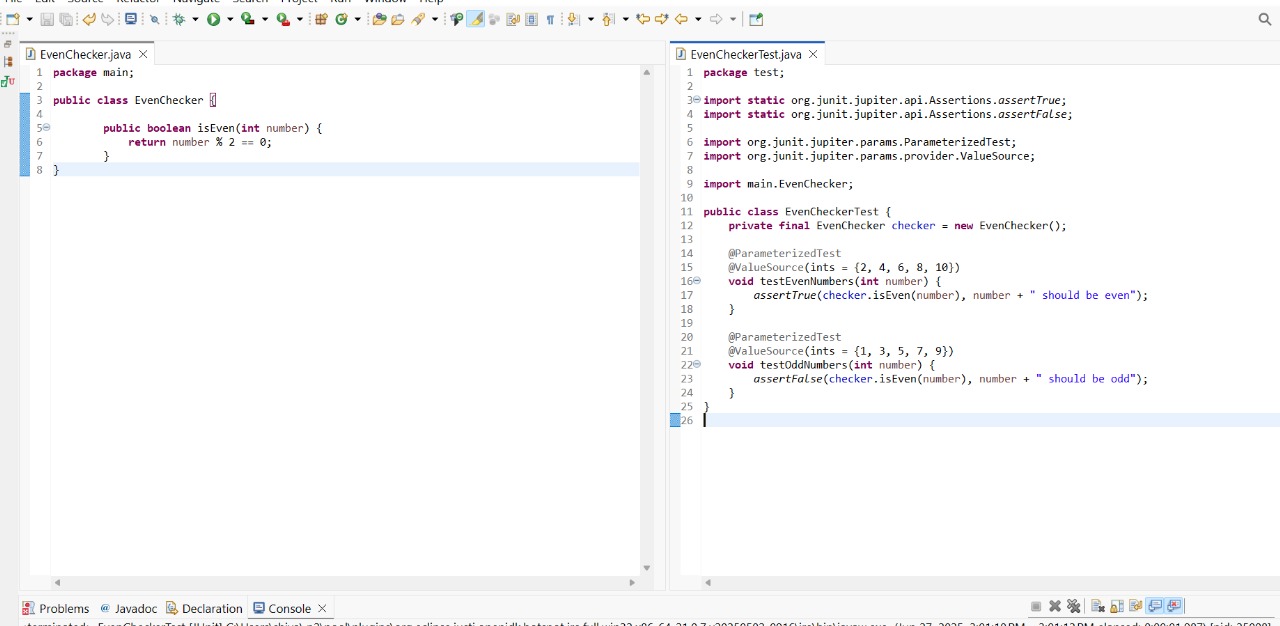
**Advanced JUnit Testing Exercises**

**Exercise 1: Parameterized Tests**

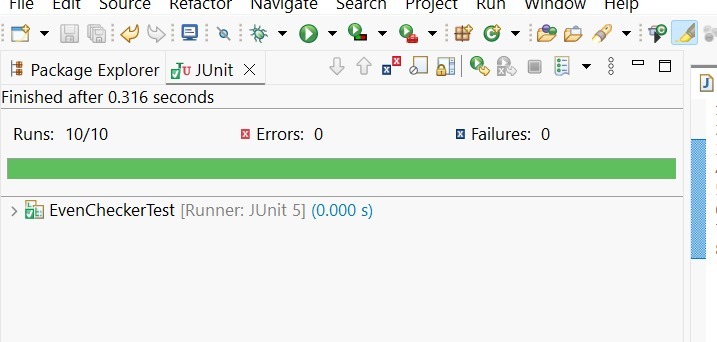
**Scenario:** You want to test a method that checks if a number is even. Instead of writing multiple test cases, you will use parameterized tests to run the same test with different inputs**.**

Steps:

1. **Create a new Java class `EvenChecker` with a method `isEven(int number)` and Create a parameterized test class `EvenCheckerTest` that tests the `isEven` method with different inputs.**



**Output**

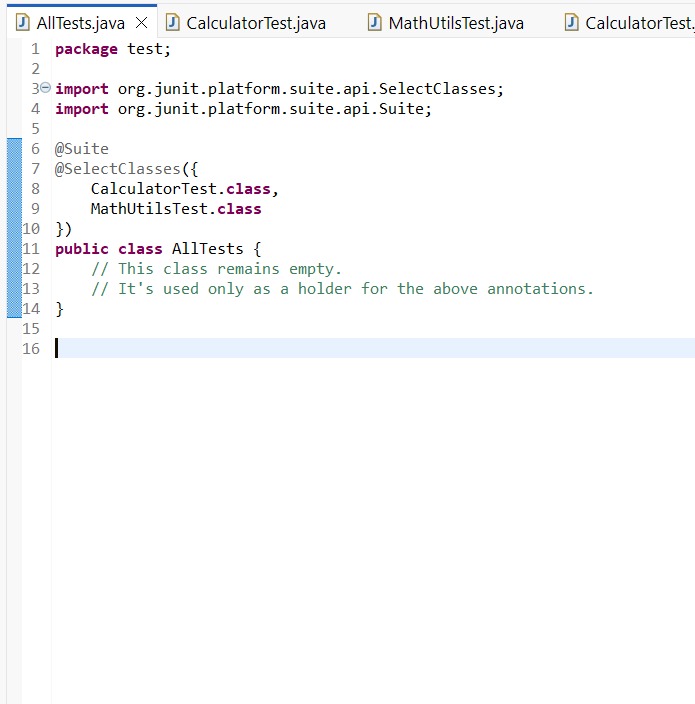


**Exercise 2: Test Suites and Categories**

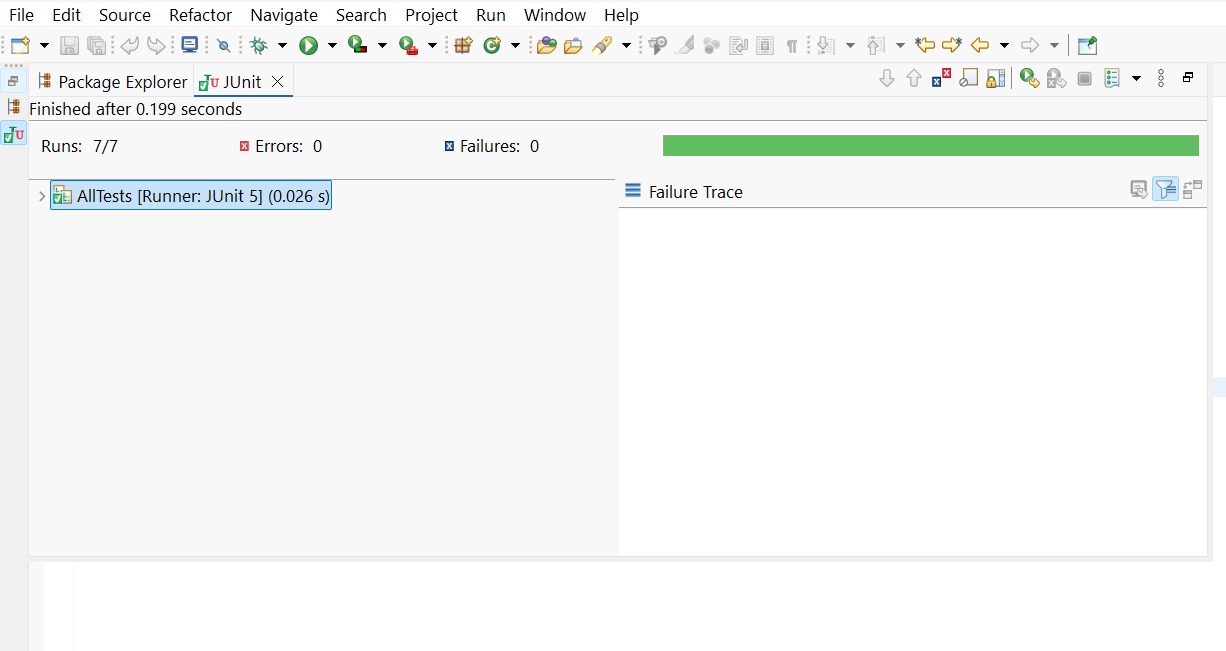
**Scenario:** You want to group related tests into a test suite and categorize them.

Steps:

1. **Create a new test suite class `AllTests`.**



**Output**

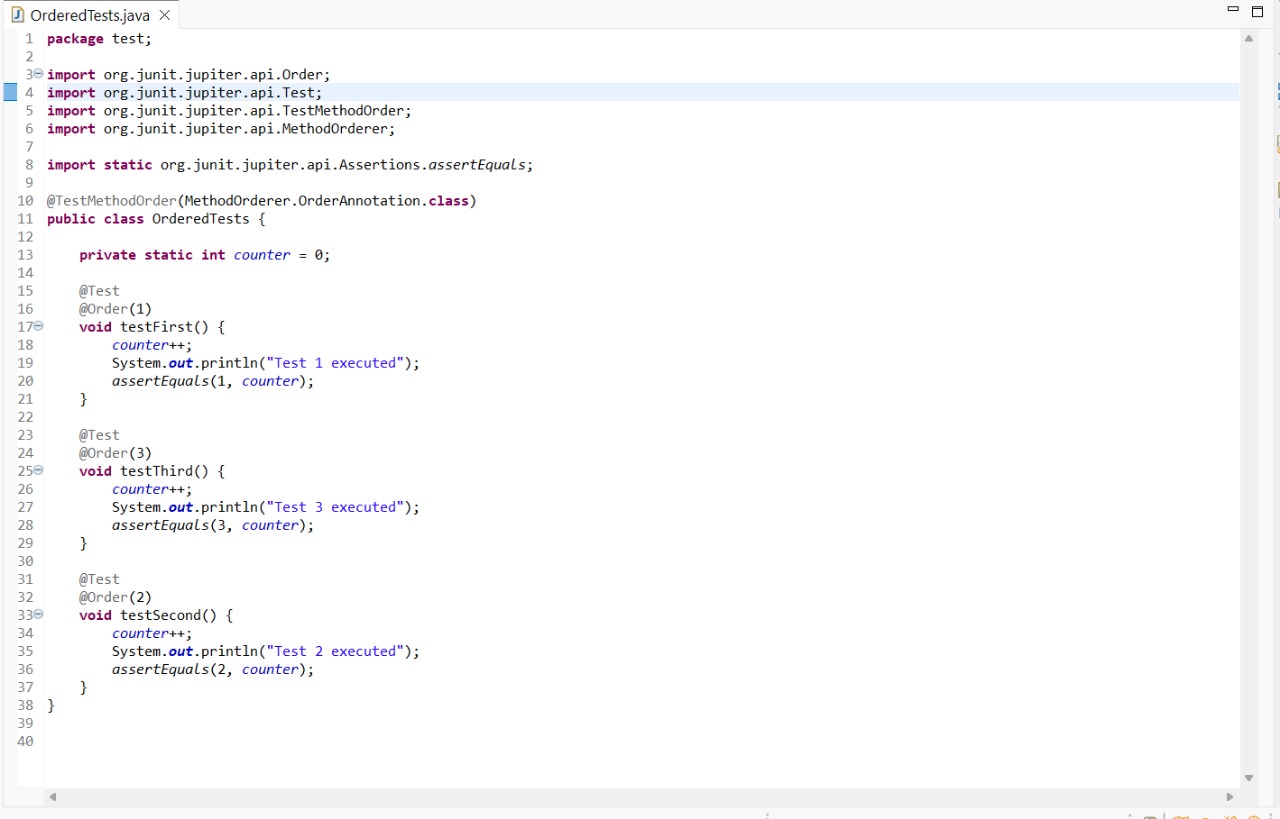


**Exercise 3: Test Execution Order**

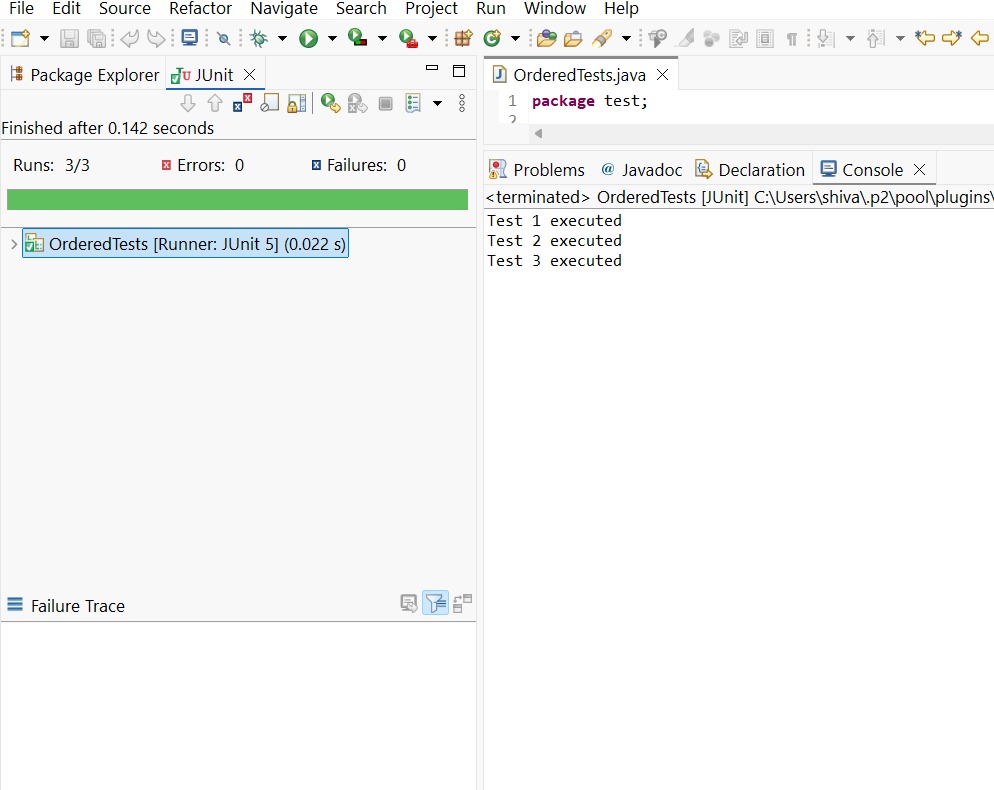
**Scenario:** You want to control the order in which tests are executed.

Steps:

1. **Create a test class `OrderedTests`.**



**Output**

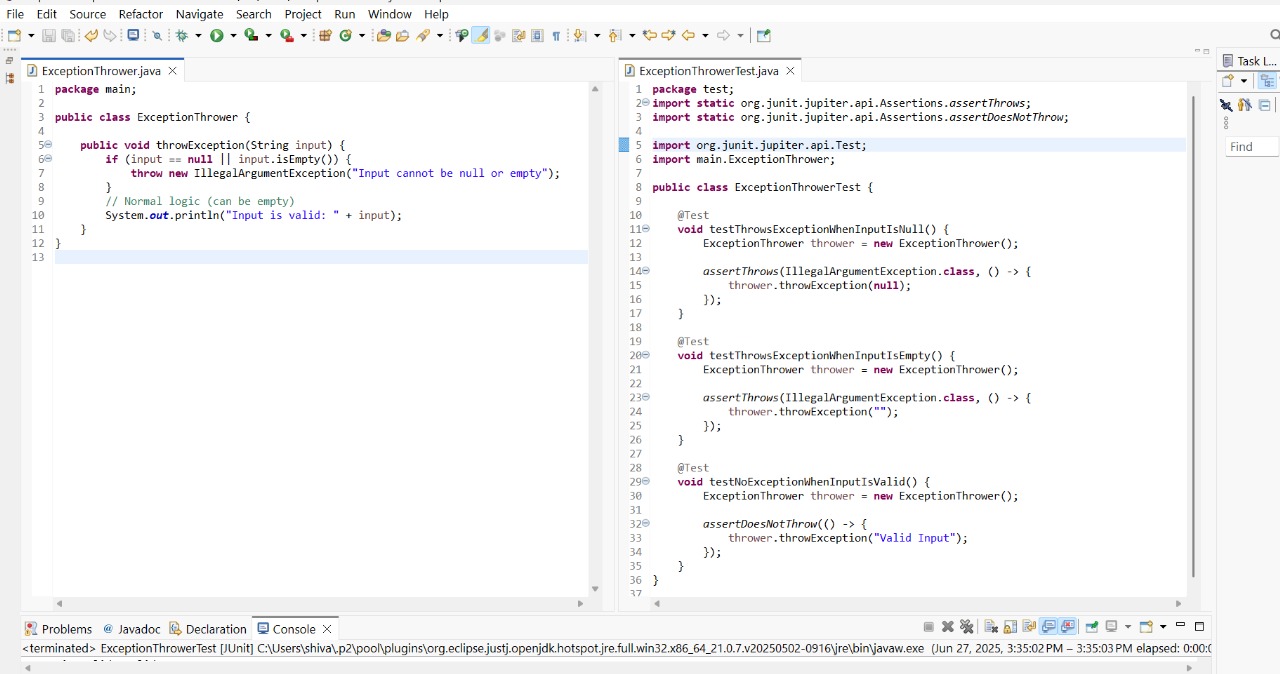


**Exercise 4: Exception Testing**

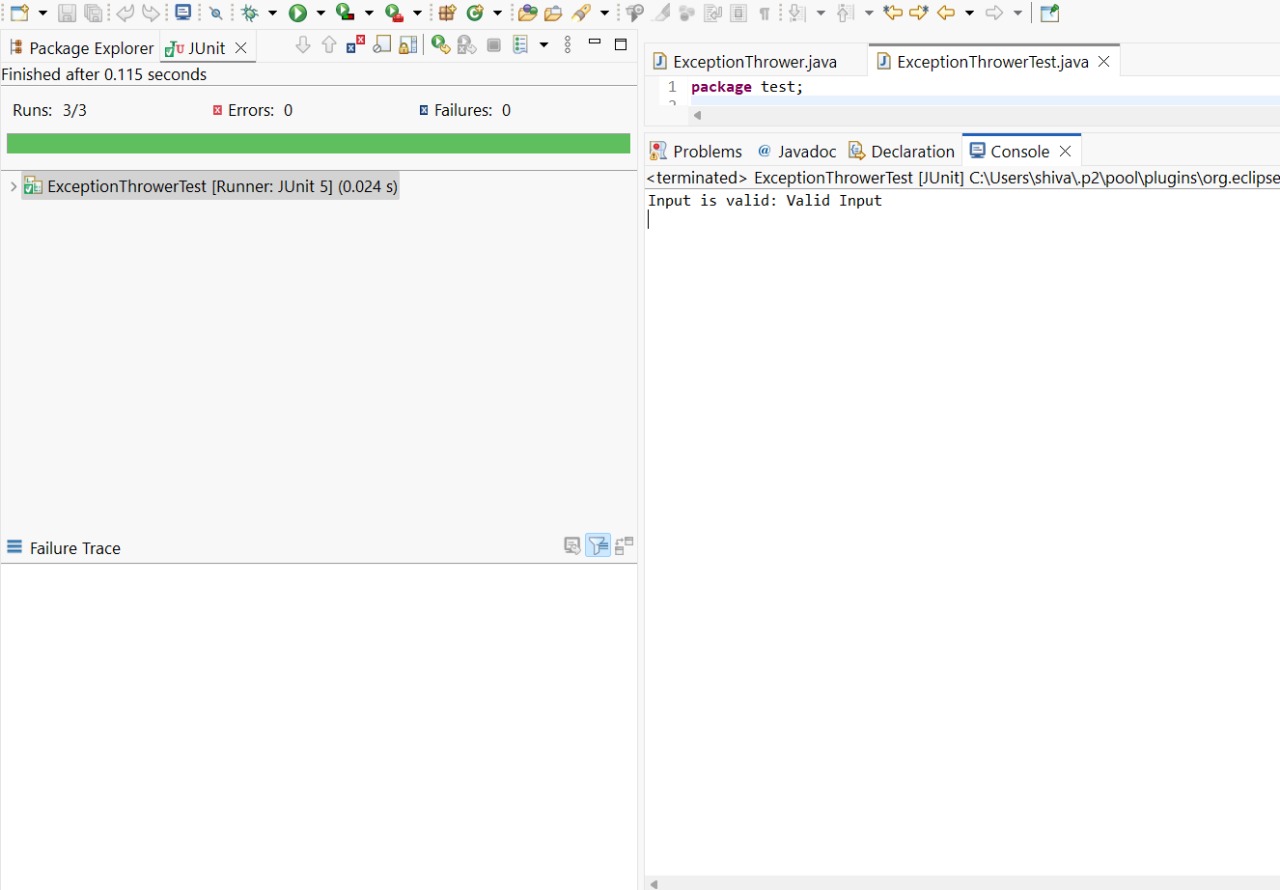
**Scenario:** You want to test that a method throws the expected exception.

Steps:

1. **Create a main class ‘ExceptionThrower’ with a method ‘throwException’.**
2. **Create a test class ‘ExceptionThrowerTest’ that tests the method for the expected exception.**



**Output**



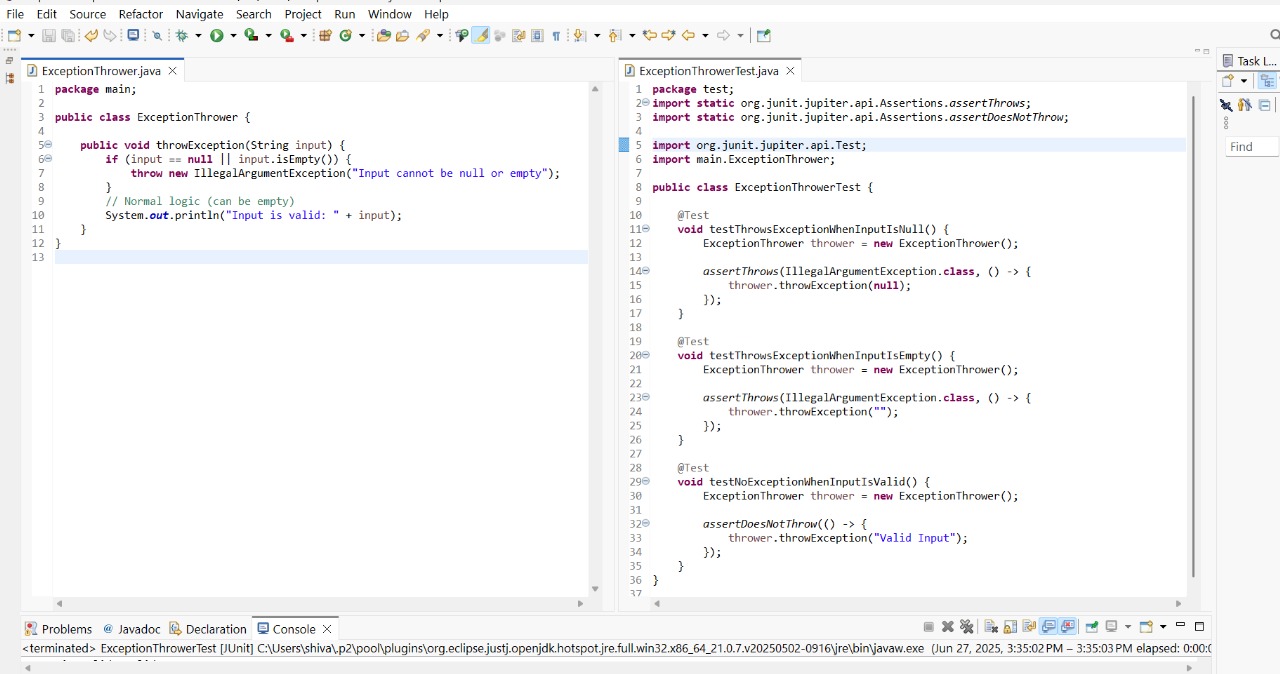
**Exercise 5: Timeout and Performance Testing**

**Scenario:** You want to ensure that a method completes within a specified time limit.

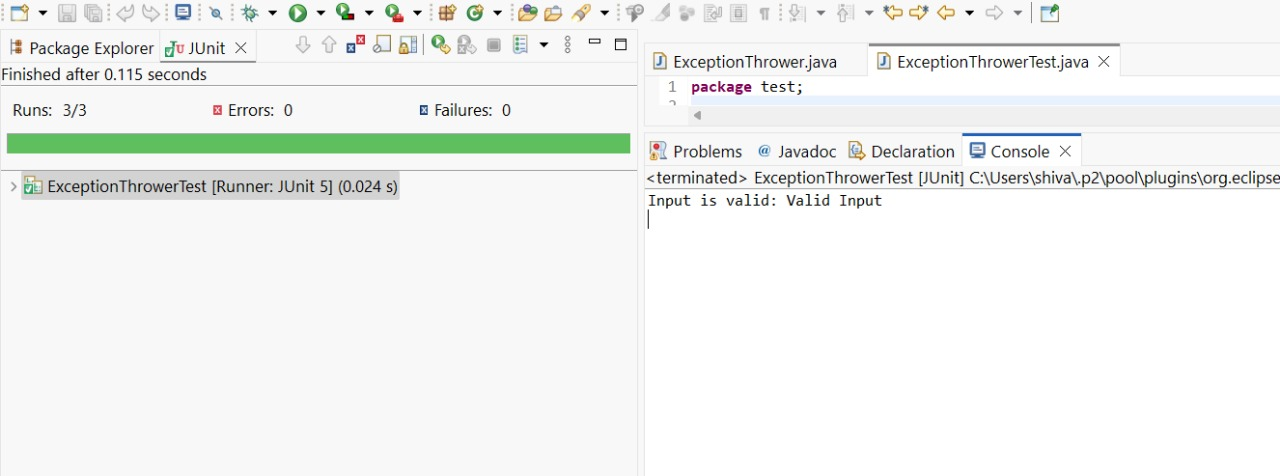
Steps:

**1. Create a class `PerformanceTester` with a method `performTask`.**

**2. Write a test class `PerformanceTesterTest` that tests the method for timeout.**



**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:**

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);

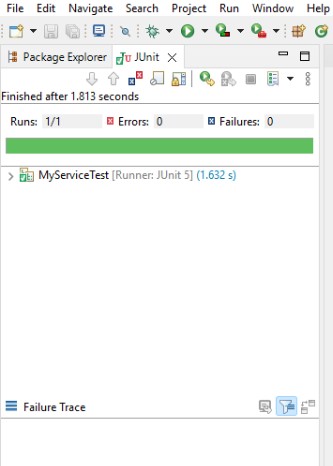
String result = service.fetchData();

assertEquals("Mock Data", result);

}

}

**Output**



**Exercise 2: Verifying Interactions**

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

**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);

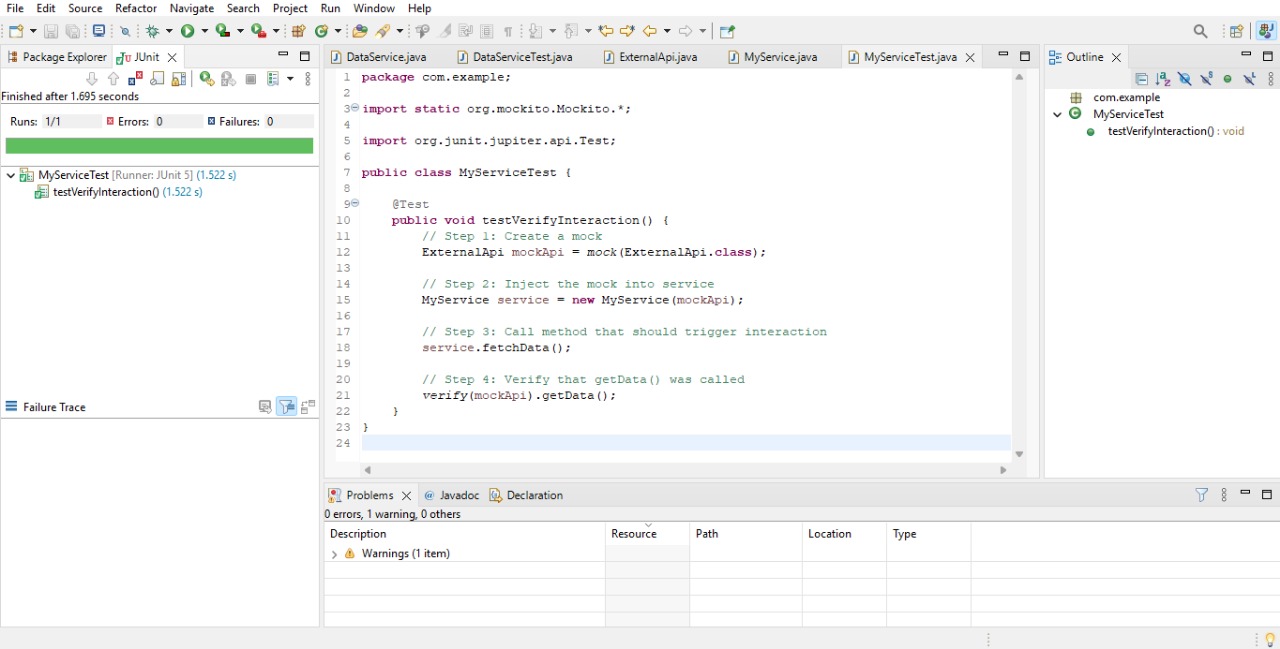
MyService service = new MyService(mockApi);

service.fetchData();

verify(mockApi).getData();

}

}



**Exercise 3: Argument Matching**

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

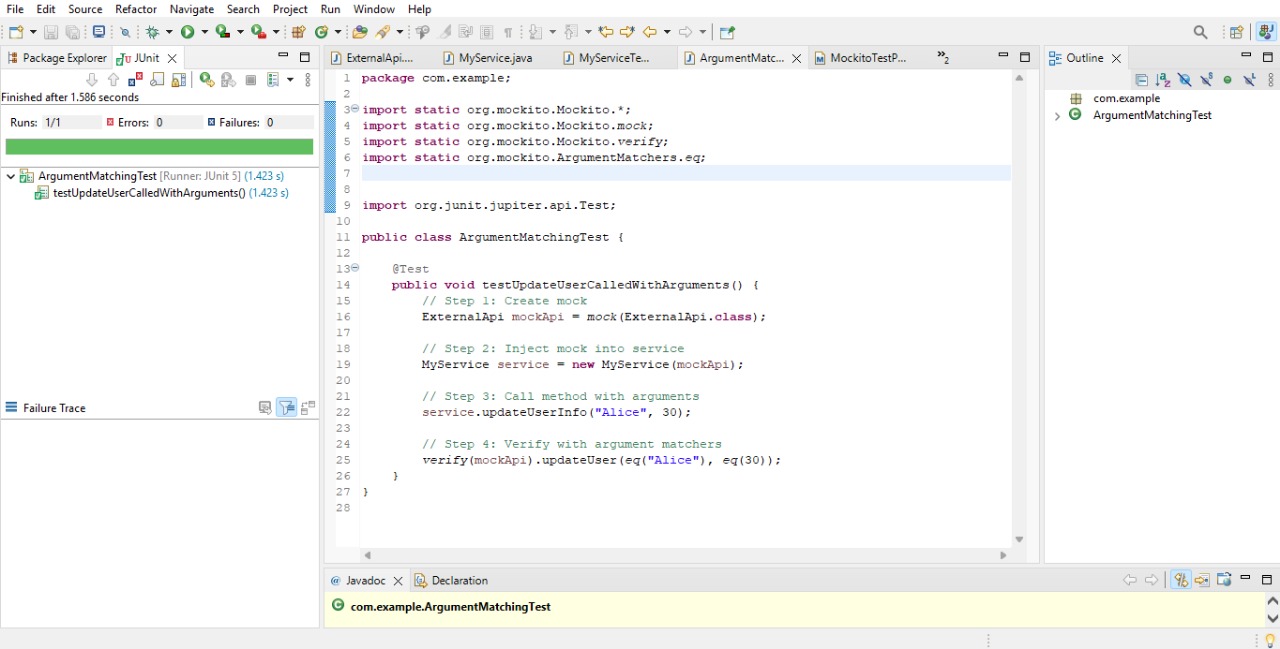
**Steps:**

1. Create a mock object.

2. Call the method with specific arguments.

3. Use argument matchers to verify the interaction.

**Output**



**Exercise 4: Handling Void Methods**

**Scenario:** You need to test a void method that performs some action.

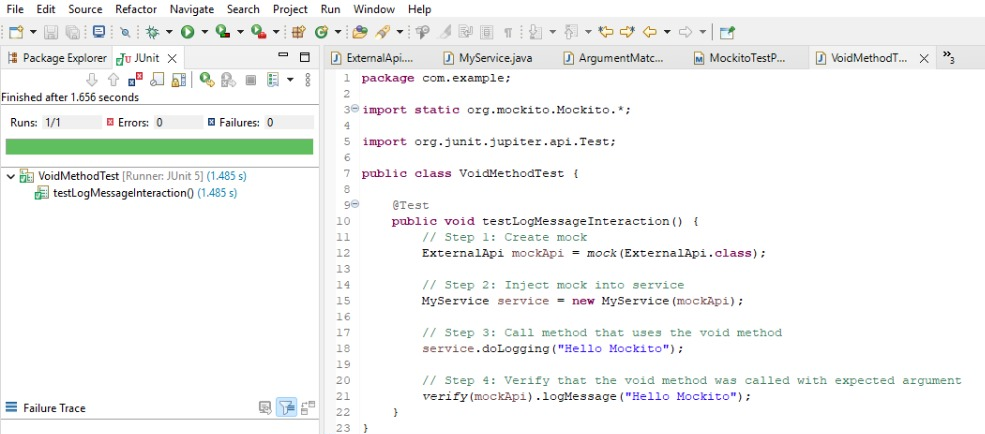
**Steps:**

1. Create a mock object.

2. Stub the void method.

3. Verify the interaction.

**Code and Output**



**Exercise 5: Mocking and Stubbing with Multiple Returns**

**Scenario:** You need to test a service that depends on an external API with multiple return values.

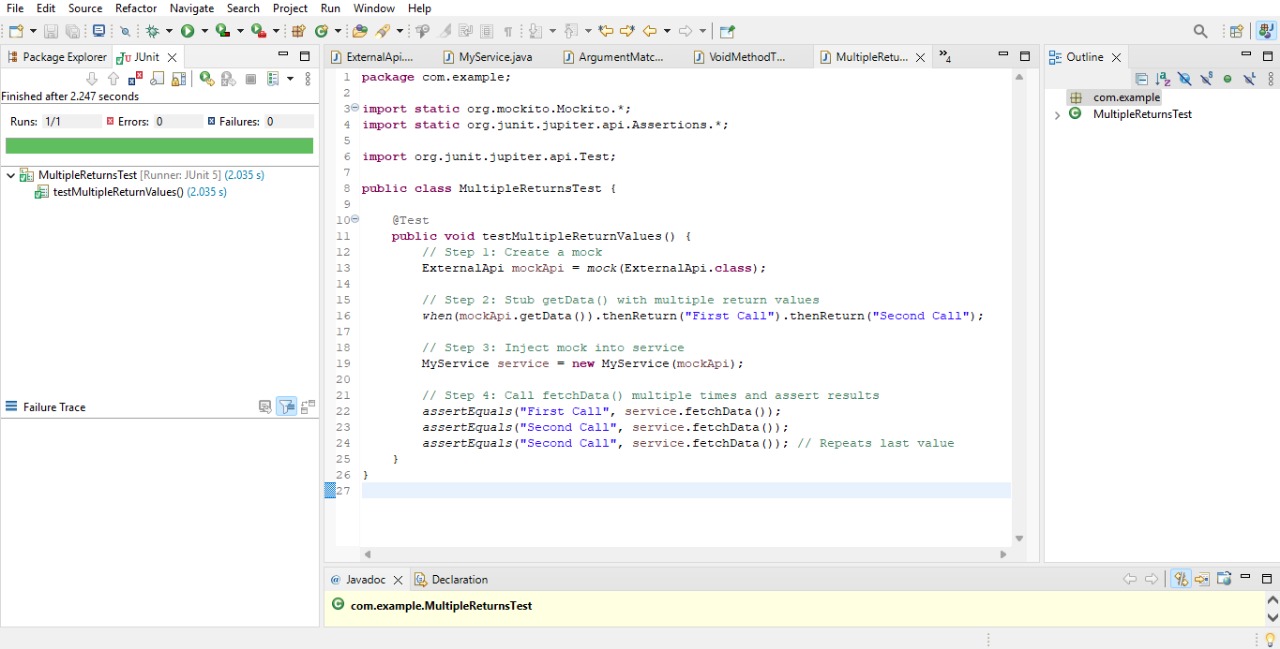
**Steps:**

1. Create a mock object for the external API.

2. Stub the methods to return different values on consecutive calls.

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

**Code and Output**



**Exercise 6: Verifying Interaction Order**

**Scenario:** You need to ensure that methods are called in a specific order.

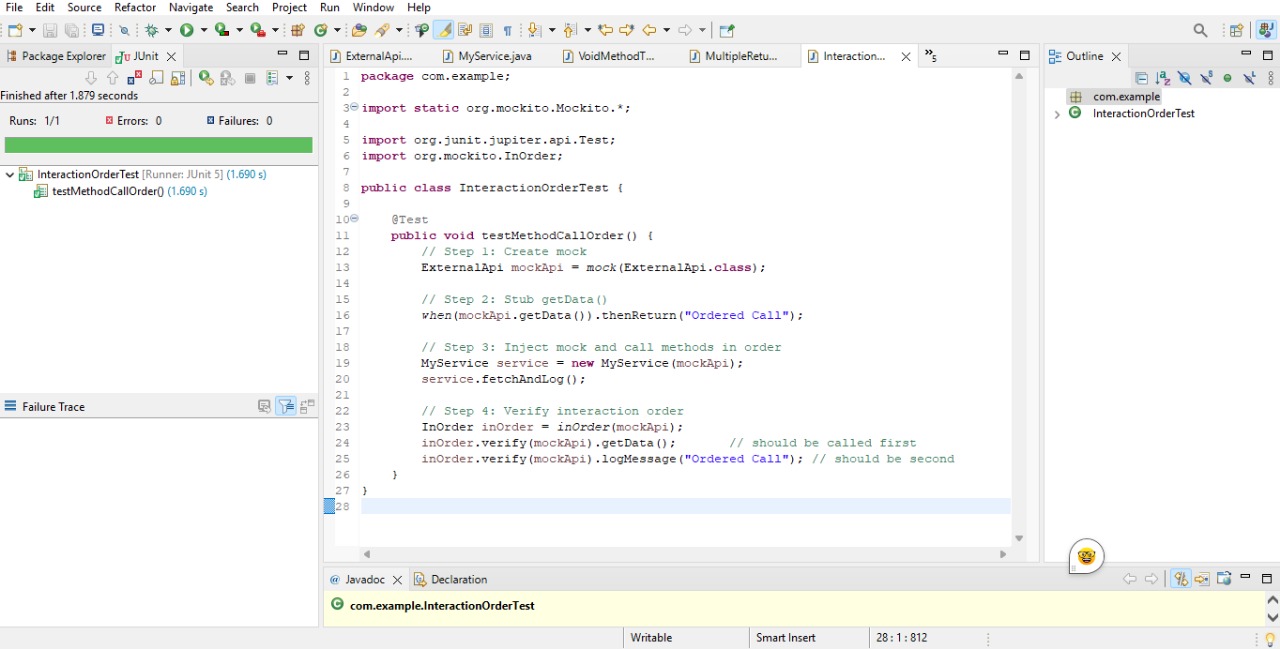
**Steps:**

1. Create a mock object.

2. Call the methods in a specific order.

3. Verify the interaction order.

**Code and Output**



**Exercise 7: Handling Void Methods with Exceptions**

**Scenario:** You need to test a void method that throws an exception.

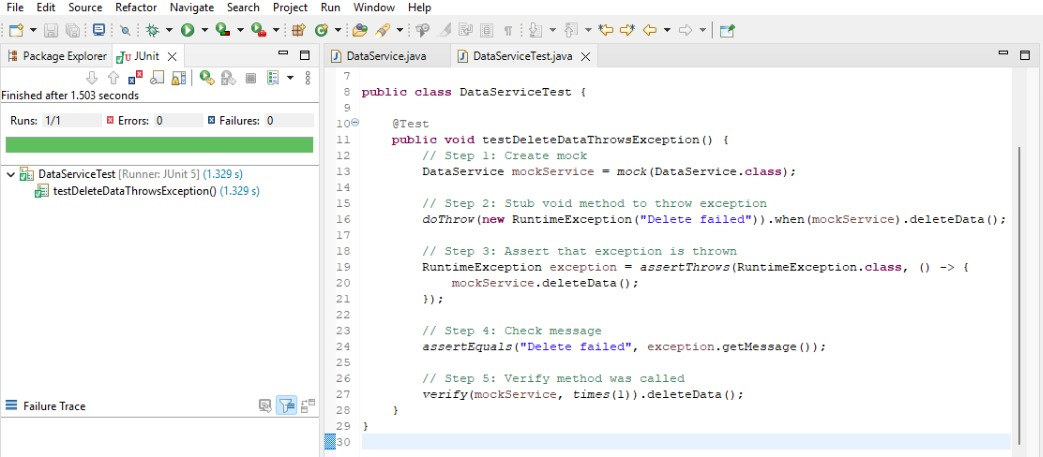
**Steps:**

1. Create a mock object.

2. Stub the void method to throw an exception.

3. Verify the interaction.

**Code and Output**



**Advanced Mockito Hands-On Exercises**

**Exercise 1: Mocking Databases and Repositories**

**Scenario**: You need to test a service that interacts with a database repository.

**Steps:**

1. Create a mock repository using Mockito.

2. Stub the repository methods to return predefined data.

3. Write a test to verify the service logic using the mocked repository.

**Solution Code:**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class ServiceTest {

@Test

public void testServiceWithMockRepository() {

Repository mockRepository = mock(Repository.class); when(mockRepository.getData()).thenReturn("Mock Data");

Service service = new Service(mockRepository);

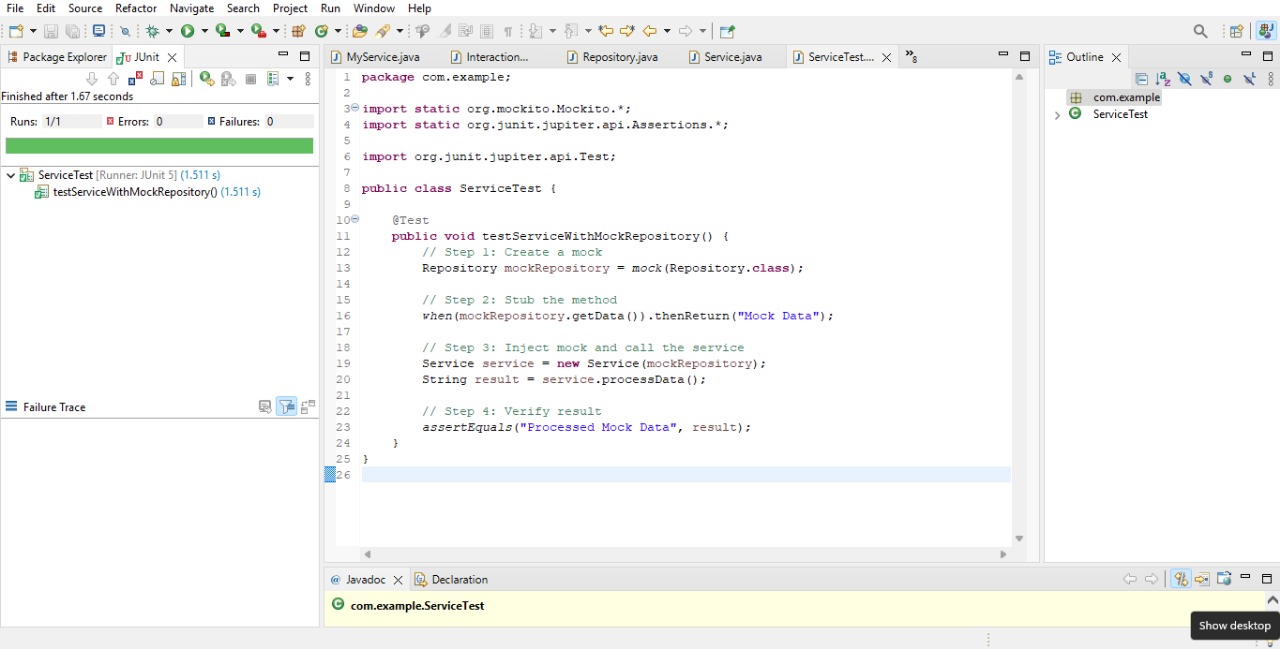
String result = service.processData();

assertEquals("Processed Mock Data", result);

}

}

**Code and Output**



**Exercise 2: Mocking External Services (RESTful APIs)**

**Scenario:** You need to test a service that calls an external RESTful API.

**Steps:**

1. Create a mock REST client using Mockito.

2. Stub the REST client methods to return predefined responses.

3. Write a test to verify the service logic using the mocked REST client.

**Solution Code:**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

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

public class ApiServiceTest {

@Test

public void testServiceWithMockRestClient() {

RestClient mockRestClient = mock(RestClient.class);

when(mockRestClient.getResponse()).thenReturn("Mock Response");

ApiService apiService = new ApiService(mockRestClient);

onse", result);

}

}

**Code and Output**

