**TDD Using JUnit5 and Mockito**

**Exercise 1 : Setting up JUnit**

**Setting Up JUnit with JAR Files (Without Maven/Gradle)**

If you are not using Maven or Gradle, you can set up JUnit in your Java project manually by following these steps:

**1. Download JUnit and Hamcrest JARs**

* Go to the [JUnit 5 Releases page](vscode-file://vscode-app/c:/Users/Kushagra%20Chaturvedi/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) or [JUnit 4 Downloads](vscode-file://vscode-app/c:/Users/Kushagra%20Chaturvedi/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html).
* Download the required JAR files (for JUnit 4: [junit-4.x.x.jar](vscode-file://vscode-app/c:/Users/Kushagra%20Chaturvedi/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) and hamcrest-core-x.x.x.jar; for JUnit 5, download [junit-platform-console-standalone-x.x.x.jar](vscode-file://vscode-app/c:/Users/Kushagra%20Chaturvedi/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)).

**2. Add JARs to Your Project**

* Place the downloaded JAR files in a [lib](vscode-file://vscode-app/c:/Users/Kushagra%20Chaturvedi/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) folder inside your project directory (or anywhere you prefer).

**3. Add JARs to Classpath**

* When compiling and running your tests, include the JAR files in your classpath.
* Example compile command (Windows):

javac -cp ".;lib/junit-4.13.2.jar;lib/hamcrest-core-1.3.jar" Calculator.java CalculatorTest.java

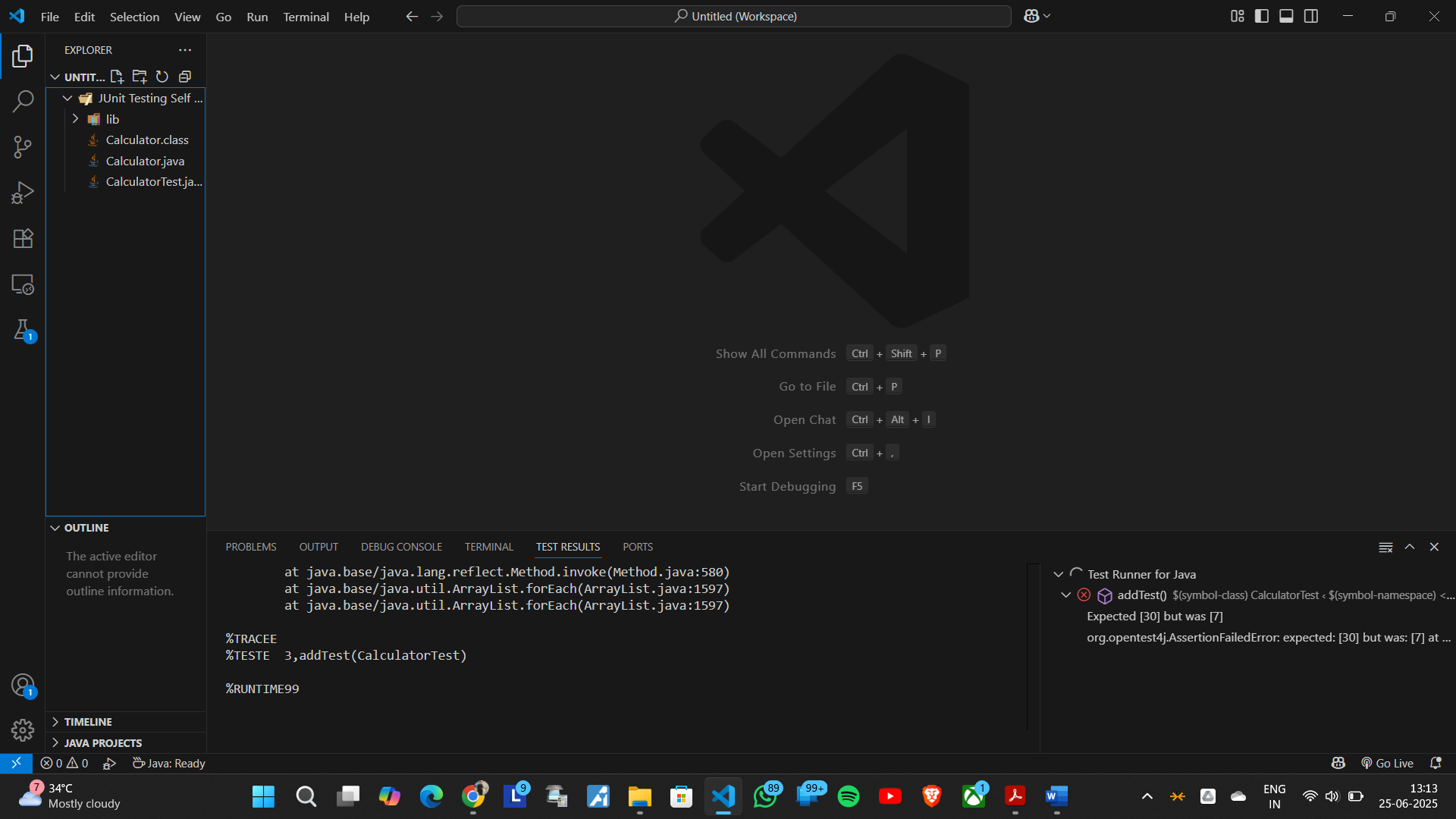
**4. Write Your Test Classes**

* Use @Test annotations and assertions as usual.

**5. Run Your Tests**

* Use the command line as shown above, or configure your IDE to include the JARs in the classpath.

**NOTE** : We can directly setup Jar in VS Code with the help of VS Code Extension such as **Extension Pack for Java and Test Runner for Java.** After selecting the jar icon on sidebar , click on enable Java Test and it will  **automatically** download all required **Junit Jupiter Test Packages** and add It as **.jar** file in lib folder of source file



**Exercise 3 : Assertions in JUnit**

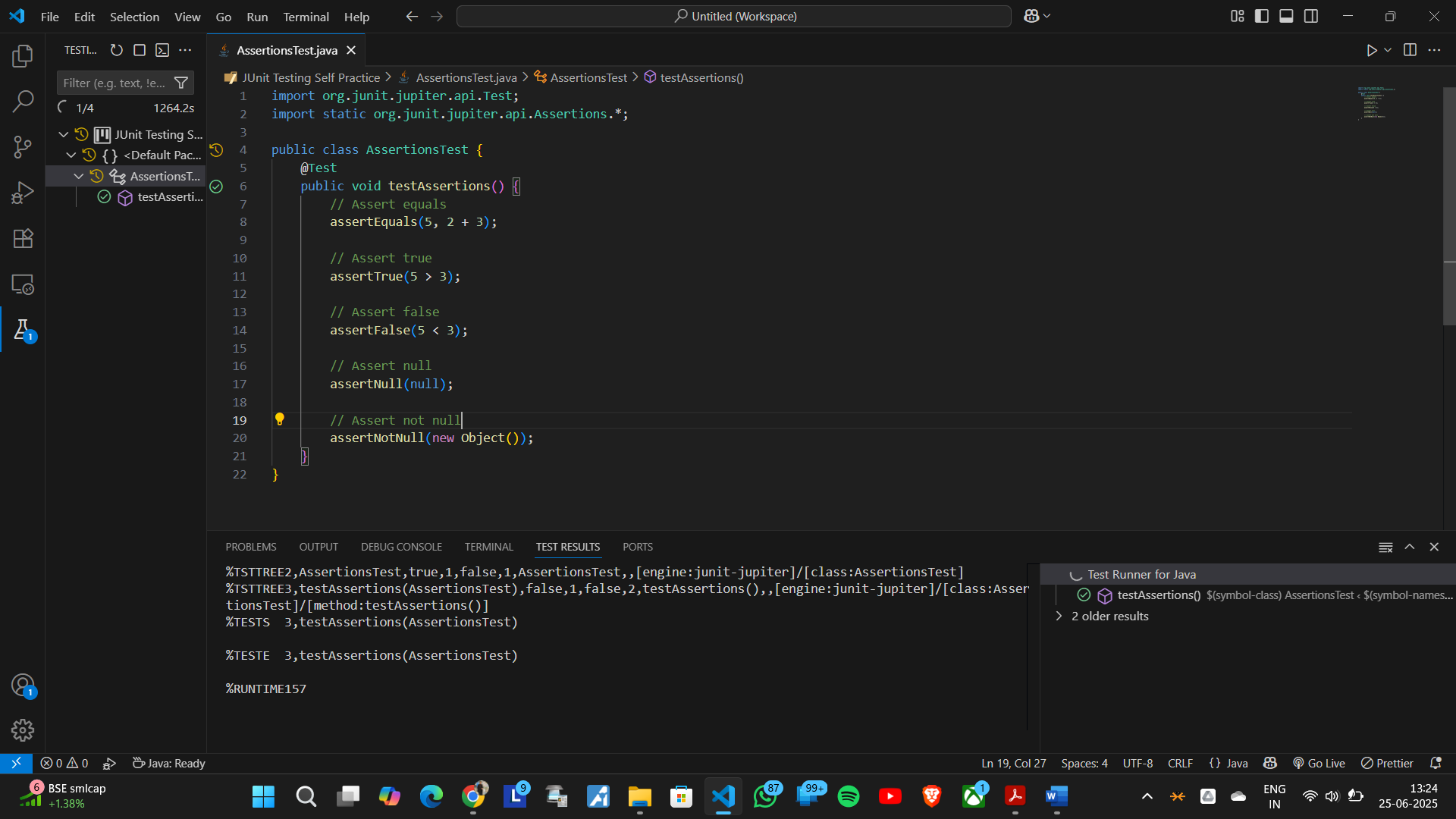
**Key Points:**

* **JUnit 5 Imports:**
  + **@Test** is used to mark the method as a test case.
  + Static imports from **org.junit.jupiter.api.Assertions** provide assertion methods.

**Assertions Used:**

* **assertEquals(expected, actual):**Verifies that two values are equal.
* **assertTrue(condition):**Checks that the condition is true.
* **assertFalse(condition):**Checks that the condition is false.
* **assertNull(object):**Asserts that the object is null.
* **assertNotNull(object):**Asserts that the object is not null.

Example Test Method:

The testAssertions() method demonstrates all the above assertions with simple examples.

**Exercise 3 : 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.**

**Code Snippet :**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class Exercise4 {

    private int number;

    @Before

    public void setUp() {

        number = 10;

    }

    @After

    public void tearDown() {

        number = 0;

    }

    @Test

    public void testAdd() {

        int result = number + 5;

        assertEquals(15, result);

    }

    @Test

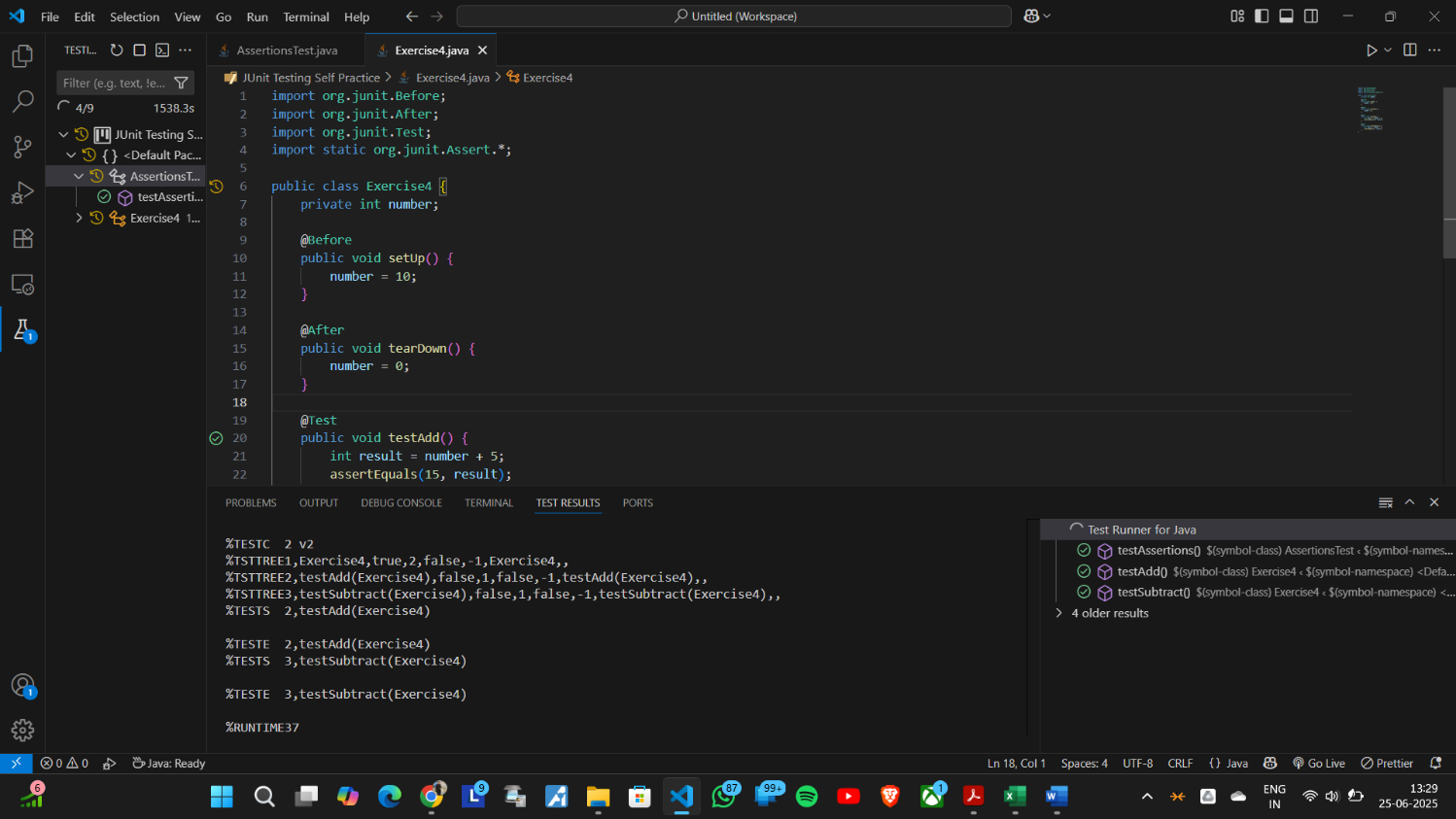
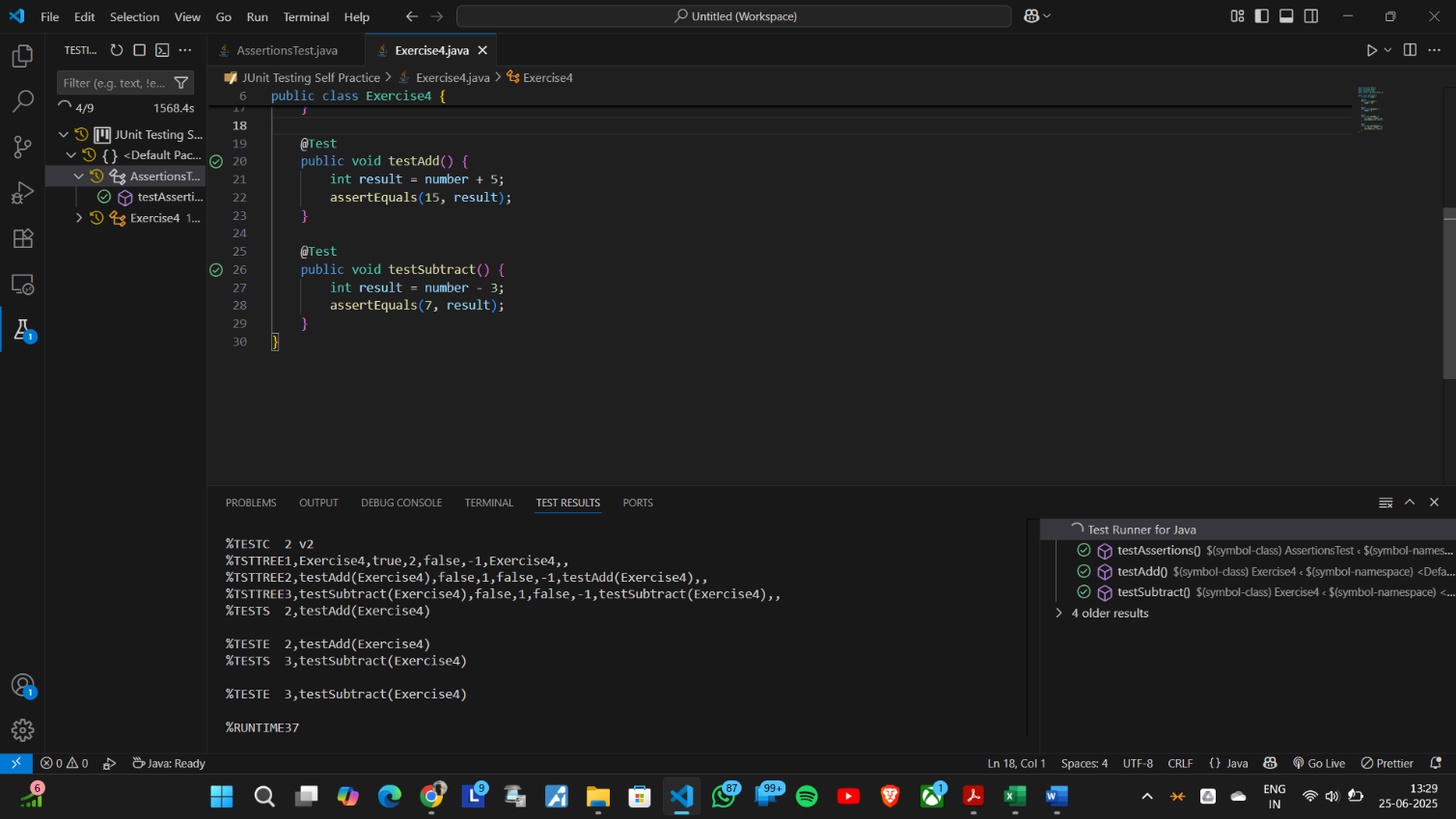
    public void testSubtract() {

        int result = number - 3;

        assertEquals(7, result);

}

}



**Mockito exercises**

**Exercise 1 : Mocking and Stubbing**

This file demonstrates how to use Mockito with JUnit to test a service that depends on an external API.

**🡪 Components**

**- ExternalApi:**

A simple class simulating an external dependency with a `getData()` method.

**- MyService:**

A service class that depends on `ExternalApi`. It has a `fetchData()` method that calls `getData()` from the API.

**- Ex1 (Test Class):**

Contains a JUnit test method `**testExternalApi()**` that:

1. Creates a mock of `**ExternalApi**` using Mockito.

2. Stubs the `**getData()**` method to return `"Mock Data"`.

3. Injects the mock into `**MyService**`.

4. Calls `**fetchData()**` and asserts that the returned value is `"**Mock Data**"`.

**🡪 Purpose**

- **Mocking**: Isolates `**MyService**` from the real `**ExternalApi**` for testing.

- **Stubbing**: Controls the return value of `**getData**()` to test `**MyService**` behavior.

- **Unit Testing:** Ensures `**MyService**` interacts correctly with its dependency.

🡪**Code Snippet :**

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

class ExternalApi {

    public String getData() {

        return "Real Data";

    }

}

class MyService {

    private ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public String fetchData() {

        return api.getData();

    }

}

public class Ex1 {

    @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);

    }

}

**Exercise 2 : Verifying Interactions**

**🡪Components**

* **ExternalApi**  
  A simple class that simulates an external dependency with a **getData**() method.
* **MyService**  
  A service class that depends on ExternalApi. It has a **fetchData**() method that calls **getData**() from the API.
* **Ex2** (Test Class)  
  Contains a JUnit test method **testVerifyInteraction**() that:
  1. Creates a **mock** of ExternalApi using Mockito.
  2. Injects the mock into MyService.
  3. Calls **fetchData**() on the service.
  4. Uses **verify**() to check that **getData**() was called on the mock.

**🡪Purpose**

* **Mocking**: Isolates MyService from the real ExternalApi for testing.
* **Interaction Verification**: Ensures that MyService correctly calls the **getData**() method of its dependency.

**🡪Usage**

* Run the test method **testVerifyInteraction**(). The test will pass if **getData**() is called as expected.

**🡪Code Snippet:**

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

class ExternalApi {

    public String getData() {

        return "Real Data";

    }

}

class MyService {

    private ExternalApi api;

    public MyService(ExternalApi api) {

        this.api = api;

    }

    public String fetchData() {

        return api.getData();

    }

}

public class Ex2 {

    @Test

    public void testVerifyInteraction() {

        ExternalApi mockApi = mock(ExternalApi.class);

        MyService service = new MyService(mockApi);

        service.fetchData();

        verify(mockApi).getData();

    }

}

**SLF4J logging framework**

**Exercise 1 : Logging Error Messages and Warning Levels**

**🡪 Steps**

1. Add **SLF4J and Logback dependencies** to your project.

<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>  
If you use **Maven**, add the dependencies to your **pom.xml**.  
If not, download the JAR files and add them to your classpath.

1. **Create a Java class and use SLF4J for logging.**  
   Example:

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**

* The logger is created for the class.
* Use logger.error to log error messages.
* Use logger.warn to log warning messages.
* When you run the program, the messages will appear in the console.

**🡪 Requirements**You need the **SLF4J API** and **Logback Classic JARs** in your classpath.  
No extra configuration is needed for basic logging to the console.