**CSCI 391 Microservices, Lab 5: Testing with JUnit**

**Objective**

In this lab, you will learn how to write, run, and interpret your own JUnit tests using **JUnit 4** to test a class that implements the **Stack** interface: ArrayStack. You will use **IntelliJ IDEA** with **Maven** as your build tool to manage dependencies and run tests.

By the end of this lab, you will:

* Understand the process of writing unit tests for Java classes.
* Be able to configure a Maven project in IntelliJ IDEA to use JUnit.
* Gain experience with the JUnit framework for writing and running automated tests.

**Prerequisites**

* **IntelliJ IDEA** (Community Edition or another supported version)
* **Maven** installed and configured in your system (Maven is integrated in IntelliJ IDEA)
* Basic knowledge of Java and JUnit 4

**Setup**

**1. Create a New Maven Project in IntelliJ IDEA**

* Open **IntelliJ IDEA** and select **File > New > Project**.
* Choose **Maven** from the project types and click **Next**.
* Name the project JUnitStackDemo and set the **GroupId** and **ArtifactId** according to your preference (Advanced Settings). Click **Create**.

**2. Add JUnit Dependency to the pom.xml**

Open the pom.xml file in the root of your project and add the JUnit dependency to the dependencies section.

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

This will ensure that Maven downloads JUnit when you build the project.

**3. Create the ArrayStack Classes**

In your project, create the ArrayStack class. These classes should implement the **Stack** interface (for String). Here’s a very basic outline for these classes:

public interface Stack<T> {

void push(T item);

T pop();

boolean isEmpty();

void clear();

}  
ArrayStack will provide its own implementation of this interface.

**Writing JUnit Tests**

**4. Create a New Test Class**

To create a JUnit test class:

1. Right-click on the **src/test/java** directory.
2. Select **New > Java Class**, and name it StackTest.
3. In the StackTest.java file, write your test methods using **JUnit annotations** like @Before and @Test.

Your test class should look like this:

import org.example.ArrayStack;

import org.example.Stack;

import org.junit.\*;

import static org.junit.Assert.\*;

public class StackTest {

private ArrayStack stack;

@Before

public void setUp() {

stack = new ArrayStack();

}

@Test

public void testClear() {

stack.push("One");

stack.push("Two");

stack.push("Three");

assertFalse(stack.isEmpty());

stack.clear();

assertTrue(stack.isEmpty());

}

@Test

public void testAddRemove() {

stack.push("One");

stack.push("Two");

stack.push("Three");

assertEquals("Three", stack.pop());

assertEquals("Two", stack.pop());

assertEquals("One", stack.pop());

assertTrue(stack.isEmpty());

}

}**5. Understanding the Test Methods**

* **@Before annotation**: The setUp() method initializes the stack object before each test method runs. In this case, it initializes an ArrayStack.
* **@Test annotation**: Marks a method as a JUnit test method. Each test method should contain assertions to check whether the behavior of the stack is correct.
  + assertTrue() and assertFalse() check boolean conditions.
  + assertEquals() checks whether the actual value matches the expected value.

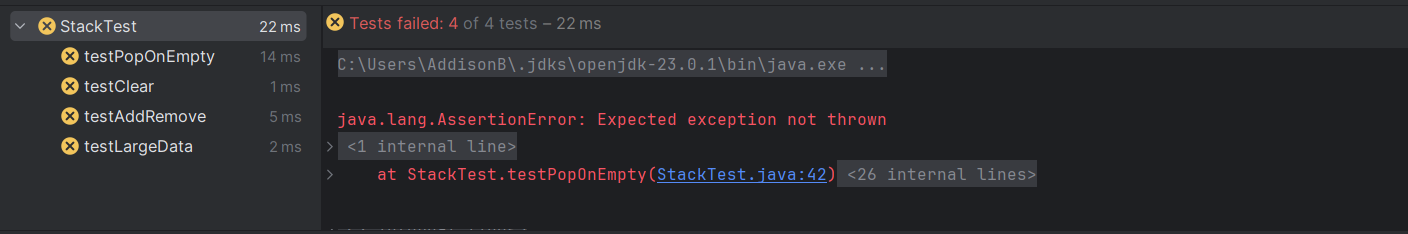
**Running the Tests**

**6. Running Your Tests in IntelliJ IDEA**

After writing your test methods, you can run them directly from IntelliJ IDEA.

1. **Run All Tests:**
   * Right-click on the StackTest.java file in the **Project** view and select **Run 'StackTest'**.
   * Alternatively, click the **Run** button at the top right of the IntelliJ window.
2. **View the Results:**
   * IntelliJ IDEA will display the test results in the **Run** window.
   * A green bar indicates all tests passed, while a red bar indicates that some tests failed.
   * You can view detailed information about each test by expanding the output in the **Run** window.

Your tests will look like this…



**Testing Details**

**7. Testing Method: clear**

In this test, you will:

* Push three items onto the stack.
* Use clear() to empty the stack.
* Use assertions to check if the stack is empty after clearing.

**Test Code:**

@Test

public void testClear() {

stack.push("One");

stack.push("Two");

stack.push("Three");

assertFalse(stack.isEmpty());

stack.clear();

assertTrue(stack.isEmpty());

}

**8. Testing Method: addRemove**

This method will:

* Push three items onto the stack.
* Pop each item off the stack and verify they come off in the reverse order.
* Check if the stack is empty after all items are removed.

**Test Code:**

@Test

public void testAddRemove() {

stack.push("One");

stack.push("Two");

stack.push("Three");

assertEquals("Three", stack.pop());

assertEquals("Two", stack.pop());

assertEquals("One", stack.pop());

assertTrue(stack.isEmpty());

}

**9. Handling Exceptions in Test Methods**

If your stack is empty and you try to pop an element, a NoSuchElementException should be thrown. You can test this scenario as follows:

@Test

public void testPopOnEmpty() {

try {

stack.pop();

fail("Expected exception not thrown");

} catch (NoSuchElementException e) {

// Test passes if exception is thrown

}

}

**Additional Exercises**

**10. Test Performance with Large Data**

To further test your implementation, push and pop large numbers of items, such as 1,000 integers, and verify that each item is correctly removed.

@Test

public void testLargeData() {

for (int i = 0; i < 1000; i++) {

stack.push(Integer.toString(i));

}

for (int i = 999; i >= 0; i--) {

assertEquals(Integer.toString(i), stack.pop());

}

assertTrue(stack.isEmpty());

}

**Conclusion**

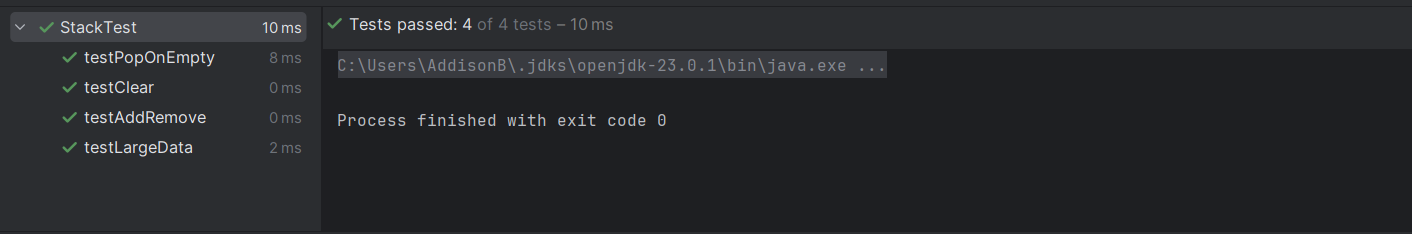
In this lab, you have written JUnit tests for stack implementations using **JUnit 4** and **IntelliJ IDEA with Maven**. You've learned how to set up a Maven project, write test methods, run tests, and interpret test results. Unit testing is an essential skill for writing reliable software, and JUnit provides an effective framework to automate this process.

As you progress, consider writing more comprehensive tests and using test-driven development (TDD) to improve your coding and debugging practices.

**Deliverables**

1. The StackTest.java file with your JUnit tests.
2. Your implementation of ArrayStack.

Make sure your tests cover all the important functionality of the Stack interface and run them to verify that your class behaves as expected i.e., your tests should look like this…



Good luck, and happy coding!