MAHARISHI INTERNATIONAL UNIVERSITY

Engaging the Managing Intelligence of Nature

Computer Science Department

CS390 Fundamental Programming Practices (FPP)

Prof. Paul Corazza and
Prof. Ankhtuya Ochirbat

Lecture 2 Appendix: Unit testing with JUnit

Wholeness of the Lesson

Sufficient unit testing ensures that the individual classes and methods behave correctly; the Junit tool aids developers in creating test cases for their programs. Natural systems function according to laws that ensure that the whole of creation is maintained in perfect order when natural law is not violated.

What is JUnit?

JUnit is the **standard testing component** for Java.

It helps you:

- Write and run unit tests
- Automatically verify that your code works correctly
- Integrate testing into your development workflow

Testing Code Using JUnit

- In the SampleClass file, we might want to test the method

```
int calculateSum(int[ ] arr)
```

- To do this, we create a TestSampleClass class with a method

```
void testCalculateSum()
```

- Testing with JUnit requires us to load the JUnit libraries into our IntelliJ project.

Adding JUnit JAR Manually (No Maven/Gradle)

Steps:

1. Right-click the project root (e.g., InClassExercises) in the **Project Explorer**.
2. Click **"Open Module Settings"** (or press F4).
3. In the left panel, select your **Module** > go to the **Dependencies** tab.
4. Click the **+ icon > JARs or directories...**
5. Select the downloaded junit-4.8.1.jar file.
6. Click **OK > Apply > OK**.

After Adding:
The JAR is now added to your project classpath. You can now write JUnit test classes, and IntelliJ will recognize annotations like @Test.

Add JUnit via Maven

Step 1: Create or Open a Maven Project

If you're starting a new project:

1. Go to File > New > Project
2. Select Maven and Set a GroupId, ArtifactId, and project name
3. Click Finish

If you already have a Maven project, just open it.

Step 2: Add JUnit Dependency in pom.xml

Open the pom.xml file in your project root. Add the following inside the <dependencies> section:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>  <!-- or latest version -->
        <scope>test</scope>
    </dependency>
</dependencies>
```

IntelliJ will automatically download the dependency and attach it to the classpath.

Creating and Running JUnit Tests in IntelliJ

- Right-click src/test/java > New > Java Class
- Name it something like MyTest

```
//JUnit4
import org.junit.Test;
import static org.junit.Assert.*;

public class MyTest{

    @Test
    public void testAddition() {
        assertEquals(4, 2 + 2);
    }
}
```

```
//JUnit5
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyTest{

    @Test
    void testAddition() {
        assertEquals(4, 2 + 2);
    }
}
```

JUnit assertion methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

@Before, @After

- before/after each test case method is called

```
@Before
  public void name() { ... }
@After
  public void name() { ... }
```

- once before/after the entire test class runs

```
@BeforeClass
  public static void name() { ... }
@AfterClass
  public static void name() { ... }
```

```
package myTestPkg;

import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class MyTestClass {
    private Collection<Object> collection;

    @Before
    public void setUp() {
        collection = new ArrayList<Object>();
    }
    @Test
     public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }
}
```

Possible Execution Order

- Order doesn't matter because each method gets its own instance of the collection
- The ordering of test-method invocations is not guaranteed, could be as follows:

```
setUp()
testOneItemCollection()
setUp()
testEmptyCollection()
```

After Methods

The JUnit framework automatically invokes any @After methods after each test is run

```
@Before
public void createOutputFile() {
    output = new File(...);
}
@After
public void deleteOutputFile() {
    output.delete();
}
@Test
public void testSomethingWithFile() { ... }
```

Exceution Order

- Would be as follows:

```
createOutputFile()
testSomethingWithFile()
deleteOutputFile()
```

How to test a method that doesn't return anything

- If a method doesn't return a value, it will have some side effect (otherwise does nothing)
- There is usually a way to verify that the side effect actually occurred as expected

```
@Test
public void testCollectionAdd() {
    Collection collection = new ArrayList();
    assertEquals(0, collection.size());
    collection.add("itemA");
    assertEquals(1, collection.size());
    collection.add("itemB");
    assertEquals(2, collection.size());
}
```

Testing an expected exception is thrown

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexException() {
    ArrayList emptyList = new ArrayList();
    Object o = emptyList.get(0);
}
```

JUnit reports
Only the first failure in a single test

- One test method with several assertions might be improved if split into several test methods
- I.e., may be necessary to divide a test up into several parts

```
public class MyTestCase {
   @Test
   public void testSomething() {
         // Set up for the test
         assertTrue(condition1);
         assertTrue(condition2);
         assertTrue(condition3);
   }
}
```

When the test results are all needed, separate the assertions into separate methods

Sometimes Necessary to do the Following:

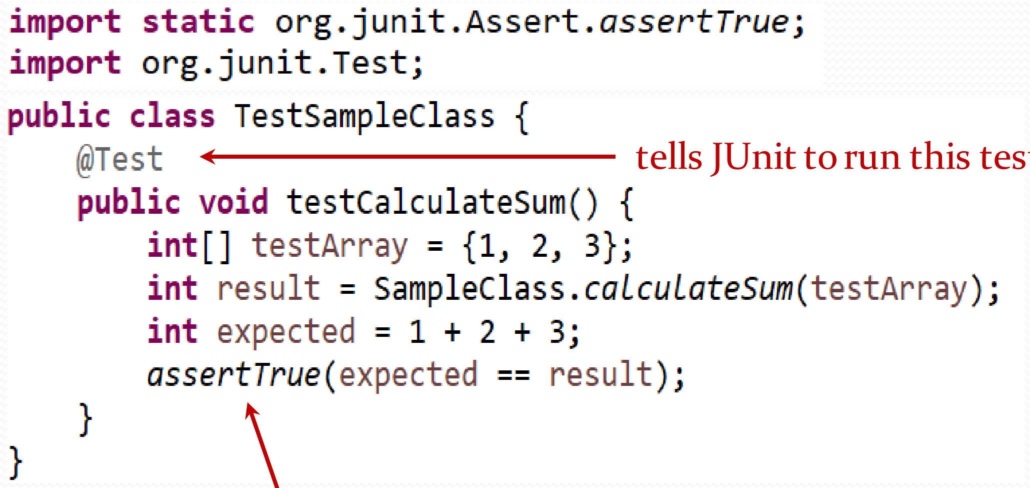
```
public class MyTestCase {
    @Before
    public void setUp() {
            // Set up for the tests
     }
    @Test
    public void testCondition1() {
            // Set up for the test
            assertTrue(condition1);
     }
    @Test
    public void testCondition2() {
            assertTrue(condition2);
    }
    @Test
    public void testCondition3() {
            assertTrue(condition3);
    }
}
```

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only

[based on materials by Marty Stepp, M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia]

Using JUnit – Testing SampleClass



```
import static org.junit.Assert.assertTrue;
import org.junit.Test;
public class TestSampleClass {
    @Test
    public void testCalculateSum() {
        int[] testArray = {1, 2, 3};
        int result = SampleClass.calculateSum(testArray);
        int expected = 1 + 2 + 3;
        assertTrue(expected == result);
    }
}
```

tells JUnit to run this test, uses import

- method from JUnit library
- accepts a boolean and expects that this boolean evaluates to *true*
- if boolean is false, JUnit declares that the test failed
- static import used to make accessible

- In a test class, you can ask JUnit to test a method of your class by marking the test method with @*Test*
- Within the test method, set up some test data, run the test class's method on this data and compare the result to the correct answer by using JUnit's *assertTrue* (or *assertEquals*) method
- If class's computation does not match expected result, test fails.
- See Demo

In-Class Exercise 1.2

- In this exercise you will create a test class TestMyClass to test the two methods in MyClass, which you implemented in Exercise 1.1.
- Be sure to add the JUnit JAR file to the project before trying to access the *@Test* annotation and the *assertTrue/assertEquals* method.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. OO programs need to be tested to ensure proper behavior.
2. Tools like Junit assist developers and simplify the task of creating large sets of tests for classes in a system of interdependent and interacting Java objects

---

3. <u>Transcendental Consciousness:</u> by its very nature, is the state of perfect orderliness.

4. <u>Wholeness moving within itself</u>: In Unity Consciousness one perceives the Universe in terms of one's own self-interacting consciousness and, on this basis, spontaneously acts in accord with Natural Law, i.e., in a life supporting manner.