UECS2354 Software Testing
Lab 02: Unit Testing and JUnit

Part A (Estimate completion time: 1.5 hours)

The Knowledge

1.1 Introduction to unit testing

Unit testing is a core aspect of test driven development and is also known as component or module testing. Unit tests are designed to test a unit of work, which is usually a single method. A unit of work is a task that isn't directly dependent on the completion of any other task. Unit tests therefore examine a single unit in isolation from other units. Each unit test will usually test for one single requirement on that method; in the event that the method fulfils more than one requirement. A method that is properly implemented should only fulfil one purpose or requirement. The unit test is typically based on the user requirements or design specifications. The unit of work tested is usually part of a larger entity, called the system under test (SUT) or class under test (CUT)

Unit tests often focus on testing whether a method follows the terms of its API contract. An API contract views an application programming interface (API) as a formal agreement between the user of the API and the implementer of the API. It is based on the signature of the method and requires its callers to provide specific arguments (which can be object references or primitive values), with the expectation that the method returns a specific object reference or primitive value.

If the method does not return the expected result, the test is said to have failed and some appropriate action will be taken by the test code, for e.g. throwing an exception. We say that the method has broken its contract. Often the unit tests help define the API contract by demonstrating the expected behaviour of the method.

1.2 Performing a unit test

Therefore to perform a unit test, what is required in advance is:

1. Knowledge of the signature of the unit (or method) that is to be tested
2. Inputs (or arguments) to be passed to this method
3. The required initial state of the class containing the method to be tested must be set up properly if the method requires the state to have a certain value in order to function in a specific manner. The state of the class is given by the value of all its member variables. This initial state is sometimes known as the test preconditions.
4. The expected result that will be returned from this method when it is called (or invoked) with a particular set of input arguments must be known in advance. This can be determined from the specification of the method functionality from the software design documents; which will be provided by the developer or the business end user.
5. The expected state of the class containing the method after the method has being invoked.

To run the test:

1. Invoke the method being tested and pass it to the set of arguments required by its signature
2. Check that the result returned from the method is equal to the expected result. The nature of this equality will depend on the result returned and the functionality of the method.
3. If the result is equal, the test is said to have passed or is successful. Otherwise, the test is said to have failed.
4. If there is an expected state that the class must be in after the method has being invoked, the actual state of the class is compared with this expected state. Again, if the result is equal, the test is said to have passed or is successful. Otherwise, the test is said to have failed. If the state of the class after the method has being invoked is not relevant, this step can be skipped.

UECS2354 Software Testing
Lab 02: Unit Testing and JUnit

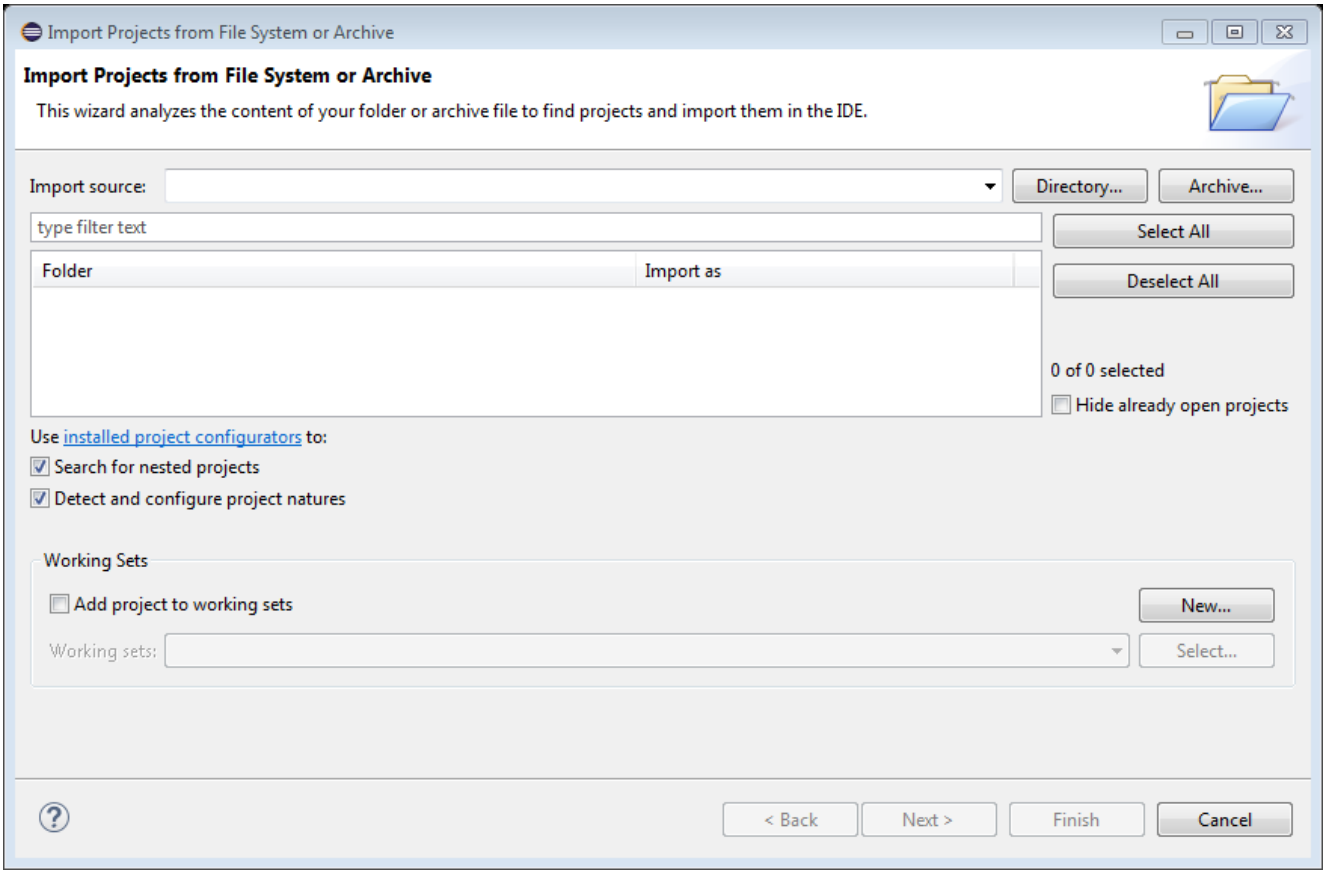
Whenever a test fails, it could be due to two reasons:

a) A bug exists in the method being tested (the most common case). In this case, the developer then proceeds to perform debugging to locate and remove the bug causing the test failure.

b) A bug exists in the test itself. Remember the test itself is also a piece of code, and can therefore be incorrectly implemented; just like the method being tested. For e.g. the method is called with the wrong arguments, the expected result is not correct because the functionality of the method is not understood correctly, and so on. Although this is less likely to occur, it is always important to consider this possibility before proceeding to start debugging the method when a test failure occurs.

Exercise:

Open the Lab2 project. **File > Open Projects from File System…**



Click on [Directory] to select the project folder (Lab02), leave other settings as default and click [Finish].

1. Write a separate class to test each method in the Calculator application.
   - It does not need to get input from user, pass the necessary values to the method instead.
   - Use the if-else structure to compare the actual result with expected result (based on the values passed to the method). Output appropriate message based on the comparison.

2. Study another 3 classes (FindLargestNumber, CountString, CombineString) in the same project and write tests to test their methods.

UECS2354 Software Testing
Lab 02: Unit Testing and JUnit

Part B: JUnit basics

The Knowledge

Three discrete goals for the framework:

- The framework must help us write useful tests.
- The framework must help us create tests that retain their value over time.
- The framework must help us lower the cost of writing tests by reusing code.

JUnit has many features that make it easy to write and run tests.

- Separate test class instances and class loaders for each unit test
- JUnit annotations to provide resource initialisation and reclamation methods: `@Before`, `@BeforeClass`, `@After`, and `@AfterClass`
- A variety of assert methods to make it easy to check the results of your tests
- Integration with popular tools like Ant and Maven, and popular IDEs like Eclipse, NetBeans, IntelliJ, and JBuilder

The test class, Suite, and Runner form the backbone of the JUnit framework.

1. A **test class** or test case is a class that contains one or more tests represented by methods annotated with **`@Test`**. Use a test class to group together tests that exercise common behaviours. There's usually a one-to-one mapping between a production class and a test class.

2. Suite or test suite usually groups test classes from the same package. A test suite is a convenient way to group together tests that are related. If you don't define a test suite for a test class, JUnit automatically provides a test suite that includes all tests found in the test class.

3. Runner or test runner executes your tests. A test runner allows you to tell JUnit how a test should be run. To run a basic test class, you needn't do anything special; JUnit uses a test runner on your behalf to manage the lifecycle of your test class, including creating the class, invoking tests, and gathering results.

The requirements to define a **test class** are:

- **the class must be public** and
- **contain a zero-argument constructor**.

If we don't define any other constructors, we don't need to explicitly define the zero-argument constructor; Java creates it for us implicitly.

The requirements to create a **test method** are:

- it must be **annotated with `@Test`**
- be **public**
- **take no arguments** and
- **return void**

JUnit creates a new instance of the test class before invoking each `@Test` method. This helps provide independence between test methods and avoids unintentional side effects in the test code. Because each test method runs on a new test class instance, we can't reuse instance variable values across test methods.

To perform **test validation**, we use the **assert methods** provided by the JUnit **`Assert`** class. We can either statically import these methods in our test class or import the JUnit `Assert` class itself.

The table below lists some of the most popular assert methods. Assert methods usually have overloaded versions with two or three parameters in the method signature. For the 3 parameter version, the first parameter is a string that is displayed as an error message if an assertion fails. The next two parameters in the 3 parameter version (as well as the 2 parameter version) follow a standard pattern: the first parameter is the expected value, and the second parameter is the actual value.

`assertXXX` method	Purpose
assertArrayEquals("message", A, B)	Asserts the equality of the `A` and `B` arrays.
assertEquals("message", A, B)	Asserts the equality of objects `A` and `B` by invoking the `equals()` method on the first object against the second.
assertNotEquals("message", A, B)	Uses `equals()` method to verify that `A` and `B` are not identical.
assertSame("message", A, B)	Asserts that the `A` and `B` objects are the same object. Whereas the previous assert method checks to see that `A` and `B` have the same value (using the `equals` method), the `assertSame` method checks to see if the `A` and `B` objects are one and the same object (using the == operator).
assertNotSame("message", A, B)	Uses == to verify that objects `A` and `B` are not the same.
assertTrue("message", A)	Asserts that the condition `A` is true.
assertFalse("message", A)	Asserts that the condition `A` is false.
assertNull("message", A)	Asserts that the object `A` is null.
assertNotNull("message", A)	Asserts that the object `A` isn't null.

** "message" is optional

NOTE:
assertEquals() for double values requires additional parameter, delta or epsilon.

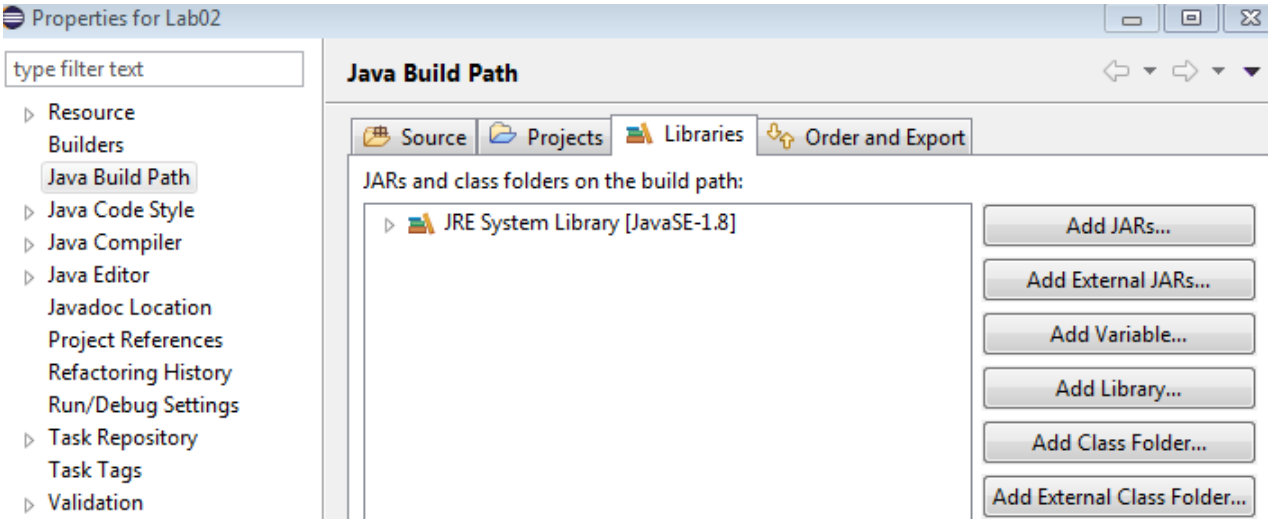
assertEquals(String message, double expected, double actual, double **delta**)

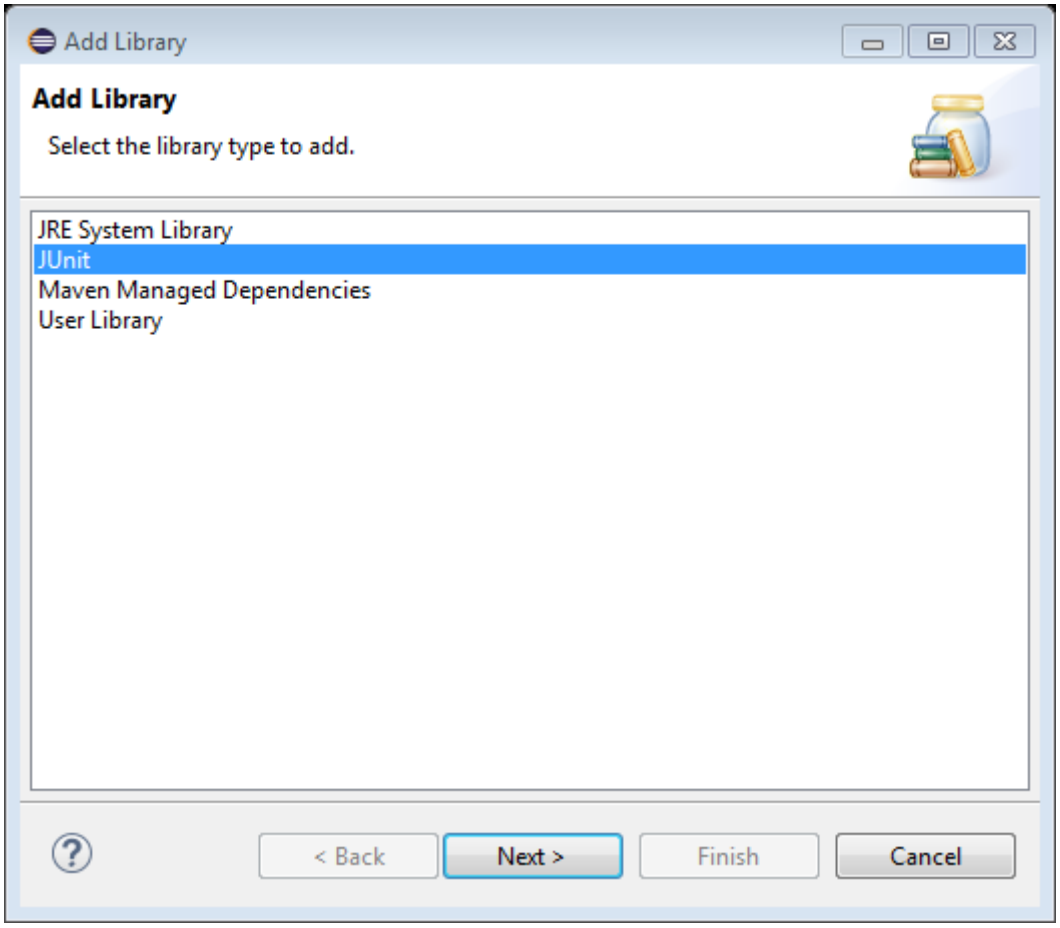
delta – the maximum difference between `expected` and `actual` for which both are still considered equal.

1.3 Creating basic JUnit unit tests

1) Eclipse Neon contains the JUnit package to create a JUnit unit test. However, if the JUnit package is not available, we can get the relevant jar files and add them to the build path.
   - `hamcrest-core-1.3.jar (classes)`
   - `hamcrest-core-1.3-javadoc.jar (Javadoc)`
   - `hamcrest-core-1.3-sources.jar (source code)`
   - `junit-4.12.jar (classes)`
   - `junit-4.12-javadoc.jar (Javadoc)`
   - `junit-4.12-sources.jar (source code)`

2) The library can be added during the creation of class using the wizard. If we want to add it later, **right-click** on **project name** in the package explorer and select **Build Path -> Add Libraries**. Select the **Junit** from the list. Click **[Next]** and then **[Finish]**. You should see the Junit library has been added into the project in the package explorer.





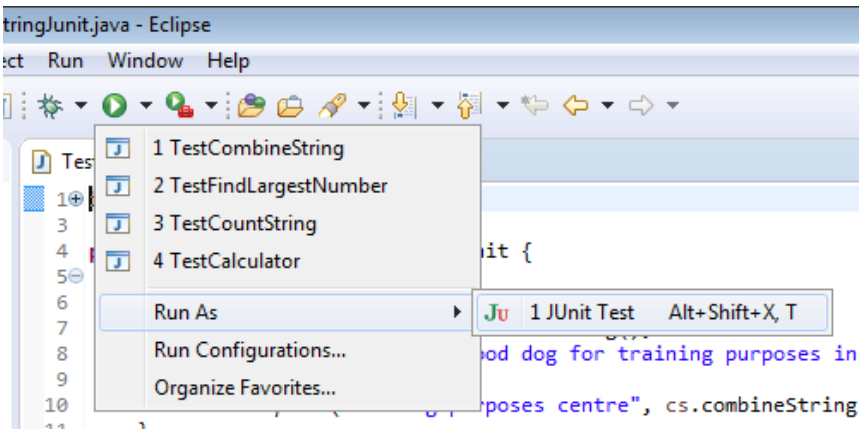
```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestCalculator{
  @Test
  public void testAddTwoNumbers() {
        Calculator calc = new Calculator();
        int result = calc.add(10, 15);
        assertEquals(25, result);
  }
}
```

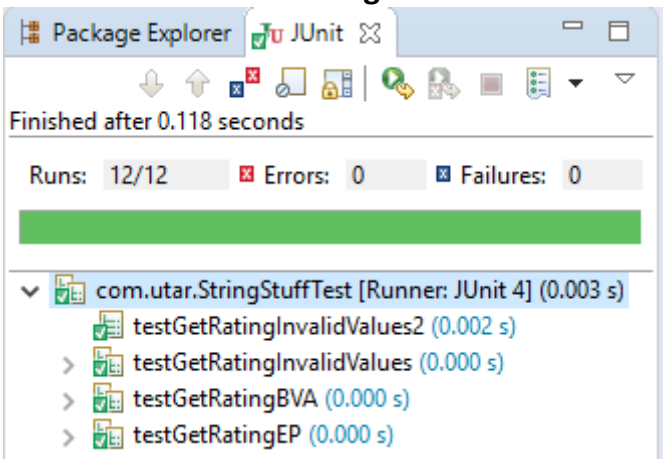
3) The above is an example of a simple JUnit test. Relevant classes and methods are imported from the **`org.junit.*`** package structure, which is part of the JUnit package. `TestCalculator` is a test class that contains a test methods (`testAddTwoNumbers()`) annotated with **`@Test`**.

4) All test methods marked with a `@Test` are executed as test by the JUnit framework. The object from the class under test is created, and the method under test is invoked in the usual manner; with the returned result stored in a variable. To check the result of the test, we call an `assertEquals` method, which we imported with a static import from the Assert class (`import static org.junit.Assert.*;`). The first parameter to this method is the expected result (25), and the second parameter is the actual returned value (result).

5) To run the test method, right-click on `TestCalculator` in the Package Explorer and then select **Run As -> JUnit Test**.



A JUnit view opens up in the stacked view area next to the Package Explorer, where you can get an overview of the results of the test run, including the number of failures and errors, as well as the stack trace of the errors. The tool bar for this view provides options for rerunning the test again, showing failures and so on. At the moment, there should be a green bar indicating that the entire test completed successfully. Notice how writing a unit test in JUnit is simpler than writing it manually. Also, a more intuitive interface is provided for checking and following up on test failures.



6) For a test failure, the failure trace at the bottom half of the view indicates the discrepancy between the expected result and the actual returned value, and also indicates the line number at which the `assertEquals` method throws this `AssertionError` exception. The `assertEquals` method will always throw this exception when the two parameters that it is comparing is not equal.

7) Online documentation for JUnit is available at http://junit.sourceforge.net/javadoc/overview-summary.html, or you can unzip them from the `junit-4.12-javadoc.jar` for offline viewing.

Exercises

1. Modify `Calculator` to introduce a bug and rerun `TestCalculator`.
2. Correct the error and rerun TestCalculator to make sure the error is fixed.
3. Write JUnit tests for others classes/methods in the project Lab02.