German University in Cairo
Media Engineering and Technology
Prof. Dr. Slim Abdennadher

Computer Programming Lab,
JUnit Testing

A **unit test** is a piece of code written by a developer that executes a specific functionality in the code to be tested. The percentage of code which is tested by unit tests is typically called **test coverage**.

A unit test targets a small unit of code, e.g., a method or a class, (local tests).

Unit tests ensure that code works as intended. They are also very helpful to ensure that the code still works as intended in case you need to modify code for fixing a bug or extending functionality. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

1 Using JUnit

1.1 The JUnit framework

JUnit in version 4.x is a test framework which uses annotations to identify methods that specify a test.

Typically a JUnit test is a method contained in a class which is only used for testing. This is called a Test class.

To write a test with the JUnit 4.x framework you annotate a method with the @org.junit.Test annotation.

In this method you use a method provided by the JUnit framework to check the expected result of the code execution versus the actual result.

1.2 Example JUnit test

Given the following code:

```
public class MyAddingAlgorithm {
    public int add(int x, int y){
        return x+y;
    }
}
```

We would like to make sure that the code does output Ḧello World:

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

public class TestCode{

   MyAddingAlgorithm instance;

   @Before
   public void setUp(){
      instance= new MyAddingAlgorithm();
   }
```

```
    @Test
    public void test1(){
    assertEquals("The output should be 5", 5, instance.add(2,3));
    }

    @Test
    public void test2(){
    assertEquals("The output should be -6", -6, instance.add(-2,-4));
    }

}
```

1.3 Annotation

JUnit 4.x uses annotations to specify what a method in the test class does. Annotation are used to mark methods and to configure the test run. The following table gives an overview of the most important available annotations.

Annotation	**Description**
@Test public void method()	The @Test annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.
@Ignore	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

1.4 Assert statements

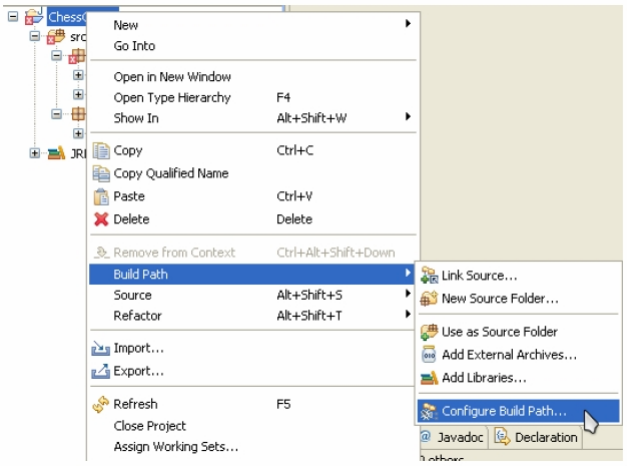
JUnit provides static methods in the Assert class to test for certain conditions. These assertion methods typically start with assert and allow you to specify the error message, the expected and the actual result. An assertion method compares the actual value returned by a test to the expected value, and throws an AssertionException if the comparison test fails. The following table gives an overview of these methods. Parameters in [] brackets are optional.

Assertion Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The String parameter is optional.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertFalse([message], boolean condition)	Checks that the boolean condition is false.
assertEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is null.
assertNotNull([message], object)	Checks that the object is not null.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.
assertNotSame([String], expected, actual)	Checks that both variables refer to different objects.

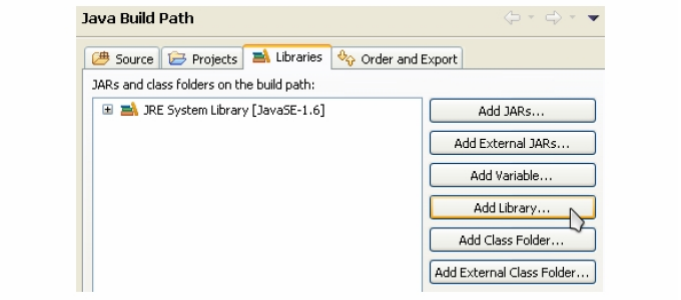
1.5 Configuring your project to use JUnit

Eclipse needs to be told that it can use the JUnit library. You need to configure the project to include the JUnit 4 library.

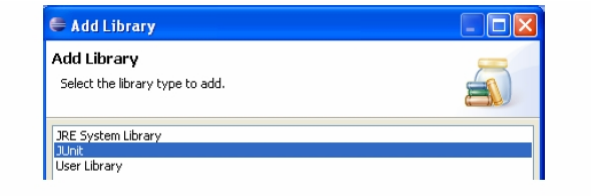
- Right click on the project and choose the "Configure Build Path.." menu.



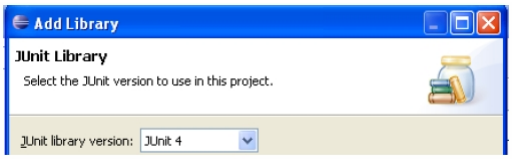
- Under the "Libraries" tab, choose to add a library



- Choose to add a JUnit library



- Choose JUnit 4 from the drop down list and click finish



1.6 Let's write a test file!

The class `ArrayHelperMethods` is posted on the MET website. The class has a method `getArrayValuesAverage` which returns the average of the elements of an array of integer values.

Now we would like to write a test class that makes sure the code is written correctly.

- What is the basic functionality you would like to test?
- what are the different cases for this problem?
- Are there any unchecked exceptions that can occur in the code? If yes, make sure to test that!