Workshop 2 : JUnit testing

In this exercise, we will cover the functionality of the testing framework JUnit and how basic unit tests are written for the framework. Most IDE's support JUnit, use any you like.

Preparation

1. Download the project "ws_junit.zip" (a Maven project) from the course home page > Workshops and import it into your IDE of choice.

2. We will work with a pair of already implemented classes, List and Node (we only test List). If you haven't heard of a linked list, check out!

3. Inspect the code. Take special note of the toString() method inside the Node class. This is a good method to use during debugging and development. Testing of the List class

Writing Tests

Now we can begin to write some JUnit tests.

1. In class TestList, edit the testAdd() method to look like the following piece of code (possibly need to fix some imports, the IDE should be able to handle this, lightbulbs or similar...).:

   ```
   @Test
   public void testAdd() {
         List l = new List();
         l.add(1);     // Call method to test
         assertTrue(l.getLength() == 1); // The logical check
   }
   ```

   The last row in the method is called an assertion. If the expression holds true, it means that the unit test was successful. If, on the other hand, the assertion fails, an error is reported.

   Run the test! It should pass.

   NOTE 1: This is how all unit tests work. The tests either fail or succeed. Manual reporting (using for example System.out.println()) should never be needed except during development and under heavy testing when writing the unit test.
   However, the printout should always be removed once the test is completed.

NOTE 2: The testAdd() method is testing a void method. This means that there is no return value that can be checked. Consequently, there needs to be some other way to check if the method operates properly (in this case using getLength()).
If there is no method available to check, a method may be added to the class in order for it to be testable (for this course).

2. Checkout this and this.

3. You can induce a failure by modifying the assertion inside the testAdd() method to something that should fail. Do and run again (then restore)!

4. Next, let's write the test for the remove() method of the List class. The method removes the first node in the List and returns the content (or value) of that node. Make use of the return value when you write the test. The test should have at least two assertions (the more checks the better). If test fails, use the debugger to debug and correct the method. How to debug in IntelliJ and NetBeans.

5. You should now write a test for the get() method. You should do the following operations in the test:

    a) Add five values to the list.
    b) Return (and check) the value for index 2 in the list.

    Something is wrong with the get() method and it does not work correctly. Use the debugger in order to try to find out what's wrong.

6. It is also important that we test that methods behave correctly (and report an error) when they get invalid data. The get() method of the List class actually has such an error check. To test it, we can create the following test:

```
@Test(expected=IllegalArgumentException.class)
public void testGetBadIndex() {
      // Get a list then ...
      list.get(-1); // Exception!!!
}
```

    The test checks that the expected exception occurs when we request an index that doesn't exist.

7. Create the method copy() inside the List class. This method should create and return a deep copy of the list.

8. Create a unit test for the copy() method.

Fixtures

Many tests need some kind of initialization before they can run. In certain cases, a test method might also need to run some shutdown code in order to free up resources that were needed during the test. Operations such as these can be done inside special methods that are executed by JUnit in preparation of each test. These are called fixtures.

1. To try out this feature, add the code below to the ListTest class. Note that the printouts below were added just for demonstration purposes and should never be part of a JUnit test class under normal conditions.

```
@BeforeClass
public static void beforeClass(){  //First of all
      out.println(”Before class”);
}

@AfterClass
public static void afterClass(){   //Last of all
      out.println(”After class”);
}

@Before
public void before(){  //Before each test method
      out.println(”Before”);
}

@After
public void after(){ //After each test method
      out.println(”After”);
}
```