JUnit Tutorial

Table of Contents

1. Import JUnit in Eclipse project

- **import junit.jar**
  - Right-click on the project name, and choose Properties.
  - In the tree on the left, select Java Build Path. Next,
  - Choose Add External JARs… and browse to find junit.jar.
  - It will be located in <eclipsedir>\plugins\org.junit_<version number>\junit.jar.
  - Once you successfully import junit.jar, close the Properties page.

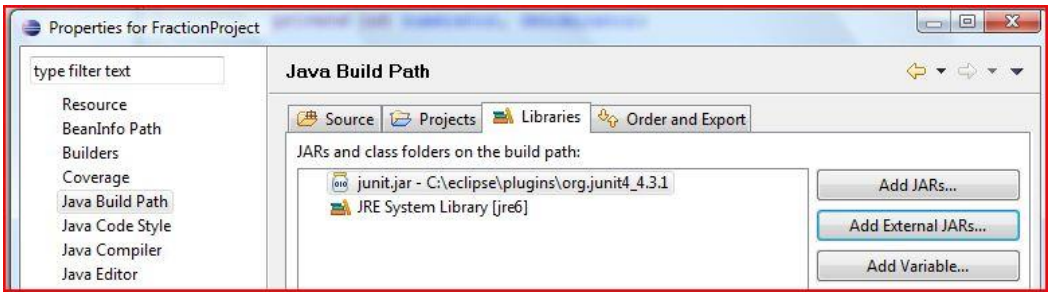

Figure 1 Import JUnit in the current project

2. Create a test case using JUnit

- Right-click the package you wish to place the test in, and choose New…, then select Other…
- In the tree on the left, expand the Java branch, and choose JUnit.
- Then in the list on the right select TestCase. This allows us to create a new test case for the selected class.

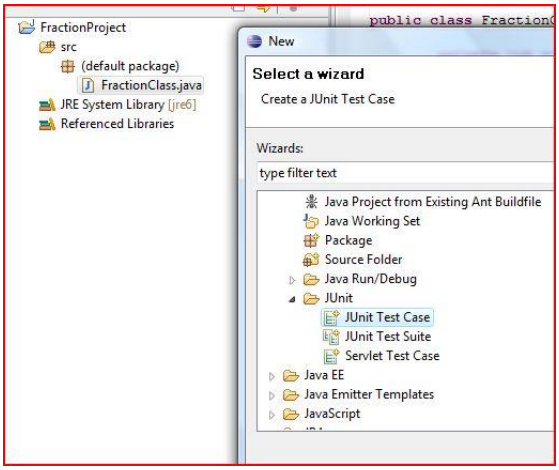

Figure 2 Create a Test Case in JUnit (step a))

- A Wizard will appear for creating a JUnit TestCase.
- Click on the Browse… button beside Test Class to select the class for testing.
- Appending "Test" to the name of the class being tested is the default test case naming scheme in JUnit.
- The check-boxes at the bottom allow you to add stubs for additional useful methods.

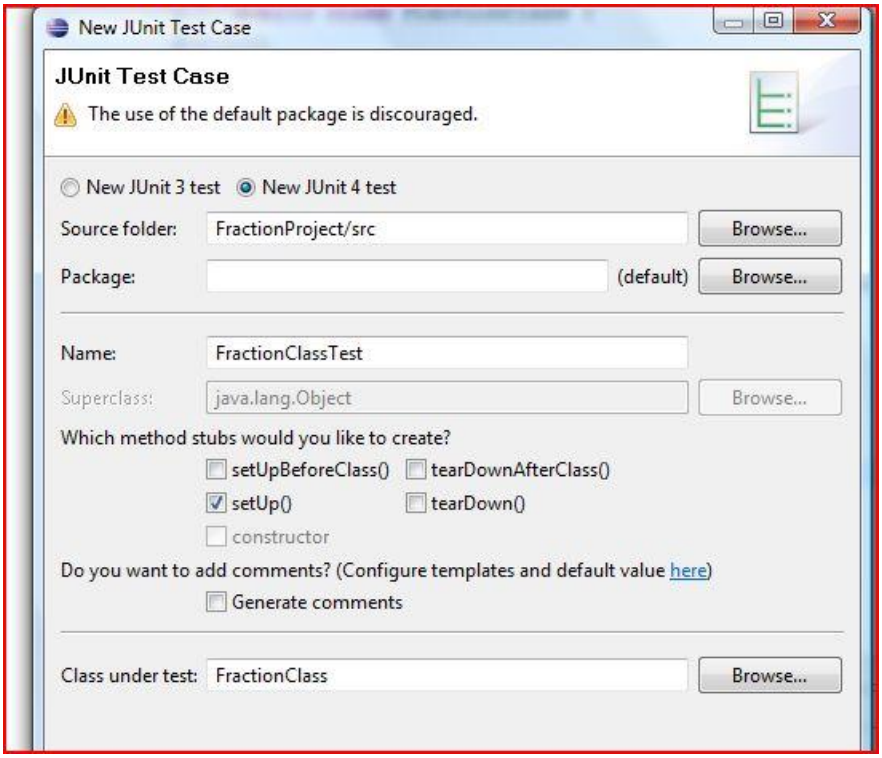

Figure 3 Create a Test Case in JUnit (step b))

- The next screen allows you to select which of class's methods you want test stubs generated for.

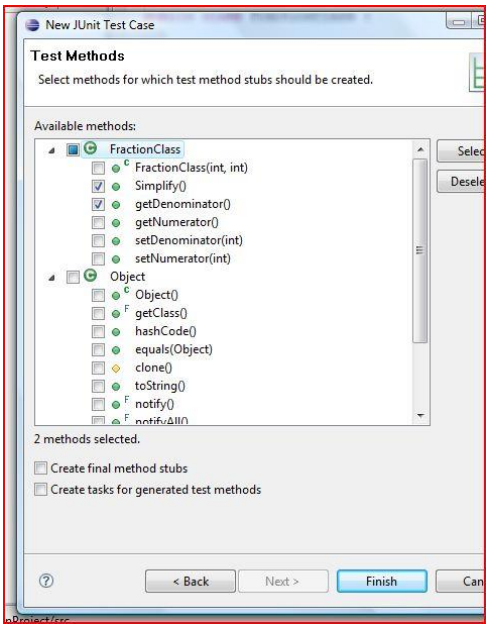

Figure 4 Create a Test Case in JUnit (step c))

- The class is created with the appropriate method stubs generated for us. All we need to do now is write the tests.

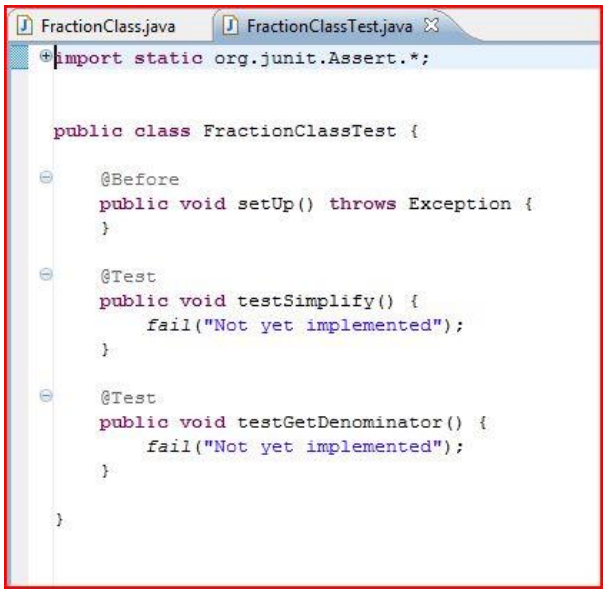

Figure 5 Create a Test Case in JUnit (step d))

3. Writing the Tests

- **First example**

  We write tests for the testGetDenominator() first. We will create a couple of fractions and call getDenominator on each. Create two private FractionClass member variables like the following:

```
private FractionClass fc1, fc2;
```

In the setup method, add two lines to construct the fractions.

```
fc1 = new FractionClass(12,30);
fc2 = new FractionClass(-25,7);
```

Each time a test method is called, the setUp() will run these two lines first, so at the start of each test the fractions will always be 12/30 and -21/5.

Now to implement our testGetDenominator() method, add the following code to the method body.

```
int result = fc1.getDenominator();
assertTrue("getDenominator() returned " + result + " instead of 30.",
result == 30);
result = fc2.getDenominator();
assertEquals(7, result);
```

Since FractionClassTest is a subclass of the junit.framework.TestCase, it inherits a variety of methods for testing assertions (i.e. conditions that should be true) about the state of variables in the test. These include assertEquals, assertTrue/False, assertSame/NotSame, and assertNull/NotNull. There is also a fail() method that just causes the current TestCase to fail. These methods generally come in two flavours: a plain version, and a version with a String parameter to provide details about the assertion that failed. In the code above, you can see examples of each. Whenever the assertion in one of the methods fails, the execution of that test method is terminated, and JUnit will record that test as having failed.

- **Second example**

Next we will implement testSimplify().

```
fc1.Simplify();
assertEquals(2, fc1.getNumerator());
assertEquals(5, fc1.getDenominator());
```

This will verify that our Simplify() method correctly reduces 12/30 to 2/5.

4. Running the Tests

Now that the tests have been written, we would like to run them. The process for running JUnit tests is very similar to that for running regular Java Applications. From the *Run As* select JUnitTest. The screen should look like the following, assuming FractionClassTest was selected when you opened the Run menu.

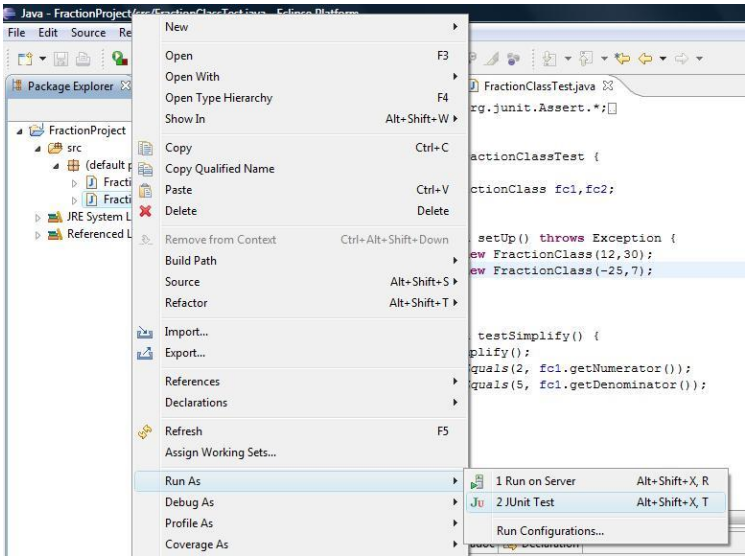

Figure 6. Executing the created Test cases

The test cases run to completion successfully, and a green bar is displayed in the JUnit view.

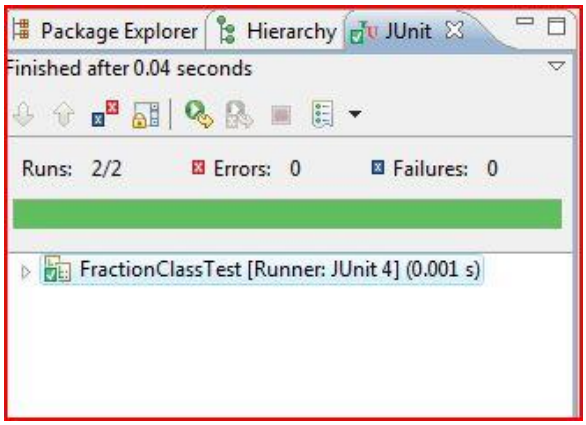

Figure 7. "Successfully" executing the test cases – green bar

- Now, let's add another piece of code to our testSimplify() method.

```
fc2.Simplify();
assertEquals(-25, fc2.getNumerator());
assertEquals(7, fc2.getDenominator());
```

This looks very similar to the previous code in testSimpify(). However, consider what happens when we run the tests again. The JUnit view will be displayed on the left of the screen (if it is not already visible) and it will show a red bar indicating that one or more tests failed.

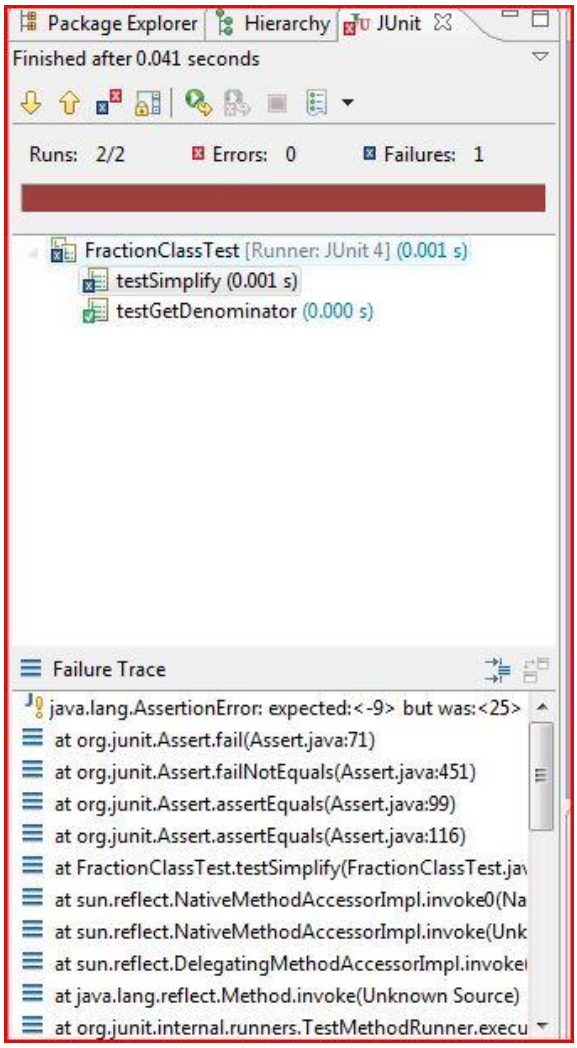

Figure 8. ''UnSuccessfully'' executing the test cases – red bar

The top part of the view displays the names of the test method(s) that failed. By doubleclicking on the method name, the corresponding method will be opened in the Java editor. The bottom of the view shows a stack trace of the method calls leading up to the failure. Here we can see that since we used the assertEquals method, it gives us a message about what the expected value was versus the value received. Double-clicking on the second line from the top of the stack trace will jump to the specific line at which the failure occurred.

The error occurred in the testSimplify() method, and it looks as though the absolute value of the result was correct, but the sign was wrong. By commenting out the failed assertion, we can verify that the sign of the denominator is also reversed from what was expected. A simple fix for this is to check if the denominator is ever negative, and if so negate both numerator and denominator, effectively "moving" the negative sign to the denominator. The code for this is commented out in the Simplify() method of FractionClass.java, so we can uncomment it now.

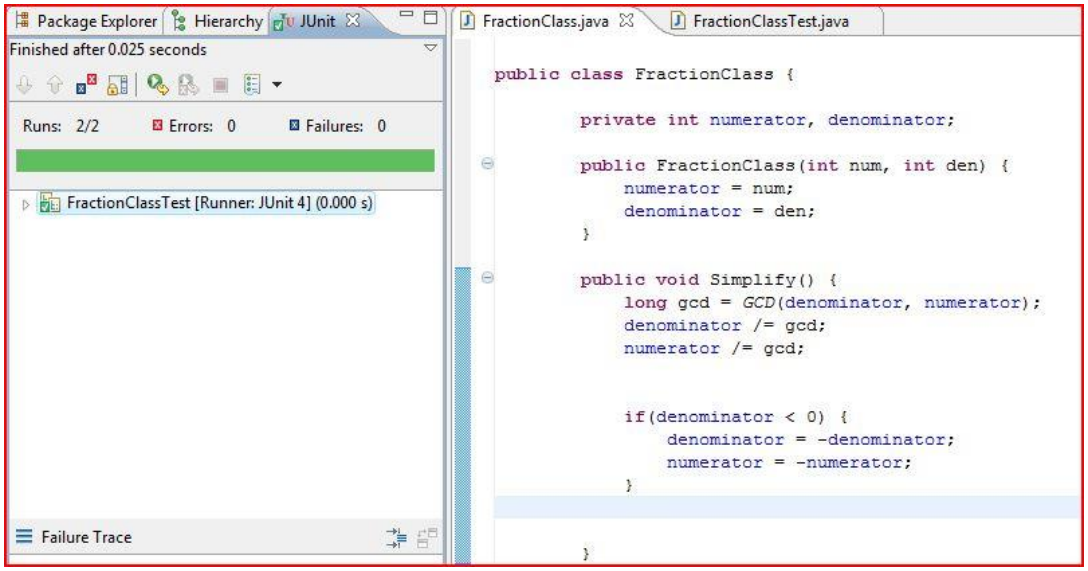

Figure 9. ''Successfully'' executing the test cases after eliminating the source of failure

5. JUnit 3 vs. JUnit 4

- JUnit 4 – annotations are introduced
- A comparison between JUnit 3 and JUnit 4 is provided next.

JUnit 3	JUnit 4	Observații
Crearea clasei Test Case		
`public class FractionClassTest_old extends TestCase { …}`	`public class FractionClassTest { …}`	
- orice clasă de testare este derivată din `TestCase`	- fără constrângeri referitoare la derivare	
Adnotarea @BeforeClass		
`public static void setUpBeforeClass() throws Exception { System.out.println("Setup for all subsequent tests..."); //setup }`	`@BeforeClass public static void setUpAll() { System.out.println("Setup for all subsequent tests..."); //setup }`	metodă statică ce se va executa o singură dată, înainte de rularea vreunui Test Case din clasă (e.g., conectarea la baza de date);
- metoda este denumită (obligatoriu) `setUpBeforeClass`;	- metoda este precedată de adnotarea `@BeforeClass`; - fără constrângeri referitoare la stabilirea identificatorului metodei;	
Adnotarea @AfterClass		
`public static void tearDownAfterClass() throws Exception { System.out.println("\ntearing all down"); }`	`@AfterClass public static void tearDownAll() { System.out.println("\ntearing all down"); }`	metodă statică ce se va executa o singură dată, după rularea tuturor **Test Case**-urilor din clasă (e.g., deconectarea de la baza de date);
- metoda este denumită (obligatoriu) `tearDownAfterClass`;	- metoda este precedată de adnotarea `@AfterClass`; - fără constrângeri referitoare la stabilirea identificatorului metodei;	
Adnotarea @Before		
`public void setUp() { fc1 = new FractionClass(12,30); fc2 = new FractionClass(-25,7); }`	`@Before public void setup() { fc1 = new FractionClass(12,30); fc2 = new FractionClass(-25,7); }`	metodă care se va executa înainte de fiecare **Test Case** din clasă (e.g., inițializarea cu date de intrare);
- metoda este denumită (obligatoriu) `setUp`;	- metoda este precedată de adnotarea `@Before`; - fără constrângeri referitoare la stabilirea identificatorului metodei;	

<table>
<tr><th colspan="3">Adnotarea @After</th></tr>
<tr><td>

```
public void tearDown() {
	fc1 = fc2 = null;
	System.out.println(fc1);
	System.out.println(fc2);
}
```

</td><td>

```
@After
public void teardown() {
	fc1 = fc2 = null;
	System.out.println(fc1);
	System.out.println(fc2);
}
```

</td><td rowspan="2">metodă care se execută după fiecare Test Case din clasă (e.g., ștergerea/ dealocarea variabilelor temporare);</td></tr>
<tr><td>- metoda este denumită (obligatoriu) <code>tearDown</code>;</td><td>- metoda este precedată de adnotarea <code>@After</code>;
- fără constrângeri referitoare la stabilirea identificatorului metodei;</td></tr>
<tr><th colspan="3">Adnotarea @Test</th></tr>
<tr><td>

```
public void testSimplify() {
	System.out.println("\ntestSimplify");
	fc1.Simplify();
	assertEquals(2, fc1.getNumerator());
	assertEquals(5, fc1.getDenominator());
}
```

</td><td>

```
@Test
public void mySimplifyTest() {
	System.out.println("\ntestSimplify");
	fc1.Simplify();
	assertEquals(2, fc1.getNumerator());
	assertEquals(5, fc1.getDenominator());
}
```

</td><td rowspan="2">metodă Test Case propriu-zisă (e.g., testează comportamentul funcției <code>simplify()</code> pentru anumite date de intrare care, de exemplu, au fost inițializate într-o metodă <code>setUp()</code> sau <code>setup()</code>);</td></tr>
<tr><td>- metoda începe (obligatoriu) cu prefixul <code>test</code> (e.g., <code>testSimplify</code>, <code>testGetDenominator</code>), altfel nu este considerată un Test Case;</td><td>- metoda este precedată de adnotarea <code>@Test</code>;
- fără constrângeri referitoare la stabilirea identificatorului metodei;</td></tr>
<tr><th colspan="3">Adnotarea @Test (expected = <<ClassException>>.class)</th></tr>
<tr><td>

```
public void testDivisionException(){
	System.out.println("\ntestDivisionException"
);
	fc2.setDenominator(0);
	try {
		fc1.div(fc2);
	} catch (Exception e) {
		e.printStackTrace();
	assertTrue(e.getMessage().equals("Division
by zero!"));
	}
}
```

</td><td>

```
@Test (expected = Exception.class)
public void testDivisionException() throws
Exception //passed {
	System.out.println("\ntestDivisionExceptio
n");
	fc2.setDenominator(0);
	fc1.div(fc2);
}
SAU
@Test(expected=IndexOutOfBoundsException.class)
public void outOfBounds()//passed {
	new ArrayList<Object>().get(1);
```

</td><td>metodă Test Case propriu-zisă care pune în evidență aruncarea unei excepții</td></tr>
</table>

<table>
<tr><td></td><td>}</td><td></td></tr>
<tr><td>- metoda începe (obligatoriu) cu prefixul <code>test</code> ;
- se foloseşte un apel assertAAA(…) care pune în evidență succesul sau eşecul testului;</td><td>- metoda este precedată de adnotarea <code>@Test</code> cu clauza <code>expected</code>, prin care se precizează tipul excepției aşteptate la execuție;
- dacă metoda testată aruncă excepție, Test Case-ul va avea starea passed;
- fără constrângeri referitoare la stabilirea identificatorului metodei;</td><td></td></tr>
<tr><td colspan="3">Adnotarea @Test (timeout = <<value>>)</td></tr>
<tr><td></td><td><pre>@Test (timeout=10)//passed
public void testDivision() {
	System.out.println("\ntestDivision");
	try {
		fc1.div(fc2);
	} catch (Exception e) {
		e.printStackTrace();
	}
	assertEquals(-14, fc1.getNumerator());
	assertEquals(125, fc1.getDenominator());
}
SAU
@Test(timeout=100)//failed
public void infinity() {
	while(true);
}</pre></td><td>metodă Test Case propriu-zisă care pune în evidență execuția într-un interval de timp precizat</td></tr>
<tr><td></td><td>- metoda este precedată de adnotarea <code>@Test</code> cu clauza <code>timeout</code>, prin care se precizează timpul maxim de execuție aşteptat, exprimat în milisecunde;
- dacă la execuție timpul depăşeşte valoarea dată, Test Case-ul va avea starea failed;
- fără constrângeri referitoare la stabilirea identificatorului metodei;</td><td></td></tr>
<tr><td colspan="3">Adnotarea @Ignore</td></tr>
<tr><td><pre>public void ttestGetDenominator() {
	System.out.println("\ntestGetDenominator");</pre></td><td><pre>@Ignore
@Test</pre></td><td>metodă Test Case</td></tr>
</table>

<table>
<tr>
<td>

```
      int result = fc1.getDenominator();
      assertTrue("getDenominator() returned " +
result + " instead of 30.", result == 30);
      result = fc2.getDenominator();
      assertEquals(7, result);
      }
```

</td>
<td>

```
public void testGetDenominator() {
      System.out.println("\ntestGetDenominator")
;
      int result = fc1.getDenominator();
      assertTrue("getDenominator() returned " +
result + " instead of 30.", result == 30);
      result = fc2.getDenominator();
      assertEquals(7, result);
}
```

</td>
<td rowspan="2">care va fi ignorată la rularea testelor (e.g., se foloseşte atunci când codul sursă se modifică, iar Test Case-ul corespunzător nu s-a adaptat încă).</td>
</tr>
<tr>
<td>- orice metodă care nu are prefixul <code>test</code> (e.g., <code>ttestGetDenominator()</code>) nu va fi considerată un Test Case şi va fi ignorată.</td>
<td>- metoda este precedată de adnotarea <code>@Ignore</code>;
- fără constrângeri referitoare la stabilirea identificatorului metodei.</td>
</tr>
</table>