Java

Testing

FSR Informatik

October 11, 2016

Overview

Testing

Testing helps to minimize bugs in your program.
A good program should be tested.

Tests never show the absence of bugs.

Test Case

A test case is a test of a part of a program with specified input. It always includes an expected result. It can succeed or fail.

A test case should not be created by the person who implemented the software under the test.
A test case should be written before the implementation.

After a tested part is altered the test case is reused to examine if the new part still works as intended.

Covering the Cases

A test case includes the normal case.
Every type of possible input should be tested. But do not write tests for every possible input.

Focus on edge cases and corner cases of your software.
Always ask yourself: *What I expect my program to do with defective input.*

Getting JUnit

JUnit is part of your Eclipse installation.

Alternatively download ***junit-4.12.jar*** *from the Maven homepage:*
`http://search.maven.org/#browse|599447922`
and follow the instructions on `http://junit.org`

Example Class

First we need a class we want to test.

```
1       public class Calc {
2
3           public int add(int x, int y) {
4                   return x + y;
5           }
6       }
7
```

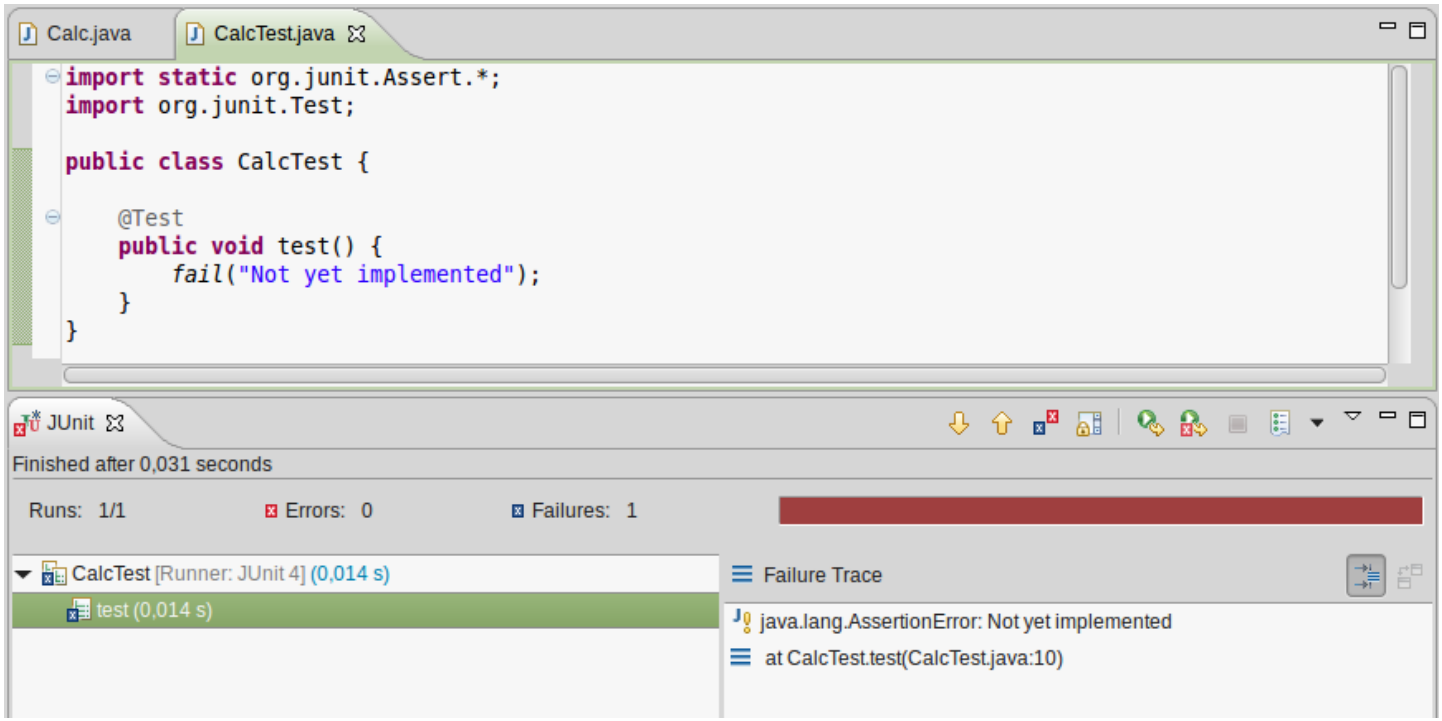
1. Create a test via File > New > JUnit TestCase.
2. Select *New JUnit 4 Test*.
3. The *Name* is **CalcTest** and the *Class under test* is **Calc**.
4. Now press Finish and confirm the addition of the JUnit libary.

Test Class

This is an empty test class. One test is performed by the method `test()`. The method is annoted with **@Test**.

```
1     import static org.junit.Assert.*;
2     import org.junit.Test;
3
4     public class CalcTest {
5
6         @Test
7         public void test() {
8             fail("Not yet implemented");
9         }
10    }
11
```

You can run the test with Ctrl + F11. The test will fail.


Calc.java
CalcTest.java
import static org.junit.Assert.*;
import org.junit.Test;
public class CalcTest {
@Test
public void test() {
fail("Not yet implemented");
}
}
JUnit
Finished after 0,031 seconds
Runs: 1/1
Errors: 0
Failures: 1
CalcTest [Runner: JUnit 4] (0,014 s)
test (0,014 s)
Failure Trace
java.lang.AssertionError: Not yet implemented
at CalcTest.test(CalcTest.java:10)

The first Test

`assertTrue()` checks if the condition is true.
`assertFalse()` checks if the condition is false.

```
1   import static org.junit.Assert.*;
2   import org.junit.Test;
3
4   public class CalcTest {
5
6       @Test
7       public void testNormalAddition() {
8           Calc calc = new Calc();
9           assertTrue(calc.add(3, 5) == 8);
10          assertFalse(calc.add(3, 5) == 0);
11      }
12  }
13
```

If and only if all assert methods add up the test will succeed.

Classic Import

If you use *import* instead of **import static** you have to call the assert methods via *Assert.assertTrue()* instead of **assertTrue()**.

```
1   import org.junit.Assert;
2   import org.junit.Test;
3
4   public class CalcTest {
5
6       @Test
7       public void testNormalAddition() {
8           Calc calc = new Calc();
9           Assert.assertTrue(calc.add(3, 5) == 8);
10          Assert.assertFalse(calc.add(3, 5) == 0);
11      }
12  }
13
```

Multiple Tests

You can put multiple tests in one test class.

```
1   public class CalcTest {
2
3       @Test
4       public void testNormalAddition() {
5           Calc calc = new Calc();
6           assertTrue(calc.add(3, 5) == 8);
7       }
8
9       @Test
10      public void testNegativeAddition() {
11          Calc calc = new Calc();
12          assertTrue(calc.add(3, -5) == -2);
13      }
14  }
15
```

Fixture - Before

A method annoted with **@Before** will run before the test methods.
Use a method to set up an environment for the tests.
Now the tests will become smaller and easier to read.

```
1   public class CalcTest {
2
3       private Calc calc;
4
5       @Before
6       public void init() {
7           this.calc = new Calc();
8       }
9
10      @Test
11      public void testNormalAddition() {
12          assertTrue(this.calc.add(3, 5) == 8);
13      }
14  }
15
```

Fixture - After

A method annoted with **@After** will run after the test methods. Use the method if there are things to tidy up:

- close database connections
- close file streams
- delete test databases or test files

Example Bookshelf

```
1       public class Bookshelf {
2
3           private String[] books;
4
5           public Bookshelf () {
6               this.books = new String[10];
7               this.books[0] = "Harry Potter";
8           }
9
10          public String getBook(int number) {
11              return this.books[number];
12          }
13      }
14
```

Test Bookshelf

If your software throws exceptions you have to verify the proper throwing of these exceptions.

```
1   public class BookshelfTest {
2
3       private Bookshelf bookshelf;
4
5       @Before
6       public void init() {
7           this.bookshelf = new Bookshelf();
8       }
9
10      @Test(expected = ArrayIndexOutOfBoundsException.
    class)
11      public void testOutOfBounds() {
12          this.bookshelf.getBook(15);
13      }
14  }
15
```

Test Bookshelf - in Detail

```
1       @Test(expected = ArrayIndexOutOfBoundsException.class)
2       public void testOutOfBounds() {
3           this.bookshelf.getBook(15);
4       }
5
```

expected has to be a subclass of the interface *Throwable*.

`ArrayIndexOutOfBoundsException.class` shows the class of the exception the test expects to be thrown.

There is no assert method needed in this test case.

Why Unit Testing?

Unit tests do not need human interaction. The automation of testing needs less time than testing by hand.
It is easy to rerun the tests when software is altered.

Links

JUnit Javadoc:
`http://junit.org/javadoc/latest/index.html`

An extensive JUnit tutorial:
`http://www.vogella.com/tutorials/JUnit/article.html`