

Forum:[Hack Forums](#)

Thread: [Introduction to Unit Testing in Java with JUnit Tests \[Red Lions\]](#)

SIM-2015410 - October 25, 2015, 6:53 pm

Unit Testing in Java with JUnit Tests

Introduction:

Testing software is an extremely important stage of development. You cannot just write a piece of code and hope it works. Now, you might think that you could just test your code manually, but when you are developing huge pieces of software and have to run many, many tests, that can become tedious. That is where unit testing with JUnit comes in. It allows you to write as many tests as you like, and whenever you run it it will test them all and let you know which pass and fail. This is really useful to make sure that you don't break unexpected things during development of large projects. So even if you are writing a simple program, it is a good habit to get into.

Getting started

Here is an example of a class we might want to test:

```
Code public class Calculator {
    public int multiply(int x, int y) {
        return x * y;
    }
    public int add(int x, int y) {
        return x + y;
    }
    public int subtract(int x, int y) {
        return x - y;
    }
}
```

We have a simple calculator class which can add, subtract and multiply 2 integers. Now, let's test it.

Here is an example test class:

```
Code import org.junit.Test;
import static org.junit.Assert.assertEquals;
```

```
public class ExampleTests {
```

```
    @Test
```

```
    public void additionTest() {
```

```
        Calculator testCalc = new Calculator();
```

```
        // assert statements
```

```
        assertEquals("1 plus 2 equals 3", 3, testCalc.add(1, 2));
```

```
        assertEquals("80 plus 10 equals 90", 90, testCalc.add(80, 10));
```

```
        assertEquals("-3 plus 8 equals 5", 5, testCalc.add(-3, 8));
```

```
        assertEquals("-12 plus -13 equals -25", -25, testCalc.add(-12, -13));
```

```
        assertEquals("1 plus 0 equals 1", 1, testCalc.add(1, 0));
```

```
    }
```

```
}
```

We have 5 assert statements in that one test. We use the assertEquals method to test if something is true. The first parameter is a message about the test so we know which one is failing/passing, the second parameter is the expected answer, and the third is what we are testing. The test will fail if any of the 5 assertEquals statements fail. Testing is only as good as the tests you write, so you need to become good at figuring out what edge cases might break your code and testing them all. Think of the weird cases, like the maximum and minimum values, etc.

Annotations

The @Test annotation, as you might guess, identified the method as a Test. You can also add failure cases to the annotation, for example timeouts and runtime exceptions:

Code @Test (expected = Exception.class) //the test will fail if the exception is NOT thrown. You can use try-catch and the fail() statement instead

@Test(timeout=100) //the test will fail if it takes longer than 100 milliseconds

If you need to initialize variables, etc. before the tests are executed, you can create another method with the @Before annotation.

Code @BeforeClass

```
public static void startup() {  
    // this is run only once, before any tests are run  
}
```

@Before

```
public void setup() {  
    // this will be executed before every test is run  
}
```

Of course, we also have corresponding @After annotations which do the opposite of the @Before:

Code @After // run after each test

@AfterClass // must be static, used to clean up after all tests

Finally, we have the @Ignore annotation, which pretty much does the same thing as commenting the test out. It is just used, as the name suggests, when you want to ignore a test. You can add a reason in the brackets like this:

Code @Ignore("Reason")

Assert Statements

You have already seen one type of assert statement - the assertEquals("message", expected, actual). It tests that two values are the same (for arrays, the reference is checked not the contents). There is a 4th, optional parameter which is the tolerance: if you are using floating point numbers, this sets the number of decimal places to compare (otherwise rounding errors might cause the test to fail).

Next up we have a simple statement: fail("message"). All it does, is causes the test to fail. You can use this if you write conditional statements yourself in the test rather than using the assert statements, to fail if certain parts of the code are reached (for example, an else clause or a catch clause).

Some other assert statements are available: There are a couple of assert statements for booleans - assertTrue("message", condition) and assertFalse("message", condition) which just check whether the condition is true or false respectively.

assertNull("message", object) and assertNotNull("message", object) check whether an object is null or not null respectively.

assertSame("message", expected, actual) and assertNotSame("message", expected, actual) check whether the two parameters reference the same object.

Running the Tests

After you have written tests, you can run them by choosing Run > Run as > JUnit Test. This will open a JUnit Test window in Eclipse, where the outcome of all the tests will be visible.

If you have multiple test classes, you can use a TestSuite to run them all.

There are more advanced uses of JUnit tests, for example using parameterized tests, categories, and JUnit rules, but I won't cover them here. This tutorial should give you a good introduction to JUnit tests and their uses. For further reading, check out the [JUnit website](#).

This tutorial was written by MeshCollider ?

??Austin?? - *October 26, 2015, 3:27 am*

Really HQ tutorial man. Before I never considered to use JUnit simply because I thought it was "pointless" but I will have to try it in my next big project!

Remix - *October 26, 2015, 7:53 am*

Great tutorial mate.

Keep up the good work.