**Assert Methods**.by [Jakob Jenkov](http://jakob.jenkov.com)

The secret of implementing JUnit unit tests, is in the use of the assert methods in the class org.junit.Assert. In this text I will take a closer look at what assert methods are available in this class.

Here is a list of the assert methods:

* [**assertArrayEquals()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertArrayEquals)
* [**assertEquals()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertEquals)
* [**assertTrue() + assertFalse()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertTrue)
* [**assertNull() + assertNotNull()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertNull)
* [**assertSame() + assertNotSame()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertSame)
* [**assertThat()**](http://tutorials.jenkov.com/java-unit-testing/asserts.html#assertThat)

Throughout the rest of this text I will explain how these assert methods work, and show you examples of how to use them. The examples will test an imaginary class called MyUnit. The code for this class is not shown, but you don't really need the code in order to understand how to test it.

**assertArrayEquals()**

The assertArrayEquals() method will test whether two arrays are equal to each other. In other words, if the two arrays contain the same number of elements, and if all the elements in the array are equal to each other.

To check for element equality, the elements in the array are compared using their equals() method. More specifically, the elements of each array are compared one by one using their equals() method. That means, that it is not enough that the two arrays contain the same elements. They must also be present in the same order. Here is an example:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testGetTheStringArray() {

MyUnit myUnit = new MyUnit();

String[] expectedArray = {"one", "two", "three"};

String[] resultArray = myUnit.getTheStringArray();

assertArrayEquals(expectedArray, resultArray);

}

}

First the expected array is created. Second the myUnit.getTheStringArray() method is called, which is the method we want to test. Third, the result of the myUnit.getTheStringArray() method call is compared to the expected array.

If the arrays are equal, the assertArrayEquals() will proceed without errors. If the arrays are not equal, an exception will be thrown, and the test aborted. Any test code after the assertArrayEquals() will not be executed.

**assertEquals()**

The assertEquals() method compares two objects for equality, using their equals() method.

Here is a simple example:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testConcatenate() {

MyUnit myUnit = new MyUnit();

String result = myUnit.concatenate("one", "two");

assertEquals("onetwo", result);

}

}

First the myUnit.concatenate() method is called, and the result is stored in the variable result.

Second, the result value is compared to the expected value "onetwo", using the assertEquals() method.

If the two objects are equal according to their implementation of their equals() method, the assertEquals() method will return normally. Otherwise the assertEquals() method will throw an exception, and the test will stop there.

This example compared to String objects, but the assertEquals() method can compare any two objects to each other. The assertEquals() method also come in versions which compare primitive types like int and float to each other.

**assertTrue() + assertFalse()**

The assertTrue() and assertFalse() methods tests a single variable to see if its value is either true, or false. Here is a simple example:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testGetTheBoolean() {

MyUnit myUnit = new MyUnit();

assertTrue (myUnit.getTheBoolean());

assertFalse(myUnit.getTheBoolean());

}

}

As you can see, the method call to myUnit.getTheBollean() is inlined inside the assertTrue() assertFalse() calls.

If the getTheBoolean() method returns true, the assertTrue() method will return normally. Otherwise an exception will be thrown, and the test will stop there.

If the getTheBoolean() method returns false, the assertFalse() method will return normally. Otherwise an exception will be thrown, and the test will stop there.

Of course the above test will fail in either the assertTrue() or assertFalse() call, if the getTheBoolean() method returns the same value in both calls. One of the assertTrue() or assertFalse() calls will fail.

**assertNull() + assertNotNull()**

The assertNull() and assertNotNull() methods test a single variable to see if it is null or not null. Here is an example:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testGetTheObject() {

MyUnit myUnit = new MyUnit();

assertNull(myUnit.getTheObject());

assertNotNull(myUnit.getTheObject());

}

}

The call to myUnit.getTheObject() is inlined in the assertNull() and assertNotNull() calls.

If the myUnit.getTheObject() returns null, the assertNull() method will return normally. If a non-null value is returned, the assertNull() method will throw an exception, and the test will be aborted here.

The assertNotNull() method works oppositely of the assertNull() method, throwing an exception if a null value is passed to it, and returning normally if a non-null value is passed to it.

**assertSame() and assertNotSame()**

The assertSame() and assertNotSame() methods tests if two object references point to the same object or not. It is not enough that the two objects pointed to are equals according to their equals() methods. It must be exactly the same object pointed to. Here is a simple example:

import org.junit.Test;

import static org.junit.Assert.\*;

public class MyUnitTest {

@Test

public void testGetTheSameObject() {

MyUnit myUnit = new MyUnit();

assertSame (myUnit.getTheSameObject(),

myUnit.getTheSameObject());

assertNotSame(myUnit.getTheSameObject(),

myUnit.getTheSameObject());

}}

The calls to myUnit.getTheSameObject() are inlined into the assertSame() and assertNotSame() method calls.

If the two references points to the same object, the assertSame() method will return normally. Otherwise an exception will be thrown and the test will stop here.

The assertNotSame() method works oppositely of the assertSame() method. If the two objects do not poin to the same object, the assertNotSame() method will return normally. Othersise an exception is thrown and the test stops here.

**assertThat()**

The assertThat() method compares an object to an org.hamcrest.Matcher to see if the given object matches whatever the Matcher requires it to match.

**Hamcrest Matchers**

Matchers is an external addition to the JUnit framework. Matchers were added by the framework called Hamcrest. JUnit 4.8.2 ships with Hamcrest internally, so you don't have to download it, and add it yourself.

Matchers are used with the org.junit.Assert.assertThat() method, which looks like this:

public void assertThat(Object o, Matcher matcher){

...

}

You can implement your own Matcher's, but JUnit (Hamcrest) comes with a few builtin matchers you can use. Here are a few examples:

import static org.junit.Assert.\*;

import static org.hamcrest.CoreMatchers.\*;

import org.junit.Test;

public class MyMatcherTest {

@Test

public void testWithMatchers() {

assertThat("this string", is("this string"));

assertThat(123, is(123));

}

}

The assertThat() method takes an object and a Matcher implementation. It is the Matcher that determines if the test passes or fails. The assertThat() method just takes care of the "plumming" - meaning calling the Matcher with the given object.

JUnit (Hamcrest, really) comes with a collection of builtin matchers you can use. In the example above, the org.hamcrest.CoreMatchers.is() method is used to create a Matcher. The Matcher returned by is() returns true, if the two values compared are equal, and false if not.

The assertThat method throws an exception if the Matcher returns false. If the Matcher returns true, the assertThat() method returns normally. How the Matcher returns true or false, I will get back to later in this text. Right now just imagine that it can.

**Chaining Matchers**

You can chain some matchers, for instance like this:

@Test

public void testWithMatchers() {

assertThat(123, not( is(345) ) );

}

Notice the chained call not( is(345) ). This is really two matchers combined. The is() method returns one matcher, and the not() method returns another. The matcher returned by not() negates the matcher output of the matcher given as input. In this case, it is the output of the matcher returned by the is() method, that is negated.

**Core Matchers**

Before you start implementing your own Matcher's, you should look at the core matchers that come with JUnit already. Here is a list of the matcher methods:

|  |
| --- |
| **Core** |
|  | any() | Matches anything |
|  | is() | A matcher that checks if the given objects are equal. |
|  | describedAs() | Adds a descrption to a Matcher |
| **Logical** |  |  |
|  | allOf() | Takes an array of matchers, and all matchers must match the target object. |
|  | anyOf() | Takes an array of matchers, and at least one of the matchers must report that it matches the target object. |
|  | not() | Negates the output of the previous matcher. |
| **Object** |  |  |
|  | equalTo() | A matcher that checks if the given objects are equal. |
|  | instanceOf() | Checks if the given object is of type X or is compatible with type X |
|  | notNullValue() +  nullValue() | Tests whether the given object is null or not null. |
|  | sameInstance() | Tests if the given object is the exact same instance as another. |

Actually, all of the above are static methods which take different parameters, and return a Matcher.

You will have to play around with matchers a little, before you get the hang of them. They can be quite handy.

**Custom Matchers**

You can write your own matchers and plug into the assertThat() method. Here is an example:

public static Matcher matches(final Object expected){

return new BaseMatcher() {

protected Object theExpected = expected;

public boolean matches(Object o) {

return theExpected.equals(o);

}

public void describeTo(Description description) {

description.appendText(theExpected.toString());

}

};

}

The static method matches() creates a new matcher and returns it.

This matcher is an anonymous subclass of the class BaseMatcher. The JUnit documentation states that you should always extend BasreMatcher rather than implement the Matcher interface yourself. Thus, if new methods are added to Matcher in the future, BaseMatcher can implement them. Your subclasses will then automatically get those methods too. That will avoid breaking your code.

Here is how to use this custom matcher:

@Test

public void testThat() {

MyUnit myUnit = new MyUnit();

assertThat(myUnit.getTheSameObject(), **matches("constant string")**);

}

Simple, isn't it? You just embed the call to the static method matches() inside the assertThat() method.