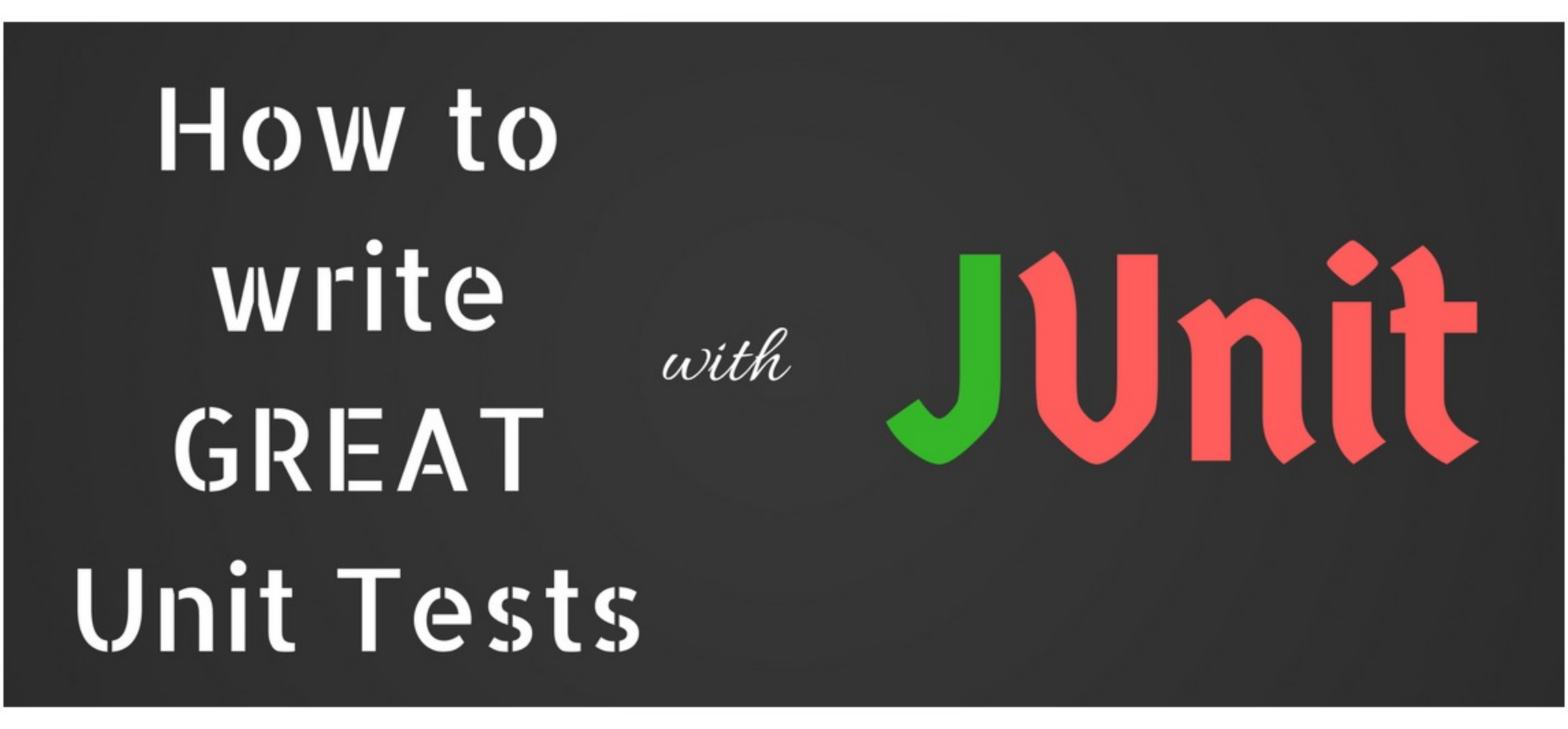
How to write great unit tests with JUnit (examples explaining 4 major features of JUnit 4)



JUnit 4 tutorial - We would look into the essentials of JUnit 4 so that we could write some quality unit tests that makes sense and are easy to maintain.

JUnit4 - Basic annotations

Following are the most commonly used annotations and their usage in a basic unit test written in JUnit 4.

- `@Test` - Marks the method as a test method.
- `@Before` and `@After` sandwiches **each test method** in the class.
- `@BeforeClass` and `@AfterClass` sandwiches **all of the test methods** in a JUnit test class.
- So when you run the JUnit test class below, the execution order is:
  1. Method annotated with `@BeforeClass`
  2. Method annotated with `@Before`
  3. First method annotated with `@Test` i.e. `test1()`.
  4. Method annotated with `@After`
  5. Method annotated with `@Before`
  6. Second method annotated with `@Test` i.e. `test2()`.
  7. Method annotated with `@After`
  8. Method annotated with `@AfterClass`

```
    public class SampleTest {

        @BeforeClass
        public static void setUpBeforeClass() throws Exception {


            //Method annotated with `@BeforeClass` will execute once before any of the test methods in this class.

            //This method could be used to set up any test fixtures that are computationally expensive and shared by several te
st methods. e.g. establishing database connections

            //Sometimes several tests need to share computationally expensive setup (like logging into a database). While this
 can compromise the independence of tests, sometimes it is a necessary optimization. From http://junit.sourceforge.net/javadoc/
org/junit/BeforeClass.html

        }

        @AfterClass
        public static void tearDownAfterClass() throws Exception {

            //Method annotated with `@AfterClass` will execute once after all of the test methods are executed in this class.

            //If you allocate expensive external resources in a BeforeClass method you need to release them after all the tests
 in the class have run. Annotating a public static void method with @AfterClass causes that method to be run after all the test
s in the class have been run. All @AfterClass methods are guaranteed to run even if a BeforeClass method throws an exception. F
rom http://junit.sourceforge.net/javadoc/org/junit/AfterClass.html
        }

        @Before
        public void setUp() throws Exception {
             //Method annotated with `@Before` will execute before each test method in this class is executed.

             //If you find that several tests need similar objects created before they can run this method could be used to do
 set up those objects (aka test-fixtures).
        }

        @After
        public void tearDown() throws Exception {

             //Method annotated with `@After` will execute after each test method in this class is executed.

             //If you allocate external resources in a Before method you must release them in this method.
        }

        @Test
        public void test1() {

           //A public void method annotated with @Test will be executed as a test case.
        }

        @Test
        public void test2() {

            //Another test cases
        }

    }
```

JUnit4 - Assertions

When it comes to assertions, there is the set of old JUnit assertions like:

- org.junit.Assert.assertArrayEquals
- org.junit.Assert.assertEquals
- org.junit.Assert.assertFalse
- org.junit.Assert.assertNotNull
- org.junit.Assert.assertNotSame
- org.junit.Assert.assertNull
- org.junit.Assert.assertSame
- org.junit.Assert.assertTrue

And the org.junit.Assert.assertThat method (available in JUnit4) which uses matchers and is better than old style assertions because it provides:

- Better readability
  - `assertThat(actual, is(equalTo(expected)));` is better than `assertEquals(expected, actual);`
  - `assertThat(actual, is(not(equalTo(expected))));` is better than `assertFalse(expected.equals(actual));`
- Better failiure messages
  - `java.lang.AssertionError: Expected: is "hello" but: was "hello world"` is better than

    `org.junit.ComparisonFailure: expected:<hello[]> but was:<hello[ world]>`
- Flexbility
  - Multiple conditions could be asserted using matchers like `anyOf` or `allOf`.

    eg: `assertThat("hello world", anyOf(is("hello world"), containsString("hello")));` In this case, the test will pass if either the actual string is "hello world" or if it contains the word "hello".

Following is a list of hamcrest coreMatchers from the hamcrest docs.

- allOf
- any
- anyOf
- anything
- both
- containsString
- describedAs
- either
- endsWith
- equalTo
- everyItem
- hasItems
- instanceOf
- is
- isA
- not
- notNullValue
- nullValue
- sameInstance
- startsWith
- theInstance

Example useage of a few of the above matchers

```
@Test
public void testAssetThatExamples() {

    // 'theString' should contain 'S' and 'r'
    assertThat("theString", both(containsString("S")).and(containsString("r")));

    List<String> items = Arrays.asList("John", "James", "Julia", "Jim");

    // items list should have James and Jim
    assertThat(items, hasItems("James", "Jim"));

    // Every item in the list should have the character 'J'
    assertThat(items, everyItem(containsString("J")));

    // check all of the matchers
    assertThat("Once", allOf(equalTo("Once"), startsWith("O")));

    // negation of all of the matchers
    assertThat("Once", not(allOf(equalTo("test"), containsString("test"))));
}
```

JUnit4 wiki for Assertions contains a list of examples for each of the assertions mentioned above. Also this is a comprehensive post on `assertThat`. I like the table at the end the most, which is a comparison of the `assertThat` with the old style assert methods, very useful.

JUnit4 - Exceptions testing

Does your method throw exceptions? There are a few different ways to verify whether expected exceptions are thrown, given the conditions. For example, we need a method which reads a file and it throws file not found exception with the message "The file 'file_name' does not exist!". We can test if the file not found exception is thrown in a number of ways. The first is the simplest and the most straight forward way which is preferred, but if we need to test the exception message as well, we could make use of the other two.

Following are the three different ways you can test that your method would throw the expected exception.

see bottom of this post to download these examples

1. Set the `expected` parameter `@Test(expected = FileNotFoundException.class)`.

```
@Test(expected = FileNotFoundException.class)
    public void testReadFile() throws IOException {
        FileReader reader = new FileReader("test.txt");
        reader.read();
        reader.close();
    }
```

2. Using `try` `catch`

```
@Test
    public void testReadFile2() {
        try {
            FileReader reader = new FileReader("test.txt");
            reader.read();
            reader.close();
            fail("Expected an IOException to be thrown");
        } catch (IOException e) {
            assertThat(e.getMessage(), is("test.txt (No such file or directory)"));
        }

    }
```

3. Testing with `ExpectedException` Rule.

```
 @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void testReadFile3() throws IOException {

        thrown.expect(IOException.class);
        thrown.expectMessage(startsWith("test.txt (No such file or directory)"));
        FileReader reader = new FileReader("test.txt");
        reader.read();
        reader.close();
    }
```

You could read more about exceptions testing in JUnit4 wiki for Exception testing and bad.robot - Expecting Exceptions JUnit Rule.

JUnit4 - Parameterized tests

Often times we need to test a single method with several different test data or inputs and `Parameterized` tests are very useful to maintain a very clean and readable tests in such cases.

e.g. In the following example, the `getTotalCharactersWithoutSpaces` method will count the number of characters ignoring any whitespace. We need to test this with different test input samples. Without Parameterized test, we would have to repeat the assertion for each of the test data which would make tests less readable and maintainable over time.

```
@RunWith(Parameterized.class)
public class GreetingTest {

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
                { "hello world", 10 }, { "helloworld", 10 }, { "hello", 5 }
                //The first item in the array is the input, and second is the expected outcome.
            });
    }

    private String input;
    private int expected;

    //This constructor must be provided for the parameterized tests to work.
    public GreetingTest(String input, int expected) {
        this.input = input;
        this.expected = expected;
    }

    @Test
    public void test() {

        Greeting greeting = new Greeting();
        assertThat(greeting.getTotalCharactersWithoutSpaces(input), is(expected));
    }

}
```

Again more elaborated examples could be found in the JUnit4 wiki for Parameterized tests

Have you tried **TDD**? check this out! Get started with Test Driven Development (A beginner's guide)

Please leave a comment and let me know if you liked it!

Please share your thoughts...

risolerh • 8 months ago
Interesante, voy a bajar el codigo para correrlo y probar y hacer mis propias pruebas.
Reply • Share ›

Dilini Rajapaksha → risolerh • 7 months ago
@risolerh Gracias por el comentario.
Reply • Share ›