ABOUT KONSTANTINA DIMTSA

Konstantina has graduated from the Department of Informatics and Telecommunications in National and Kapodistrian University of Athens (NKUA) and she is currently pursuing M.Sc studies in Advanced Information Systems at the same department. She is also working as a research associate for NKUA in the field of telecommunications. Her main interests lie in software engineering, web applications, databases and telecommunications.

JUnit Tutorial for Unit Testing – The ULTIMATE Guide (PDF Download)

Posted by: Konstantina Dimtsa in Core Java November 10th, 2014

EDITORIAL NOTE: *We have provided plenty of JUnit tutorials here at Java Code Geeks, like JUnit Getting Started Example, JUnit Using Assertions and Annotations Example, JUnit Annotations Example and so on.*

However, we prefered to gather all the JUnit features in one detailed guide for the convenience of the reader. We hope you like it!

Table Of Contents

1. Unit testing introduction

1.1. What is unit testing?

A unit can be a function, a class, a package, or a subsystem. So, the term unit testing refers to the practice of testing such small units of your code, so as to ensure that they work as expected. For example, we can test whether an output is what we expected to see given some inputs or if a condition is true or false.

This practice helps developers to discover failures in their logic behind their code and improve the quality of their code. Also, unit testing can be used so as to ensure that the code will work as expected in case of future changes.

1.2. Test coverage

In general, the development community has different opinion regarding the percentage of code that should be tested (test coverage). Some developers believe that the code should have 100% test coverage, while others are comprised with a test coverage of 50% or less. In any case, you should write tests for complex or critical parts of your code.

1.3. Unit testing in Java

The most popular testing framework in Java is JUnit. As this guide is focused to JUnit, more details for this testing framework will presented in the next sections. Another popular testing framework in Java is TestNG.

2. JUnit introduction

JUnit is an open source testing framework which is used to write and run repeatable automated tests, so that we can be ensured that our code works as expected. JUnit is widely used in industry and can be used as stand alone Java program (from the command line) or within an IDE such as Eclipse.

JUnit provides:

- Assertions for testing expected results.
- Test features for sharing common test data.
- Test suites for easily organizing and running tests.
- Graphical and textual test runners.

JUnit is used to test:

- an entire object
- part of an object – a method or some interacting methods
- interaction between several objects

2.1. JUnit Simple Example using Eclipse

In this section we will see a simple JUnit example. First we will present the class we would like to test:

Calculate.java

```
01 package com.javacodegeeks.junit;
02
03 public class Calculate {
04
05     public int sum(int var1, int var2) {
06         System.out.println("Adding values: " + var1 + " + " + var2);
07         return var1 + var2;
08     }
09
10 }
```

In the above source code, we can notice that the class has one public method named

```
sum()
```

, which gets as inputs two integers, adds them and returns the result. So, we will test this method. For this purpose, we will create another class including methods that will test each one of the methods of the previous class (in this case, we have only one method to be tested). This is the most common way of usage. Of course, if a method is very complex and extended, we can have more than one test methods for this complex method. The details of creating test cases will be presented in the next sections. Below, there is the code of the class named

```
CalculateTest.java
```

, which has the role of our test class:

CalculateTest.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04
05 import org.junit.Test;
06
07 public class CalculateTest {
08
09     Calculate calculation = new Calculate();
10     int sum = calculation.sum(2, 5);
11     int testSum = 7;
12
13     @Test
14     public void testSum() {
15         System.out.println("@Test sum(): " + sum + " = " + testSum);
16         assertEquals(sum, testSum);
17     }
18
19 }
```

Let's explain the above code. Firstly, we can see that there is a

```
@Test
```

annotation above the

```
testSum()
```

method. This annotation indicates that the public void method to which it is attached can be run as a test case. Hence, the

```
testSum()
```

method is the method that will test the

```
sum()
```

public method. We can also observe a method called

```
assertEquals(sum, testsum)
```

. The method

```
assertEquals ([String message], object expected, object actual)
```

takes as inputs two objects and asserts that the two objects are equal.

If we run the test class, by right-clicking in the test class and select *Run As -> Junit Test*, the program output will look like that:

```
1 Adding values: 2 + 5
2 @Test sum(): 7 = 7
```

To see the actual result of a JUnit test, Eclipse IDE provides a JUnit window which shows the results of the tests. In this case where the test succeeds, the JUnit window does not show any errors or failures, as we can see in the image below:

Figure 1: JUnit window after a successful test.

Now, if we change this line of code:

```
1 int testSum = 10;
```

so that the integers to be tested are not equal, the output will be:

```
1 Adding values: 2 + 5
2 @Test sum(): 7 = 10
```

And in the JUnit window, an error will appear and this message will be displayed:

```
1 java.lang.AssertionError: expected: but was:
2 at com.javacodegeeks.junit.CalculateTest.testSum(CalculateTest.java:16)
```

2.2. JUnit annotations

In this section we will mention the basic annotations supported in Junit 4. The table below presents a summary of those annotations:

Annotation	Description
`@Test` `public void method()`	The `Test` annotation indicates that the public void method to which it is attached can be run as a test case.
`@Before` `public void method()`	The `Before` annotation indicates that this method must be executed before each test in the class, so as to execute some preconditions necessary for the test.
`@BeforeClass` `public static void method()`	The `BeforeClass` annotation indicates that the static method to which is attached must be executed once and before all tests in the class. That happens when the test methods share computationally expensive setup (e.g. connect to database).
`@After` `public void method()`	The `After` annotation indicates that this method gets executed after execution of each test (e.g. reset some variables after execution of every test, delete temporary variables etc)
`@AfterClass` `public static void method()`	The `AfterClass`

	annotation can be used when a method needs to be executed after executing all the tests in a JUnit Test Case class so as to clean-up the expensive set-up (e.g disconnect from a database). Attention: The method attached with this annotation (similar to `BeforeClass`) must be defined as static.
`@Ignore` `public static void method()`	The `Ignore` annotation can be used when you want temporarily disable the execution of a specific test. Every method that is annotated with `@Ignore` won't be executed.

Let's see an example of a test class with some of the annotations mentioned above.

AnnotationsTest.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04 import java.util.*;
05 import org.junit.*;
06
07 public class AnnotationsTest {
08
09     private ArrayList testList;
10
11     @BeforeClass
12     public static void onceExecutedBeforeAll() {
13         System.out.println("@BeforeClass: onceExecutedBeforeAll");
14     }
15
16     @Before
17     public void executedBeforeEach() {
18         testList = new ArrayList();
19         System.out.println("@Before: executedBeforeEach");
20     }
21
22     @AfterClass
23     public static void onceExecutedAfterAll() {
24         System.out.println("@AfterClass: onceExecutedAfterAll");
25     }
26
27     @After
28     public void executedAfterEach() {
29         testList.clear();
30         System.out.println("@After: executedAfterEach");
31     }
32
33     @Test
34     public void EmptyCollection() {
35         assertTrue(testList.isEmpty());
36         System.out.println("@Test: EmptyArrayList");
37
38     }
39
40     @Test
41     public void OneItemCollection() {
42         testList.add("oneItem");
43         assertEquals(1, testList.size());
44         System.out.println("@Test: OneItemArrayList");
45     }
46
47     @Ignore
48     public void executionIgnored() {
49
50         System.out.println("@Ignore: This execution is ignored");
51     }
52 }
```

If we run the above test, the console output would be the following:

```
1 @BeforeClass: onceExecutedBeforeAll
2 @Before: executedBeforeEach
3 @Test: EmptyArrayList
4 @After: executedAfterEach
5 @Before: executedBeforeEach
6 @Test: OneItemArrayList
7 @After: executedAfterEach
8 @AfterClass: onceExecutedAfterAll
```

2.3. JUnit assertions

In this section we will present a number of assertion methods. All those methods are provided by the `Assert` class which extends the `class java.lang.Object`

and they are useful for writing tests so as to detect failures. In the table below there is a more detailed explanation of the most commonly used assertion methods.

Assertion	Description
`void assertEquals([String message], expected value, actual value)`	Asserts that two values are equal. Values might be type of int, short, long, byte, char or java.lang.Object. The first argument is an optional String message.
`void assertTrue([String message], boolean condition)`	Asserts that a condition is true.
`void assertFalse([String message],boolean condition)`	Asserts that a condition is false.
`void assertNotNull([String message], java.lang.Object object)`	Asserts that an object is not null.
`void assertNull([String message], java.lang.Object object)`	Asserts that an object is null.
`void assertSame([String message], java.lang.Object expected, java.lang.Object actual)`	Asserts that the two objects refer to the same object.
`void assertNotSame([String message], java.lang.Object unexpected, java.lang.Object actual)`	Asserts that the two objects do not refer to the same object.
`void assertArrayEquals([String message], expectedArray, resultArray)`	Asserts that the array expected and the resulted array are equal. The type of Array might be int, long, short, char, byte or java.lang.Object.

Let's see an example of some of the aforementioned assertions.

AssertionsTest.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04 import org.junit.Test;
05
06 public class AssertionsTest {
07
08     @Test
09     public void test() {
10         String obj1 = "junit";
11         String obj2 = "junit";
12         String obj3 = "test";
13         String obj4 = "test";
14         String obj5 = null;
15         int var1 = 1;
16         int var2 = 2;
17         int[] arithmetic1 = { 1, 2, 3 };
18         int[] arithmetic2 = { 1, 2, 3 };
19
20         assertEquals(obj1, obj2);
21
22         assertSame(obj3, obj4);
23
24         assertNotSame(obj2, obj4);
25
26         assertNotNull(obj1);
27
28         assertNull(obj5);
29
30         assertTrue(var1  var2);
31
32         assertArrayEquals(arithmetic1, arithmetic2);
33     }
34
35 }
```

In the class above we can see how these assert methods work.

- The

  `assertEquals()`

  method will return normally if the two compared objects are equal, otherwise a failure will be displayed in the JUnit window and the test will abort.

- The
  `assertSame()`
  and
  `assertNotSame()`
  methods tests if two object references point to exactly the same object.
- The
  `assertNull()`
  and
  `assertNotNull()`
  methods test whether a variable is null or not null.
- The
  `assertTrue()`
  and
  `assertFalse()`
  methods tests if a condition or a variable is true or false.
- The
  `assertArrayEquals()`
  will compare the two arrays and if they are equal, the method will proceed without errors. Otherwise, a failure will be displayed in the JUnit window and the test will abort.

3. JUnit complete example using Eclipse

In this section we will show a complete example of using JUnit. We will see in detail how to create and run tests and we will show how to use specific annotations and assertions of JUnit.

3.1. Initial Steps

Let's create a java project named *JUnitGuide*. In the *src* folder, we right-click and select *New -> Package*, so as to create a new package named

```
com.javacodegeeks.junit
```

where we will locate the class to be tested. For the test classes, it is considered as good practice to create a new source folder dedicated to tests, so that the classes to be tested and the test classes will be in different source folders. For this purpose, right-click your project, select *New -> Source Folder*, name the new source folder test and click *Finish*.

Figure 2: Create a new source folder named *test*.

Tip
Alternatively, you can create a new source folder by right-clicking your project and select *Properties -> Java Build Path*, select the tab *Source*, select *Add Folder -> Create New Folder*, write the name *test* and press *Finish*.
You can easily see that there are two source folders in your project:

Figure 3

You can also create a new package in the newly created test folder, which will be called

```
com.javacodegeeks.junit
```

, so that your test classes won't be located to the default package and we are ready to start!

3.2. Create the java class to be tested

Right-click the *src* folder and create a new java class called

```
FirstDayAtSchool.java
```

. This will be the class whose public methods will be tested.

FirstDayAtSchool.java

```
01 package com.javacodegeeks.junit;
02
03 import java.util.Arrays;
04
05 public class FirstDayAtSchool {
06
07     public String[] prepareMyBag() {
08         String[] schoolbag = { "Books", "Notebooks", "Pens" };
09         System.out.println("My school bag contains: "
10                 + Arrays.toString(schoolbag));
11         return schoolbag;
12     }
13
14     public String[] addPencils() {
15         String[] schoolbag = { "Books", "Notebooks", "Pens", "Pencils" };
16         System.out.println("Now my school bag contains: "
17                 + Arrays.toString(schoolbag));
18         return schoolbag;
19     }
20 }
```

3.3. Create and run a JUnit test case

To create a JUnit test case for the existing class

```
FirstDayAtSchool.java
```

, right-click on it in the Package Explorer view and select *New → JUnit Test Case*. Change the source folder so that the class will be located to *test* source folder and ensure that the flag *New JUnit4 test* is selected.

Figure 4: Create a new test class.

Then, click *Finish*. If your project does not contain the JUnit library in its classpath, the following message will be displayed so as to add the JUnit library to the classpath:

Figure 5: Add JUnit4 library to your project's build path.

Below, there is the code of the class named

```
FirstDayAtSchoolTest.java
```

, which is our test class:

FirstDayAtSchool.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04
05 import org.junit.Test;
06
07 public class FirstDayAtSchoolTest {
08
09     FirstDayAtSchool school = new FirstDayAtSchool();
10     String[] bag1 = { "Books", "Notebooks", "Pens" };
11     String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils" };
12
13     @Test
14     public void testPrepareMyBag() {
15         System.out.println("Inside testPrepareMyBag()");
16         assertArrayEquals(bag1, school.prepareMyBag());
17     }
18
19     @Test
20     public void testAddPencils() {
21         System.out.println("Inside testAddPencils()");
22         assertArrayEquals(bag2, school.addPencils());
23     }
24
25 }
```

Now we can run the test case by right-clicking on the test class and select *Run As -> JUnit Test*.

The program output will look like that:

```
1 Inside testPrepareMyBag()
2 My school bag contains: [Books, Notebooks, Pens]
3 Inside testAddPencils()
4 Now my school bag contains: [Books, Notebooks, Pens, Pencils]
```

and in the JUnit view will be no failures or erros. If we change one of the arrays, so that it contains more than the expected elements:

```
1 String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils", "Rulers"};
```

and we run again the test class, the JUnit view will contain a failure:

Figure 6: Test failure

Else, if we change again one of the arrays, so that it contains a different element than the expected:

```
1 String[] bag1 = { "Books", "Notebooks", "Rulers" };
```

and we run again the test class, the JUnit view will contain once again a failure:

Figure 7: Test failure

3.4. Using

```
@Ignore
```

annotation

Let's see in the above example how can we use the

```
@Ignore
```

annotation. In the test class

```
FirstDayAtSchoolTest
```

we will add the

```
@Ignore
```

annotation to the

```
testAddPencils()
```

method. In that way, we expect that this testing method will be ignored and won't be executed.

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04
05 import org.junit.Ignore;
06 import org.junit.Test;
07
08 public class FirstDayAtSchoolTest {
09
10     FirstDayAtSchool school = new FirstDayAtSchool();
11     String[] bag1 = { "Books", "Notebooks", "Pens" };
12     String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils" };
13
14     @Test
15     public void testPrepareMyBag() {
16         System.out.println("Inside testPrepareMyBag()");
17         assertArrayEquals(bag1, school.prepareMyBag());
18     }
19
20     @Ignore
21     @Test
22     public void testAddPencils() {
23         System.out.println("Inside testAddPencils()");
24         assertArrayEquals(bag2, school.addPencils());
25     }
26
27 }
```

Indeed, this is what happens according to the output:

```
Inside testPrepareMyBag()
My school bag contains: [Books, Notebooks, Pens]
```

Now, we will remove the

```
@Ignore
```

annotation from the

```
testAddPencils()
```

method and we will annotate the whole class instead.

```
package com.javacodegeeks.junit;

import static org.junit.Assert.*;

import org.junit.Ignore;
import org.junit.Test;

@Ignore
public class FirstDayAtSchoolTest {

    FirstDayAtSchool school = new FirstDayAtSchool();
    String[] bag1 = { "Books", "Notebooks", "Pens" };
    String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils" };

    @Test
    public void testPrepareMyBag() {
        System.out.println("Inside testPrepareMyBag()");
        assertArrayEquals(bag1, school.prepareMyBag());
    }


    @Test
    public void testAddPencils() {
        System.out.println("Inside testAddPencils()");
        assertArrayEquals(bag2, school.addPencils());
    }

}
```

The whose test class won't be executed, so no result will be displayed int the console output and in the junit view:

Figure 8

3.5. Creating suite tests

In this section, we will see how to create suite tests. A test suite is a collection of some test cases from different classes that can be run all together using

```
@RunWith
```

and

```
@Suite
```

annotations. This is very helpful if you have many test classes and you want to run them all together instead of running each test one at a time.

When a class is annotated with

```
@RunWith
```

, JUnit will invoke the class in which is annotated so as to run the tests, instead of using the runner built into JUnit.

Based on the classes of the previous sections, we can create two test classes. The one class will test the public method

```
prepareMyBag()
```

and the other test class will test the method

```
addPencils()
```

. Hence, we will eventually have the classes below:

PrepareMyBagTest.java

```
01 package com.javacodegeeks.junit;
02
03 import org.junit.Test;
04 import static org.junit.Assert.*;
05
06 public class PrepareMyBagTest {
07
08     FirstDayAtSchool school = new FirstDayAtSchool();
09
10     String[] bag = { "Books", "Notebooks", "Pens" };
11
12     @Test
13     public void testPrepareMyBag() {
14
15         System.out.println("Inside testPrepareMyBag()");
16         assertArrayEquals(bag, school.prepareMyBag());
17
18     }
19
20 }
```

AddPencilsTest.java

```
01 package com.javacodegeeks.junit;
02
03 import org.junit.Test;
04 import static org.junit.Assert.*;
05
06 public class AddPencilsTest {
07
08     FirstDayAtSchool school = new FirstDayAtSchool();
09
10     String[] bag = { "Books", "Notebooks", "Pens", "Pencils" };
11
12     @Test
13     public void testAddPencils() {
14
15         System.out.println("Inside testAddPencils()");
16         assertArrayEquals(bag, school.addPencils());
17
18     }
19
20 }
```

Now we will create a test suite so as to run the above classes together. Right-click the *test* source folder and create a new java class named

```
SuiteTest.java
```

with the following code:

SuiteTest.java

```
01 package com.javacodegeeks.junit;
02
03 import org.junit.runner.RunWith;
04 import org.junit.runners.Suite;
05
06 @RunWith(Suite.class)
07 @Suite.SuiteClasses({ PrepareMyBagTest.class, AddPencilsTest.class })
08 public class SuitTest {
09
10 }
```

With the

```
@Suite.SuiteClasses
```

annotation you can define which test classes will be included in the execution.

So, if you right-click the test suite and select *Run As -> JUnit Test*, the execution of both test classes will take place with the order that has been defined in the

```
@Suite.SuiteClasses
```

annotation.

3.6. Creating parameterized tests

In this section we will see how to create parameterized tests. For this purpose, we will use the class mentioned in section 2.1 which provides a public method for adding integers. So, this will be the class to be tested.

But when a test class can be considered as a parameterized test class? Of course, when it fullfills all the following requirements:

- The class is annotated with

  `@RunWith(Parameterized.class)`

  .
  As explained in the previous section,

  `@RunWith`

  annotation enables JUnit to invoke the class in which is annotated to run the tests, instead of using the runner built into JUnit.

  `Parameterized`

  is a runner inside JUnit that will run the same test case with different set of inputs.
- The class has a single constructor that stores the test data.
- The class has a static method that generates and returns test data and is annotated with the

  `@Parameters`

  annotation.
- The class has a test, which obviously means that it needs a method annotated with the

  `@Test`

  annotation.

Now, we will create a new test class named

`CalculateTest.java`

, which will follow the guidelines mentioned above. The source code of this class follows.

CalculateTest.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.assertEquals;
04 import java.util.Arrays;
05 import java.util.Collection;
06
07 import org.junit.Test;
08 import org.junit.runner.RunWith;
09 import org.junit.runners.Parameterized;
10 import org.junit.runners.Parameterized.Parameters;
11
12 @RunWith(Parameterized.class)
13 public class CalculateTest {
14
15     private int expected;
16     private int first;
17     private int second;
18
19     public CalculateTest(int expectedResult, int firstNumber, int secondNumber) {
20         this.expected = expectedResult;
21         this.first = firstNumber;
22         this.second = secondNumber;
23     }
24
25     @Parameters
26     public static Collection addedNumbers() {
27         return Arrays.asList(new Integer[][] { { 3, 1, 2 }, { 5, 2, 3 },
28                 { 7, 3, 4 }, { 9, 4, 5 }, });
29     }
30
31     @Test
32     public void sum() {
33         Calculate add = new Calculate();
34         System.out.println("Addition with parameters : " + first + " and "
35                 + second);
36         assertEquals(expected, add.sum(first, second));
37     }
38 }
```

As we can observe in the class above, it fullfills all the above requirements. The method

`addedNumbers`

annotated with

`@Parameters`

returns a Collection of Arrays. Each array includes the inputs/output numbers of each test execution. The number of elements in each array must be the same with the number of parameters in the constructor. So, in this specific case, each array includes three elements, two elements that represent the numbers to be added and one element for the result.

If we run the

```
CalculateTest
```

test case, the console output will be the following:

```
Addition with parameters : 1 and 2
Adding values: 1 + 2
Addition with parameters : 2 and 3
Adding values: 2 + 3
Addition with parameters : 3 and 4
Adding values: 3 + 4
Addition with parameters : 4 and 5
Adding values: 4 + 5
```

As we see in the output, the test case is executed four times, which is the number of inputs in the method annotated with

```
@Parameters
```

annotation.

3.7. Rules

In this section we present a new feature of JUnit called *Rules* which allows very flexible addition or redefinition of the behavior of each test method in a test class. For this purpose,

```
@Rule
```

annotation should be used so as to mark public fields of a test class. Those fields should be of type

```
MethodRule
```

, which is an alteration in how a test method is run and reported. Multiple

```
MethodRules
```

can be applied to a test method.

```
MethodRule
```

interface has a lot of implementations, such as

```
ErrorCollector
```

which allows execution of a test to continue after the first problem is found,

```
ExpectedException
```

which allows in-test specification of expected exception types and messages,

```
TestName
```

which makes the current test name available inside test methods, and many others. Except for those already defined rules, developers can create their own custom rules and use them in their test cases as they wish.

Below we present the way we can use one of the existing rules named

```
TestName
```

in our own tests.

```
TestName
```

is invoked when a test is about to start.

NameRuleTest.java

```
01 package com.javacodegeeks.junit;
02
03 import static org.junit.Assert.*;
04
05 import org.junit.*;
06 import org.junit.rules.TestName;
07
08 public class NameRuleTest {
09     @Rule
10     public TestName name = new TestName();
11
12     @Test
13     public void testA() {
14         System.out.println(name.getMethodName());
15         assertEquals("testA", name.getMethodName());
16
17     }
18
19     @Test
20     public void testB() {
21         System.out.println(name.getMethodName());
22         assertEquals("testB", name.getMethodName());
23     }
```

```
}
```

We can see that the

```
@Rule
```

annotation marks the public field

```
name
```

which is of type

```
MethodRule
```

and specifically,

```
TestName
```

type. Then, we can use in our tests this

```
name
```

field and find for example the name of the test method, in this specific case.

3.8. Categories

Another new feature of JUnit is called *Categories* and allows you to group certain kinds of tests together and even include or exclude groups (categories). For example, you can separate slow tests from fast tests. To assign a test case or a method to one of those categories the

```
@Category
```

annotation is provided. Below there is an example of how we can use this nice feature of JUnit, based on the release notes of JUnit 4.8.

```
public interface FastTests { /* category marker */
}
```

```
public interface SlowTests { /* category marker */
}
```

Firstly, we define two categories, FastTests and SlowTests. A category can be either a class or an interface.

```
public class A {
  @Test
  public void a() {
    fail();
  }

  @Category(SlowTests.class)
  @Test
  public void b() {
  }
}
```

In the above code, we mark the test method

```
b()
```

of class

```
A
```

with

```
@Category
```

annotation so as to indicate that this specific method belongs to category

```
SlowTests
```

. So, we are able to mark not only whole classes but also some of their test methods individually.

```
@Category({ SlowTests.class, FastTests.class })
public class B {
  @Test
  public void c() {
  }
}
```

In the above sample of code, we can see that the whole class

```
B
```

is annotated with

```
@Category
```

annotation . Annotating a test class with

```
@Category
```

annotation automatically includes all its test methods in this category. We can also see that a test class or a test method can belong to more than one categories.

```
@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@SuiteClasses({ A.class, B.class })
// Note that Categories is a kind of Suite
public class SlowTestSuite {
  // Will run A.b and B.c, but not A.a
}
```

In this sample of code, we notice that there is a suite test named

```
SlowTestSuite
```

. Basically, categories are a kind of suite. In this suite, we observe a new annotation called

```
@IncludeCategory
```

, indicating which categories will be included in the execution. In this specific case, methods belonging to SlowTests category will be executed. Hence, only the test method

```
b()
```

of class

```
A
```

will be executed as well as the test method

```
c()
```

of class

```
B
```

, which both belong to SlowTests category.

```
@RunWith(Categories.class)
@IncludeCategory(SlowTests.class)
@ExcludeCategory(FastTests.class)
@SuiteClasses({ A.class, B.class })
// Note that Categories is a kind of Suite
public class SlowTestSuite {
  // Will run A.b, but not A.a or B.c
}
```

Finally, we change a little bit the test suite and we add one more new annotation called

```
@ExcludeCategory
```

, indicating which categories will be excluded from the execution. In this specific case, only the test method

```
b()
```

of class

```
A
```

will be executed, as this is the only test method that belongs explicitly to SlowTests category.

We notice that in both cases, the test method

```
a()
```

of class

```
A
```

won't be executed as it doesn't belong to any category.

4. Run JUnit tests from command line

You can run your JUnit test outside Eclipse, by using the

```
org.junit.runner.JUnitCore
```

class. This class provides the

```
runClasses()
```

method which allows you to execute one or several test classes. The return type of

```
runClasses()
```

method is an object of the type

```
org.junit.runner.Result
```

. This object can be used to collect information about the tests. Also, in case there is a failed test, you can use the object

```
org.junit.runner.notification.Failure
```

which holds description of the failed tests.

The procedure below shows how to run your test outside Eclipse.

Create a new Java class named

```
JunitRunner.java
```

with the following code:

JunitRunner.java

```
01 package com.javacodegeeks.junit;
02
03 import org.junit.runner.JUnitCore;
04 import org.junit.runner.Result;
05 import org.junit.runner.notification.Failure;
06
07 public class JunitRunner {
08
09     public static void main(String[] args) {
10
11         Result result = JUnitCore.runClasses(AssertionsTest.class);
12         for (Failure fail : result.getFailures()) {
13             System.out.println(fail.toString());
14         }
15         if (result.wasSuccessful()) {
16             System.out.println("All tests finished successfully...");
17         }
18     }
19 }
```

As an example, we choose to run the

```
AssertionsTest
```

test class.

- Open command prompt and move down directories so as to find the directory where the two classes are located.
- Compile the Test class and the Runner class.

```
1 C:\Users\konstantina\eclipse_luna_workspace\JUnitGuide\test\com\javacodegeeks\junit>javac -
  classpath "C:\Users\konstantina\Downloads\junit-
  4.11.jar";"C:\Users\konstantina\Downloads\hamcrest-core-1.3.jar"; AssertionsTest.java
  JunitRunner.java
```

As we did in Eclipse, we should also include library jars of JUnit to our classpath.

- Now run the

  ```
  JunitRunner
  ```

  .

```
1 C:\Users\konstantina\eclipse_luna_workspace\JUnitGuide\test\com\javacodegeeks\junit>java -
  classpath "C:\Users\konstantina\Downloads\junit-
  4.11.jar";"C:\Users\konstantina\Downloads\hamcrest-core-1.3.jar"; JunitRunner
```

Here is the output:

```
1 All tests finished successfully...
```

5. Conclusions

This was a detailed guide about JUnit testing framework, the most popular testing framework in Java.

If you enjoyed this, then subscribe to our newsletter to enjoy weekly updates and complimentary whitepapers! Also, check out JCG Academy for more advanced training!

Download
You can download the full source code of this guide here : JUnitGuide.zip

Tagged with: JUNIT TESTING ULTIMATE