Get Acquainted with the New Advanced Features of JUnit 4

Learn how to migrate from JUnit 3.8 to JUnit 4. Discover version 4's new features, including extensive use of annotations, and find out the status on IDE integration.

by Antonio Goncalves

Does anybody need an introduction on JUnit? No? OK, so I'll assume that you know this Java unit testing framework created by Kent Beck and Erich Gamma and skip the introduction. Instead I will focus on the migration process from JUnit 3.8 to the latest version, JUnit 4, and its integration in IDEs and Ant.

JUnit 4 is a completely different API from the versions that came before it and depends on new features of Java 5.0 (annotations, static import…). As you'll see, JUnit 4 is simpler, richer, and easier to use and introduces more flexible initialization and cleanup, timeouts, and parameterized test cases.

Nothing beats a bit of code for clarification. I'll use an example that I can use to illustrate different test cases throughout the article: a calculator. The sample calculator is very simple, inefficient, and even has a few bugs; it only manipulates integers and stores the result in a static variable. Substract method does not return a valid result, multiply is not implemented yet, and it looks like there is a bug on the squareRoot method: It loops infinitely. These bugs will help illustrate the efficiency of the tests in JUnit 4. You can switch this calculator on and off and you can clear the result. Here is the code:

```
package calc;

public class Calculator {

    private static int result;          // Static variable where the result is stored

    public void add(int n) {
        result = result + n;
    }
```

```
    public void substract(int n) {
        result = result - 1;          //Bug : should be result = result - n
    }

    public void multiply(int n) {}     //Not implemented yet

    public void divide(int n) {
        result = result / n;
    }

    public void square(int n) {
        result = n * n;
    }

    public void squareRoot(int n) {
        for (; ;) ;                    //Bug : loops indefinitely
    }

    public void clear() {               // Cleans the result
        result = 0;
    }

    public void switchOn() {          // Swith on the screen, display "hello", beep
        result = 0;                    // and do other things that calculator do nowadays
    }

    public void switchOff() { }      // Display "bye bye", beep, switch off the screen

    public int getResult() {
        return result;
    }
}
```

Migrating a Test Class

Now I'll take a simple test class already written in JUnit 3.8 and migrate it to JUnit 4. This class has some flaws: It does not test all the business methods and it looks like there is a bug in the testDivide method (8/2 is not equal to 5). Because the implementation of multiply is not ready, its test is written but ignored.

The differences between the two frameworks are highlighted in bold (see Table 1).

Table 1. CaculatorTest in JUnit 3.8 and JUnit 4

JUnit 3.8	JUnit 4

```
package junit3;

import calc.Calculator;
import junit.framework.TestCase;



public class CalculatorTest extends TestCase {

  private static Calculator calculator =
            new Calculator();

  @Override
  protected void setUp() {
      calculator.clear();
  }


  public void testAdd() {
    calculator.add(1);
    calculator.add(1);
    assertEquals(calculator.getResult(), 2);
  }


  public void testSubtract() {
    calculator.add(10);
    calculator.subtract(2);
    assertEquals(calculator.getResult(), 8);
  }


  public void testDivide() {
    calculator.add(8);
    calculator.divide(2);
     assert  calculator.getResult()  ==  5;
  }

  public void testDivideByZero()  {
    try {
      calculator.divide(0);
       fail();
     } catch (ArithmeticException e) {
    }
```

```
package junit4;

import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import  static  org.junit.Assert.*;

public class CalculatorTest {

  private static Calculator calculator =
            new Calculator();

  @Before
  public void clearCalculator() {
    calculator.clear();
  }

  @Test
  public void add() {
    calculator.add(1);
    calculator.add(1);
    assertEquals(calculator.getResult(), 2);
  }

  @Test
  public void subtract() {
    calculator.add(10);
    calculator.subtract(2);
    assertEquals(calculator.getResult(), 8);
  }

  @Test
  public void divide() {
    calculator.add(8);
    calculator.divide(2);
    assert calculator.getResult() == 5;
  }

  @Test(expected  =  ArithmeticException.class)
  public void divideByZero() {
    calculator.divide(0);
  }
```

```
  }

  public void notReadyYetTestMultiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
    }
}
```

```
 @Ignore("not ready yet")
 @Test
 public void multiply() {
   calculator.add(10);
   calculator.multiply(10);
   assertEquals(calculator.getResult(), 100);
 }
}
```

Packages
First of all, you can see that JUnit 4 uses org.junit.* package while JUnit 3.8 uses junit.framework.*. Of course, for backward compatibility, the JUnit 4 jar file ships with both packages.

Inheritance
Test classes do not have to extend junit.framework.TestCase anymore. In fact, they don't have to extend anything. JUnit 4 has decided to use annotations instead. To be executed as a test case, a JUnit 4 class needs at least one @Test annotation. For example, if you write a class with only @Before and @After annotations without at least one @Test method, you will get an error when trying to execute it: java.lang.Exception: No runnable methods.

Assert Methods
Because in JUnit 4 a test class doesn't inherit from TestCase (where assertEquals() methods are defined in JUnit 3.8), you have to use the prefixed syntax (e.g. Assert.assertEquals()) or, thanks to JDK 5.0, import statically the Assert class. In doing so, you can then use assertEquals methods exactly the way you have used them previously (e.g. assertEquals ()).

There are two new assertion methods in JUnit 4. They are used to compare arrays of objects. Both arrays are equal if each element they contain is equal.

```
public static void assertEquals(String message, Object[] expecteds, Object[] actuals);
public static void assertEquals(Object[] expecteds, Object[] actuals);
```

Twelve assertEquals methods have totally disappeared thanks to the autoboxing of the JDK 5.0. Methods such as assertEquals(long, long) in JUnit 3.8 use the assertEquals(Object, Object) in JUnit 4. It is the same for assertEquals(byte, byte), assertEquals(int, int), and so on. This will help prevent the antipattern "Using the Wrong Assert" (see http://www-128.ibm.com/developerworks/opensource/library/os-junit/ and http://www.exubero.com/junit/antipatterns.html).

A novelty with JUnit 4 is the neat integration of the assert keyword (divide() method in our example). You can use it as you would use the assertEquals methods, because they both throw the same exception (java.lang.AssertionError). JUnit 3.8 assertEquals would throw a junit.framework.AssertionFailedError. Note that when using assert, you must specify the –ea parameter to Java; if not, asserts are ignored (see "Programming with Asserts").

Fixture
Fixtures are methods to initialize and release any common objects during tests. In JUnit 3.8 you would use setUp() for initialization before running each test and then tearDown() for cleaning purposes after each test had completed. Both methods are overridden from the

TestCase class and therefore are uniquely defined. Note that I'm using the Java 5.0 built-in @Override annotation for the setup method—this annotation indicates that the method declaration is intended to override the method declaration in a superclass. Instead, JUnit 4 uses @Before and @After annotations. These methods can be called by any name (clearCalculator() in our example). I'll explain more about these annotations later in the article.

Tests
JUnit 3.8 recognizes a test method by analyzing its signature: The method name has to be prefixed with 'test', it must return void, and it must have no parameters (e.g. public void testDivide()). A test method that doesn't follow this naming convention is simply ignored by the framework and no exception is thrown, indicating a mistake has been made.

JUnit 4 doesn't use the same conventions. A test method does not have to be prefixed with 'test' but instead uses the @Test annotation. As in the previous framework, a test method must return void and have no parameters. With JUnit 4 this is controlled at runtime and throws an exception if not respected:

```
java.lang.Exception: Method xxx should have no parameters
java.lang.Exception: Method xxx should be void
```

The @Test annotation supports the optional expected parameters. It declares that a test method should throw an exception. If it doesn't or if it throws a different exception than the one declared, the test fails. In our example, dividing an integer by zero should raise an ArithmeticException.

Ignoring a Test
Remember that the multiply method is not implemented. However, you don't want the test to fail, you just want it ignored. How do you temporarily disable a test with JUnit 3.8? By commenting it or changing the naming conventions so that the runner doesn't find it. In my example I used the method name notReadyYetTestMultiply(). It doesn't start with 'test' so it won't be recognized. The problem is that in the middle of thousands of other tests, you might not remember to rename this method.

To ignore a test in JUnit 4 you can comment a method or delete the @Test annotation (you can't change the naming conventions anymore or an exception would be thrown). However, the problem will remain: The runner will not report such a test. You can now add the @Ignore annotation in front or after @Test. Test runners will report the number of ignored tests, along with the number of tests that ran and the number of tests that failed. Note that @Ignore takes an optional parameter (a String) if you want to record why a test is being ignored.

Running the Tests
In JUnit 3.8 you could choose from several runners: text, AWT, or Swing. JUnit 4 only uses text runners. Remember the tagline underneath the JUnit logo? "Keep the bar green to keep the code clean." Well, JUnit 4 will not display any green bar to inform you that your tests have succeeded. If you want to see any kind of green you'll need to use JUnit extensions or an IDE that integrates JUnit such as IDEA or Eclipse"

First, I want to run the JUnit 3.8 test class with the good old junit.textui.TestRunner (with the –ea parameter to take into account the assert keyword).

```
java -ea junit.textui.TestRunner junit3.CalculatorTest

..F.E.
```

```
There was 1 error:
1) testDivide(junit3.CalculatorTest)java.lang.AssertionError
        at junit3.CalculatorTest.testDivide(CalculatorTest.java:33)

There was 1 failure:
1) testSubtract(junit3.CalculatorTest)junit.framework.AssertionFailedError: expected:<9> but was:<8>
        at junit3.CalculatorTest.testSubtract(CalculatorTest.java:27)

FAILURES!!!
Tests run: 4,  Failures: 1,  Errors: 1
```

TestDivide produces an error because assert ensures that 8/2 does not equal 5. TestSubstract produces a failure because 10-2 should be equal to 8 but there is a bug in the implementation and it returns 9.

Now I'll run both classes with the new org.junit.runner.JUnitCore runner, which acts like a facade for running tests. It can execute JUnit 4 and JUnit 3.8 tests as well as a mixture of both.

```
java –ea org.junit.runner.JUnitCore junit3.CalculatorTest
JUnit version 4.1

..E.E.

There were 2 failures:
1) testSubtract(junit3.CalculatorTest)
junit.framework.AssertionFailedError: expected:<9> but was:<8>
        at junit.framework.Assert.fail(Assert.java:47)
2) testDivide(junit3.CalculatorTest)
java.lang.AssertionError
        at junit3.CalculatorTest.testDivide(CalculatorTest.java:33)

FAILURES!!!
Tests run: 4,  Failures: 2

***


java –ea org.junit.runner.JUnitCore junit4.CalculatorTest
JUnit version 4.1

...E.EI

There were 2 failures:
1) subtract(junit4.CalculatorTest)
java.lang.AssertionError: expected:<9> but was:<8>
        at org.junit.Assert.fail(Assert.java:69)
```

```
2) divide(junit4.CalculatorTest)
java.lang.AssertionError
        at junit4.CalculatorTest.divide(CalculatorTest.java:40)

FAILURES!!!
Tests run: 4,  Failures: 2
```

The first visible difference is that the JUnit version number is displayed in the console (4.1). The second is that JUnit 3.8 differentiates failures and errors. JUnit 4 makes it simpler by only using failures. A novelty is the letter "I", which indicates that a test has been ignored.

Advanced Tests
Now I'll demonstrate some advanced features of JUnit 4. Listing 1 is a new test class, AdvancedTest, that extends AbstractParent.

Advanced Fixture
Both classes use the new annotations @BeforeClass and @AfterClass as well as @Before and @After. The main differences between these annotations are shown in Table 2.

Table 2. @BeforeClass/@AfterClass vs. @Before/@After

@BeforeClass and @AfterClass	@Before and @After
Only one method per class can be annotated.	Multiple methods can be annotated. Order of execution is unspecified. Overridden methods are not run.
Method names are irrelevant	Method names are irrelevant
Runs once per class	Runs before/after each test method
@BeforeClass methods of superclasses will be run before those of the current class. @AfterClass methods declared in superclasses will be run after those of the current class.	@Before in superclasses are run before those in subclasses. @After in superclasses are run after those in subclasses.
Must be public and static.	Must be public and non static.
All @AfterClass methods are guaranteed to run even if a @BeforeClass method throws an exception.	All @After methods are guaranteed to run even if a @Before or @Test method throws an exception.

@BeforeClass and @AfterClass can be very useful if you need to allocate and release expensive resources only once. In our example the AbstractParent starts and stops the entire test system using these annotations on startTestSystem() and stopTestSystem() methods. And it initializes and cleans the system using @Before and @After. The child class AdvancedTest also uses a mixture of these annotations.

It is not good practice to have System.out.println in your test code, but in this case it helps to understand the order these annotations are called. When I run AdvancedTest I get:

```
Start test system               //@BeforeClass of parent
  Switch on calculator          //@BeforeClass of child
```

```
    Initialize test system       //First test
    Clear calculator

    Initialize test system       //Second test
    Clear calculator
    Clean test system

    Initialize test system       //Third test
    Clear calculator
    Clean test system

    Initialize test system       //Forth test
    Clear calculator
    Clean test system

  Switch off calculator          //@AfterClass of child
Stop test system                 //@AfterClass of parent
```

As you can see @BeforeClass and @AfterClass are only called once, meanwhile @Before and @After are called for each test.

Timeout Tests
In the previous example I wrote a test case for the squareRoot() method. Remember that there is a bug in this method which causes it to loop indefinitely. I want this test to exit after 1 second if there is no result. That's what the timeout parameter does. This second optional parameter of the @Test annotation (the first one was expected), causes a test to fail if it takes longer than a specified amount of clock time (milliseconds). When I run the test I get:

```
There was 1 failure:
1) squareRoot(junit4.AdvancedTest)
java.lang.Exception: test timed out after 1000 milliseconds
        at org.junit.internal.runners.TestMethodRunner.runWithTimeout(TestMethodRunner.java:68)
        at org.junit.internal.runners.TestMethodRunner.run(TestMethodRunner.java:43)

FAILURES!!!
Tests run: 4,  Failures: 1
```

Parameterized Tests
In Listing 1 I tested the squareRoot <<it is the square method not the squareRoot>> method by creating several test methods (square2, square4, square5), which do exactly the same thing, parameterized by some variables. This copy/paste technique can now be optimized using a parameterized test case (see Listing 2).

The test case in Listing 2 uses two new annotations. When a class is annotated with @RunWith, JUnit will invoke the class referenced to run the tests instead of the default runner. To use a parameterized test case, you need to use the runner org.junit.runners.Parameterized. To know which parameters to use, the test case needs a public static method (here data() but the name is irrelevant) that returns a Collection and is annotated with @Parameters. You also need a public constructor that takes these parameters.

When running this class, the output is:

```
java org.junit.runner.JUnitCore junit4.SquareTest
JUnit version 4.1

.......E

There was 1 failure:
1) square[6](junit4.SquareTest)
java.lang.AssertionError: expected:<48> but was:<49>
        at org.junit.Assert.fail(Assert.java:69)

FAILURES!!!
Tests run: 7,  Failures: 1
```

There are seven tests executed (the seven dots '.'), as if seven individual square methods were written. Note that we have a failure in our test because the square of 7 is 49, not 48.

Suite
To run several test classes into a suite in JUnit 3.8 you had to add a suite() method to your classes. With JUnit 4 you use annotations instead. To run the CalculatorTest and SquareTest you write an empty class with @RunWith and @Suite annotations.

```
package junit4;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
        CalculatorTest.class,
        SquareTest.class
        })
public class AllCalculatorTests {
}
```

Again, the @RunWith annotation is telling JUnit to use the org.junit.runner.Suite. This runner allows you to manually build a suite containing tests from many classes. The names of these classes are defined in the @Suite.SuiteClass. When you run this class, it will run CalculatorTest and SquareTest. The output is:

```
java -ea org.junit.runner.JUnitCore junit4.AllCalculatorTests
JUnit version 4.1
```

```
...E.EI.......E

There were 3 failures:
1) subtract(junit4.CalculatorTest)
java.lang.AssertionError: expected:<9> but was:<8>
        at org.junit.Assert.fail(Assert.java:69)
2) divide(junit4.CalculatorTest)
java.lang.AssertionError
        at junit4.CalculatorTest.divide(CalculatorTest.java:40)
3) square[6](junit4.SquareTest)
java.lang.AssertionError: expected:<48> but was:<49>
        at org.junit.Assert.fail(Assert.java:69)

FAILURES!!!
Tests run: 11,  Failures: 3
```

Runner

It may not be obvious but JUnit 4 uses runners extensively. If @RunWith is not specified, your class will still be executed with a default runner (org.junit.internal.runners.TestClassRunner). The original Calculator class doesn't explicitly declare a runner, so therefore it uses the default. A class containing a method with @Test has a @RunWith by implication. In fact, you could add the following code to the Calculator class and the output would be exactly the same.

```
import org.junit.internal.runners.TestClassRunner;
import org.junit.runner.RunWith;

@RunWith(TestClassRunner.class)
public class CalculatorTest {
...
}
```

In the case of the @Parameterized and @Suite I needed a special runner to execute my test cases. That's why I explicitly annotated them.

Tools Integration (or Lack Thereof)

As I'm writing this article, JUnit 4 integration in IDEs is not yet perfect. In fact if you try to run the test classes we've just seen, they will not work in any IDE because they are not recognized as being test classes. For forward compatibility JUnit 4 comes with an adapter (junit.framework.JUnit4TestAdapter) that you have to use in a suite() method. Here is the code you have to add in every class to make them understandable by IDEs, Ant, and JUnit 3.8 text runner:

```
    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(CalculatorTest.class);
    }
```

Intellij IDEA
IDEA 5 does not integrate JUnit 4. We will have to wait for IDEA 6. In the meantime I've used the early access version (Demetra build 5321). The parameterized test case didn't work. Figure 1 shows the CalculatorTest executing (the ignored test is represented with a different icon).

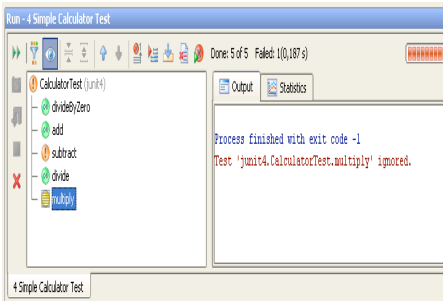

Figure 1. IDEA Demetra is only running CalculatorTest.

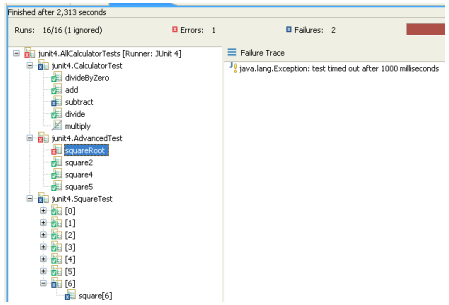

Figure 2. Eclipse 3.2RC7 is running the suite class AllCalculatorTests.

Eclipse
I've used the version 3.2 RC7 of Eclipse. It is not a stable version but the integration with JUnit 4 is much better than with IDEA. Figure 2 shows what you get when running the AllCalculatorTests class.

As you can see, the parameterized test case (SquareTest) is represented as seven individual tests.

Ant Integration
The junit task currently only supports JUnit 3.8 style tests, meaning you also have to wrap your JUnit 4 tests in a JUnit4TestAdapter in order to run them in Ant. The <junit> task is used in exactly the same way as in JUnit 3.8:

```
<!-- Test -->
<target name="test" depends="compile">
    <junit fork="yes" haltonfailure="yes">
        <test name=" junit4.AllCalculatorTests"/>
        <formatter type="plain" usefile="false"/>
        <classpath refid="classpath"/>
    </junit>
</target>
```

JUnit: Losing Market or Coming Back Strong?
For a long time JUnit was the defacto unit testing framework. But lately not much has happened to this framework: no major release, no notable new features. This is possibly the reason why other frameworks, such as Test-NG have started taking over.

With this new version, JUnit is back on track. It has new APIs and now uses annotations, making it easier to develop test cases. In fact, the JUnit developers have started to think of new, future annotations. For example, you could add a @Prerequisite annotation for a test case that depends on prerequisites (e.g. you need to be online to execute this test); or add a @Repeat annotation that would specify the number of repetitions along with a timeout (e.g. repeat a test five times to make sure there is a real timeout problem); or even add a platform parameter to the @Ignore annotation (e.g. @Ignore(platform=macos), which would ignore a test only if you run on a MacOS platform). As

you can see, JUnit is still alive with a promising future.

Antonio Goncalves is a senior architect specialized in Java/J2EE. Former BEA consultant he now helps insurance, finance and telecommunication clients set up their architectures. He also teaches J2EE at CNAM University in Paris.