Jump into JUnit 4

Streamlined testing with Java 5 annotations

Skill Level: Intermediate

Andrew Glover (aglover@stelligent.com)
President
Stelligent Incorporated

20 Feb 2007

JUnit 4 has dropped the strict naming conventions and inheritance hierarchies of old, in favor of the streamlined flexibility of Java™ 5 annotations. In this tutorial, a supplement to his popular series on code quality, testing fanatic Andrew Glover shows you how to leverage the new features enabled by annotations, including parametric tests, exception tests, and timed tests. He also introduces JUnit 4's flexible fixtures and shows you how to use annotations, rather than suites, to logically group tests before running them. The tutorial includes several sample tests run in Eclipse and instructions for running JUnit 4 tests in older, incompatible versions of Ant.

Section 1. Before you start

About this tutorial

The introduction of Java 5 annotations brought dramatic changes to JUnit, transforming it from the testing framework developers had grown to know and love into something more streamlined but less familiar. In this tutorial, I discuss the most important changes to JUnit 4 and introduce the exciting new features you've probably heard about but may not yet be using.

Objectives

This tutorial guides you step-by-step through the fundamental concepts of JUnit 4, with emphasis on the new Java 5 annotations. At the conclusion of this one-hour

© Copyright IBM Corporation 1994, 2007. All rights reserved.

tutorial, you will understand the major changes to JUnit 4, as well as being familiar with features such as exception testing, parametric testing, and the new flexible fixture model. You will know how to declare a test, how to use annotations (rather than suites) to logically group tests prior to running them, and how to run tests in Eclipse 3.2 or Ant, as well as from the command line.

Prerequisites

To get the most from this tutorial, you should be familiar with Java development in general. This tutorial also assumes that you understand the value of developer testing and are familiar with basic pattern matching. To follow the section on running JUnit 4 tests, you should be able to work with Eclipse 3.2 as an IDE and with Ant 1.6 or greater. You need not be familiar with previous versions of JUnit to follow the tutorial.

System requirements

To follow along and try out the code for this tutorial, you need a working installation of Sun's JDK 1.5.0_09 (or later) or the IBM developer kit for Java technology 1.5.0 SR3. For the sections on running JUnit 4 in Eclipse, you need a working installation of Eclipse 3.2 or later. For the section on Ant, you need version 1.6 or greater.

The recommended system configuration for this tutorial is as follows:

- A system supporting either the Sun JDK 1.5.0_09 (or later) or the IBM developer kit for Java technology 1.5.0 SR3 with at least 500MB of main memory
- At least 20MB of disk space to install the software components and examples covered

The instructions in the tutorial are based on a Microsoft Windows operating system. All the tools covered in the tutorial also work on Linux and UNIX systems.

Section 2. What's new in JUnit 4

Thanks to Java 5 annotations, JUnit 4 is more lightweight and flexible than ever. It has dropped its strict naming conventions and inheritance hierarchies in favor of some exciting new functionality. Here's a quick list of what's new in JUnit 4:

- Parametric tests
- Exception tests

© Copyright IBM Corporation 1994, 2007. All rights reserved.

- Timeout tests
- Flexible fixtures
- An easy way to ignore tests
- A new way to logically group tests

I start by explaining the most significant, overarching changes to JUnit 4, in preparation for introducing these and more new features in the later sections.

Out with the old

Prior to the addition of Java 5 annotations in JUnit 4, the framework had established two conventions that were essential to its ability to function. The first was that JUnit implicitly required that any method written to function as a logical test begin with the word *test*. Any method beginning with that word, such as `testUserCreate`, would be executed according to a well-defined test process that guaranteed the execution of a fixture both before and after the test method. Second, for JUnit to recognize a class object containing tests, the class itself was required to extend from JUnit's `TestCase` (or some derivation thereof). A test that violated either of these two conventions *would not run*.

Listing 1 shows a JUnit test written prior to JUnit 4.

Listing 1. Does it have to be so hard?

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import junit.framework.TestCase;

public class RegularExpressionTest extends TestCase {

 private String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private Pattern pattern;

 protected void setUp() throws Exception {
  this.pattern = Pattern.compile(this.zipRegEx);
 }

 public void testZipCode() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101");
  boolean isValid = mtcher.matches();
  assertTrue("Pattern did not validate zip code", isValid);
 }
}
```

> Many argue that JUnit 4's use of annotations was influenced by TestNG as well as .NET's NUnit. See Resources to learn more about annotations in other testing frameworks.

In with the new

JUnit 4 uses Java 5 annotations to completely eliminate both of these conventions.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

The class hierarchy is no longer required and methods intended to function as tests need only be decorated with a newly defined `@Test` annotation.

Listing 2 shows the same test seen in Listing 1 but redefined using annotations:

Listing 2. Testing with annotations

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class RegularExpressionTest {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
 }

 @Test
 public void verifyGoodZipCode() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101");
  boolean isValid = mtcher.matches();
  assertTrue("Pattern did not validate zip code", isValid);
 }
}
```

The test in Listing 2 may not be any easier to code, but it certainly is easier to comprehend.

Documentation simplified

One useful side effect of annotations is that they clearly document what each method is intended to do *without* requiring a deep understanding of the framework's internal model. What could be more clear than decorating a test method with `@Test`? This is a big improvement on old-style JUnit, which required a fair amount of familiarity with JUnit conventions, even if all you wanted was to understand each method's contribution to an overall test case.

Annotations are a big help when it comes to parsing a test that has already been written, but they become even more compelling when you see the extra kick they bring to the process of writing tests.

Section 3. Testing with annotations

Java 5 annotations make JUnit 4 a notably different framework from previous versions. In this section, I familiarize you with using annotations in key areas such as

© Copyright IBM Corporation 1994, 2007. All rights reserved.

test declaration and exception testing, as well as for timeouts and ignoring unwanted or unusable tests.

Test declaration

Declaring a test in JUnit 4 is a matter of decorating a test method with the `@Test` annotation. Note that you need not extend from any specialized class, as Listing 3 shows:

Listing 3. Test declaration in JUnit 4

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertFalse;

public class RegularExpressionTest {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
 }

 @Test
 public void verifyZipCodeNoMatch() throws Exception{
  Matcher mtcher = this.pattern.matcher("2211");
  boolean notValid = mtcher.matches();
  assertFalse("Pattern did validate zip code", notValid);
 }
}
```

A note about static imports

I used Java 5's static import feature to import the `Assert` class's `assertFalse()` method in Listing 3. This is because test classes do not extend from `TestCase` as they did in previous versions of JUnit.

Testing for exceptions

As with previous versions of JUnit, it's usually a good idea to specify that your test throws `Exception`. The only time you want to ignore this rule is if you're trying to test for a particular exception. If a test throws an exception, the framework reports a failure.

If you'd actually like to test for a particular exception, JUnit 4's `@Test` annotation supports an `expected` parameter, which is intended to represent the exception type the test should throw upon execution.

A simple comparison demonstrates what a difference the new parameter makes.

Exception testing in JUnit 3.8

© Copyright IBM Corporation 1994, 2007. All rights reserved.

The JUnit 3.8 test in Listing 4, aptly named `testZipCodeGroupException()`, verifies that attempting to obtain the third group of the regular expression I've declared will result in an `IndexOutOfBoundsException`:

Listing 4. Testing for an exception in JUnit 3.8

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import junit.framework.TestCase;

public class RegularExpressionTest extends TestCase {

 private String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private Pattern pattern;

 protected void setUp() throws Exception {
  this.pattern = Pattern.compile(this.zipRegEx);
 }

 public void testZipCodeGroupException() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  try{
   mtcher.group(2);
   fail("No exception was thrown");
  }catch(IndexOutOfBoundsException e){
  }
 }
}
```

This older version of JUnit requires me to do quite a bit of coding for such a simple test -- namely writing a `try/catch` and failing the test if the exception isn't caught.

Exception testing in JUnit 4

The exception test in Listing 5 is no different from the one in Listing 4, except that it uses the new `expected` parameter. (Note that I was able to retrofit the test from Listing 4 by passing in the `IndexOutOfBoundsException` exception to the `@Test` annotation.)

Listing 5. Exception testing with the 'expected' parameter

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;

public class RegularExpressionJUnit4Test {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
 }

 @Test(expected=IndexOutOfBoundsException.class)
 public void verifyZipCodeGroupException() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  mtcher.group(2);
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
  }
 }
```

Testing with timeouts

In JUnit 4, a test case can take a timeout value as a parameter. As you can see in Listing 6, the `timeout` value represents the maximum amount of time the test can take to run: if the time is exceeded, the test fails.

Listing 6. Testing with a timeout value

```
@Test(timeout=1)
public void verifyFastZipCodeMatch() throws Exception{
 Pattern pattern = Pattern.compile("^\\d{5}([\\-]\\d{4})?$");
 Matcher mtcher = pattern.matcher("22011");
 boolean isValid = mtcher.matches();
 assertTrue("Pattern did not validate zip code", isValid);
}
```

Testing with timeouts is easy: Simply decorate a method with `@Test` followed by a `timeout` value and you've got yourself an automated timeout test!

Ignoring tests

Prior to JUnit 4, ignoring broken or incomplete tests was a bit of a pain. If you wanted the framework to ignore a particular test, you had to alter its name so as to not follow the test nomenclature. For instance, I often found myself placing a "_" in front of a test method to indicate that the test wasn't made to be run at the current moment.

JUnit 4 has introduced an annotation aptly dubbed `@Ignore`, which forces the framework to ignore a particular test method. You can also pass in a message documenting your decision for unsuspecting developers who happen upon the ignored test.

The @Ignore annotation

Listing 7 shows how easy it is to ignore a test whose regular expression isn't yet working:

Listing 7. Ignore this test

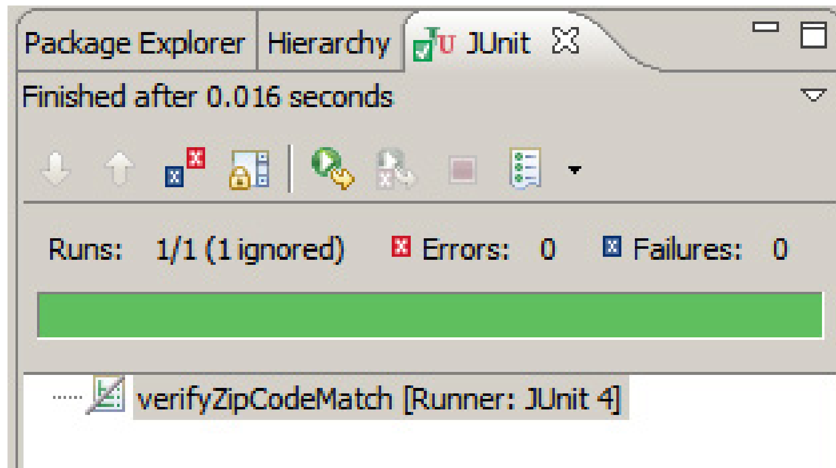
```
@Ignore("this regular expression isn't working yet")
@Test
public void verifyZipCodeMatch() throws Exception{
 Pattern pattern = Pattern.compile("^\\d{5}([\\-]\\d{4})");
 Matcher mtcher = pattern.matcher("22011");
 boolean isValid = mtcher.matches();
 assertTrue("Pattern did not validate zip code", isValid);
}
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

It's documented

Attempting to run this test in Eclipse (for example) will report an ignored test, as shown in Figure 1:

Figure 1. How an ignored test shows up in Eclipse



Section 4. Test fixtures

Test fixtures aren't new to JUnit 4, but the fixture model is new and improved. In this section, I explain why and where you might want to employ fixtures and then show you the difference between the inflexible fixtures of old and JUnit 4's sparkly new model.

Why use fixtures?

Fixtures foster reuse through a contract that ensures that particular logic is run either before or after a test. In older versions of JUnit, this contract was implicit regardless of whether you implemented a fixture or not. JUnit 4, however, has made fixtures explicit through annotations, which means the contract is only enforced if you actually decide to use a fixture.

Through a contract that ensures fixtures can be run either before or after a test, you can code reusable logic. This logic, for example, could be initializing a class that you will test in multiple test cases or even logic to populate a database before you run a data-dependent test. Either way, using fixtures ensures a more manageable test case: one that relies on common logic.

Fixtures come in especially handy when you are running many tests that use the same logic and some or all of them fail. Rather than sifting through each test's set-up logic, you can look in one place to deduce the cause of failure. In addition, if some tests pass and others fail, you might be able to avoid examining the fixture logic as a

© Copyright IBM Corporation 1994, 2007. All rights reserved.

source of the failures altogether.

Inflexible fixtures

Older versions of JUnit employed a somewhat inflexible fixture model, where you had to wrap every test method by `setUp()` and `tearDown()` methods. You can see a potential downside of this model in Listing 8, where the `setUp()` method is implemented and therefore run twice -- once for each test defined:

Listing 8. Inflexible fixtures

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import junit.framework.TestCase;

public class RegularExpressionTest extends TestCase {

 private String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private Pattern pattern;

 protected void setUp() throws Exception {
  this.pattern = Pattern.compile(this.zipRegEx);
 }

 public void testZipCodeGroup() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  assertEquals("group(1) didn't equal -5051", "-5051", mtcher.group(1));
 }

 public void testZipCodeGroupException() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  try{
   mtcher.group(2);
   fail("No exception was thrown");
  }catch(IndexOutOfBoundsException e){
  }
 }
}
```

Working around it

In prior versions of JUnit, it was possible to specify that a fixture run only once, using the `TestSetup` decorator, but it was a cumbersome operation, as shown in Listing 9 (note the required `suite()` method):

Listing 9. TestSetup prior to JUnit 4

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import junit.extensions.TestSetup;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.textui.TestRunner;
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
public class OneTimeRegularExpressionTest extends TestCase {

 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 public static Test suite() {
  TestSetup setup = new TestSetup(
    new TestSuite(OneTimeRegularExpressionTest.class)) {
     protected void setUp() throws Exception {
      pattern = Pattern.compile(zipRegEx);
     }
    };
  return setup;
 }

 public void testZipCodeGroup() throws Exception {
  Matcher mtcher = pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  assertEquals("group(1) didn't equal -5051", "-5051", mtcher.group(1));
 }

 public void testZipCodeGroupException() throws Exception {
  Matcher mtcher = pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  try {
   mtcher.group(2);
   fail("No exception was thrown");
  } catch (IndexOutOfBoundsException e) {
  }
 }
}
```

Suffice it to say that prior to JUnit 4, getting around fixtures was almost more trouble than it was worth.

Flexibility in 4.0

JUnit 4 uses annotations to cut out a lot of the overhead of fixtures, allowing you to run a fixture for every test or just once for an entire class or not at all. There are four fixture annotations: two for class-level fixtures and two for method-level ones. At the class level, you have `@BeforeClass` and `@AfterClass`, and at the method (or test) level, you have `@Before` and `@After`.

The test case in Listing 10 includes a fixture that is run for both tests using the `@Before` annotation:

Listing 10. Flexible fixtures using annotations

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;

public class RegularExpressionJUnit4Test {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @Before
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
  }

  @Test
  public void verifyZipCodeNoMatch() throws Exception{
   Matcher mtcher = this.pattern.matcher("2211");
   boolean notValid = mtcher.matches();
   assertFalse("Pattern did validate zip code", notValid);
  }

  @Test(expected=IndexOutOfBoundsException.class)
  public void verifyZipCodeGroupException() throws Exception{
   Matcher mtcher = this.pattern.matcher("22101-5051");
   boolean isValid = mtcher.matches();
   mtcher.group(2);
  }
 }
```

One-time fixtures

What if you wanted to run a fixture just once? Rather than implementing an old-style decorator, as shown in Listing 9, you could instead use the `@BeforeClass` annotation, as shown in Listing 11:

Listing 11. One-time setup in JUnit 4

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;

public class RegularExpressionJUnit4Test {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
 }

 @Test
 public void verifyZipCodeNoMatch() throws Exception{
  Matcher mtcher = this.pattern.matcher("2211");
  boolean notValid = mtcher.matches();
  assertFalse("Pattern did validate zip code", notValid);
 }

 @Test(expected=IndexOutOfBoundsException.class)
 public void verifyZipCodeGroupException() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101-5051");
  boolean isValid = mtcher.matches();
  mtcher.group(2);
 }
}
```

Tear it down

In case you're wondering, the old `tearDown()` functionality hasn't gone away under the new fixture model. If you wanted to do a `tearDown()`, you could simply create a new method and use the `@After` or `@AfterClass` as needed.

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Flexibility squared

You can specify more than one fixture for a test case in JUnit 4. The new annotation-driven fixtures do nothing to hold you back from creating multiple `@BeforeClass` fixture methods. Keep in mind, however, that as of the current version of JUnit 4, you cannot specify which fixture methods are to be run first, which can get tricky if you decide to use more than one.

Section 5. Run it: Testing in JUnit 4

One of the most striking features of the new and improved JUnit 4 is the lack of suites -- the mechanism used to logically group tests and run them as a single unit. In this section, I introduce the new, streamlined annotations that have replaced suites and show you how to run your JUnit 4 tests in both Eclipse and Ant.

An old-fashioned suite

So you can see the difference, check out the old-style JUnit suite shown in Listing 12 (this suite groups two logical test classes and runs them as a single unit):

Listing 12. An old-fashioned JUnit suite

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class JUnit3Suite {

 public static Test suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(OneTimeRegularExpressionTest.suite());
  suite.addTestSuite(RegularExpressionTest.class);
  return suite;
 }
}
```

Two sweet new annotations

In JUnit 4, suite semantics have been replaced with two new annotations. The first one, `@RunWith`, is designed to facilitate having different runners (other than the ones built into the framework) execute a particular test class. JUnit 4 bundles a suite runner, named `Suite`, that you must specify in the `@RunWith` annotation. What's more, you must provide another annotation, named `@SuiteClasses`, which takes as a parameter a list of classes intended to represent the test suite.

Listing 13. These annotations are pretty sweet

© Copyright IBM Corporation 1994, 2007. All rights reserved.

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

@RunWith(Suite.class)
@SuiteClasses({ParametricRegularExpressionTest.class,
      RegularExpressionTest.class,
      TimedRegularExpressionTest.class})
public class JUnit4Suite {

}
```

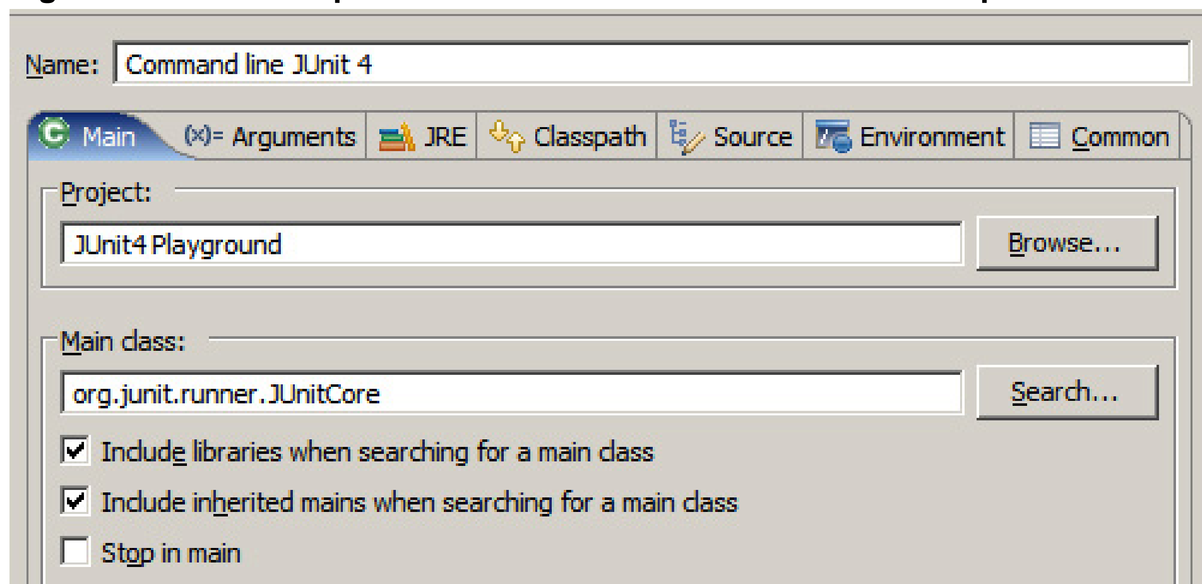
If the new JUnit 4 annotations seem weird to you, don't worry. Apparently the creators of JUnit 4 feel similarly. In the Javadocs, they explain that they expect the runner API to change "as we learn how people really use it." At least they're honest!

Running JUnit 4 tests in Eclipse

You have the option to run a JUnit 4 test class through an IDE like Eclipse or through the command line. You can run JUnit tests in Eclipse versions 3.2 and above by selecting the **Run As JUnit** test option. Running tests through the command line requires executing the `org.junit.runner.JUnitCore` class and passing as an argument the fully qualified name of a test.

For example, in Eclipse, if you didn't want to use the bundled JUnit runner, you could define a new run configuration and first specify the `JUnitCore` class, as I do in Figure 2:

Figure 2. The first step to run JUnit 4 command-line tests in Eclipse

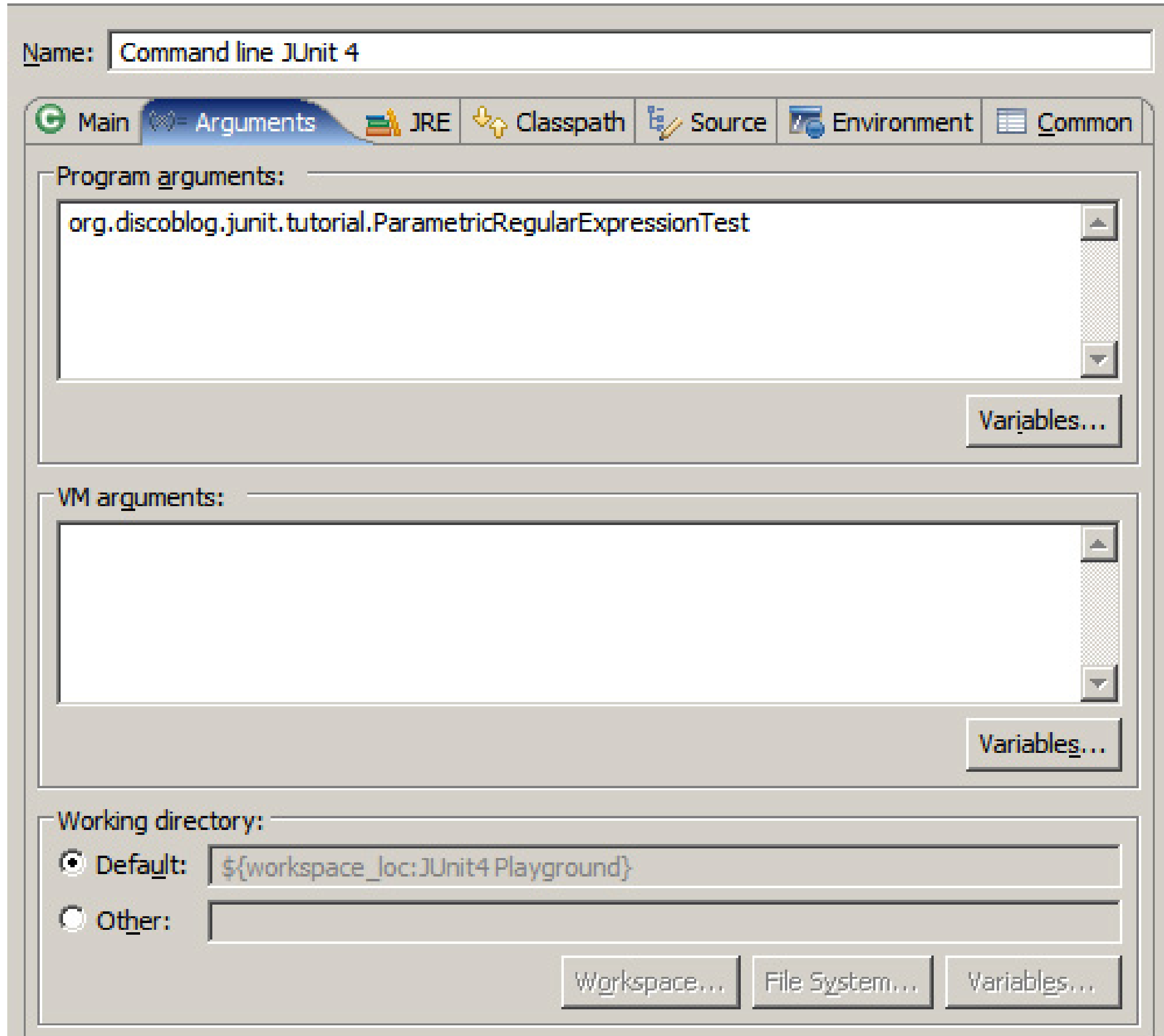


Specifying a test

Next, you would need to specify which test to run, by adding the fully qualified name of the test to the "Program arguments" text box of the Arguments tab, as shown in Figure 3:

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Figure 3. Second step to run JUnit command line tests in Eclipse



Ant and JUnit 4

Ant and JUnit have been a great team for a while now, and many developers expected the relationship would only get better with the introduction of JUnit 4. As it turns out, though, there's a hitch. If you're running any version of Ant prior to 1.7, you cannot simply run JUnit 4 tests out of the box. That isn't to say you can't run the tests -- only that you can't run them out of the box.

An ill-fitted pairing

Running the JUnit 4 test (in Listing 14) in Ant (pre 1.7) yields some interesting results:

Listing 14. A simple JUnit 4 test class

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```


© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class RegularExpressionTest {
 private static String zipRegEx = "^\\d{5}([\\-]\\d{4})?$";
 private static Pattern pattern;

 @BeforeClass
 public static void setUpBeforeClass() throws Exception {
  pattern = Pattern.compile(zipRegEx);
 }

 @Test
 public void verifyGoodZipCode() throws Exception{
  Matcher mtcher = this.pattern.matcher("22101");
  boolean isValid = mtcher.matches();
  assertTrue("Pattern did not validate zip code", isValid);
 }
}
```

Failed on many counts

Using the venerable `junit` task in Ant yields the errors in Listing 15:

Listing 15. A bunch of errors

```
[junit] Running test.com.acme.RegularExpressionTest
[junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.047 sec
[junit] Testsuite: test.com.acme.RegularExpressionTest
[junit] Tests run: 1, Failures: 1, Errors: 0, Time elapsed: 0.047 sec

[junit] Testcase: warning took 0.016 sec
[junit]     FAILED
[junit] No tests found in test.com.acme.RegularExpressionTest
[junit] junit.framework.AssertionFailedError: No tests found in
  test.com.acme.RegularExpressionTest
[junit] Test test.com.acme.RegularExpressionTest FAILED
```

A retro solution

If you want to run your JUnit 4 tests on a version of Ant prior to 1.7, you must retrofit the test case with a `suite()` method that returns an instance of `JUnit4TestAdapter`, as shown in Listing 16:

Listing 16. A new use for an old method

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

You must fully qualify the return type of `Test` in this instance because of the similarly named `@Test` annotation. Once the `suite()` method is in place, however, any version of Ant will happily run your JUnit 4 tests!

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Section 6. Parametric testing

Every once in a while, an application's business logic requires that you write a hugely varying number of tests to ensure it is sound. In previous versions of JUnit, this sort of scenario could be very inconvenient, mainly because varying the parameter groups to a method under test meant writing a test case for each unique group.

JUnit 4 introduces an excellent new feature that lets you create generic tests that can be fed by parametric values. As a result, you can create a single test case and run it multiple times -- once for every parameter you've created.

Parametric simplicity

It takes just five steps to create a parametric test in JUnit 4:

1. Create a generic test that takes no parameters.
2. Create a `static` feeder method that returns a `Collection` type and decorate it with the `@Parameter` annotation.
3. Create class members for the parameter types required in the generic method defined in Step 1.
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3.
5. Specify the test case be run with the `Parameterized` class via the `@RunWith` annotation.

I'll walk through these steps one by one.

Step 1. Create a generic test

Listing 17 shows a generic test to verify various values against a regular expression. Note the `phrase` and `match` values aren't defined.

Listing 17. A generic test

```
@Test
public void verifyGoodZipCode() throws Exception{
 Matcher mtcher = this.pattern.matcher(phrase);
 boolean isValid = mtcher.matches();
 assertEquals("Pattern did not validate zip code", isValid, match);
}
```

Step 2. Create a feeder method

© Copyright IBM Corporation 1994, 2007. All rights reserved.

The next step is to create a feeder method, which you must declare as `static` and return a `Collection` type. You need to decorate the method with the `@Parameters` annotation. Inside the method, you simply create a multi-dimensional `Object` array and convert it to a `List`, shown in Listing 18:

Listing 18. The feeder method with the @Parameters annotation

```
@Parameters
public static Collection regExValues() {
 return Arrays.asList(new Object[][] {
  {"22101", true },
  {"221x1", false },
  {"22101-5150", true },
  {"221015150", false }});
}
```

Step 3. Create two class members

Because the parameters are of types `String` and `boolean`, your next step is to create two class members:

Listing 19. Declare two class members

```
private String phrase;
private boolean match;
```

Step 4. Create a constructor

The constructor you create next links the class members to your parameter values, as shown in Listing 20:

Listing 20. A constructor that matches values

```
public ParametricRegularExpressionTest(String phrase, boolean match) {
 this.phrase = phrase;
 this.match = match;
}
```

Step 5. Specify the Parameterized class

Last, specify at the class level that this test must be run with the `Parameterized` class, as shown in Listing 21:

Listing 21. Specifying Parameterized and the @RunWith annotation

```
@RunWith(Parameterized.class)
public class ParametricRegularExpressionTest {
 //...
}
```

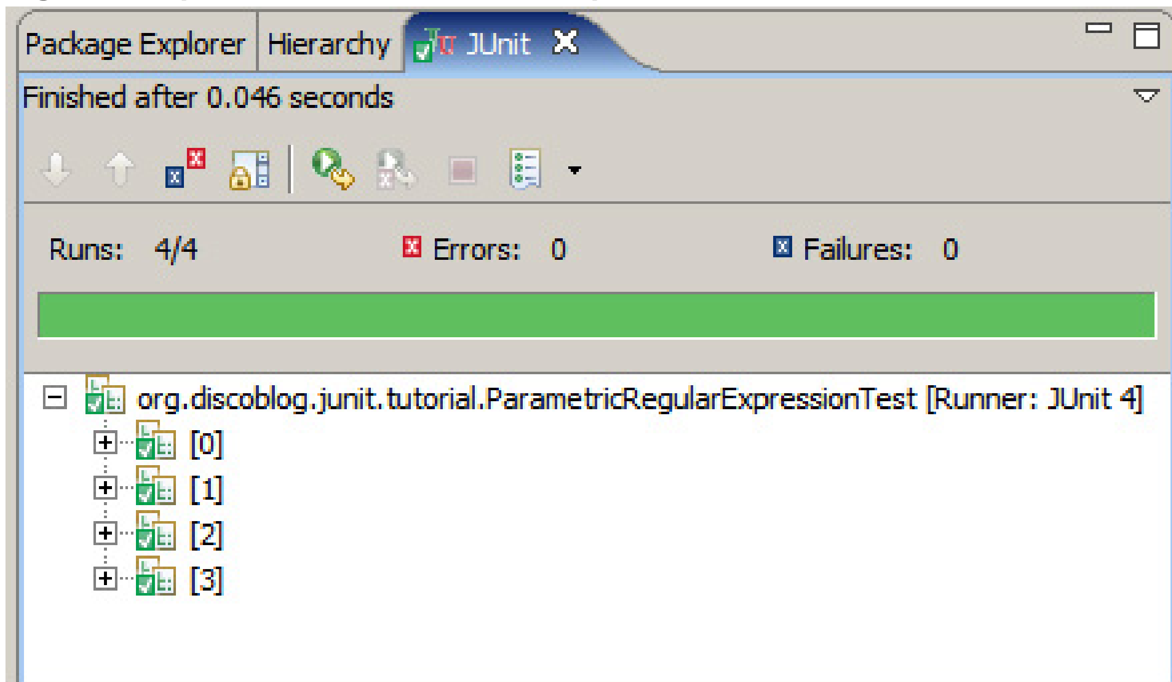
Run the test

© Copyright IBM Corporation 1994, 2007. All rights reserved.

When you execute the test class, the generic `verifyGoodZipCode()` test method runs four times, once for each value pair defined in the `regExValues()` data feeder method in Listing 18.

If you ran this test in Eclipse, for example, it would report four test runs, as shown in Figure 4:

Figure 4. A parametric tests run in Eclipse



Section 7. What else is new?

In addition to the important changes discussed so far, JUnit 4 also has introduced a few minor ones; namely the addition of a new assert method and the elimination of a terminal state.

A new assert

JUnit 4 has added a new assert method for comparing array contents. It isn't a big addition, but it does mean you'll never again have to iterate over the contents of an array and assert each individual item.

For example, the code in Listing 22 isn't possible in older versions of JUnit. This test case fails because of the slight difference in the second element of each array.

Listing 22. assertEquals supports arrays now in JUnit 4

© Copyright IBM Corporation 1994, 2007. All rights reserved.

```
@Test
public void verifyArrayContents() throws Exception{
 String[] actual = new String[] {"JUnit 3.8.x", "JUnit 4", "TestNG"};
 String[] var = new String[] {"JUnit 3.8.x", "JUnit 4.1", "TestNG 5.5"};
 assertEquals("the two arrays should not be equal", actual, var);
}
```

No more errors!

A small but sweet change in JUnit 4 is that it eliminates the notion of errors. Whereas previous versions would report both the number of failures and the number of errors, in JUnit 4, a test either passes or it fails.

Interestingly, while one state has been eliminated, a new one has been added, this time because of the ability to ignore tests. When you execute a series of tests, JUnit 4 reports the number of tests run, the number of failures, and the number of tests ignored.

Section 8. Wrap up

JUnit 4's radical departure from its original design doesn't mean the framework operates all that differently -- the power and simplicity of the original framework is intact. In fact, as you dig deeper into the framework, you should find that it hasn't sacrificed any of the core principles that ignited the developer testing revolution, although it has added some compelling new features.

In this tutorial, you walked through the process of getting to know JUnit 4, from test declaration to parametric testing. You discovered new features such as timed tests and exception testing and learned about changes to familiar ones like fixtures and logical grouping. You also saw how a test run looks in Eclipse and learned a simple workaround that allows you to run your tests in any version of Ant, even those prior to 1.7.

If there is any one thing I hope you learned from this tutorial, it's that annotations don't detract a thing from JUnit and they do add considerable ease of use. Give annotations a try: they'll have you jumping in and out of test writing before you know it!

© Copyright IBM Corporation 1994, 2007. All rights reserved.

Resources

- Participate in the discussion forum for this content.
- "An early look at JUnit 4" (Elliotte Rusty Harold, developerWorks, September 2005): Elliotte Harold takes JUnit 4 out for a spin.
- "In pursuit of code quality: JUnit 4 vs. TestNG" (Andrew Glover, developerWorks, August 2006): Is it true that JUnit 4 has borrowed all of its best tricks from TestNG?
- "TestNG makes Java unit testing a breeze" (Filippo Diotalevi, developerWorks, January 2005): TestNG isn't just powerful, innovative, extensible, and flexible; it also illustrates an interesting application of Java annotations.
- "Annotations in Tiger, Part 1: Add metadata to Java code" (Brett McLaughlin, developerWorks, September 2004): Brett McLaughlin explains why metadata is so useful, introduces you to annotations in the Java language, and delves into Java 5's built-in annotations.
- "Classworking toolkit: Annotations vs. configuration files" (Dennis Sosnoski, developerWorks, August 2005): Dennis Sosnoski explains why configuration files still have their uses, especially for aspect-like functions that cut across the source-code structure of an application.
- "JUnit Reloaded" (Ralf Stuckert, Java.net, December 2006): Compares JUnit 4 to earlier versions.
- "JUnit 4 you" (Fabiano Cruz, Fabiano Cruz's Blog, June 2006): An overview of tool and IDE support for JUnit 4.
- "Limiting asserts in test cases" (thediscoblog.com): A best practice for JUnit, TestNG, or any other framework that fails fast.
- "DbUnit with JUnit 4" (testearly.com): Just because JUnit 4 is different doesn't mean you can't use it with extension frameworks built for older versions of JUnit.
- "Using JUnit extensions in TestNG" (Andrew Glover, thediscoblog.com, March 2006): Just because a framework claims to be a JUnit extension doesn't mean it can't be used within TestNG.
- *In pursuit of code quality* series (Andrew Glover, developerWorks): See all the articles in this series ranging from code metrics to testing frameworks to refactoring.

About the author

Andrew Glover
Andrew Glover is president of Stelligent Incorporated, which helps companies address software quality with effective developer testing strategies and continuous

© Copyright IBM Corporation 1994, 2007. All rights reserved.

integration techniques that enable teams to monitor code quality early and often. Check out Andy's blog for a list of his publications.

© Copyright IBM Corporation 1994, 2007. All rights reserved.