IBM

An early look at JUnit 4

Upcoming release promises evolution in testing

Level: Intermediate

Elliotte Harold (elharo@metalab.unc.edu), Adjunct Professor, Polytechnic University

13 Sep 2005

JUnit is the *de facto* standard unit testing library for the Java™ language. JUnit 4 is the first significant release of this library in almost three years. It promises to simplify testing by exploiting Java 5's annotation feature to identify tests rather than relying on subclassing, reflection, and naming conventions. In this article, obsessive code tester Elliotte Harold takes JUnit 4 out for a spin and details how to use the new framework in your own work. Note that this article assumes prior experience with JUnit.

JUnit, developed by Kent Beck and Erich Gamma, is almost indisputably the single most important third-party Java library ever developed. As Martin Fowler has said, "Never in the field of software development was so much owed by so many to so few lines of code." JUnit kick-started and then fueled the testing explosion. Thanks to JUnit, Java code tends to be far more robust, reliable, and bug free than code has ever been before. JUnit (itself inspired by Smalltalk's SUnit) has inspired a whole family of xUnit tools bringing the benefits of unit testing to a wide range of languages. nUnit (.NET), pyUnit (Python), CppUnit (C++), dUnit (Delphi), and others have test-infected programmers on a multitude of platforms and languages.

However, JUnit is just a tool. The real benefits come from the ideas and techniques embodied by JUnit, not the framework itself. Unit testing, test-first programming, and test-driven development do not have to be implemented in JUnit, any more than GUI programming must be done with Swing. JUnit itself was last updated almost three years ago. Although it's proved more robust and longer lasting than most frameworks, bugs have been found; and, more importantly, Java has moved on. The language now supports generics, enumerations, variable length argument lists, and annotations--features that open up new possibilities for reusable framework design.

JUnit's stasis has not been missed by programmers who are eager to dethrone it. Challengers range from Bill Venners's Artima SuiteRunner to Cedric Beust's TestNG. These libraries have some features to recommend them, but none have achieved the mind or market share held by JUnit. None have broad out-of-the-box support in products like Ant, Maven, or Eclipse. So Beck and Gamma have commenced work on an updated version of JUnit that takes advantage of the new features of Java 5 (especially annotations) to make unit testing even simpler than it was with the original JUnit. According to Beck, "The theme of JUnit 4 is to encourage more developers to write more tests by further simplifying JUnit." Although it maintains backwards compatibility with existing JUnit 3.8 test suites, JUnit 4 promises to be the most significant innovation in Java unit testing since JUnit 1.0.

Note: The changes to the framework are quite bleeding-edge. Although the broad outlines of JUnit 4 are clear, the details can still change. This is meant to be an early sneak peek at JUnit 4, not the final word.

Test methods

All previous versions of JUnit use naming conventions and reflection to locate tests. For example, this code tests that 1+1 equals 2:

```
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  public void testAddition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

By contrast, in JUnit 4, tests are identified by an `@Test` annotation, as shown here:

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void testAddition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

The advantage to using annotations is that you no longer need to name all your methods `testFoo()`, `testBar()`, etc. For instance, the following approach also works:

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void additionTest() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

As does this one:

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

This lets you follow the naming convention that best fits your application. For instance, some samples I've seen have adopted a convention where the test class uses the same names for its test methods as the class being tested does. For example, `List.contains()` is tested by `ListTest.contains()`; `List.addAll()` is tested by `ListTest.addAll()`; and so forth.

The `TestCase` class still works, but you're no longer required to extend it. As long as you annotate test methods with `@Test`, you can put your test methods in any class at all. However, you'll need to import the `junit.Assert` class to access the various assert methods, as shown here:

```
import org.junit.Assert;

public class AdditionTest {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    Assert.assertEquals(2, z);
  }

}
```

You also can use the new static import feature in JDK 5 to make this just as simple as the old version:

```
import static org.junit.Assert.assertEquals;

public class AdditionTest {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

This approach makes testing protected methods much easier because the test case class can now extend the class that contains the protected methods.

SetUp and TearDown

JUnit 3 test runners automatically invoke the `setUp()` method before running each test. This method typically initializes fields, turns on logging, resets environment variables, and so forth. For example, here's the `setUp()` method from XOM's `XSLTransformTest`:

```
protected void setUp() {

    System.setErr(new PrintStream(new ByteArrayOutputStream()));

    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");

}
```

In JUnit 4, you can still initialize fields and configure the environment before each test method is run. However, the method that does it no longer needs to be called `setUp()`. It just needs to be denoted with the `@Before` annotation, as shown here:

```
@Before protected void initialize() {

    System.setErr(new PrintStream(new ByteArrayOutputStream()));

    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");

}
```

You can even have multiple methods noted `@Before`, each of which is run before each test:

```
@Before protected void findTestDataDirectory() {
    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");
}

 @Before protected void redirectStderr() {
    System.setErr(new PrintStream(new ByteArrayOutputStream()));
}
```

Cleanup is similar. In JUnit 3, you use a `tearDown()` method like this one I use in XOM for a test that consumes large quantities of memory:

```
protected void tearDown() {
  doc = null;
  System.gc();
```

```
}
```

With JUnit 4, I can give it a more natural name and annotate it with `@After`:

```
@After protected void disposeDocument() {
  doc = null;
  System.gc();
}
```

As with `@Before`, you can have multiple cleanup methods annotated `@After`, each of which is run after each test.

Finally, you no longer need to explicitly call the initialization and cleanup methods in the superclass. As long as they're not overridden, the test runner will call these for you automatically as necessary. `@Before` methods in superclasses are invoked before the `@Before` methods in subclasses. (This mirrors the order of constructor invocation.) `@After` methods run in reverse: Methods in subclasses are invoked before methods in superclasses. Otherwise, the relative order of multiple `@Before` or `@After` methods is not guaranteed.

Suite-wide initialization

JUnit 4 also introduces a new feature that really has no equivalent in JUnit 3: class-scoped `setUp()` and `tearDown()` methods. Any method annotated `@BeforeClass` will run exactly once before the test methods in that class run, and any method annotated with `@AfterClass` will run exactly once after all the tests in the class have been run.

For instance, suppose each test in the class uses a database connection, a network connection, a very large data structure, or some other resource that's expensive to initialize or dispose of. Rather than re-creating it before each and every test, you can create it once and tear it down once. This approach will make some test cases run a lot faster. As an example, when I test error-handling in code that calls into third-party libraries, I often like to redirect `System.err` before the tests begin so the output is not cluttered with expected error messages. Then I restore it after the tests end like so:

```
// This class tests a lot of error conditions, which
// Xalan annoyingly logs to System.err. This hides System.err
// before each test and restores it after each test.
private PrintStream systemErr;

@BeforeClass protected void redirectStderr() {
    systemErr = System.err; // Hold on to the original value
    System.setErr(new PrintStream(new ByteArrayOutputStream()));
}

@AfterClass protected void tearDown() {
    // restore the original value
    System.setErr(systemErr);
}
```

There's no need to do this before and after each and every test. Do be careful with this feature, however. It has the potential to violate the independence of the tests and introduce unexpected coupling. If one test somehow changes an object that is initialized by `@BeforeClass`, it may affect the outcome of other tests. It can introduce order dependence into the test suite and hide bugs. As with any optimization, only implement this after profiling and benchmarking have proven you have a real problem. That being said, I have seen more than one test suite that takes so long to run that it's not run as often as it needs to be, especially tests that need to make many network and database connections. (The LimeWire test suite takes over two hours to run, for example.) Anything you can do to speed up these test suites so programmers run them more often will reduce bugs.

Testing exceptions

Exception testing is one of the biggest improvements in JUnit 4. The old style of exception testing is to wrap a `try` block around the code that throws the exception, then add a `fail()` statement at the end of the `try` block. For example, this method tests that division by zero throws an `ArithmeticException`:

```
public void testDivisionByZero() {

    try {
        int n = 2 / 0;
        fail("Divided by zero!");
    }
    catch (ArithmeticException success) {
        assertNotNull(success.getMessage());
    }

}
```

Not only is this method ugly, but it tends to trip up code-coverage tools because whether the tests pass or fail, some code isn't executed. In JUnit 4, you can now write the code that throws the exception and use an annotation to declare that the exception is expected:

```
@Test(expected=ArithmeticException.class)
  public void divideByZero() {
    int n = 2 / 0;
}
```

If the exception isn't thrown (or a different exception is thrown), the test will fail. However, you'll still need to use the old `try-catch` style if you want to test the exception's detail message or other properties.

Ignored tests

Perhaps you have a test that takes an excessively long time to run. It's not that the test should run faster, just that what it's doing is fundamentally complex or slow. Tests that access remote network servers often fall into this category. If you're not working on things that are likely to break that test, you might want to skip the long-running test method to speed up your compile-test-debug cycle. Or perhaps a test is failing for reasons beyond your control. For instance, the W3C XInclude test suite tests for autorecognition of a couple of Unicode encodings Java does not yet support. Rather than being forced to stare at the red bar repeatedly, such tests can be annotated as `@Ignore`, as shown here:

```
// Java doesn't yet support
// the UTF-32BE and UTF32LE encodings
    @Ignore public void testUTF32BE()
      throws ParsingException, IOException, XIncludeException {

        File input = new File(
          "data/xinclude/input/UTF32BE.xml"
        );
        Document doc = builder.build(input);
        Document result = XIncluder.resolve(doc);
        Document expectedResult = builder.build(
          new File(outputDir, "UTF32BE.xml")
        );
        assertEquals(expectedResult, result);

    }
```

The test runner will not run these tests, but it will indicate that the tests were skipped. For instance, when using the text interface, an "I" (for ignore) will be printed instead of the period that's printed for a passing test, or the "E" that's printed for a failing test:

```
$ java -classpath .:junit.jar org.junit.runner.JUnitCore
  nu.xom.tests.XIncludeTest
JUnit version 4.0rc1
.....I..
Time: 1.149

OK (7 tests)
```

Do be careful, however. Presumably there's a reason these tests were written in the first place. If you ignore the tests forever, the code they're expected to test may break and the breakage may not be detected. Ignoring tests is a temporary stopgap--not a real solution to any problem you have.

Timed tests

Testing performance is one of the thorniest areas of unit testing. JUnit 4 doesn't solve this problem completely, but it does help. Tests can be annotated with a timeout parameter. If the test takes longer than the specified number of milliseconds to run, the test fails. For example, this test fails if it takes longer than half a second to find all the elements in a document that was previously set up in a fixture:

```
@Test(timeout=500) public void retrieveAllElementsInDocument() {
    doc.query("//*");
}
```

Besides simple benchmarking, timed tests are also useful for network testing. If a remote host or database a test is trying to connect to is down or slow, you can bypass that test so as not to hold up all the other tests. Good test suites execute quickly enough that programmers can run them after each and every significant change, potentially dozens of times a day. Setting a timeout makes this more feasible. For example, the following test times out if parsing http://www.ibiblio.org/xml takes longer than 2 seconds:

```
@Test(timeout=2000)
  public void remoteBaseRelativeResolutionWithDirectory()
   throws IOException, ParsingException {
      builder.build("http://www.ibiblio.org/xml");
  }
```

New assertions

JUnit 4 adds two `assert()` methods for comparing arrays:

```
public static void assertEquals(Object[] expected, Object[] actual)
public static void assertEquals(String message, Object[] expected,
Object[] actual)
```

These methods compare arrays in the most obvious way: Two arrays are equal if they have the same length and each element is equal to the corresponding element in the other array; otherwise, they're not. The case of one or both arrays being null is also handled.

What's missing

JUnit 4 is a radical new framework, not a an upgraded version of the old framework. JUnit 3 developers are likely to find themselves searching for a few things that just aren't there.

The most obvious omission is a GUI test runner. If you want to see a comforting green bar when your tests pass or an anxiety-inducing red bar when they fail, you'll need an IDE with integrated JUnit support such as Eclipse. Neither the Swing nor the AWT test runners will be updated or bundled with JUnit 4.

The next surprise is that there's no longer any distinction between failures (anticipated errors checked for by assert methods) and errors (unanticipated errors indicated by exceptions). While JUnit 3 test runners can still distinguish these cases, JUnit 4 runners will not be able to.

Finally, JUnit 4 has no `suite()` methods that build a test suite out of multiple test classes. Instead, variable-length argument lists are used to allow an unspecified number of tests to be passed to the test runner.

I'm not too happy about the elimination of the GUI test runners, but the other changes seem likely to increase JUnit's simplicity. Just consider how much of the documentation and FAQ are currently dedicated to explaining these points, and then consider that with JUnit 4 you never need to explain any of this again.

Building and running JUnit 4

Currently, there are no binary releases of JUnit 4. If you want to experiment with the new version, you'll need to check it out of the CVS repository on SourceForge. The branch is "Version4" (see Resources). Be warned that much of the documentation has not been updated and still refers to the old 3.x way of doing things. Java 5 is required to compile JUnit 4 as it makes heavy use of annotations, generics, and other Java 5 language level features.

The syntax for running tests from the command line has changed a little since JUnit 3. You now use the `org.junit.runner.JUnitCore` class:

```
$ java -classpath .:junit.jar org.junit.runner.JUnitCore
  TestA TestB TestC...
JUnit version 4.0rc1

Time: 0.003

OK (0 tests)
```

Compatibility

Beck and Gamma have striven to maintain both forwards and backwards compatibility. The JUnit 4 test runners can run JUnit 3 tests without any changes. Just pass the fully qualified class name of each test you want to run to the test runner, just as you would for a JUnit 4 test. The runner is smart enough to figure out which test class depends on which version of JUnit and invoke it appropriately.

Backwards compatibility is trickier, but it is possible to run JUnit 4 tests in JUnit 3 test runners. This is important so that tools with integrated JUnit support such as Eclipse can handle JUnit 4 without an update. To enable your JUnit 4 tests to run in JUnit 3 environments, wrap them in a `JUnit4TestAdapter`. Adding the following method to your JUnit 4 test class should suffice:

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

Java-wise, however, JUnit 4 is not in the least bit backwards-compatible. JUnit 4 absolutely depends on Java 5 features. It will not compile or run with Java 1.4 or earlier.

Still to come ...

JUnit 4 is not finished yet. A number of important pieces are missing, including most of the documentation. I don't recommend converting your test suites to annotations and JUnit 4 just yet. Nonetheless, development is moving at a rapid pace, and JUnit 4 does look quite promising. While Java 2 programmers will be sticking with JUnit 3.8 for the foreseeable future, those who've made the move to Java 5 should soon consider adapting their test suites to this new framework to match.

Resources

Learn

- *Pragmatic Unit Testing in Java with JUnit* (Andy Hunt and Dave Thomas, Pragmatic Programmers, 2003): An excellent introduction to unit testing.
- *JUnit Recipes: Practical Methods for Programmer Testing* (J. B. Rainsberger, Manning, 2004): One of the most widely cited and referenced books on JUnit.
- TestNG: Cedric Beust's framework pioneered the annotation based testing style now used in JUnit 4.

- "TestNG makes Java unit testing a breeze" (Filippo Diotalevi, developerWorks, January 2005): An introduction to TestNG.
- "Incremental development with Ant and JUnit" (Malcolm Davis, developerWorks, November 2000): Explores the benefits of unit testing, in particular using Ant and JUnit, with code samples.
- "Demystifying Extreme Programming: Test-driven programming" (Roy Miller, developerWorks, April 2003): Find out how test-driven programming can revolutionize your productivity and quality as a programmer, and learn the mechanics of writing tests.
- "Keeping critters out of your code" (David Carew, et. al., developerWorks, June 2003): Learn how you can use JUnit in conjunction with WebSphere Application Developer.
- "Measure test coverage with Cobertura" (Elliotte Rusty Harold, developerWorks, May 2005): Learn to identify untested code and locate bugs with this handy open source tool.

Get products and technologies

- JUnit 4: Download the newest version of JUnit the SourceForge CVS repository; be sure to use the branch tag "Version4."

Discuss

- developerWorks blogs: Get involved in the developerWorks community.

About the author

Elliotte Rusty Harold is originally from New Orleans, to which he returns periodically in search of a decent bowl of gumbo. However, he resides in the Prospect Heights neighborhood of Brooklyn with his wife Beth and cats Charm (named after the quark) and Marjorie (named after his mother-in-law). He's an adjunct professor of computer science at Polytechnic University, where he teaches Java technology and object-oriented programming. His Cafe au Lait Web site has become one of the most popular independent Java sites on the Internet, and his spin-off site, Cafe con Leche, has become one of the most popular XML sites. His books include *Effective XML*, *Processing XML with Java*, *Java Network Programming*, and *The XML 1.1 Bible*. He's currently working on the XOM API for processing XML, the Jaxen XPath engine, and the Jester test coverage tool.