JUnit 4.x Howto

Blaine Simpson

JUnit 4.x Howto

Blaine Simpson

$Revision: 2049 $

Copyright © 2008, 2009 Axis Data Management Corp. [http://admc.com/]

This document may be freely served, duplicated, or distributed, as long as the contents are not modified. Usage of described software is covered by the licenses of the individual products.

Table of Contents

Preface vii
    Available formats for this document vii
    License vii
    Purpose viii
    Coverage viii
    Support viii
1. Introduction 1
    What is JUnit 1
    Obtaining and Installing JUnit 1
    Test Coverage -- quantity of test cases 1
2. Test Expressions 3
3. Class junit.framework.Assert 4
4. Test Methods 5
    Handling Throwables in your Test Methods 5
    JUnit Failures vs. Errors 6
5. Test Classes and the JUnit Test Execution Environment 8
6. Specifying which Tests to Execute 10
7. Using Test Runners 11
8. Using JUnit with Ant 12
9. Using JUnit with IDEs 14

List of Tables

1. Available formats of this document vii

List of Examples

4.1. Implementation of a JUnit Test Method 5
4.2. Generically treating generated Exceptions as test Failures 6
5.1. Example JUnit test class 8
7.1. Executing tests from the command line 11
8.1. Ant build file snippet 12

Preface

Note

In January of 2009. this document was forked off of the 3.x HOWTO, including a rewrite of 90% of the content. JUnit 4.x is clearly the way of the future. If I was new to JUnit, I wouldn't waste my time learning 3.x. Consequently, this Guide has been greatly simplified by omitting the many complex configurations supported only with JUnit 3.x.

If you notice any mistakes in this document, please email the author at blaine dot simpson at admc dot com [mailto:blaine dot simpson at admc dot com?subject=junit4x Howto] so that he can correct them. (We require you to convert the "and"s and "dot"s with the corresponding punctuation, in order to thwart spammers). See the Support section below about any other issues.

Available formats for this document

This document is available in several formats.

Note

Standalone PDF readers (those that do not display inside of a web browser frame) can't resolve relative URLs. Therefore, if you are viewing this document with a standalone PDF reader, links to external files distributed with this document will not work.

You may be reading this document right now at http://pub.admc.com/howtos, or in a distribution somewhere else. I hereby call the document distribution from which you are reading this, your *current distro*.

http://pub.admc.com/howtos hosts the latest production versions of all available formats. If you want a different format of the same *version* of the document you are reading now, then you should try your current distro. If you want the latest production version, you should try http://pub.admc.com/howtos.

Sometimes, distributions other than http://pub.admc.com/howtos do not host all available formats. So, if you can't access the format that you want in your current distro, you have no choice but to use the newest production version at http://pub.admc.com/howtos.

Table 1. Available formats of this document

format	your distro	at http://pub.admc.com/howtos
Chunked HTML	index.html	http://pub.admc.com/howtos/junit4x/
All-in-one HTML	junit4x.html	http://pub.admc.com/howtos/junit4x/junit4x.html
PDF	junit4x.pdf	http://pub.admc.com/howtos/junit4x/junit4x.pdf

If you are reading this document now with a standalone PDF reader, the your distro links may not work.

License

This HOWTO document is copyrighted by Axis Data Management Corp. [http://admc.com/] and may not be modified.

Purpose

I am writing this HOWTO because, in my opinion, JUnit is very simple and easy to learn, but there is no tutorial or HOWTO out there which is

- concise
- explains JUnit architecture accurately enough so that readers can figure out how to use JUnit in new situations
- accurate

(Due to both design improvements as well as decisions not to support several features with JUnit versions 4.x, the second item above is now largely obsolete).

The tutorial that I used to learn JUnit spent about 80% of my time learning the program that was to be tested as an example, and the resultant tutorial therefore took five times longer to digest than it should have. There is no need to learn to understand somebody else's data structures just to learn JUnit. If you have an hour to learn to use a terrific product, then proceed.

Coverage

Most importantly **this document covers only versions 4.x of JUnit**. There are business and technical situations which require using JUnit 3.x, most importantly when you must support Java 1.3. For those situations, you should see our JUnit 3.x Howto [http://pub.admc.com/howtos/junit3x/]. I have purposefully dropped coverage of all 3.x-specific features from the Howto you are reading now, in order to make it more accessible (and less scary) to the mainstream users who wish to use the current version of JUnit. A few aspects I cover may be sub-version-specific, and in that case, what I say applies to JUnit version 4.5.

Feature-wise, I cover all of the procedures and features necessary for meaningful unit testing. In addition, I cover how to execute tests from the command-line, from Ant, and from IDEs.

I only cover how to execute tests graphically from IDE's, since JUnit 4.x dropped direct support for graphical test runners. It is not difficult to use the 3.x graphical test-runners in a JUnit 4.x environment, but I'm not going to spend time trying to support environments which the JUnit dev team is aborting. I am not covering test suites for the same reason-- that they are not directly supported by JUnit 4.x. I would very much like to cover the very useful features *Categories* and *Assumptions*, however, these features are still experimental. Out of concern for your time and my time, I don't want to waste time explaining features which are likely to change drastically. (Been burned by that one-too-many times).

I'm not covering the new `assertThat` syntax, because I think it is less intuitive than the traditional JUnit assertion syntax (and consider the assertion that it is *More readable* to be arguable). The JUnit team promises to support the traditional syntax indefinitely. I am not covering *Theories* because the field of application is rather limited, and the design is invasive and heavy-handed. It's hard enough to get unit testing adopted by a development team without throwing in high-maintenance components.

Support

Use the designated topic at the support forum at http:/admc.com/jforum/ with questions, suggestions, etc., about this document or its subject. Axis Data Management Corp. [http://admc.com/] provides professional development, support, and custom education for JUnit implementations, as well as for many other subjects in the realms of computer systems and software development.

Chapter 1. Introduction

What is JUnit

JUnit is a program used to perform *unit testing* of virtually any software. JUnit testing is accomplished by writing test cases using Java, compiling these test cases and running the resultant classes with a JUnit Test Runner.

I will explain a teensy bit about software and unit testing in general. What and how much to test depends on how important it is to know that the tested software works right, in relation to how much time you are willing to dedicate to testing. Since you are reading this, then for some reason you have decided to dedicate at least some time to unit testing.

As an example, I'm currently developing an SMTP server. An SMTP server needs to handle email addresses of the format specified in RFC documents. To be confident that my SMTP server complies with the RFC, I need to write a test case for each allowable email address type specified in the RFC. If the RFC says that the character "#" is prohibited, then I should write a test case that sends an address with "#" and verifies that the SMTP server rejects it. You don't need to have a formal specification document to work from. If I wasn't aiming for *compliance*, I could just decide that it's good enough if my server accepts email addresses consisting only of letters, numbers and "@", without caring whether other addresses succeed or fail.

Another important point to understand is that, everything is better if you make your tests before you implement the features. For my SMTP server, I did some prototyping first to make sure that my design will work, but I did not attempt to satisfy my requirements with that prototype. I am now writing up my test cases and running them against existing SMTP server implementations (like Sendmail). When I get to the implementation stage, my work will be extremely well defined. Once my implementation satisfies all of the unit tests, just like Sendmail does, then my implementation will be completed. At that point, I'll start working on new requirements to surpass the abilities of Sendmail and will work up those tests as much as possible before starting the second implementation iteration. (If you don't have an available test target, then there sometimes comes a point where the work involved in making a dummy test target isn't worth the trouble... so you just develop the test cases as far as you can until your application is ready).

Everything I've said so far has to do with unit testing, not JUnit specifically. Now on to JUnit...

Obtaining and Installing JUnit

There is very little to say about installing JUnit. All you really need from the distribution is the jar file named like `junit-4.5.jar`. There's a good chance that you already have this jar bundled with an IDE or a Java project that you work with. A typical user will not need any of the other jars (which include source code, or lack a third-party library which you probably do need). The only reason to permanently install the distribution is if you want a local copy of the documentation which is readily available online at http://junit.sourceforge.net/#Documentation .

Note

The README.html in the 4.5 distribution incorrectly states the version (over and over) as 4.6. That's a rather unfortunate mistake, seeing as this file is the main documentation facade for the distribution documentation.

Test Coverage -- quantity of test cases

With respect to test coverage, an ideal test would have test cases for every conceivable variation type of input, *not every possible instance of input*. You should have test cases for combinations or permutations of all variation types of the implemented operations. You use your knowledge of the method implementation (including possible implementation changes which you want to allow for) to narrow the quantity of test cases way down.

I'll discuss testing the multiplication method of a calculator class as an example. First off, if your multiplcation method hands its parameters (the multiplication operands) exactly the same regardless of order (now and in the future), then you have just dramatically cut your work from covering all permutations to covering all combinations.

You will need to cover behavior of multiplying by zero, but it will take only a few test cases for this. It is extremely unlikely that the multiplication method code does anything differently for an operand value of 2 different than it would for an operand value of 3, therefore, there is no reason to have a test for "2 x 0" and for "3 x 0". All you need is one test with a positive integer operand and zero, like "46213 x 0". (Two test cases if your implementation is dependent upon input parameter sequence). You may or may not need additional cases to test other *types* of non-zero operands, and for the case of "0 x 0". Whether a test case is needed just depends upon whether your current and future code could ever behave differently for that case.

Chapter 2. Test Expressions

The most important thing is to write tests like

```
    (expectedResult == obtainedResult)
```

or

```
    expectedResult.equals(obtainedResult)
```

You can do that without JUnit, of course. For the purpose of this document, I will call the smallest unit of testing, like the expression above, a *test expression* . All that JUnit does is give you flexibility in grouping your test expressions and convenient ways to capture and/or summarize test failures.

Chapter 3. Class junit.framework.Assert

JUnit-specific Test Expressions

The Assert class has a bunch of static convenience methods to help you with individual test expressions. For example, without JUnit I might code

```
    if (obtainedResult == null || !expectedResult.equals(obtainedResult))
        throw new MyTestThrowable("Bad output for # attempt");
```

With JUnit, you can code

```
import static org.junit.Assert.*;
...
    assertEquals("Bad output for # attempt", expectedResults, obtainedResults);
```

The static `import` statement is not required. It just allows you to code more concisely with `assert...` instead of `Assert.assert...`. Every JUnit-supplied assert method throws a `java.lang.AssertionError`s when the test fails.

There are lots of assert* methods to take care of the several common Java types. They take care of checking for nulls. Both `assert()` failures and `failure()`s (which are effectively the same thing) throw `java.lang.AssertionError Throwable`s, which the test-runners and listeners know what to do with. If a test expression fails, the containing test method quits (by virtue of throwing the `AssertionError` internally, then the test-runner performs cleanup executes the next test method in the sequence starts (with the error being noted for reporting now or later). Take a look at the API spec for `junit.framework.Assert [http://junit.sourceforge.net/javadoc_40/org/junit/Assert.html]` .

If `Assert` doesn't supply a specific expression test which you need, then code up your own expression and use `Assert.assertTrue()` or `Assert.assertFalse()` on the boolean result of your expression. Or you can just call one of the `Assert.fail()` methods once you determine a failure case is at hand. If you want to generalize custom failure recognition, you can make your own utility methods or catch blocks which generate and throw a new `AssertionError`. In this case, you will often want to run `initCause()` on the `AssertionError` instance before throwing it, to indicate the original source location of the failure.

Chapter 4. Test Methods

Composition and Execution of Test Methods

The smallest groupings of test expressions are the methods that you put them in. Whether you use JUnit or not, you need to put your test expressions into Java methods, so you might as well group the expressions, according to any criteria you want, into methods. If you designate such a test method with the `"@org.junit.Test"` annotation, then JUnit will execute it as explained in the Test Classes and the JUnit Test Execution Environment chapter (and that is the ultimate purpose of doing so). For this document, I call such a method a *test method*.

The general recommendation is to test only one aspect of behavior in a method. (I know how vague this is). If that can be done with a single expression, then your test method should contain just one expression. There are two main reasons for keeping test methods very fine-grained.

First, your report output is only as granular as the test methods. If you have 10 methods each of which tests handling of a different type of input character, your test report will have a count (at least) for the success or failure of each character type. If you put the expressions to test all ten character types into a single test method, you will only get a single test result for all character types. The difference is drastic. If your first test fails, none of the other tests will even execute in the single-method scenario.

Secondly, setup and execution of one test may cause unintended consequences for a test which follows. This risk is usually acceptable for a single-author class, but for jointly developed software, or test classes which will outlast the tenure of the original author, the risk can be considerable.

With JUnit version 4, we implement test methods as follows.

Example 4.1. Implementation of a JUnit Test Method

```
    @org.junit.Test
    public void testDoubler() {
        TargetClass targetObject = new TargetClass();
        targetObject.init();   // Any setup needed to prepare the target object
        assertEquals("Target object failed to double 'xyz'",
                "xyzxyz", targetObject.double("xyz"));
    }
```

There are several important observations to be made here. It must be an instance method of type void with no parameters. Unlike JUnit 3.x, **the name of the method doesn't matter** to the JUnit 4.x test-runners (or to anything else). So, you can implement your own method naming convention. I didn't prefix the method name with "test" to comply with any JUnit rule, but because it makes the methods easy to recognize (both for human readers, and for scripts, Ant build files, etc.). Your test methods are allowed to throw anything by the rules of JUnit, but a good test method won't declare any `Throwables` (more on this topic below).

Test methods are single-threaded, but different test methods must be independent and can't depend on state (including values of instance or static variables) set by another test method. If you are going to make use of instance variables, you must take pains to initialize the variable values for each test method run. The following chapter explains how the test-runners set up and execute your test methods, and how you can write shared setup and shut down code.

Handling Throwables in your Test Methods

Caution

Be careful when catching `Throwables` or `Errors` in your test methods. If you catch `java.lang.AssertionError` and don't rethrow it, you will have intercepted JUnit's test failure indication.

If your test method calls other methods which declare checked exceptions, you should not just declare it in your test method, but should handle it in either normal Java try/catch fashion, or with the JUnit *expected attribute*.

If you have elicited a `Throwable` on purpose, you can catch it to detect that, but if your test method is fine-grained (as it should be), you can accomplish the same thing more gracefully by telling JUnit that your test method should throw the specified `Throwable`.

```
    @org.junit.Test(expected=IOException.class)
    public void myTestMethod() {
```

In this example, we assume the source file has an import statement for `java.io.IOException`.

If, on the other hand, production of the `Throwable` indicates a test failure, then you should catch it from a try block and fail the test with something like `Assert.fail(String)` [http://junit.sourceforge.net/javadoc_40/org/junit/Assert.html#fail(java.lang.String)] .

Here is a code snippet that handles the very common case where the code being tested could throw any of various `Exception`s, and they should all be considered test failures, not an error in the test system. This idiom can save you a lot of work and still create well-behaved tests.

Example 4.2. Generically treating generated Exceptions as test Failures

```
    public void testWithParams() {
        try {
            rb1.validate();
            rb2.validate();
            rb1.setMissingPosValueBehavior(
                    ValidatingResourceBundle.EMPTYSTRING_BEHAVIOR);
            rb2.setMissingPosValueBehavior(
                ValidatingResourceBundle.NOOP_BEHAVIOR);
            assertEquals(SUBSTITUTED_CONN_MSG,
                    rb1.getString(SqltoolRB.JDBC_ESTABLISHED, testParams));
            assertEquals(SUBSTITUTED_CONN_MSG,
                    rb2.getString(SqltoolRB.JDBC_ESTABLISHED, testParams));
            rb1.setMissingPosValueBehavior(
                 RefCapablePropertyResourceBundle.THROW_BEHAVIOR);
            assertEquals(SUBSTITUTED_CONN_MSG,
                    rb1.getString(SqltoolRB.JDBC_ESTABLISHED, testParams));
        } catch (Exception e) {
            AssertionError ae = new AssertionError(
                "RB system failed to generate end-user message");
            ae.initCause(e);
            throw ae;
        }
  }
```

Notice that I made my own `AssertionErorr` instead of just executing `fail()`. This is so that the JUnit failure report will identify the source of the original problem instead of the location of the `fail()` call, whether the original `Throwable` is generated by your test code (like a NPE) or fifty levels deep in the call stack.

You can easily insert additional catch statements for specialized reporting of especially-expected exception types; and you can also broaden it for code which is known to throw non-Exception `Throwable`s. In that case, you must specifically catch and re-throw `AssertionError`s, or you will incapacitate JUnit.

JUnit Failures vs. Errors

JUnit test reports differentiate failures vs. errors. A failure is a test which your code has explicitly failed by using the mechanisms for that purpose (as described above). Generation of a *failure* indicates that your time investment is paying off-- it points you right to an anticipated problem in the program that is the target of your testing. A JUnit *error*, on the other hand, indicates an unanticipated problem. It is either a resource problem such as is normally the domain of Java unchecked throwables; or it is a problem with your implementation of the test. When you run a test and get an error, it means you really need to fix something, and usually not just the program that you intended to test. Either a problem has occurred outside of your test method (like in test class instantiation, or the test setup methods described in the next chapter), or your test method has thrown an exception (as described above, your methods should only throw unchecked exceptions if you have used the `expected` mechanism described above).

(With version 3, there used to be lifecycle side-effects when Errors were produced. I believe that these problems have been fixed with version 4).

Chapter 5. Test Classes and the JUnit Test Execution Environment

JUnit sets up the execution environment for tests at the test class file level. I hereby define a *Test Class* to be any class that contains *test methods* (test methods were defined in the previous chapter). Test Classes in version JUnit version 4 are a great improvement over version 3.x test classes, which had to subclass a JUnit-provided class. This makes the framework much less invasive and lets you leverage inheritance and other object-oriented features if and where that makes sense.

Example 5.1. Example JUnit test class

```
import java.sql.Connection;
public class HackerTest extends AbstractClientTest {
    private Connection mkConnection() {  // A utility method
        ...

    @org.junit.Test
    void testPasswordGuessing() {
        Connection conn1 = mkConnection();
        ...
        assertEquals(...
    }

    @org.junit.Test(expected=InterruptedException)
    void testTimeout() {
        ...

    @org.junit.After
    protected void deallocateCacheSpace() {
        ....

    @org.junit.After
    protected void deallocateNameTokens() {
        ...
```

Your test class may extend some other class. As explained earlier, you may name your test methods anything you wish to. Our sample test class contains a normal instance method named `mkConnection`. This method must be careful not to depend on the state of instance variables between test runs, the same as the test methods themselves. The test methods must be independent of one another. With JUnit version 4, it isn't even possible to specify the sequence in which your test methods will be executed.

Instance objects which are set up for each test method execution (by @Before methods, and perhaps cleaned up with @After methods) are called *Fixtures*.

The class contains two test methods, `testPasswordGuessing` and `testTimeout`. It also contains two JUnit lifecycle methods, `deallocateCacheSpace` and `deallocateNameTokens`. The `@Before [http://junit.sourceforge.net/javadoc_40/org/junit/Before.html]` and `@After [http://junit.sourceforge.net/javadoc_40/org/junit/After.html]` notations specify methods which will be executed before and after each of your test methods. So, in this case, the method execution sequence would be

- `testPasswordGuessing`
- `deallocateCacheSpace`
- `deallocateNameTokens`
- `testTimeout`

- `deallocateCacheSpace`
- `deallocateNameTokens`

except that the sequence of the first three and last three could be exchanged (since JUnit doesn't guarantee the sequence in which test methods are selected), and the sequence of `deallocateCacheSpace` and `deallocateNameTokens` could be exchanged (since JUnit doesn't guarantee the sequence of multiple methods which have the same lifecycle attribute).

JUnit catches all `AssertionError`s thrown from test methods, and that is what makes test "failures". If any fixture or test method throws anything else, JUnit will catch it, and that is what makes test "errors". Errors indicate that there is something wrong with your test setup or environment, not with the intended subject of the unit test.

The use of attributes is a great improvement over JUnit verision 3.x, where the single allowed setup and single allowed cleanup methods were pre-defined.

There are also @BeforeClass and @AfterClass lifecycle attributes. `@BeforeClass [http://junit.sourceforge.net/javadoc_40/org/junit/BeforeClass.html]` doesn't do anything that you couldn't do just as well in the class constructor. but `@AfterClass [http://junit.sourceforge.net/javadoc_40/org/junit/AfterClass.html]` methods can be useful for performing final cleanup after all test methods (and @Afters) in a single test class have completed.

Chapter 6. Specifying which Tests to Execute

This topic used to occupy most of this HOWTO guide. For good or for ill, JUnit version 4 hasn't obligated to a way to declaratively or flexibly organize and nest test methods. There is very little here for you to learn. The still evolving JUnit feature of *Catalogs* should some day take care of the use case previously served by *Test Suites*.

You just tell your test runner what Test Classes to execute. For each specified Test Class, an instance of the Test Class is constructed; @BeforeClass methods are executed (if any); then the cycle of @Before-methods + @Test-method + @After-methods gets executed for each test method in the class; and finally all @AfterClass methods execute (if any). You control the test methods which get executed in the class simply by adding or removing @Test attributes to/from methods.

You can also temporarily disable a test method by prevising @Test with "@Ignore". This is a very useful practice, because it produces a message upon test execution so the test method won't be forgotten when it is in disabled state.

Chapter 7. Using Test Runners

Running non-graphical tests from the command-line is as simple as possible. Make sure the `junit-4.5.jar` file, plus your test class, plus all needed resources (including, of course, the test object), are in your CLASSPATH. Then run `org.junit.runner.JUnitCore [http://junit.sourceforge.net/javadoc_40/org/junit/runner/JUnitCore.html]` , giving the absolute class names of all the Test Classes to execute. `JUnitCore` is JUnit version 4's test-runner manager. It examines each specified classes and invokes a test-runner to execute the test mods of the class.

Example 7.1. Executing tests from the command line

```
    export CLASSPATH=classes:/local/libjunit-4.5.jar
            java org.junit.runner.JUnitCore com.acme.LoadTester com.acme.PushTester
```

If you want your test class to be self-executable, you can write up a main method in your test class like this

```
public static void main(String args[]) {
      org.junit.runner.JUnitCore.main(LoadTester.class.getName());
    }
```

where your test class file is `LoadTester.java`.

As noted above, `JUnitCore` inspects the specified test classes and instructs test runner classes to execute the test methods. If you use JUnit version 4.5 and follow the methods explained in this guide, JUnitCore will always use a `BlockJUnit4ClassRunner`. If JUnitCore detects test classes or suites coded for JUnit version 3, it will select a JUnit-3-compatible test runner. There are also ways to specify the test runner implementation to use for a class (in order to use the graphical version 3 test-runner, for example), but since I am covering only version of JUnit, I am wrapping up the discussion here.

Chapter 8. Using JUnit with Ant

JUnit works great from Ant, as long as you don't expect graphical output in real-time. The Ant <junit> [http://ant.apache.org/manual/OptionalTasks/junit.html] task is an *Optional* task, not a *Core* task, but with Ant 1.7, this just means you need to provide the `ant-4.5.jar` file. Ant 1.7 comes with wiring for the <junit> task out-of-the-box. Just specify the path to your `ant-4.5.jar` in your <junit> task. (There are several other ways to get it on the classpath, but this method is the last invasive, and least likely to have side-effects on your other applications).

You can download or view this complete Ant build file [resources/build.xml]. The part that runs JUnit tests is captured below.

Example 8.1. Ant build file snippet

```
<target name="unittest" description="Execute unit tests"
        depends="compile-tests">
  <ivy:cachepath conf="test" pathid="test.refid"/>
  <mkdir dir="tmp/rawtestoutput"/>
  <junit printsummary="true" failureproperty="junit.failure">
    <!-- N.b. use failureproperty instead of haltonfailure, because if we use
         the former, we will get no detailed report about the failure.
         If the test fails, the fail element below will still assure that
         the Ant run will exit with error status.
    -->
    <classpath refid="test.refid"/>
    <classpath path="classes:test-classes"/>
    <!-- Ant provides several ways to set the classpath.  The critical thing
         is just that the final classpath for the junit task must include
         the junit-4.x jar file, the test classes, and all classes referred
         to directly or indirectly by your test classes.  -->

    <batchtest todir="tmp/rawtestoutput">
      <fileset dir="test-classes"/>
      <formatter type="xml"/>
    </batchtest>
    <!-- In the unlikely case that you just have a single test class,
         use a test element like this instead of the batchtest element
         above:   <test name='com.admc.jamama.smtp.SMTPTest'/>
         You can nest the formatter inside it, just like batchtest.
    -->

    <!-- You can use sysproperty elements to pass configuration settings
         to your test classes, or to appplication classes they will run:
   <sysproperty key="targetserver.test" value="mercedes"/>
   -->
  </junit>
  <junitreport todir="tmp">
    <fileset dir="tmp/rawtestoutput"/>
    <report todir="test-reports"/>
  </junitreport>
  <fail if="junit.failure" message="Unit test(s) failed.  See reports!"/>:w
</target>
```

This is a powerful snippet. I've used variations of this for very large, distributed Java projects. Here are the important things to notice.

- The printsummary attr. causes the junit task to produce useful feed back to the person running the test, in real time.
- The combination of the failureproperty attr + the <fail> element at the end ensures that Ant will exit with failure status if a test fails. Even a sleepy builder will notice that there was a test failure.

- As alluded to previously, there are many ways to set the global Ant classpath, but it's cleaner to add the junit jar file only to the classpath for your junit task. Use <junit>'s `classpath` or `classpathref` attributes or nested elements for this purpose. The sample describes what the resultant classpath must contain.
- For most systems complex enough to warrant unit testing, you will want to use a `batchtest` to flexibly select the test classes to run test-runners on. If you are just playing or something, you may want to use the `test` element instead, as described in the comment.
- The combination of the formatter plus the following batchtest task generates the pretty HTML reports.

How things work to generate HTML reports is.

1. The xml formatter element nested in the junit element produces raw data XML files.
2. The following junitreport task generates a summary XML file named TESTS-TestSuites.xml
3. The report element nested in the junitreport taskgenerates the final HTML and css files

Sometimes you just want to see the report results on your screen. Maybe you're repeatedly running unit tests with Ant in one window, while you have your editor up, adjusting code to satisfy the tests. In that case, take out the junitreport element, and use just

```
    <junit printsummary="true">
```

or, for more detail to the screen, use a formatter but set `usefile="false"`

You usually do not want to set attribute `showoutput="true"`. This sends the stderr and stdout of your test class and your target class (or whatever your test classes run) to the screen. If your classes do produce any output, it usually makes a big mess when mashed together with the JUnit output.

Chapter 9. Using JUnit with IDEs

Your IDE should have built-in support for JUnit version 4. You may have to specify to user version 4 in global settings, project-specific settings, or on the screens for performing the individual JUnit tasks.

There will be an operation to create a new JUnit test class. If you specify the target class to be tested by your new test class (often defaults to the class you currently have selected), you can then select methods in the target class *to be tested*, and the IDE will make template test methods intended to test each one. In most cases, you will want more than one test method to test a single method of the target class. You can, of course, make as many additional test methods as you want to, manually.

To execute the test-runner, you specify the target test class or test classes, using any of various methods. With Eclipse, you do this under the Run / Run Configurations... menu.