JUnit 3.x Howto

Blaine Simpson

JUnit 3.x Howto

Blaine Simpson

$Revision: 2422 $

Table of Contents

List of Tables

List of Examples

Preface

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=junit3x 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/junit3x/
All-in-one HTML	junit3x.html	http://pub.admc.com/howtos/junit3x/junit3x.html
PDF	junit3x.pdf	http://pub.admc.com/howtos/junit3x/junit3x.pdf

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

License

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

The second item above is especially important with JUnit versions 3.x (as opposed to 4.x, where its far more easy to apply JUnit to new situations without understanding the architecture of JUnit itself.

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 a couple hours to learn to use a terrific product, then proceed.

Coverage

Most importantly **this document covers only versions 3.x of JUnit**. If you do not need to support JUnit 3.x (for example, if you must run with Java version 1.3 or earlier), then I really recommend you save yourself a lot of trouble and just learn JUnit 4.x, the JUnit of the future. See our JUnit 4.x Howto [http://pub.admc.com/howtos/junit3x/] to learn JUnit 4.x.

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 (graphical and non-graphical), and from Ant. Graphical test-runners (without need for an IDE) is one of the features that the 3.x series of JUnit provides and the 4.x series does not.

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.jar` from a 3.x distribution of JUnit. You may already have this jar bundled with an IDE or a Java project that you work with-- just make sure that it is a 3.x series jar, not a 4.x series. A typical user will not need any of the other jars (which include source code, or do not contain the entire JUnit product). You may want to install the distribution just to have a local copy of the documentation, since much of the documentation you find online will be for the 4.x series (and this trend will only increase in the future).

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

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 MyTestException("Bad output for # attempt");
```

With JUnit, you can code

```
Assert.assertEquals("Bad output for # attempt",
        expectedResults, obtainedResults);
```

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 `junit.framework.AssertionFailedError` `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 `AssertionFailedError` internally, then the test-runner performs cleanup before executing the next test method in the sequence (with the error being noted for reporting now or later). Take a look at the API spec for `junit.framework.Assert`.

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 `AssertionFailedError`. 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. Grouping test expressions into 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 specify such a method (perhaps inplicitly by nameing the method `test*`), JUnit will execute it as explained in the Abstract class junit.framework.TestCase chapter For the purpose of this document, I will such a method a "test method".

If you have 20 different test expressions to run, and five need to run over a single URLConnection, then it would make sense to put those into one method where you create the URLConnection, run the five test expressions, then clean up the URLConnection. The other methods may be in other methods in the same class, or in some other class. You have to pay attention to whether the different expressions should be run independently. Often test expressions have to be run sequentially with one set of resources-- that's what I'm talking about here. This all applies whether you are using JUnit or not. The point is, if a set of test expressions needs to be run sequentially using the same resource instance(s), they should be grouped in one method.

Handling Throwables in your Test Methods

Caution

Be careful when catching `Throwables` or `Errors` in your test methods. If you catch `junit.framework.AssertionFailedError` 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 normal Java try/catch fashion.

If 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)`.

Here is a real-life code snippet that handles the very common case where an exception could be thrown from any of many lines of test code. You don't want to have to write a try/catch block around each statement just to be able to identify the point of failure, yet the `Throwables` must be caught or they will be misinterpreted as errors instead of failures (see the next section about that). This idiom can save you a lot of work and still create well-behaved tests.

Example 4.1. Generically treating generated Exceptions as test Failures

```
    public void testDetailedSimpleQueryOutput() {
        try {
            verifySimpleQueryOutput();
        } catch (SQLException se) {
            junit.framework.AssertionFailedError ase
                = new junit.framework.AssertionFailedError(se.getMessage());
            ase.initCause(se);
            throw ase;
        }
    }
```

Notice that I made my own `AssertionFailedErorr` 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 widen the caught `Throwable` type for more general handling, or insert additional catch statements for more specialized reporting of especially-expected exception types. For the former case, you must specifically catch and re-throw `AssertionFailedError`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 following chapters), or your test method has thrown an exception.

Note

With some 3.x versions of JUnit, there are/were lifecycle side-effects when Errors were produced. Unless you know otherwise, don't assume that cleanup methods will run after test failure. (This is not a concern with JUnit 4.x, which is reliable in this respect).

Chapter 5. Interface junit.framework.Test

An object that you can *run* with the JUnit infrastructure is a *Test*. But you can't just implement Test and run that object. You can only *run* specially created instances of TestCase. (*Running* other implementations of Test will compile, but produce runtime errors.)

Chapter 6. Abstract class junit.framework.TestCase

If you have test(s), each grouping of which needs *independent* resource(s) set up the same way (to avoid side-effects or for other reasons), then you can satisfy that need by making a setUp method and a cleanup method. For every grouping needing these resources set up, you run

```
    setUp();
    assert(something);
    assert(somethingelse);
    cleanup();
```

JUnit accomplishes this by having you extend the TestCase class and grouping your *tests* into methods that share those resource(s); and using the methods setUp() and tearDown() for the obvious purposes. All the *tests* in one method share the same setUp-ed resources. The setUp and tearDown method implementations need to match the signatures in the API spec for TestCase. JUnit will catch Throwables from every method it executes on your behalf. Whenever it catches a `junit.framework.AssertionFailedError` from a test method, that causes a test failure. If it catches any other throwable, that causes a test error. This sequence of setUp() + yourTestMethod() + tearDown(), with the described catch-handling, is the exact purpose of Test.run().

The tricky thing to understand is, the only elemental Tests that can actually be run are TestCase implementations that were created with TestCase(String). Since TestCase is an abstract class, you can't just call TestCase(String)-- you have to implement TestCase and make a constructor that does super(String). The object "x" below is a runnable test.

```
    class X extends TestCase {
    ...
        public X(String s) {
            super(s);
        }
```

Then, in any class...

```
        X x = new X("astring");
```

Not very intuitive, but that's how it is. The string argument to TestCase(String) must be a method in the constructing class. For example, if X extends TestCase, then `new X("astring" )` will have a runtime failure unless X has a method named "astring" that takes no arguments. This concept is very weird, but the concept is crucial to be able to figure out the many different ways that one can code JUnit unit tests. Therefore, let's solidify this with an exercise.

I urge you to do this very simple exercise to prove to yourself that TestCase instances created like `new X("astring")` are really runnable Tests.

1. Create a public class X that extends `junit.framework.TestCase`.
2. Make a public constructor for X that takes a String argument, and which just calls the super constructor of the same form.
3. Make an object method named "meth" that takes no arguments and throws Exception. Just have it print out something so that you can see when it is invoked.
4. Add a `public static void main(String[])` method that throws Exception so that you can invoke the program.
5. In your main() method, run `(new X("meth")).run();`

This proves that `new X("meth")` really creates a runnable Test, and that when it runs, the following happens.

```
    anonymous.setUp();
    anonymous.meth();
    anonymous.tearDown();
```

Note that the setUp() and tearDown() methods executed are from the same exact class that your method ("meth" in our example) resides in. You can instantiate the Test and run it from some other class, but the three methods that get run come from the TestSuite-implementating class. An important implication of this is, if you code any setUp or tearDown method in a TestCase class, you should only put test methods into this class which need your setUp/tearDown methods to run before/after it. If you have a method which doesn't need these prep/cleanup actions, then move them into a TestCase that doesn't implement setUp/tearDown.

TestCase itself extends Assert, so your individual `tests` can be coded like `assertEquals(this, that);` , since *this* is now an Assert.

If you understand everything so far, then you understand JUnit much better than most JUnit users do.

You just run

```
    (new myTestCase("myTestMethodName")).run();
```

for every test you want to run. You can have as many TestCase implementation files with as many test methods as you want in each one, in addition to non-test methods. In these classes or in other classes, you can invoke any of these tests. You could have a test manager class with a main(String[]) method that invokes a variety of test methods in any of your TestCase implementation files.

At this point, you can write and execute all the JUnit tests you could ever want to. You probably want to use a JUnit supplied Test Runner though, to collect and display test results as it runs tests. We'll get to that later.

Chapter 7. Class junit.framework.TestSuite

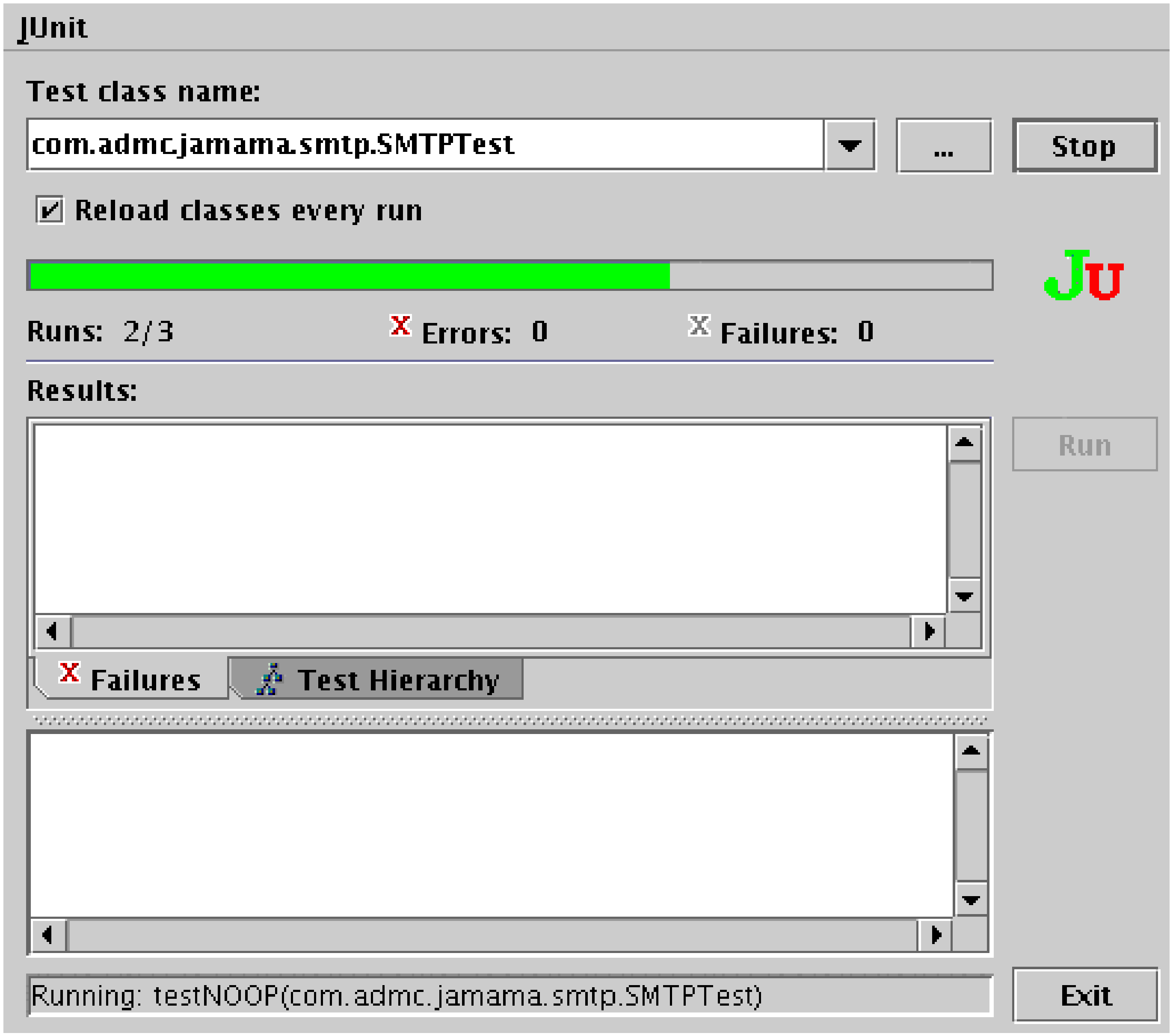
A TestSuite is just an object that contains an ordered list of runnable Test objects. TestSuites also implement Test() and are runnable. To *run* a TestSuite is just to run all of the elemental Tests in the specified order, where by *elemental tests*, I mean Tests for a single method like we used in the Abstract class junit.framework.TestCase chapter. TestSuites can be nested. Remember that for every elemental Test run, three methods are actually invoked, setUp + test method + tearDown.

This is how TestSuites are used.

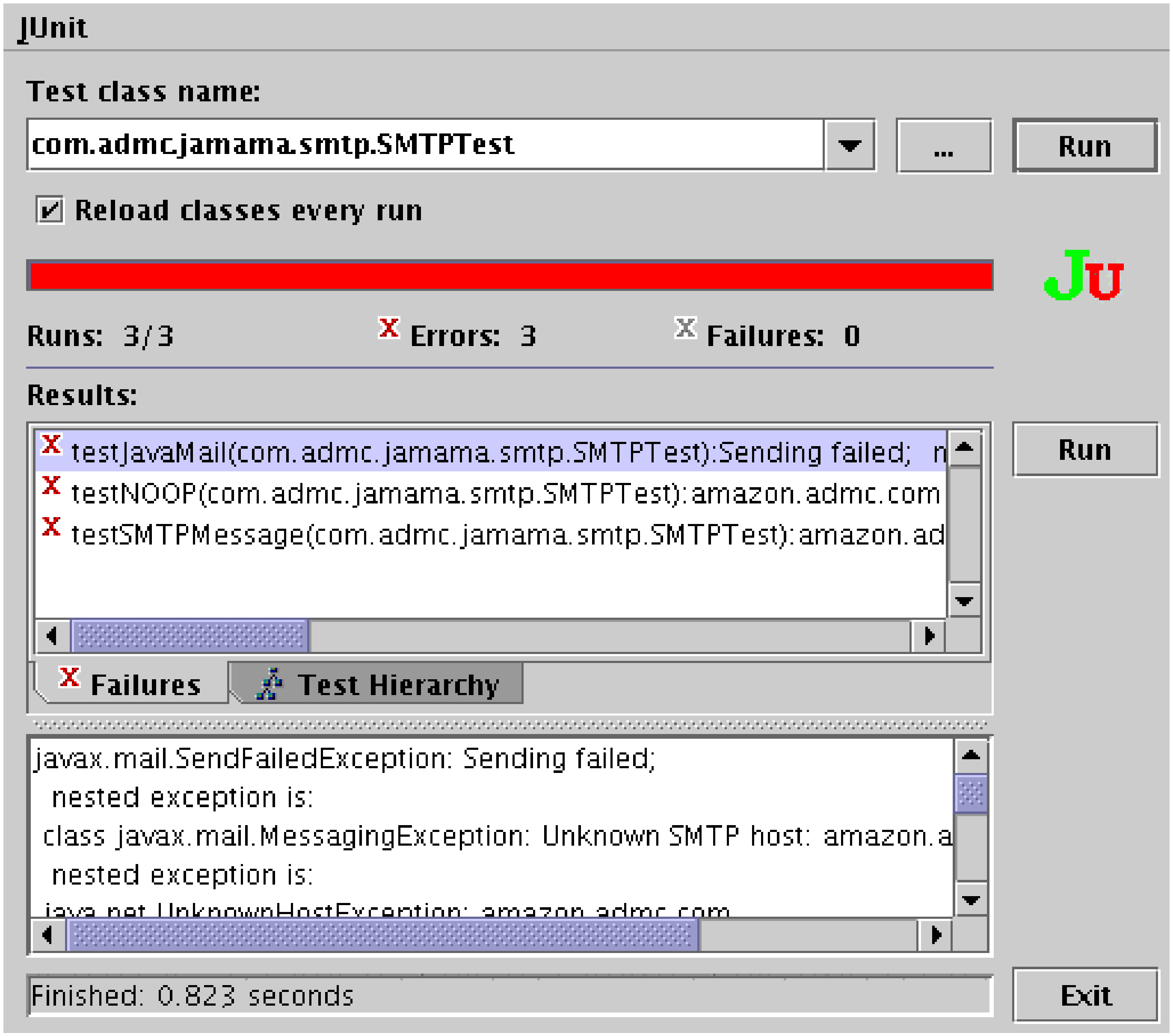
```
TestSuite s = new TestSuite();
s.addTest(new TC("tcm"));
s.addTest(new TD("tdm"));
s.addTestSuite(AnotherTestSuiteImplementation.class);
s.run(new TestResult());
```

Chapter 8. Making your TestCases Test-Runner-Friendly

Test Runners execute Tests-- TestSuites and TestCases. They run Test.run() just like you did in our example, but they also collect, summarize, and format the results nicely. There are also graphical test runners. Test Runner in this sense is not a real class name. By *Test Runner*, I mean a Java class that can run JUnit tests and collect, summarize, and format the results. The JUnit Test Runners that I know extend junit.runner.BaseTestRunner. This is what your tests will look like if you use the JUnit Swing Test Runner.



A successful Swing TestRunner.



```
testJavaMail(com.admc.jamama.smtp.SMTPTest):Sending failed; n
testNOOP(com.admc.jamama.smtp.SMTPTest):amazon.admc.com
testSMTPMessage(com.admc.jamama.smtp.SMTPTest):amazon.ad
```

```
javax.mail.SendFailedException: Sending failed;
  nested exception is:
 class javax.mail.MessagingException: Unknown SMTP host: amazon.a
  nested exception is:
 java.net.UnknownHostException: amazon.admc.com
```

```
Finished: 0.823 seconds
```

A failed Swing TestRunner run.

Your current objective is to learn the best way to tell the Test Runners which TestSuites to run. There are many different methods-- nearly all of these methods are either ugly or inflexible (the former requiring you to implement inner classes with methods inside every single test method, the latter requiring that your test methods follow a naming convention and be run in a specific order with no omissions).

The one method that is extremly simple and so flexible that you will never need another method, is to just create a static method TestSuite.suite() that returns the TestSuite that you want run. You give the name of this TestCase class to a Test Runner and it will do exactly what you want it to do.

If a TestCase class has a *suite()* method, then when you run a Test Runner on that TestCase, it will run

```
    setUp();
    yourMethod();
    tearDown();
```

for each elemental test method in the TestSuite that your *suite()* method returns. Your suite() method creates and populates a new TestSuite object exacly as described in the Class junit.framework.TestSuite chapter, and returns it.

This method is very flexible because you can specify any subset of your methods in any sequence you wish. The methods can even come from different files. (Keep in mind that the setUp() and tearDown() methods for each test method come from the file containing the test method). You can name the methods as you wish, but they can't take any input parameters.

Here's exactly what you need to do to make a TestCase class that Test Runners can work with. You just add a (String) constructor to your TestCase subclass (because you need to call `new YourTestCase(String)` next). Then implement suite() by populating and returning a TestSuite.

```
    public SMTPTest(String s) { super(s); }

    static public junit.framework.Test suite() {
        junit.framework.TestSuite newSuite = new junit.framework.TestSuite();
        newSuite.addTest(new SMTPTest("testJavaMail"));
        newSuite.addTest(new SMTPTest("testNOOP"));
        newSuite.addTest(new SMTPTest("testSMTPMessage"));
        return newSuite;
    };
```

Chapter 9. Using Test Runners

There are other ways to feed TestSuites to Test Runners, but, in my opinion, the strategy of returning a TestSuite via TestCase.suite(), as explained in the Making your TestCases Test-Runner-Friendly chapter is by far the best. Therefore, in this chapter I will explain how to use Test Runners with TestCases/TestSuites implemented in that way.

You can use any of the executable programs that JUnit comes with to select your TestCase class (with an implementation of *suite()*), and it will do exactly what you want.

You need to have the junit jar file and your own classes available in your classpath.

```
# To run your TestCase suite:
java -cp whatever:/path/to/junit.jar junit.textui.TestRunner MyTestClass

# To run your TestCase suite in a Gui:
java -cp whatever:/path/to/junit.jar junit.swingui.TestRunner MyTestClass

# To run the TestRunner Gui and select your suite or test methods
# from within the Gui:
java -cp whatever:/path/to/junit.jar junit.swingui.TestRunner
```

Note

(Note that you can't supply more than one TestCase on the command-line, though after the Swing TestRunner is running, you can load multiple TestCases into it).

It is also very easy to make your TestCase class *execute* by just running it.

```
# This runs the suite:
java -cp whatever:/path/to/junit.jar org.blah.your.TestClass

# This does the same thing with the fancy Gui:
java -cp whatever:/path/to/junit.jar org.blah.your.TestClass -g
```

To have this ability, just copy the following code into any TestCase source file that you have.

```
static public void main(String[] sa) {
    if (sa.length > 0 && sa[0].startsWith("-g")) {
        junit.swingui.TestRunner.run(SMTPTest.class);
    } else {
        junit.textui.TestRunner runner = new junit.textui.TestRunner();
        System.exit(
                runner.run(runner.getTest(SMTPTest.class.getName())).
                        wasSuccessful() ? 0 : 1
        );
    }
}
```

Just replace SMTPTest with the name of your class. (I use the full package paths to the JUnit classes so as to not pollute your namespace and require you to add import statements).

Enjoy.

Chapter 10. Using JUnit with Ant

The Ant <junit> target is non-graphical. (It's packed in the *Optional* Ant jar, not with the core tasks, but that usually doesn't matter since the optional jar file will always be in Ant's classpath unless somebody has monkeyed with the Ant distribution). You can, of course, invoke a graphical Test Runner using the <java> task, but be aware that the Test Runner gui never works from my Ant JVM. I set `fork='true'` and then it works fine. Here are some samples showing how to invoke JUnit tests in various ways. The gui test invokes my TestCase using the main() method that I described above. The non-gui test runs a batch of test suites, but has a commented-out line showing how to invoke a single test suite.

```
<!-- This one runs the designated TestCase suite in a Gui -->
<target name='guitest' depends='-test-shared, compile-tests'
        description='Run Unit Tests in the Junit Swing Gui'>
    <!-- note the fork='true' in the following <java> task -->
    <java classname='com.admc.jamama.smtp.SMTPTest' fork='true'>
        <classpath>
            <pathelement path='${jar.junit}'/>
            <pathelement path='testclasses'/>
            <pathelement path='classes'/>
            <fileset dir='lib'/>
        </classpath>
        <!-- the following stuff required by my specific application -->
        <arg value='-g'/>
        <sysproperty key='test.targetserver' value='${test.targetserver}'/>
        <sysproperty key='test.targetport' value='${test.targetport}'/>
    </java>
</target>

<!-- This one runs all of the designated TestCases non-graphically -->
<target name='test' depends='-test-shared, compile-tests'
        description='Run unit tests in non-graphical mode'>
    <junit printsummary='true' haltonfailure='true'>
        <formatter type='brief' usefile='false'/>
        <classpath>
            <pathelement path='${jar.junit}'/>
            <pathelement path='testclasses'/>
            <pathelement path='classes'/>
            <fileset dir='lib'/>
        </classpath>
        <!-- the following stuff required by my specific application -->
        <sysproperty key='test.targetserver' value='${test.targetserver}'/>
        <sysproperty key='test.targetport' value='${test.targetport}'/>
        <!-- <test name='com.admc.jamama.smtp.SMTPTest'/> Single test -->
        <batchtest>
            <fileset dir='testclasses' includes='**/*Test.class'/>
        </batchtest>
    </junit>
</target>
```

My version always tells me the summary whether I specify printsummary or not (contrary to the API spec). Do try different combinations of formatters and showoutout, because the output from the Ant target is very different from the standalone Test Runners.