6 Running JUnit tests

This chapter covers

- Monitoring JUnit tests as they execute
- Executing individual tests
- Executing tests that need to reload classes
- Ignoring tests
- Using the JUnit-addons test runner

There are a number of ways to execute your tests, including a large variety of test runners, not all of which have the same set of features. JUnit provides three test runners: a text-based one, an AWT-based one, and a Swing-based one. We will describe each in turn. In addition to the ones that JUnit provides, a number of people have built their own test runners that include special features not found in the originals. The JUnit test runners were not built to be easily extended, and so whenever someone has wanted to add features, it has been shown to be easier to build a new test runner from the ground up. A person builds a custom test runner because he is either having a problem running tests in the current project or environment, or because a particular feature is needed that isn't available. You might find yourself in either of these same situations. The recipes in this chapter focus not only on solving such problems with your test runner, but also on finding the special features you might need in a test runner. For basic tutorials on the various test runners, refer to their web sites. Now, let's take a tour through the various test runners.

The basic test runners

JUnit provides one text-based and two graphical test runners. For most purposes, the AWT-based test runner has been entirely superseded by the Swing-based test runner; so we ignore the AWT-based test runner in this discussion, leaving us the text-based test runner and the Swing-based runner.

The text-based runner is implemented by the class `junit.textui.TestRunner`. It reports test progress and test results in text format to the console. In chapter 1 we used the text-based test runner to run JUnit's own tests as a way to verify the JUnit installation. The text-based runner is a candidate to be integrated with an automated build process such as Ant, Cruise Control,or Anthill.

The Swing-based runner is implemented by the class `junit.swingui.TestRunner`. It reports test progress graphically using a progress bar. The progress bar starts out green and turns red only when a test fails or an error occurs. This is the genesis of the slang "green bar" for "the tests pass 100%" and "red bar"for "some test fails." Figure 6.1 shows the Swing-based test runner with both success and failure.

When you launch either the text-based or Swing-based test runner, you pass the fully qualified name (package name and all) of the test suite you would like to execute as a command-line parameter. This can either be a test case class or any class that provides the `suite()` method. As an example, to launch the Swing-based runner in a Windows environment, issue the following command:

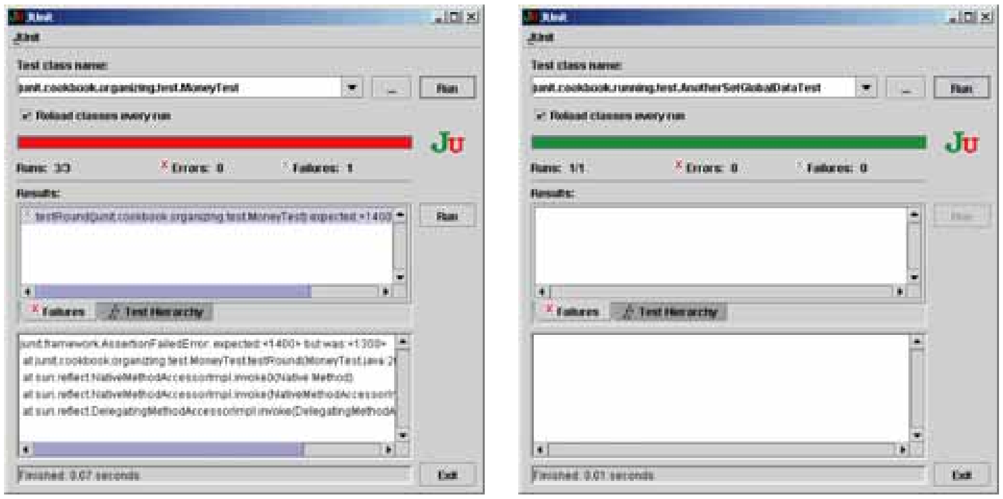
```
java -classpath <your classes>;%JUNIT_HOME%/junit.jar
⇛  junit.swingui.TestRunner <your test suite name>
```

Here we assume that you have an environment variable named `JUNIT_HOME` that points to the directory containing junit.jar. You do not need to use an environment variable—you could just hard code the path to JUnit in your command. If you plan to invoke JUnit test runners from a script you intend to use on multiple machines, we recommend referring to the location of JUnit through an environment variable; otherwise, you would either have to install JUnit to the same location on every machine or change the script on every machine to point to the location where JUnit is installed.

Using Ant

To execute your tests with Ant (http://ant.apache.org), you have two options: use the `<junit>` task or use the `<java>` task passing the class name of a test runner, such as `junit.textui.TestRunner`. Refer to the Ant manual (http://ant.apache.org/manual) for details on configuring and using the `<junit>` task.

The `<junit>` task allows you to use the `<batchtest>` task to create a test suite from tests on the file system. If you do not like the way Ant provides this feature, you can always launch the text-based test runner with `<java>` and use either GSBase's `RecursiveTestSuite` or JUnit-addons's `DirectorySuiteBuilder` to do the same thing, so neither approach can claim an advantage here. See chapter 4, "Managing Test Suites" for a discussion of these two test suite-building utilities.



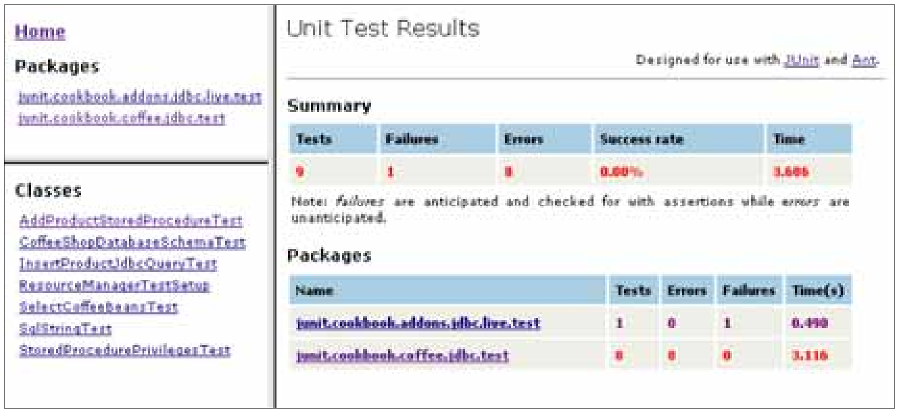
Test class name: junit.cookbook.organizing.test.MoneyTest
✓ Reload classes every run
Runs: 3/3 Errors: 0 Failures: 1
Results:
testRound(junit.cookbook.organizing.test.MoneyTest): expected:<1400

```
junit.framework.AssertionFailedError: expected:<1400> but was:<1300>
 at junit.cookbook.organizing.test.MoneyTest.testRound(MoneyTest.java:2
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorI
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodA
```

Finished: 0.07 seconds

Test class name: junit.cookbook.running.test.AnotherSetGlobalDataTest
✓ Reload classes every run
Runs: 1/1 Errors: 0 Failures: 0
Results:
Finished: 0.01 seconds

Figure 6.1 Swing-based test runner; left: one test error with a red bar; right: all tests pass with a green bar


Home

Packages

junit.cookbook.addons.jdbc.live.test
junit.cookbook.coffee.jdbc.test

Classes

AddProductStoredProcedureTest
CoffeeShopDatabaseSchemaTest
InsertProductJdbcQueryTest
ResourceManagerTestSetup
SelectCoffeeBeansTest
SqlStringTest
StoredProcedurePrivilegesTest

Unit Test Results

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
9	1	0	0.00%	3.606

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)
junit.cookbook.addons.jdbc.live.test	1	0	1	0.490
junit.cookbook.coffee.jdbc.test	8	0	0	3.116

Figure 6.2 A sample test execution report created by Ant's `<junitreport>` task.

The `<junit>` task does not report test progress to the console in the same manner that the JUnit text-based test runner does, and we feel that seeing that progress is very comforting, so we prefer it. This causes us to lean in the direction of using the `<java>` task to launch a text-based test runner.

If you want to publish build results to a web site, or if you simply like to see your test results in a format similar to Javadoc, then you want to use the `<junitreport>` task in conjunction with the `<junit>` task. JUnitReport takes XML output from the `<junit>` XML-based results formatter then applies an XSL stylesheet to it, yielding a summary much like Javadoc. You can see sample output in figure 6.2. Once again, refer to the Ant manual for details on using `<junitreport>`.

As with any trade-off, your best bet is to try both approaches and measure the difference. We tend to use the text-based test runner until we decide we want something more sophisticated, then we change.

JUnit-addons Test Runner

The JUnit-addons project (http://junit-addons.sourceforge.net) provides a test runner built with an open architecture designed to replace the JUnit text-based test runner. Although there are more command-line parameters than with the JUnit test runner, it is not necessary to specify them all when you use it. When you execute this test runner without any parameters you receive this message:

```
JUnit-addons Runner 1.0-alpha2 by Vladimir Ritz Bossicard
Usage: junitx.runner.TestRunner [-verbose]
                                 -runner.properties={filepath}
                                 -test.properties={filepath}
                                 -class classname
```

The `-class` option corresponds to the single parameter you pass to the JUnit test runner: the name of the test suite class to execute.

The "runner properties" direct the JUnit-addons test runner to configure itself with monitors (which can pause and resume the execution of tests) and listeners (which can obtain information about each test as it executes). You can use monitors and listeners to implement simple extensions such as custom test report formats. We describe the JUnit-addons test listener in detail in chapter 16, "JUnit-addons," but in the meantime, we describe a simple reporting extension in recipe 6.2. (In particular, see recipe 16.6, "Report the name of each test as it executes," which describes a minor defect in the JUnit-addons test runner documentation.)

The "test properties" help you specify test data paths, tool paths, and other environment settings on which your tests might depend. We do not discuss test properties in detail in this chapter, but do describe the general concept of using test properties in chapter 5, "Working with Test Data." The JUnit-addons test runner provides a small framework for organizing these files.

We provide recipes throughout this book that describe how to leverage the features in JUnit-addons. The JUnit-addons recipe in this chapter describes how to disable tests without removing test code (see recipe 6.6, "Ignore a test").

6.1 See the name of each test as it executes

◆ Problem

You would like to see the progress of the test run while it executes, including the name of the currently executing test.

◆ Background

If you introduce an infinite loop or deadlock into your code then the tests will eventually stop making progress. No failure message, nothing. If this happens you need to work "outside the system" to isolate the test (and therefore the production code) causing the problem. One easy way to get this information is to print out the name of each test as it starts. If there are no project or environmental constraints regarding the test runner you use to execute your tests, we have a simple solution for you.

◆ Recipe

If you need this feature, then we recommend simply executing the tests using a graphical test runner, such as the JUnit Swing-based test runner. The test runner's status bar displays the name of the currently executing test, so you can simply look at the point where the tests stop progressing and read the name of the offending test. Now you know where to look for the cause of the problem.

◆ *Discussion*

If, for some reason, you cannot or prefer not to use a Swing-based test runner, you can achieve the desired result with a little extra work. We describe our recommended technique in recipe 6.2.

◆ *Related*

- 6.2—See the name of each test as it executes with a text-based test runner
- 16.6—Report the name of each test as it executes

6.2 *See the name of each test as it executes with a text-based test runner*

◆ *Problem*

You would like to monitor the progress of your tests, including the name of the currently executing test, but you need or want to use a text-based test runner.

◆ *Background*

If you introduce an infinite loop or deadlock into your code then the tests will eventually stop making progress. No failure message, nothing. If this happens you need to work “outside the system” to isolate the test (and therefore the production code) causing the problem. One easy way to get this information is to print out the name of each test as it starts. This turns out to be easy to do if you can use a graphical test runner, but if your test environment assumes a text-based test runner then you have to solve this problem another way.

◆ *Recipe*

Because you’re debugging, you probably have a short-term need for this feature. In this case, the easiest way to achieve this is to add a line of code to the `setUp()` method of your test case class that prints out the name of each test during execution.

```
public class MyNameIsTest extends TestCase {
    protected void setUp() {
        System.out.println(getName());
    }
    // ... your tests ...
}
```

If you would prefer to see the class name and the test name, then call `toString()`, rather than `getName()`. It depends on whether you need the additional context; `toString()` returns something like `MyNameIsTest(testNumberOne)`, whereas `getName()` returns just `testNumberOne`.

◆ *Discussion*

If you need a more permanent solution, the next step is to override the method `TestCase.runTest()` to print the name of the test just prior to executing the test. You could easily add "entry and exit" messages in this way.

```
public class MyNameIsTest extends TestCase {
    protected void runTest() throws Throwable {
        System.out.println("Starting test " + toString());
        super.runTest();
        System.out.println("Ending test " + toString());
    }
}
```

Remember to invoke `super.runTest()`; otherwise, your test will not execute! It is remarkably easy to forget to invoke the superclass's implementation when overriding a method, making this a slightly dangerous way to implement this feature. The good news is that you only need to do this once: push this method all the way up in your test case class hierarchy so that all your classes can use it (see recipe 3.6, "Introduce a Base Test Case"). The bad news is that by implementing this through inheritance, you constrain your design, as Java is a single-inheritance language. It would be nice to implement this either as a Decorator or as a runtime-configurable parameter—a feature you can easily add or remove as needed. The runtime-configurable parameter is likely the simpler solution and we describe how to add these to your tests in recipe 5.1, "Using Java system properties" as well as throughout chapter 5, "Working with Test Data."

If you do not mind changing test runners, you can use the JUnit-addons test runner as an alternative to this recipe. See recipe 16.6, "Report the name of each test as it executes."

◆ *Related*

- 3.6—Introduce a Base Test Case
- 5.1—Using Java system properties
- 6.1—See the name of each test as it executes
- 16.6—Report the name of each test as it executes