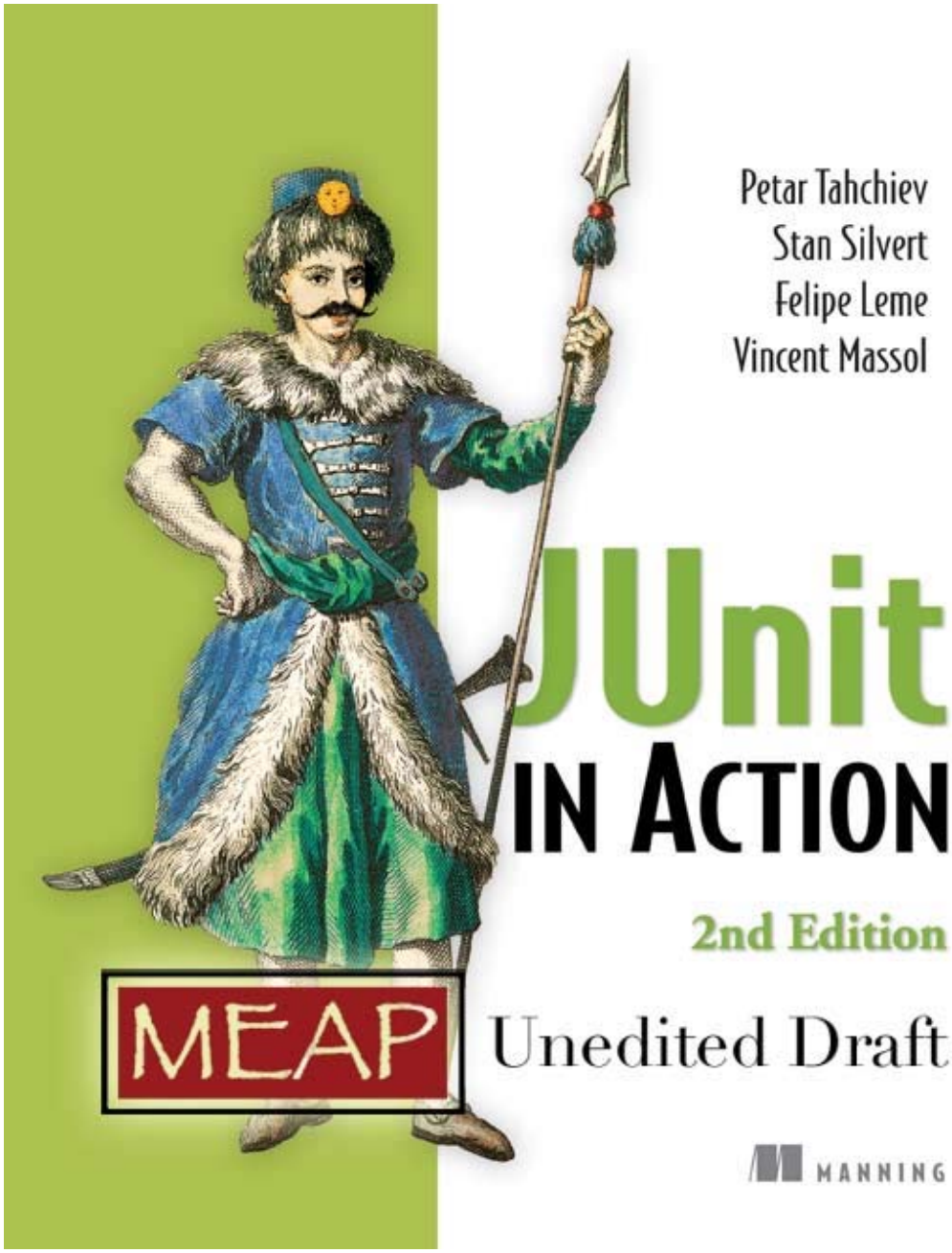

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

1

JUnit jumpstart

This chapter covers

- What JUnit is
- Writing sample tests by hand
- Installing JUnit and running tests

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

Never in the field of software development was so much owed by so many to so few lines of code.

—Martin Fowler

All code is tested.

During development, the first thing we do is run our own programmer's "acceptance test." We code, compile, and run. And when we run, we test. The "test" may just be clicking a button to see if it brings up the expected menu. But, still, every day, we code, we compile, we run…and *we test*.

When we test, we often find issues—especially on the first run. So we code, compile, run, and test again.

Most of us will quickly develop a pattern for our informal tests: We add a record, view a record, edit a record, and delete a record. Running a little test suite like this by hand is easy enough to do; so we do it. *Over and over again*.

Some programmers like doing this type of repetitive testing. It can be a pleasant break from deep thought and hard coding. And when our little click-through tests finally succeed, there's a real feeling of accomplishment: *Eureka! I found it!*

Other programmers dislike this type of repetitive work. Rather than run the test by hand, they prefer to create a small program that runs the test automatically. Play-testing code is one thing; running automated tests is another.

If you are a "play-test" developer, this book is meant for you. We will show you how creating automated tests can be easy, effective, and even fun!

If you are already "test-infected," this book is also meant for you! We cover the basics in part 1, and then move on to the tough, real-life problems in parts 2, 3, and 4.

1.1 Proving it works.

Some developers feel that automated tests are an essential part of the development process: A component cannot be *proven* to work until it passes a comprehensive series of tests. In fact, two developers felt that this type of "unit testing" was so important that it deserved its own framework. In 1997, Erich Gamma and Kent Beck created a simple but effective unit testing *framework* for Java, called JUnit. The work followed the design of an earlier framework Kent Beck created for Smalltalk, called SUnit.

> DEFINITION: *framework* — A framework is a semi-complete application. [1] A framework provides a reusable, common structure that can be shared between applications. Developers incorporate the framework into their own application and extend it to meet

[1] Ralph Johnson and Brian Foote, "Designing Reusable Classes," Journal of Object-Oriented Programming 1.5 (June/July 1988): 22–35; http://www.laputan.org/drc/drc.html.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

their specific needs. Frameworks differ from toolkits by providing a coherent structure, rather than a simple set of utility classes.

If you recognize those names, it's for good reason. Erich Gamma is well known as one of the "Gang of Four" who gave us the now classic *Design Patterns* book.[2] Kent Beck is equally well known for his groundbreaking work in the software discipline known as Extreme Programming (http://www.extremeprogramming.org).

JUnit (junit.org) is open source software, released under IBM's Common Public License Version 1.0 and hosted on SourceForge. The Common Public License is business-friendly: People can distribute JUnit with commercial products without a lot of red tape or restrictions.

JUnit quickly became the de facto standard framework for developing unit tests in Java. In fact, the underlying testing model, known as xUnit, is on its way to becoming the standard framework for any language. There are xUnit frameworks available for ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk, and Visual Basic—just to name a few!

Of course, the JUnit team did not invent software testing or even the unit test. Originally, the term *unit test* described a test that examined the behavior of a single *unit of work*.

Over time, usage of the term *unit test* broadened. For example, IEEE has defined unit testing as "Testing of individual hardware or software units *or groups of related units*" (emphasis added).[3]

In this book, we use the term *unit test* in the narrower sense of a test that examines a single unit in isolation from other units. We focus on the type of small, incremental test that programmers apply to their own code. Sometimes these are called *programmer tests* to differentiate them from quality assurance tests or customer tests (http://c2.com/cgi/wiki?ProgrammerTest).

Here's a generic description of a typical unit test from our perspective: "Confirm that the method accepts the expected range of input, and that the method returns the expected value for each test input."

This description asks us to test the behavior of a method through its interface. If we give it value *x*, will it return value *y*? If we give it value *z* instead, will it throw the proper exception?

DEFINITION: *unit test*—A unit test examines the behavior of a distinct unit of work. Within a Java application, the "distinct unit of work" is often (but not always) a single method. By contrast, integration tests and acceptance tests examine how various

---

[2] Erich Gamma et al., Design Patterns (Reading, MA: Addison-Wesley, 1995).

[3] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries (New York, IEEE, 1990).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

components interact. A unit of work is a task that is not directly dependent on the completion of any other task.

Unit tests often focus on testing whether a method is following the terms of its *API contract*. Like a written contract by people who agree to exchange certain goods or services under specific conditions, an API contract is viewed as a formal agreement made by the interface of a method. A method requires its callers to provide specific objects or values and will, in exchange, return certain objects or values. If the contract cannot be fulfilled, then the method throws an exception to signify that the contract cannot be upheld. If a method does not perform as expected, then we say that the method has broken its contract.

DEFINITION: *API contract*—A view of an Application Programming Interface (API) as a formal agreement between the caller and the callee. Often the unit tests help define the API contract by demonstrating the expected behavior. The notion of an API contract stems from the practice of *Design by Contract*, popularized by the Eiffel programming language (http://archive.eiffel.com/doc/manuals/technology/contract).

In this chapter, we'll walk through creating a unit test for a simple class from scratch. We'll start by writing some tests manually, so you can see how we used to do things. Then, we will roll out JUnit to show you how the right tools can make life much simpler.

1.2 Starting from scratch

Let's say you have just written the `Calculator` class shown in listing 1.1.

Listing 1.1 The test calculator class

```
public class Calculator {
public double add(double number1, double number2) {
return number1 + number2;
}
}
```

Although the documentation is not shown, the intended purpose of the `Calculator`'s `add(double, double)` method is to take two doubles and return the sum as a double. The compiler can tell you that it compiles, but you should also make sure it works at runtime. A core tenet of unit testing is: "Any program feature without an automated test simply doesn't exist."[4] The `add` method represents a core feature of the calculator. You have some code

[4] Kent Beck, Extreme Programming Explained: Embrace Change (Reading, MA: Addison-Wesley, 1999).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

that allegedly implements the feature. What's missing is an automated test that proves your implementation works.

But isn't the add method "too simple to possibly break"?

The current implementation of the `add` method is too simple to break. If `add` were a minor utility method, then you might not test it directly. In that case, if add did fail, then tests of the methods that used `add` would fail. The add method would be tested indirectly, but tested nonetheless. In the context of the calculator program, `add` is not just a method, it's a *program feature*. In order to have confidence in the program, most developers would expect there to be an automated test for the add feature, no matter how simple the implementation appears. In some cases, you can prove program features through automatic functional tests or automatic acceptance tests. For more about software tests in general, see chapter 3.

Yet testing anything at this point seems problematic. You don't even have a user interface with which to enter a pair of `doubles`. You could write a small command line program that waited for you to type in two `double` values and then displayed the result. Of course, then you would also be testing your own ability to type a number and add the result ourselves. This is much more than you want to do. You just want to know if this "unit of work" will actually add two `doubles` and return the correct sum. You don't necessarily want to test whether programmers can type numbers!

Meanwhile, if you are going to go to the effort of testing your work, you should also try to preserve that effort. It's good to know that the `add(double,double)` method worked when you wrote it. But what you really want to know is whether the method works when you ship the rest of the application. If we put these two requirements together, we come up with the idea of writing a simple test program for the method.

The test program could pass known values to the method and see if the result matches our expectations. You could also run the program again later to be sure the method continues to work as the application grows. So what's the simplest possible test program you could write? How about the simple `TestCalculator` program shown in listing 1.2?

Listing 1.2 A simple test calculator program

```
public class TestCalculator {
public static void main(String[] args) {
        Calculator calculator = new Calculator();
double result = calculator.add(10,50);
if (result != 60) {
System.out.println("Bad result: " + result);
}
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

The first `TestCalculator` is simple indeed! It creates an instance of `Calculator`, passes it two numbers, and checks the result. If the result does not meet your expectations, you print a message on standard output.

If you compile and run this program now, the test will quietly pass, and all will seem well. But what happens if you change the code so that it fails? You will have to carefully watch the screen for the error message. You may not have to supply the input, but you are still testing your own ability to monitor the program's output. You want to test the code, not yourself!

The conventional way to handle error conditions in Java is to throw an exception. Since failing the test is an error condition, let's try throwing an exception instead.

Meanwhile, you may also want to run tests for other `Calculator` methods that you haven't written yet, like `subtract` or `multiply`. Moving to a more modular design would make it easier to trap and handle exceptions and make it easier to extend the test program later. Listing 1.3 shows a slightly better `TestCalculator` program.

Listing 1.3 A (slightly) better test calculator program

```
public class TestCalculator {

private int nbErrors = 0;

public void testAdd() {                                          |
Calculator calculator = new Calculator();                        |
double result = calculator.add(10, 50);                         (1)
if (result != 60) {                                              |
throw new RuntimeException("Bad result: " + result);             |
}
}

public static void main(String[] args) {
TestCalculator test = new TestCalculator();
try {                                                            |
test.testAdd();                                                  |
}                                                                |
catch (Throwable e) {                                           (2)
test.nbErrors++;                                                 |
e.printStackTrace();                                             |
}
if (test.nbErrors > 0) {                                         |
throw new RuntimeException("There were " + test.nbErrors
+ " error(s)");
}
}
}
```

Working from listing 1.3, at (1) you move the test into its own method. It's now easier to focus on what the test does. You can also add more methods with more unit tests later, without making the main block harder to maintain. At (2), you change the `main` block to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

print a stack trace when an error occurs and then, if there are any errors, to throw a summary exception at the end.

1.3 Understanding unit-testing frameworks

There are several best practices that unit testing frameworks should follow. These seemingly minor improvements in the `TestCalculator` program highlight three rules that (in our experience) all unit testing frameworks should observe:

- Each unit test must run independently of all other unit tests.
- Errors must be detected and reported test by test.
- It must be easy to define which unit tests will run.

The "slightly better" test program comes close to following these rules but still falls short. For example, in order for each unit test to be truly independent, each should run in a different classloader instance.

Adding a class is also only slightly better. You can now add new unit tests by adding a new method and then adding a corresponding `try/catch` block to `main`.

A definite step up, but still short of what you would want in a *real* unit test suite. The most obvious problem is that large `try/catch` blocks are known to be maintenance nightmares. You could easily leave a unit test out and never know it wasn't running!

It would be nice if you could just add new test methods and be done with it. But how would the program know which methods to run?

Well, you could have a simple registration procedure. A registration method would at least inventory which tests are running.

Another approach would be to use Java's *reflection* and *introspection* capabilities. A program could look at itself and decide to run whatever methods are named in a certain way—like those that begin with the letters test, for example.

Making it easy to add tests (the third rule in our earlier list) sounds like another good rule for a unit testing framework.

The support code to realize this rule (via registration or introspection) would not be trivial, but it would be worthwhile. There would be a lot of work up front, but that effort would pay off each time you added a new test.

Happily, the JUnit team has saved you the trouble. The JUnit framework already supports registering or introspecting methods. It also supports using a different *classloader* instance for each test, and reports all errors on a case-by-case basis.

Now that you have a better idea of why you need unit testing frameworks, let's set up JUnit and see it in action.

1.4 Setting up JUnit

JUnit comes in the form of a jar file (`junit.jar`). In order to use JUnit to write your application tests, you'll simply need to add the junit jar to your project's compilation classpath and to your execution classpath when you run the tests.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

Let's now download the JUnit (JUnit 4.5 or newer) distribution, which contains several test samples that you will run to get familiar with executing JUnit tests. Follow these steps: Download the latest version of JUnit from junit.org, referred to in step 2 as `junit.zip`. Unzip the `junit.zip` distribution file to a directory on your computer system (for example, C:\ on Windows or `/opt/` on UNIX).

Underneath this directory, unzip will create a subdirectory for the JUnit distribution you downloaded (for example, `C:\junit4.5` on Windows or `/opt/junit4.5` on UNIX).

You are now ready to run the tests provided with the JUnit distribution. JUnit comes complete with Java programs that you can use to view the result of a test. There is a nice *textual* test runner (figure 1.1) that can be used from the command line.

To run the text test runner, open a shell in `C:\junit4.5` on Windows or in `/opt/junit4.5` UNIX, and type the appropriate command:

Windows:

```
java -cp junit.jar;. junit.samples.AllTests
```

UNIX:

```
java -cp junit.jar:. junit.samples.AllTests
```

The `AllTests` class contains a `main` method to execute the sample tests:

```
public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
}
```

And the result of the execution is shown in figure 1.1.

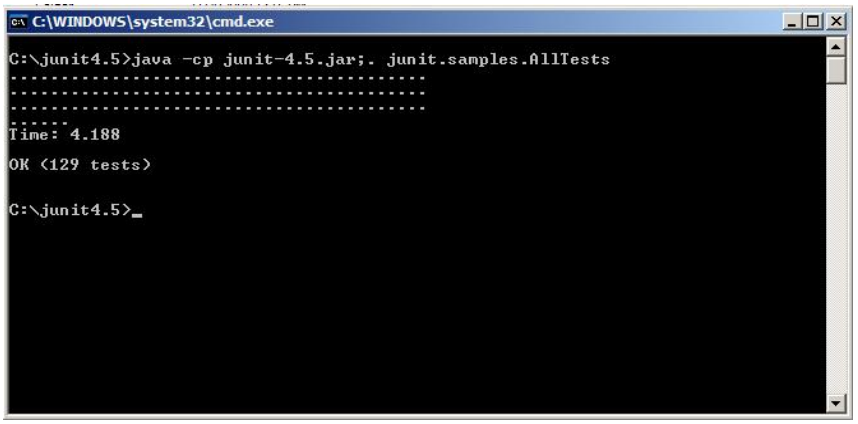

Fig 1.1 Execution of the JUnit distribution sample tests using the text test runner

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

Notice that for the text test runner, tests that pass are shown with a dot. Had there been errors, they would have been displayed with an *E* instead of a dot.

As you can see from the figures, the runners report equivalent results. The textual test runner is easier to run, especially in batch jobs, though the graphical test runner can provide more detail.

In part III of the book, we look at running tests using the Ant build tool and also the Maven build tool.

1.5 Testing with JUnit

JUnit has many features that make tests easier to write and to run. You'll see these features at work throughout this book:

- Separate classloaders for each unit test to avoid side effects.
- Standard resource initialization and reclamation methods (`setUp` and `tearDown`).
- A variety of assert methods to make it easy to check the results of your tests.
- Integration with popular tools like Ant and Maven, and popular IDEs like Eclipse, IntelliJ, and JBuilder.

Without further ado, let's turn to listing 1.4 and see what the simple `Calculator` test looks like when written with JUnit

Listing 1.4 The CalculatorTest program written with JUnit

```
import static org.junit.Assert.*;
import org.junit.Test;


public class CalculatorTest {                                    (1)

@Test                                                            (2)
public void add() {
Calculator calculator = new Calculator();                        (3)
double result = calculator.add(10, 50);                          (4)
assertEquals(60, result, 0);                                     (5)
     }
}
```

Pretty simple, isn't it? Let's break it down by the numbers.

In listing 1.4 at (1), you start by creating a test – there is absolutely no restrictions on the way you name the class or the visibility of the class.

At (2), you mark the method of being a unit test method by adding the `@Test` annotation. A best-practice is to name your test methods following the `test`*XXX* pattern.

At (3), you start the test by creating an instance of the `Calculator` class (the "object under test"), and at (4), as before, you execute the test by calling the method to test, passing it two known values.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

At (5), the JUnit framework begins to shine! To check the result of the test, you call an `assertEquals` method, which you imported with a static import on the first line of the class. The Javadoc for the `assertEquals` method is:

```
/**
* Asserts that two doubles are equal concerning a delta. If the
* expected value is infinity then the delta value is ignored.
*/
static public void assertEquals(double expected, double actual,
double delta)
```

In listing 1.4, you passed `assertEquals` these parameters:

- expected = 60
- actual = result
- delta = 0

Since you passed the calculator the values 10 and 50, you tell `assertEquals` to expect the sum to be 60. (You pass 0 as you are adding integer numbers, so there is no delta.) When you called the `calculator` object, you tucked the return value into a local double named `result`. So, you pass that variable to `assertEquals` to compare against the expected value of 60.

Which brings us to the mysterious delta parameter. Most often, the `delta` parameter can be zero, and you can safely ignore it. It comes into play with calculations that are not always precise, which includes many floating-point calculations. The `delta` provides a plus/minus factor. So if the actual is within the range (`expected-delta`) and (`expected+delta`), the test will still pass.

If you want to enter the test program from listing 1.4 into your text editor or IDE, you can try it using the text test runner. Let's assume you have entered the code from listings 1.1 and 1.4 in the `C:\junitbook\jumpstart` directory (`/opt/junitbook/jumpstart` on UNIX). Let's first compile the code by opening a shell prompt in that directory and typing the following (we'll assume you have the javac executable on your PATH):

JUnit Design Goals

The JUnit team has defined three discrete goals for the framework:

> The framework must help us write useful tests.
>
> The framework must help us create tests that retain their value over time.
>
> The framework must help us lower the cost of writing tests by reusing code.

In listing 1.4, we tried to show how easy it can be to write tests with JUnit.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

We'll return to the other goals in chapter 2.

Windows:
```
javac -cp ..\..\junit4.5\junit-4.5.jar *.java
```
UNIX:
```
javac -cp ../../junit4.5/junit-4.5.jar *.java
```

You are now ready to start the Swing test runner, by typing the following:

Windows:
```
java -cp .;..\..\junit4.5\junit-4.5.jar
➔ org.junit.runner.JUnitCore TestCalculator
```
UNIX:
```
java -cp .:../../junit4.5/junit-4.5.jar
➔org.junit.runner.JUnitCore TestCalculator
```

The result of the test is shown in figure 1.2.

The remarkable thing about the JUnit `TestCalculator` class in listing 1.4 is that the code is every bit as easy to write as the first `TestCalculator` program in listing 1.2, but you can now run the test automatically through the JUnit framework.

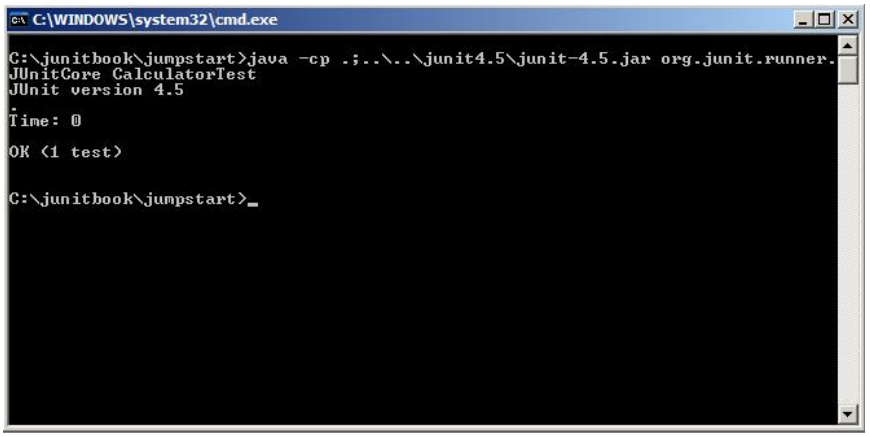

Figure 1.2 Execution of the first JUnit test TestCalculator using the text test runner

NOTE

If you are maintaining any tests written prior to JUnit version 3.8.1, you will need to add a constructor, like this:

```
public TestCalculator(String name) { super(name); }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502

It is no longer required with JUnit 3.8 and later.

1.6 Summary

Every developer performs some type of test to see if new code actually works. Developers who use automatic unit tests can repeat these tests on demand to ensure the code still works later.

Simple unit tests are not difficult to write by hand, but as tests become more complex, writing and maintaining tests can become more difficult. JUnit is a unit testing framework that makes it easier to create, run, and revise unit tests.

In this chapter, we scratched the surface of JUnit by installing the framework and stepping through a simple test. Of course, JUnit has much more to offer.

In chapter 2, we take a closer look at the JUnit framework classes (different annotations and assertion mechanisms) and how they work together to make unit testing efficient and effective. (Not to mention just plain fun!) We will also walk through the differences between the old-style JUnit and the new features of JUnit4.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=502