JUnit

Still popular after all these years...

JUnit: A Starter Guide

It started with a simple posting on a mailing list.

I hadn't expected thousands of people to read the Starter Guide each month, and I certainly hadn't expected it would become part of *JUnit Recipes: Practical Methods for Programmer Testing*. While a lot has changed since 2002, I continue to answer many of the same questions today that I answered in 2002 about how to use JUnit, how to practice test-driven development and how to design software we can confidently change. Even though I don't write Java code for food these days, I carry all the design principles and practices forward in my work that I learned by honing my craft with test-driven development as well as asking and answering a great many questions on the various Yahoo! groups. I have decided to give the Starter Guide a new home because I don't want to lose it and I because I don't it to remain anchored it in its past.

```
From jbr@diasparsoftware.com Sun Jan 20 06:26:08 2002
Date: Sun, 20 Jan 2002 09:27:40 -0500
To: junit@yahoogroups.com
Subject: A new JUnit primer

Everyone:

You can find another JUnit Primer on my web site, although I'm not sure
whether it says much more than Mike Clark's primer. I hope that the
newer JUnit users out there find this article as their starting point
-- it tries to answer a few questions that other tutorials don't.

Feedback is welcome.
J. B. Rainsberger,
President, Diaspar Software Services
Let's write software that people understand.
http://www.diasparsoftware.com/

(http://tech.groups.yahoo.com/group/junit/message/3837)
```

Please enjoy "JUnit: A Starter Guide", but please also search the web for the latest and greatest information, tutorials, books, training, and more, about JUnit, test-driven development and responsible software design. I choose not to link to specific resources, because I don't want those links to go out of date.

Thank you for reading.

J. B. Rainsberger
September 18, 2008

JUnit: A Starter Guide

JUnit is an open source testing framework for Java. It provides a very simple way to express the way you intend your code to work. By expressing your intentions in code, you can use the JUnit test runners to verify that your code behaves according to your intentions. This document should provide enough information for you to get started writing tests with JUnit.

The rhythm of a unit test

We will first talk specifically about unit test cases. A unit test case is a collection of tests designed to verify the behavior of a single unit within your program. In Java, the single unit is almost always a class. A Java unit test case, then, tests a single class.

When we talk about the "rhythm" of a test, we refer to the general structure of a test. When you sit down to write a test, you do the following:

1. Create some objects.
2. Send those objects some messages.
3. Verify some assertions.

Let us look at each step in detail.

The first thing you will do in a test is create some objects. For our purposes, we will choose a similar example to the classic JUnit example from the article Test Infected: a `Money` class. A `Money` object is essentially a quantity object whose value is a positive number and whose unit is a currency.

```
class Money {
    double getValue();
    String getCurrencySymbol();
}
```

One of your tests may be to verify what happens when you add `Money` objects together that represent the same currency. The first step in that test is to create two `Money` objects with the same currency.

Your next step in this test is to send your new `Money` object some messages. In this case, since you are testing the "add" feature of `Money`, you will likely

send one `Money` object the message "add" whose parameter is the other `Money` object.

Your final step in this test is to make some assertions about your objects after you have carried out the operations you're trying to test. In this case, now that you've added one `Money` object to the other, you have some assertions to verify:

1. The receiver `Money` object's value should be equal to its old value plus the value of the `Money` object it received through the "add" message.
2. The `Money` object sent as a parameter should not have changed in value.

Notice the use of the words "should" and "should not" above. In the Smalltalk version of the xUnit framework, SUnit, you can actually code your assertions this way

```
"If only JUnit were this simple!"
self should: [receiver value = 50]
self shouldnt: [true = false]
```

which, apart from the lack of apostrophe in "shouldn't", is plain English. By using the word "should" you are saying, "If this condition is not true, then my test fails." That is the definition of an assertion within a test: a boolean condition that must be true, otherwise the test fails.

So we have taken a simple test and shown its rhythm:

1. Create two `Money` objects with the same currency, call them receiver and parameter.
2. Send `receiver` the message `add(parameter)`.
3. `receiver`'s value should be its old value plus `parameter`'s value, and `parameter` should not have changed.

All your tests will follow this general rhythm, although they may require more object creation, message sending and assertions.

What a JUnit test looks like

In JUnit, each test is implemented as a method. Specifically, each JUnit test is implemented as a method that "asks no quarter and offers none." In other words, it takes no parameters and returns no value. The method stub for our example test would look like this.

```
public void testAddSameCurrency() {
    fail("Not yet implemented.");
}
```

Notice the declaration of the method: all test methods must be public, because they are executed by a test runner in a different package. Also notice the name of the method: it starts with the string "test". Although this is not strictly required, if you name all your tests this way, then the test runners can automatically find all the test methods within your test case class, freeing you from the need to tell the test runner which methods to execute.

Finally notice the statement I have written inside the test. The test case class inherits several methods used in expressing assertions. One of those is `fail`,

which forces the test to fail at that point. You can pass the method `fail` a message so that the test runner displays that message whenever the test fails at that point.

```
.F
Time: 0
There was 1 failure:
1) testAddSameCurrency(MoneyTestCase)junit.framework.AssertionFailedError: Not yet
implemented.
    at MoneyTestCase.testAddSameCurrency(MoneyTestCase.java:9)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

You can see that the test runner displays the failure message "Not yet implemented" -- the same message we coded in our test. We suggest using this technique to help you distinguish between tests that you have finished writing and tests that you have not finished writing. If you finish writing a test and it does not pass, then you have to write production code to make it pass; if you do not finish writing a test, then you should make sure it always fails so that you will be forced to go back and finish it.

The `TestCase` class

The central class in the JUnit test framework is `TestCase`, found in package `junit.framework`. All your test case classes will be descendants of this class. A `TestCase` object has multiple test methods, just like the one we started to write above. To create your own test case class, simply create a subclass of `TestCase`.

```
package com.diasparsoftware.junit.examples;

import junit.framework.TestCase;

public class MoneyTestCase extends TestCase {
    public MoneyTestCase(final String name) {
        super(name);
    }


    public void testAddSameCurrency() {
        fail("Not yet implemented.");
    }
}
```

Notice the constructor which takes a test name as a parameter. This is the name that the test runners display when they run the test or report a failure. All your test case classes need this constructor.

You can paste the above code into your editor, move it to a different package if you like, compile it and try it out. To compile and run the test case class, you must have the JUnit .jar file in your CLASSPATH.

```
REM JUNIT_HOME should be wherever you installed JUnit
set JUNIT_HOME=c:\junit

REM Compile the Java class as you normally would, but with
REM JUnit in the CLASSPATH.
javac -classpath %JUNIT_HOME%\junit.jar MoneyTestCase.java

REM Run the test runner of your choice, with JUnit and your
REM test case class files in the CLASSPATH
REM Below should be all one command -- all on the same line.
java -classpath <myClasses>;%JUNIT_HOME%\junit.jar
     junit.swingui.TestRunner MoneyTestCase
```

The example above uses the Swing-based test runner, although, to make this article faster to download, we have opted to use the text-based test runner for our example. The command line parameter to the `TestRunner` should always be the fully-qualified class name of the test case class to run. If your test is in a package, then you should qualify it using Java package notation, and not your operating system's directory notation. In other words, it is `com.diasparsoftware.junit.examples.MoneyTestCase` and not `com/diasparsoftware/junit/examples/MoneyTestCase.class`.

You should see the same result as in the image shown earlier in this article: the test runner runs the `MoneyTestCase` and shows one failure with the message "Not yet implemented."

COMMON PROBLEMS:

1. CLASSPATH is incorrect. Make sure that junit.jar is explicitly set in the CLASSPATH parameter you pass to the Java interpreter. Also make sure that your test case class is in that CLASSPATH. Without them, the Java interpreter cannot find your classes! If you have trouble with this step, you need to try running some simple Java programs until you become comfortable with the way the Java interpreter finds classes.
2. Test case class name is incorrect. The good news is that if you specify your test case class name incorrectly, the test runner will still be launched. The bad news is that you'll see the message "Test case class not found." In this case, simply type the fully-qualified test case class name in the text box provided by the test runner and press "Run". See the note above about specifying this test case class name correctly.
3. Warning! No tests found. If you see this message, then the test runner was unable to automatically find the test methods in your test case class. The most common reasons are: (1) the test method is not declared public -- package-level access is not enough; (2) the test method does not start with "test" -- the first four characters of the test method name must be exactly "test" in order for the test runner to find the method; (3) the test method returns a value or takes parameters -- remove them.

Assuming none of the above problems occurred, you should be ready to make the test do something interesting.

Making JUnit test something

Let us return to our example test. We would like to add two `Money` objects with the same currency together and verify the result. As we discussed earlier, the rhythm of the test is simple:

1. Create two `Money` objects with the same currency.
2. Add one `Money` object to the other.
3. Verify that one object's value is the sum of the two and the other has not changed.

Let us translate this, step by step, into JUnit code. Note that until we get to the verify step, we will simply write plain vanilla Java code. No magic here.

First, create two `Money` objects with the same currency. Since we don't care which currency that is, I'll let the `Money` object decide. We therefore write the following code.

```
public void testAddSameCurrency() {
    // STEP 1: Create some objects
    final Money money1 = new Money(20);
    final Money money2 = new Money(30);

    fail("Not yet implemented.");
}
```

Notice that we have not removed the `fail` statement from the end of the test yet. We have not finished writing the test; therefore, we keep this statement at the end of the test to remind us that we are not finished. If we run the test now, the test runner will make it clear that we should not expect the test to pass.

We have created two `Money` objects with the default currency and values 20 and 30. We naturally need a `Money` class with the appropriate constructor, so we build this class.

```
package com.diasparsoftware.junit.examples;

public class Money {
    public Money(final int aValue) {
    }
}
```

Notice that the constructor does nothing so far. This is intentional. Since we are trying to write the test, we should concentrate on that one activity and simply write enough code to make the test compile. Any more and we risk losing our train of thought. The code above is enough to make our test compile, but of course, the test is not yet finished, so it cannot pass.

The first step is complete: we have the two `Money` objects that we need to work with. The next step is to add one of them to the other. What is a simple way to do this? We come up with the code below.

```
public void testAddSameCurrency() {
    // STEP 1: Create some objects
    final Money money1 = new Money(20);
    final Money money2 = new Money(30);

    // STEP 2: Send some messages
    money1.add(money2);

    fail("Not yet implemented.");
}
```

In order for this to compile, we need to create the `add()` method on the `Money` class. Once again, in order not to distract ourselves, we only write the minimum code needed to make the test compile. We end up with the following code.

```
package com.diasparsoftware.junit.examples;

public class Money {
    public Money(final int aValue) {
    }


    public void add(final Money aMoney) {
    }
}
```

If you are concerned that we keep writing code that does not do anything, do not be concerned! We have a vision of the test in our mind and we want that vision to see the light of day before it leaves us. Just write enough to make the test compile for now.

We have finished the second step. Now it is time to verify the result of sending the "add" message to a `Money` object. In our case, we expect that when we add 30 to 20 the result is 50. We can therefore write the following assertion.

```
assertEquals(50, money1.getValue());
```

Let us look at this one line of code and understand it. If we simply read it as English, we read "assert that the value of the first money object equals 50." In simpler language, "The first money object's value should be 50." The method `assertEquals()` takes two parameters: the expected value and the actual value. Remember that the expected value is always the first parameter and the actual value is the second parameter.

In JUnit, we code assertions by calling methods whose names usually start with "assert". Although there are many such methods, the two you will most often want to use are

1. `assertTrue`, which evaluates a boolean condition and fails when that condition is false.
2. `assertEquals`, which evaluates whether two objects or values are equal and fails when they are not.

JUnit named the first method `assertTrue` so as not to collide with the keyword `assert` which has been added to the Java language for JDK 1.4. In order to assert that a condition is false, you would simply write

```
assertTrue(myCondition == false);
// or, if you prefer
assertTrue(!myCondition);
```

Unfortunately, there is no `assertNotEquals`, so you must resort to writing

```
assertTrue(!object1.equals(object2));
```

in that case. We will talk more about assertions in another article. For now, use `assertTrue` and `assertEquals` as your primary weapons for verifying results in your tests.

Speaking of which, let us now see our example test as a finished product.

```
    public void testAddSameCurrency() {
        // STEP 1: Create some objects
        final Money money1 = new Money(20);
        final Money money2 = new Money(30);

        // STEP 2: Send some messages
        money1.add(money2);

        // STEP 3: Verify the results.
        assertEquals(50, money1.getValue());
        assertEquals(30, money2.getValue());
    }
```

Notice two things: we have added two assertions and we have removed our `fail` statement. First, we verify that the first money object's value has changed to match the sum of 20 and 30, which is 50. Next, we verify that the second money object's value has not changed. Once again, we need to create a method on the `Money` object to make this test compile. Our `Money` class now looks like this.

```
package com.diasparsoftware.junit.examples;

public class Money {
    public Money(final int aValue) {
    }


    public void add(final Money aMoney) {
```

```
    }

    public int getValue() {
        return 0;
    }
}
```

Since the method `getValue()` needs to answer some value, we choose something simple for the moment. Our test is now complete as it expresses what we want to express and compiles.

Running the test

Now that we have finished writing the test, we want to make it pass. Before we do that, we should run the test and watch it fail. In fact, we want very much to watch it fail.

What?!

Simply put, if we write a test, expect it to fail and the test instead passes, something interesting has happened. Either the test is incorrect or we have more code than we thought we had. In our case, since we have been creating the `Money` class along with our test, we would have to conclude that our test was incorrect. Imagine what would happen if we wrote code that does what we wanted, ran the test and saw it fail: we would be scratching our heads because we think that the code we wrote should work. It may take us a long time to realize that it is the test, not the production code, that is faulty. We can eliminate this possibility by simply running the test as soon as it compiles and watching it fail.

The other possibility, of course, is that we had already written production code to make this test case pass. Often this means that we wrote more code than we needed to make a previous test pass. When this happens, it is a good idea to go back, read the code that makes this new test unexpectedly pass and understand what we've done. If we wrote too much code to make a previous test pass, then we may wish to be more strict about writing just enough code to make tests pass. The less code we write, the fewer things can go wrong.

Let us return to our test. We have run it and seen it fail.

```
.F
Time: 0.01
There was 1 failure:
1)
testAddSameCurrency(com.diasparsoftware.junit.examples.MoneyTestCase)junit.framework.AssertionFailedError:
expected:<50> but was:<0>
    at
com.diasparsoftware.junit.examples.MoneyTestCase.testAddSameCurrency(MoneyTestCase.java:20)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

The failure message indicates that the test expected 50, but the production code answered 0. We see that the failed assertion is at line 20 of `MoneyTestCase`, and a quick lookup shows that this assertion is the one that fails.

```
public void testAddSameCurrency() {
    // STEP 1: Create some objects
    final Money money1 = new Money(20);
    final Money money2 = new Money(30);

    // STEP 2: Send some messages
    money1.add(money2);

    // STEP 3: Verify the results.
    assertEquals(50, money1.getValue());    // <-- failing
    assertEquals(30, money2.getValue());
}
```

A JUnit test fails as soon as any assertion within the test fails. This means that the test aborts at that point and no further work is done -- in particular, the second assertion is not evaluated. You may be thinking that it is useful to know whether both assertions currently fail, so that you could fix both at the same time. Perhaps, but the JUnit behavior is intentional. The JUnit philosophy is that ultimately, it does not matter how many ways a test fails, it only matters that the test fails. Take it slowly, fix one problem at a time. It seems counterintuitive, but the fewer things you try to do at once, often the faster you complete everything. The human mind is strange.

Making the test pass

No matter. Let us simply write the production code to make this test pass. We will keep running the tests to help us know when we have finished. As soon as the test passes, we can move onto the next test.

What we do next will seem odd. Go with it for now. Since the only assertion we need to pass is the one that expects a value of 50, let us simply change `Money` so that `getValue()` answers 50. This should make the first assertion pass. We can then see what else might fail. (It sounds weird, I know -- go with it.)

```
package com.diasparsoftware.junit.examples;

public class Money {
    public Money(final int aValue) {
    }


    public void add(final Money aMoney) {
    }
```

```
    public int getValue() {
        return 50;    // <-- Just return 50.
    }
}
```

Now we recompile the tests and rerun them. If you haven't already noticed, there is a checkbox on the Swing-based test runner with the option "Reload classes every run." Make sure this option is selected, because by selecting it, we can keep the test runner window open even while we change our code. The test runner will reload the classes every time. Convenient, don't you think?

We recompile, rerun the test and we get this.

```
.F
Time: 0.01
There was 1 failure:
1)
testAddSameCurrency(com.diasparsoftware.junit.examples.MoneyTestCase)junit.framework.AssertionFailedError:
expected:<30> but was:<50>
    at
com.diasparsoftware.junit.examples.MoneyTestCase.testAddSameCurrency(MoneyTestCase.java:21)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Now the assertion on line 21 fails: this one expects 30, but gets 50. We take a closer look at the assertion.

```
public void testAddSameCurrency() {
    // STEP 1: Create some objects
    final Money money1 = new Money(20);
    final Money money2 = new Money(30);

    // STEP 2: Send some messages
    money1.add(money2);

    // STEP 3: Verify the results.
    assertEquals(50, money1.getValue());
    assertEquals(30, money2.getValue());   // <-- failing
}
```

Once an assertion passes, like the one on line 20, we cannot change the code to make it fail. In other words, we need both assertions to pass, which means that we should not just change the method `getValue()` to answer 30. Now we need the `Money` object to remember the value it was provided through the constructor. We add some code and end up with the following.

```
package com.diasparsoftware.junit.examples;

public class Money {
    private int value;


    public Money(final int aValue) {
        value = aValue;
    }


    public void add(final Money aMoney) {
    }


    public int getValue() {
        return value;
    }
}
```

Again, recompile and rerun. We hope that the test now passes. When we do this, we see the following.

```
.F
Time: 0.01
There was 1 failure:
1)
testAddSameCurrency(com.diasparsoftware.junit.examples.MoneyTestCase)junit.framework.AssertionFailedError:
expected:<50> but was:<20>
    at
com.diasparsoftware.junit.examples.MoneyTestCase.testAddSameCurrency(MoneyTestCase.java:20)
FAILURES!!!
Tests run: 1,  Failures: 1,  Errors: 0
```

Now what?! The first assertion fails again. We seem to have taken a step backward. The message tells us that the test expects `money1`'s value to be 50, but it is now 20. Notice that this was `money1`'s value before we add `money2` to it. This means that the constructor and the `getValue()` method both work, but `add()` does not. We implement this method in the simplest way that makes the test pass.

```
package com.diasparsoftware.junit.examples;

public class Money {
    private int value;
```

```
    public Money(final int aValue) {
        value = aValue;
    }


    public void add(final Money aMoney) {
        value += aMoney.getValue();    // <-- Simply add the values
    }


    public int getValue() {
        return value;
    }
}
```

Again we recompile, rerun and see that magic word: **OK**.

```
.
Time: 0
OK (1 tests)
```

If you hadn't noticed how rude JUnit is when it tells you that your tests fail, then seeing OK should be a pleasant surprise. One of the basic philosophies behind JUnit is that if all the tests pass, then a quick "OK" is all you should need to see. You'll grow to love seeing the happy rows of dots -- each one shows a running test -- and the final OK.

Note, if you have been using the Swing-based test runner, then you may have just seen the wonderful green bar of success for the first time! If you hadn't noticed the red bar of failure when you ran the tests before, then the beautiful green bar should be a pleasant surprise. That green bar is your immediate signal that all tests passed. Sometimes I like to use the graphical test runners just to see that nice green bar.

The job is not quite done

We have made our first test pass with a `Money` class that has a very simple implementation. In order to finish the job, we need to identify tests that likely do not yet pass. Notice that `Money` does not handle currencies at all yet, so we need to write a few tests that show that `Money` knows how to handle different currencies. For example, we can write a simple test to verify the default currency of a `Money` object.

```
public void testCreateDefaultCurrency() {
    assertEquals("CAD", new Money().getCurrencySymbol());
}
```

This very compact test verifies that if we create a `Money` object and specify nothing about it, then its currency symbol should be "CAD", representing Canadian dollars. We would need to create the method `getCurrencySymbol()` to make this test compile, then we can simply make the new method

answer "CAD" to make this new test pass.

Next, we may wish to ensure that if we specify a value and a currency for a `Money` object, then `getValue()` and `getCurrencySymbol()` answer exactly what we passed into the constructor.

```
public void testCreate() {
    final Money money = new Money(60, "USD");

    assertEquals(60, money.getValue());
    assertEquals("USD", money.getCurrencySymbol());
}
```

Notice two things about this test. First, it uses data we have not tried before, just to ensure that our code works for increasingly diverse data sets. Also notice its name, which follows a common naming convention we will describe in detail in this article. Read on.

Finally, we can add a test that adds two `Money` objects of different currencies. What behavior would we expect in that situation? We have at least three choices.

1. Change `Money` so that it has a value for each currency, doing conversions back and forth.
2. Change `Money` so that it has a value for each currency, but keeps them separate and does not do conversions.
3. Throw an exception indicating that it is illegal to add `Money` objects of different currency types.

Let us examine how to write the tests for each case.

The first case is quite easy. In order to do currency conversions, we need to specify a conversion rate between currencies. We can add a class-level method to `Money` that allows us to set the conversion rate. We can then add an instance-level method to `Money` that allows us to ask for its value in any currency we can convert to.

It sounds like we really have two tests: one testing the conversion and one testing the addition. First, the conversion test.

```
public void testConvert() {
    Money.convertAtRate("USD", "CAD", 1.58);
    final Money money = new Money(100, "USD");

    assertEquals(158, money.getValue("CAD"));
    assertEquals(100, money.getValue("USD"));
    assertEquals(100, money.getValue());
    assertEquals("USD", money.getCurrencySymbol());
}
```

First, we tell `Money` which conversion rate to use. One dollar US equals $1.58 Canadian. Next, we create a `Money` object for USD100. Our money object

should be worth CAD158 and USD100. The extra assertions help verify that we don't mess up code that's already written. If we ask `Money` for its Canadian dollar value, it should be $158. Its US dollar value should still be $100. In fact, if we ask for its value and do not specify currency, we should get its value in the currency used to create it. Its currency symbol should still be USD.

We need to create methods to make the test compile, watch it fail, then make it pass, step by step. As the textbooks say, we leave this as an exercise to the reader.

Our next test involves adding `Money` objects are different currencies. Imagine the rhythm of the test first, then compare your idea with our implementation.

```
public void testAddDifferentCurrency() {
    Money.convertAtRate("USD", "CAD", 1.58);
    final Money american = new Money(100, "USD");
    final Money canadian = new Money(158, "CAD");

    american.add(canadian);

    assertEquals(200, american.getValue());
    assertEquals(316, american.getValue("CAD"));
    assertEquals("USD", american.getCurrencySymbol());

    assertEquals(100, canadian.getValue("USD"));
    assertEquals(158, canadian.getValue());
    assertEquals("CAD", canadian.getCurrencySymbol());
}
```

Once again, we verify the values of both `Money` objects after we perform the add operation. For each `Money` object, we check the value in both US and Canadian dollars. We also verify that the value in each object's default currency is correct and that its default currency is as expected. Recompile, run, watch it fail, then make it pass. Another exercise for the reader.

Testing exceptions

We skip the second option, since it essentially the same as the first, except that the expected values in the tests will change. We will focus on the third option, as it forces us to consider a common idiom in JUnit tests: writing tests that expect an exception to be thrown.

There has been much debate on this topic, but ultimately those debates center on trivial issues of coding style. We present an implementation pattern here that serves us well and suggest it to you. If you prefer something different that does the same thing, then who are we to tell you what to do?

Our third design option was to have the add() method throw an exception when a client attempts to add two `Money` objects of different currencies. How would we write the test? The answer requires a little more understanding of how the framework evaluates whether a test passes or fails.

Briefly, a test passes if it does not fail. That sounds like a useless statement, but it very accurately reflects how JUnit works. The framework executes a test

method. If that test method executes without encountering any failures, then the test is considered a "pass". You may wonder, then, how does the framework know whether a test fails?

Sidebar: AssertionFailedError

The JUnit framework defines an error class called AssertionFailedError. Being an error, it is unchecked so that not every test method needs to declare that it might throw the error. All the assertion methods in the JUnit framework throw an AssertionFailedError whenever -- you guessed it -- an assertion fails! To be precise, here is a simple implementation of assertTrue().

```
public static void assertTrue(final boolean condition) {
    if (!condition) {
        throw new AssertionFailedError();
    }
}
```

The JUnit framework catches this error and reports that the test has failed. If the AssertionFailedError object has any detail about the failure, the test runners display that information to the user. For example, look at this implementation of an overloaded version of `assertTrue()`.

```
public static void assertTrue(final String message, final boolean condition) {
    if (!condition) {
        throw new AssertionFailedError(message);
    }
}
```

You can add a failure message to your assertions, giving yourself specific information about the failure. This is useful when, six months from now, you change code that makes a test fail and you would never otherwise be able to remember why it would fail.

So the JUnit framework throws and catches an AssertionFailedError whenenver one of your assertions fails, and that is how JUnit detects and reports test failures to the test runner and, ultimately, to you the programmer.

Now that you know how JUnit detects whether a test passes or fails, you can write a test that verifies whether an exception is thrown: verify that the test's execution path ends up inside the exception handler for the exception you expect, and call fail() if the test's execution path does not get there.

```
public void testAddDifferentCurrency() {
    Money.convertAtRate("USD", "CAD", 1.58);
    final Money american = new Money(100, "USD");
    final Money canadian = new Money(158, "CAD");
    try {
        american.add(canadian);

        // If the test gets here, then no exception
```

```
        // was thrown, so fail.
        fail("How did we add USD to CAD?!");
    }
    catch (final CurrencyMismatchException success) {
        // If the test gets here, then the right
        // exception was thrown, so do nothing.
    }
    // The test can only get here if the expected
    // exception was thrown, so the test would pass.
}
```

Notice the rhythm of the test.

1. Create some objects.
2. Execute the operation we're testing inside a try/catch block.
3. Catch only the exceptions we want the operation to throw.
4. Force the test to fail right after the operation executes, in case that operation does not throw an exception.

This is how to test that a method throws the correct exception.

You may be asking yourself, "What if the method throws a different exception?" We claim that our test already handles that situation. To understand why requires a little more knowledge of the JUnit framework.

Failures and Errors

In the JUnit framework, tests can either pass, fail or have errors. It should be clear by now what "pass" and "fail" mean, but what does it mean for a test to have an error?

Simply put, if a test method throws an exception that it does not itself catch, the test forces the JUnit framework to catch that exception. When this happens, the framework reports that the test has an error. You can interpret this to mean that something entirely unexpected happened -- something that may be worse than the production code simply being implemented incorrectly. It could be that the disk is full, or a file is not found, or the JVM has run out of memory. Something truly exceptional has happened and there is no way for the framework to evaluate whether the test has passed or failed, so the framework throws up its hands and says, "Something is wrong here."

In this case, you will see an error reported through the JUnit test runner, like the case below.

```
.E
Time: 0.01
There was 1 error:
1) testAddSameCurrency(com.diasparsoftware.junit.examples.MoneyTestCase)
```

```
java.lang.RuntimeException: Something strange happened.
    at com.diasparsoftware.junit.examples.Money.getValue(Money.java:18)
    at com.diasparsoftware.junit.examples.Money.add(Money.java:13)
    at
com.diasparsoftware.junit.examples.MoneyTestCase.testAddSameCurrency(MoneyTestCase.java:17)
FAILURES!!!
Tests run: 1,  Failures: 0,  Errors: 1
```

In this example, we have added code to the method `getValue()` to simulate something strange happening: it throws a `RuntimeException`, the kind of exception that usually indicates that the operating environment is not right or that corrupted data has made its way into the system. Since the test does not (and cannot) anticipate such a thing, it does not catch `RuntimeException`. Instead, the exception is thrown all the way back into the JUnit framework, which catches the exception and reports it as an error.

What do you do in case of an error?

1. Read the error message carefully and see whether your environment or data is damaged.
2. Read your test and verify that it is making the correct assertions. It may expect the production code to throw exception X, but the code is (correctly) throwing exception Y. In this case, it is your test, and not your production code, which is wrong.
3. Read your production code: it may simply be throwing the wrong exception.

It has been suggested that when we test for exceptions, we catch `Throwable` and fail when the throwable is not the one we expect. Let us point out that although this technique works, it results in extra code for not much more benefit. Since the framework already reports the unexpected exception as an error, there is little to gain by converting that error into a failure. Wrong is wrong. More to the point, when your production code acts strangely, you would likely benefit from JUnit's behavior of reporting the strangeness in a special way. You may consider possibilities that you would not otherwise consider when faced with a simple test failure. You may think, "Aha! My production code is incorrect. Let me fix it." When you get there, you see that it should work. Why, then, does the test fail? It may be several minutes before you notice that your disk is 99.98% full. Seeing an error tends to make you think of stranger things first -- our experience is that it is a time-saver.

JUnit helps you write better exceptions

One neat side-effect of writing your tests with JUnit is that it helps you write better exception objects. Few Java programmers pay attention to their custom-built exception objects. When they test their applications by hand, the mere throwing of the correct exception class makes them happy. It would be better, though, if they included more information in their exceptions. Writing JUnit tests that verify the correct exception is thrown helps programmers write more detailed, more useful exception classes. Let us return to our `add()` method for an example. Our test verified that we caught an instance of the correct exception class, but nothing else. That means that we can write extremely useless code that makes the test pass.

We can agree that this is the classic, simple implementation of an exception class.

```
package com.diasparsoftware.junit.examples;

public class CurrencyMismatchException extends RuntimeException {
```

```
    public CurrencyMismatchException(final String message) {
        super(message);
    }
}
```

The default constructor calls its parent's default constructor. The one-argument constructor calls its parent's one-argument constructor. As a Java programmer, you have written a class like this perhaps hundreds of times.

The problem, though, is that this could happen: your tests pass, but when you use the `Money` object within an application, your customer sees this error message.

```
$ java -classpath classes com.diasparsoftware.junit.examples.MoneyApp
Exception in thread "main" com.diasparsoftware.junit.examples.CurrencyMismatchEx
ception: Your mother is ugly.
        at com.diasparsoftware.junit.examples.Money.add(Money.java:49)
        at com.diasparsoftware.junit.examples.MoneyApp.go(MoneyApp.java:9)
        at com.diasparsoftware.junit.examples.MoneyApp.main(MoneyApp.java:17)
```

Your tests certainly did not catch the fact that one of your disgruntled programmers put this nice error message into your production code. It's too late to fire him: your company just went bankrupt from the poor public relations this mistake caused.

If you had verified the data in the exception object within your tests, you could have avoided this situation. Here is what we would do.

```
public void testAddDifferentCurrencyThrowsException() {
    Money.convertAtRate("USD", "CAD", 1.58);
    final Money american = new Money(100, "USD");
    final Money canadian = new Money(158, "CAD");
    try {
        american.add(canadian);

        // If the test gets here, then no exception
        // was thrown, so fail.
        fail("How did we add USD to CAD?!");
    }
    catch (final CurrencyMismatchException success) {
        // If the test gets here, then the right
        // exception was thrown, so check its data.
        assertEquals(american, success.getFirstMoney());
        assertEquals(canadian, success.getSecondMoney());
        assertEquals("Unable to add " + success.getFirstMoney()
                 + " and " + success.getSecondMoney() + " because "
                 + "they are not the same currency.",
```

```
            success.getMessage());
    }
    // The test can only get here if the expected
    // exception was thrown, so the test would pass.
}
```

At the very least, you could verify that the exception's message does not contain "mother" and "ugly". To make this test pass, we need to make `CurrencyMismatchException` demand this data from the classes that create them.

```
package com.diasparsoftware.junit.examples;

public class CurrencyMismatchException extends RuntimeException {
    private Money firstMoney;
    private Money secondMoney;


    public CurrencyMismatchException(
            final Money aMoney,
            final Money anotherMoney) {
        super(createMessage(aMoney, anotherMoney));
        firstMoney = aMoney;
        secondMoney = anotherMoney;
    }


    private static String createMessage(
            final Money aMoney,
            final Money anotherMoney) {
        return "Unable to add " + aMoney
                + " and " + anotherMoney + " because "
                + "they are not the same currency.";
    }


    public Money getFirstMoney() {
        return firstMoney;
    }


    public Money getSecondMoney() {
        return secondMoney;
    }
}
```

Now your disgruntled programmer's code does not even compile, let alone make it into your flagship product. Instead, he is forced to throw a very useful exception.

```
public void add(final Money aMoney) {
    if (!getCurrencySymbol().equals(aMoney.getCurrencySymbol())) {
        throw new CurrencyMismatchException(this, aMoney);
    }
    value += aMoney.getValue();
}
```

Not only does your test pass, but the application shows a much better message.

```
$ java -classpath classes com.diasparsoftware.junit.examples.MoneyApp
Exception in thread "main" com.diasparsoftware.junit.examples.CurrencyMismatchEx
ception: Unable to add com.diasparsoftware.junit.examples.Money@5d87b2 and com.d
iasparsoftware.junit.examples.Money@77d134 because they are not the same currenc
y.
        at com.diasparsoftware.junit.examples.Money.add(Money.java:49)
        at com.diasparsoftware.junit.examples.MoneyApp.go(MoneyApp.java:9)
        at com.diasparsoftware.junit.examples.MoneyApp.main(MoneyApp.java:17)
```

Well, it's not perfect yet, but implementing `Money.toString()` will fix that:

```
public class Money {
    ...
    public String toString() {
        return currencySymbol + value;
    }
    ...
}

[...]

$ java -classpath classes com.diasparsoftware.junit.examples.MoneyApp
Exception in thread "main" com.diasparsoftware.junit.examples.CurrencyMismatchEx
ception: Unable to add USD100 and CAD158 because they are not the same currency.

        at com.diasparsoftware.junit.examples.Money.add(Money.java:49)
        at com.diasparsoftware.junit.examples.MoneyApp.go(MoneyApp.java:9)
        at com.diasparsoftware.junit.examples.MoneyApp.main(MoneyApp.java:17)
```

Beautiful. Look at the advantage that writing your tests with JUnit has given you. You have better error messages and a useful string representation of one of

your core classes, just because you tried to write a unit test using JUnit. You will not need to run around adding all this extra good stuff in when time is tight at the end of your project.

Advanced Topics

Now that you have seen how to use JUnit to write simple tests, you likely have enough information to write many of your tests. There are some advanced topics that you will need to examine when you find that you don't quite know how to test something in your application. They are the subject of other articles.

- The contract of Object: why you need to implement `equals()`, `hashCode()` and `toString()` for almost everything.
- Test fixture: using the same objects for many tests.
- Test suites: managing multiple test classes.
- Custom assertions: evolving your own domain-specific testing language.
- Mock objects: how to test a class simply on its own.
- Performance testing with JUnit.
- System tests with JUnit.
- Web services testing with JUnit and its cousins.

Resources

Read the original article that helped launch JUnit as a testing framework.

If you haven't read Design Patterns, then stop, acquire a copy and read it — but please read the implementations section, and not just the class diagrams.

Another fine primer for JUnit, courtesy of Mike Clark at Clarkware Consulting, Inc.

Appendices

Naming test classes

The simplest way to name a unit test class is to use a standard subclass naming convention, since a test case class is a subclass of `TestCase`. Specifically, the test case class is a test for, say, the `Money` class; therefore, it is a `MoneyTestCase`.

Test fixture classes and abstract test cases — that is, test-related classes that are not themselves test cases — should be named anything reasonable that **does not** end in "TestCase". This suggestion makes automatically collecting test case classes into a larger test suite a matter of a very simple "end-of-string" match.

Naming test methods

A standard that has emerged in our work is to name test methods according to the feature they test, and not according to class and method names. This is intentional, as it helps us focus on the behavior we want, and not the methods we want.

Since each feature has multiple scenarios, all of which must work, we generally use test method names that separate the name of the feature from the name of the scenario. The separator can either be an underscore character or the word "with", depending on local custom regarding having the underscore character in method names.

As an example, consider the feature "create a Money object". Presumably you have created test case `MoneyTestCase`. If this is the "happy path" — that is, if it is the normal scenario for this feature, where methods are called correctly and data all makes sense — then we simply call the method `testCreate()`. The absence of a scenario name suggests that this is the "normal" scenario.

Now to test what happens when, say, someone tries to create a `Money` object worth -1 dollars, we call the test method `testCreate_NegativeValue()` or, if you hate the underscore, `testCreateWithNegativeValue()`.
