# Using JUnit in Eclipse [1, 2]

**by Christopher Batty**

**Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada**

**Last revised: October 30, 2003**

## Overview:

In this document, we describe how to use Eclipse's built-in JUnit tools to create unit tests for Java classes.

## Unit Testing Basics

When developing software we need to run our code to verify that it actually works as intended. As the software gets larger and larger, it becomes more likely that introducing a new change will "break" the existing code. At the same time, verifying all of a program's features by hand is a menial and time-consuming task. To reduce the need for manual testing, we can introduce unit tests which are small snippets of code designed to thoroughly exercise a particular function or class. With unit tests in place, when a change is made to the code we can simply run all the tests to ensure that nothing has been broken.

An additional motivation for unit testing is to support test-driven development. Test-driven development involves writing unit tests first, with the actual code coming after. The reasoning is that in order to write complete, valid tests, you must have a solid understanding of how the new class should work, and in particular what its interface will be. Thus, the unit tests provide an executable definition of the class. When you move on to writing the actual code, you just have to fill in implementation, and once the tests are passed, the implementation is complete.

In Java, the standard unit testing framework is known as JUnit. It was created by Erich Gamma and Kent Beck, two authors best known for design patterns and eXtreme Programming, respectively. Eclipse builds come with both JUnit and a plug-in for

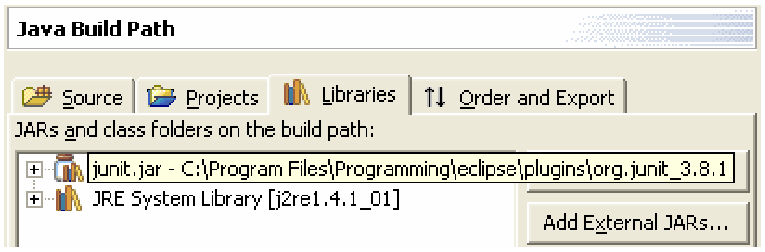
[1] This work was funded by an IBM Eclipse Innovation Grant.

[2] © Christopher Batty and David Scuse

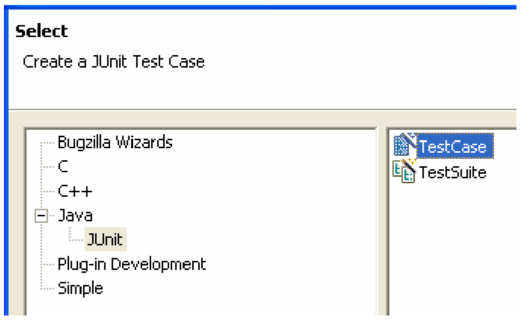
creating and working with JUnit tests. As a demonstration, we will use these tools to test a simple class for representing fractions.

## Creating a Test Case

Using Eclipse, import the sample project (JUnitDemo) provided with this tutorial. (A second project, JUnitDemoWithTests, that contains the result of adding JUnit tests is also included.) It contains just SimpleFraction.java, which has functionality for creating, modifying and simplifying a Fraction. The first thing we must do is import junit.jar, so we have access to the testing framework. Right-click on the project name, and choose Properties. In the tree on the left, select Java Build Path. Next, choose Add External JARs… and browse to find junit.jar. It will be located in <eclipsedir>\plugins\org.junit_<version number>\junit.jar. Once you successfully import junit.jar, close the Properties page.


Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
junit.jar - C:\Program Files\Programming\eclipse\plugins\org.junit_3.8.1
JRE System Library [j2re1.4.1_01]
Add External JARs...

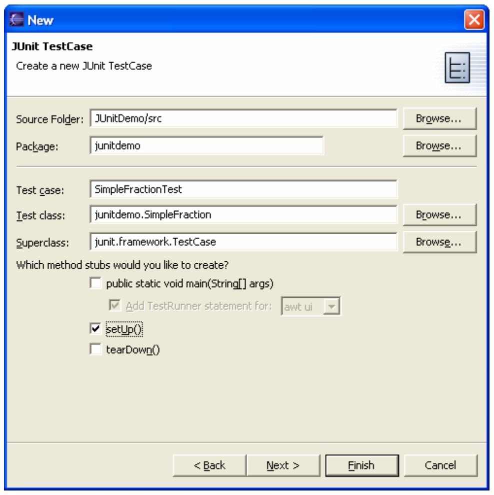
Now we can create a test for our SimpleFraction class. Right-click the package you wish to place the test in, and choose New…, then select Other… In the tree on the left, expand the Java branch, and choose JUnit. Then in the list on the right select TestCase. This allows us to create a new test case for SimpleFraction.



A Wizard will appear for creating a JUnit TestCase. Click on the Browse… button beside Test Class to select the class for testing. In our case it will be the SimpleFraction class, so type in SimpleFraction and select it from the list. Notice that the Test case name is automatically filled in as SimpleFractionTest. Appending "Test" to the name of the class being tested is the default test case naming scheme in JUnit.

The check-boxes at the bottom allow you to add stubs for additional useful methods. The first is a standard main() function which can be used to run this test individually as an application, rather than as a part of a large suite of tests. The drop-down menu gives you the choice of running with text output, an AWT-based GUI, or a more elaborate Swing-based GUI. This is unnecessary for us since we will be using Eclipse's built-in JUnit tools.

The other stubs are for setUp() and tearDown() functions, which are called before and after each test case. They can be used to reduce redundant code if the tests require the same resources or data. For demonstration purposes, we will use a setUp() method to initialize a fraction used by multiple tests. Check setUp() and choose Next.



JUnit TestCase

Create a new JUnit TestCase

Source Folder: JUnitDemo/src

Package: junitdemo

Test case: SimpleFractionTest

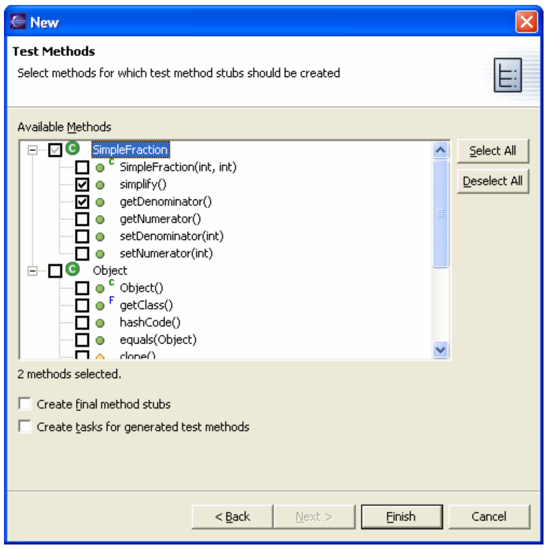
Test class: junitdemo.SimpleFraction

Superclass: junit.framework.TestCase

Which method stubs would you like to create?

- [ ] public static void main(String[] args)
  - [x] Add TestRunner statement for: awt ui
- [x] setUp()
- [ ] tearDown()

The next screen allows you to select which of SimpleFraction's methods you want test stubs generated for. For now we will just test simplify() and getDenominator(), so we check these two boxes and hit Finish.



Test Methods

Select methods for which test method stubs should be created

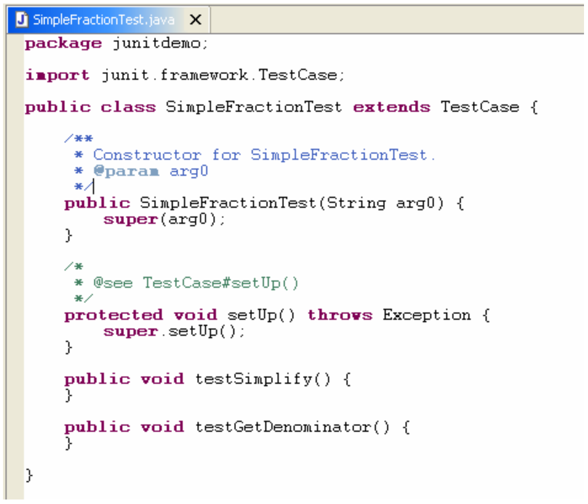
Available Methods

- [ ] SimpleFraction
  - [ ] SimpleFraction(int, int)
  - [x] simplify()
  - [x] getDenominator()
  - [ ] getNumerator()
  - [ ] setDenominator(int)
  - [ ] setNumerator(int)
- [ ] Object
  - [ ] Object()
  - [ ] getClass()
  - [ ] hashCode()
  - [ ] equals(Object)
  - [ ] clone()

2 methods selected.

- [ ] Create final method stubs
- [ ] Create tasks for generated test methods

The class is created with the appropriate method stubs generated for us. All we need to do now is write the tests.



```
package junitdemo;

import junit.framework.TestCase;

public class SimpleFractionTest extends TestCase {

    /**
     * Constructor for SimpleFractionTest.
     * @param arg0
     */
    public SimpleFractionTest(String arg0) {
        super(arg0);
    }

    /*
     * @see TestCase#setUp()
     */
    protected void setUp() throws Exception {
        super.setUp();
    }

    public void testSimplify() {
    }

    public void testGetDenominator() {
    }

}
```

## Writing the Tests

Since getDenominator() is straightforward, we will write testGetDenominator() first. We will create a couple of SimpleFractions and call getDenominator on each. Create two private SimpleFraction member variables like the following:

```
private SimpleFraction f1, f2;
```

In the setup method, add two lines to construct the fractions.

```
f1 = new SimpleFraction(15, 25);
f2 = new SimpleFraction(-27, 6);
```

Each time a test method is called, the setUp() will run these two lines first, so at the start of each test the fractions will always be 15/25 and -27/6.

Now to implement our testGetDenominator() method, add the following code to the method body.

```
int result = f1.getDenominator();
assertTrue("getDenominator() returned " + result + " instead of 25.",
       result == 25);

result = f2.getDenominator();
assertEquals(6, result);
```

Since SimpleFractionTest is a subclass of the junit.framework.TestCase, it inherits a variety of methods for testing assertions (i.e. conditions that should be true) about the state of variables in the test. These include assertEquals, assertTrue/False, assertSame/NotSame, and assertNull/NotNull.  There is also a fail() method that just causes the current TestCase to fail.  These methods generally come in two flavours: a plain version, and a version with a String parameter to provide details about the assertion that failed.  In the code above, you can see examples of each.  Whenever the assertion in one of the methods fails, the execution of that test method is terminated, and JUnit will record that test as having failed.

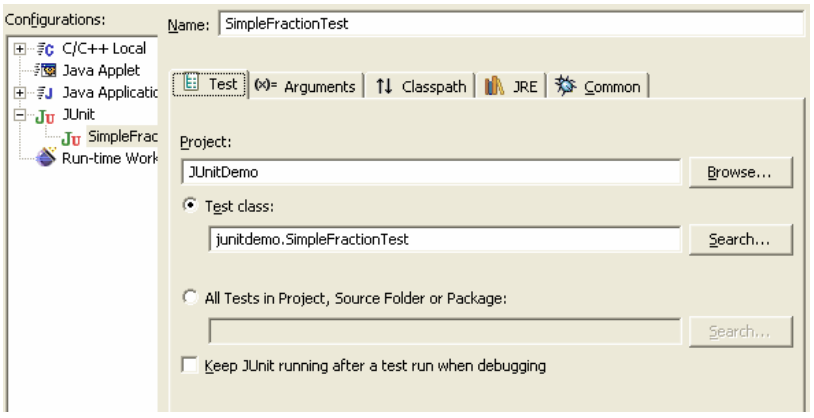
Next we will implement testSimplify().

```
f1.simplify();
assertEquals(3, f1.getNumerator());
assertEquals(5, f1.getDenominator());
```

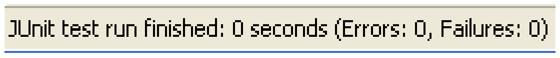
This will verify that our simplify() method correctly reduces 15/25 to 3/5.

## Running the Tests

Now that the tests have been written, we would like to run them.  The process for running JUnit tests is very similar to that for running regular Java Applications.  From the Run menu choose Run…  Select JUnit in the tree on the left, and hit New.  The screen should look like the following, assuming SimpleFractionTest was selected when you opened the Run menu.  (If not, simply browse to find SimpleFractionTest.)


Name: SimpleFractionTest
Project: JUnitDemo
Test class: junitdemo.SimpleFractionTest

Next, just choose Run, and the TestCase will be executed.  If it ran successfully, you will see a message in the status bar at the bottom of the screen that looks like the following.


JUnit test run finished: 0 seconds (Errors: 0, Failures: 0)

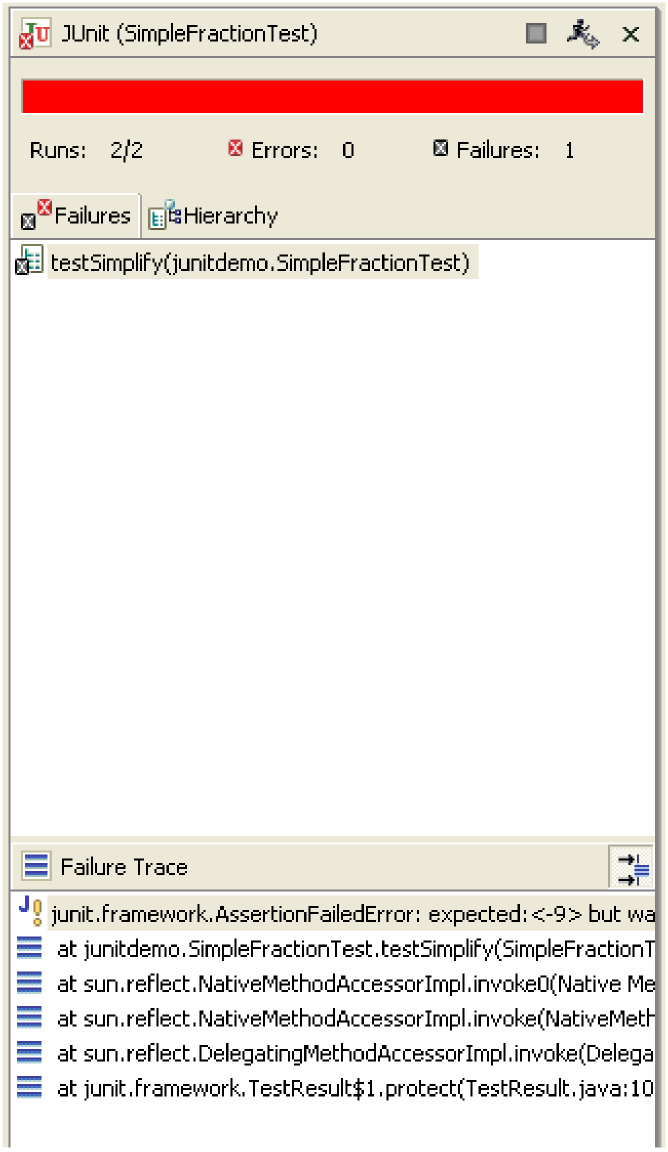
If the JUnit view is visible, it will display a green bar, indicating a successful test.  We will look at the JUnit view in more detail later on.

## Debugging a Failed Test

Now, let's add another piece of code to our testSimplify() method.

```
f2.simplify();
assertEquals(-9, f2.getNumerator());
assertEquals(2, f2.getDenominator());
```

This looks very similar to the previous code in testSimpify(). However, consider what happens when we run the tests again. The JUnit view will be displayed on the left of the screen (if it is not already visible) and it will show a red bar indicating that one or more tests failed.



```
testSimplify(junitdemo.SimpleFractionTest)
```

```
junit.framework.AssertionFailedError: expected:<-9> but wa
at junitdemo.SimpleFractionTest.testSimplify(SimpleFractionT
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Me
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMeth
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delega
at junit.framework.TestResult$1.protect(TestResult.java:10
```

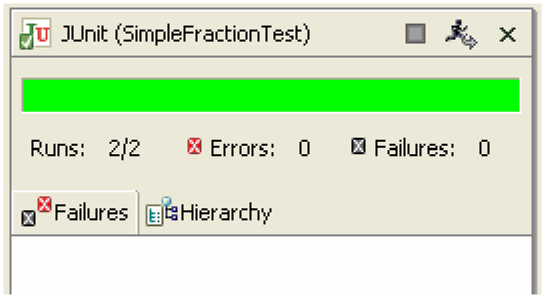
The top part of the view displays the names of the test method(s) that failed. By double-clicking on the method name, the corresponding method will be opened in the Java editor.

The bottom of the view shows a stack trace of the method calls leading up to the failure. Here we can see that since we used the assertEquals method, it gives us a message about what the expected value was versus the value received. Double-clicking on the second line from the top of the stack trace will jump to the specific line at which the failure occurred.

The error occurred in the testSimplify() method, and it looks as though the absolute value of the result was correct, but the sign was wrong. By commenting out the failed assertion, we can verify that the sign of the denominator is also reversed from what was expected. A simple fix for this is to check if the denominator is ever negative, and if so negate both numerator and denominator, effectively "moving" the negative sign to the denominator. The code for this is commented out in the simplify() method of SimpleFraction.java, so we can uncomment it now.

```
if(denominator < 0) {
      denominator = -denominator;
      numerator = -numerator;
}
```

This fix is very straightforward. We can be fairly certain it will correct the specific case in question, but will it break our existing code? Since we have our unit tests in place, we make the change, and run the tests again. This time, they run to completion successfully, and a green bar is displayed in the JUnit view. Both our new and pre-existing test cases work, so we may assume that the fix was correct.


JUnit (SimpleFractionTest)

Runs: 2/2 Errors: 0 Failures: 0

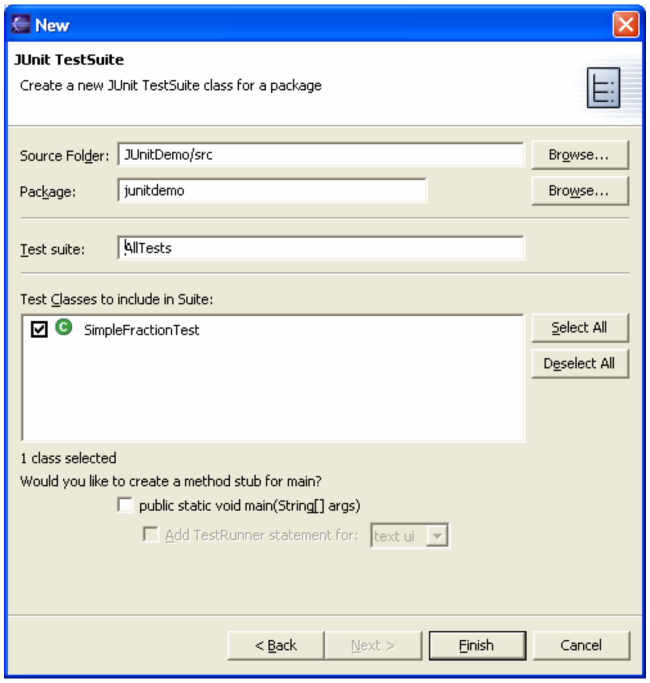
This raises an important point about the value of unit testing. A suite of tests is only useful if it is both correct and thorough. You may occasionally have a test case fail only to discover that it is the test case that is flawed and not the code being tested. Similarly, if a change to the code breaks functionality that is not covered by the test cases but is used in your actual application, it will take longer to discover. Meanwhile you may continue to make changes that further break existing code. It is therefore important to keep your unit tests up to date as you write new code.

In the case of our SimpleFraction example above, it would be wise to add some additional simplify() tests with different values and different combinations of negative and positive denominator and numerator to further verify that the change we made behaves as intended. When we later try to add new features to our SimpleFraction class, these existing tests will give us greater confidence in the correctness of any changes.

## Test Suites

Once you have created more than one TestCase, it is useful to be able to group them and run them all together. JUnit has the concept of a TestSuite for doing exactly that.

To create a Test Suite in Eclipse, right-click on the package and select New->Other… Select JUnit as before, but choose to create a TestSuite rather than a TestCase.

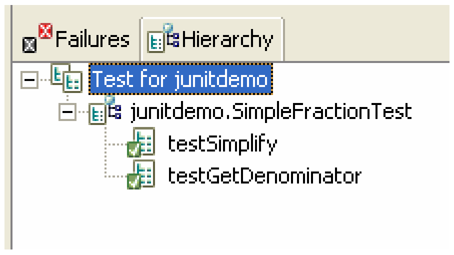


JUnit TestSuite
Create a new JUnit TestSuite class for a package

Source Folder: JUnitDemo/src
Package: junitdemo
Test suite: AllTests
Test Classes to include in Suite:
☑ SimpleFractionTest
1 class selected
Would you like to create a method stub for main?
☐ public static void main(String[] args)
☐ Add TestRunner statement for: text ui

By default, all the classes in the package containing the word "test" will be added to the suite. When you hit Finish, a new class will be created that contains a method that instantiates and returns a TestSuite with all the TestCases you selected. As with individual TestCases, the TestSuite can be launched from the Run menu.

You may have noticed the second tab on the JUnit view labeled Hierarchy. This displays the current TestSuite and its contents. In addition to adding individual TestCases to a TestSuite, you can also add TestSuites to a TestSuite. This allows the creation of a more complex, tree-like hierarchy of tests, with TestCases being the leaf nodes and TestSuites being the internal nodes. It is this hierarchy that is displayed in the Hierarchy view when a JUnit test or suite is run. In our example we have just the single suite containing the SimpleFractionTest class with two test methods.



```
Test for junitdemo
  junitdemo.SimpleFractionTest
    testSimplify
    testGetDenominator
```

## Conclusion

Due to its simplicity and usefulness, JUnit has become nearly ubiquitous in the world of Java, and ports of it exist for nearly every language available (e.g. CppUnit, NUnit, HttpUnit, PHPUnit, etc.) If you wish to learn more about JUnit, consider visiting http://www.junit.org