UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 61B **P. N. Hilfinger**
Fall 2014

Unit Testing with JUnit

1 The Basics

JUnit is a testing framework that is intended to help in writing *unit tests:* tests of individual components of a system. It provides a testing framework that allows you to add small testing methods to your programs, which the framework will find and run automatically. Its most recent form, JUnit 4.x, makes convenient use of *annotations,* a feature of Java 1.5, to mark testing code. In this document, we'll cover just the features of JUnit likely to be used in this class.

Basic use is simple. Here, for example, is a small example for testing a method from a homework: `Progs.factorSum`, which is supposed to compute the sum of all divisors of N that are $< N$:

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

public class ProgTest {

    @Test
    public void factorSum () {
        assertEquals (0, Progs.factorSum (0));
        assertEquals (0, Progs.factorSum (1));
        assertEquals (1, Progs.factorSum (2));
        assertEquals (1, Progs.factorSum (3));
        assertEquals (3, Progs.factorSum (4));
        assertEquals (1, Progs.factorSum (5));
        assertEquals (6, Progs.factorSum (6));
        assertEquals (1, Progs.factorSum (7));
        assertEquals (16, Progs.factorSum (12));
        assertEquals (22, Progs.factorSum (20));
        assertEquals (28, Progs.factorSum (28));
        assertEquals (54, Progs.factorSum (42));
    }

}
```

As you can see, a test consists of a method that is *annotated* with `@org.junit.Test` (which we can abbreviate as `@Test`, thanks to the `import` statements). Each test contains arbitrary code for performing some check (generally on an individual method). The method `assertEquals` (defined in class `org.junit.Assert`) does nothing if its two operands are equal, and otherwise causes an error.

Perhaps you have been accustomed to testing your code using `print` statements. You don't do that here; the various `test...` methods you write will communicate their results to you by the success or failure of the assertions. After compiling `ProgTest` and, of course, the class `Progs` that it tests, you can run the tests in it from the command line with

```
java org.junit.runner.JUnitCore ProgTest
```

which will find all accessible methods in `ProgTest` (and any other classes you list) that are annotated `@Test` and run them, reporting which ones fail. In my opinion, the output from JUnitCore could use some cleaning up, so we've provided an alternative, which we'll be using this semester[1]:

```
java ucb.junit.textui ProgTest
```

You can mark as many tests as you want with the `@Test` annotation. You can also put tests in `Progs.java` itself, the file being tested, and use `Progs` as an argument to `ucb.junit.textui`. The idea is to add one or more test methods to your tests with each new method you write. To be really hard-core (*test-centric*), you should write the tests after deciding on the arguments to your method and writing its comment and *before* writing its body. Initially, it will fail the test (being unimplemented), and will succeed when you get the implementation right.

The test runners (ours and theirs) report at most one error per test method, so it may not always be wise to put many `assert` calls in the same test method, as we did in the `factorSum` test. It's a matter of taste. Once your program is running, you don't expect it to fail tests, so there is no harm in giving a method many chances to fail, as it were. However, you'll get information about more errors at once if you split assertions up among several methods.

2 Assert methods

Besides `assertEquals`, There are a number of these `assert` methods provided in JUnit, the most useful of which are described in Table 1. All these methods have a variant that takes a String-valued message as its first parameter. This allows you to provide an alternative error message to the default message.

3 Dealing with exceptions and infinite loops

If a test causes an exception that it shouldn't, the framework will (as you'd hope) catch this and count it as a test failure. Sometimes, though, you *expect* an exception, as when you are testing that a certain method throws an exception on erroneous data. You can handle this by adding a parameter to the `@Test` annotation:

```
@Test(expected=IllegalArgumentException.class) public void badCall1 () {
    Account account = Bank.makeAccount (1000);
    account.deposit (-100);
}
```

(The Java notation `.class` after a class name denotes a literal of type `Class`, a basic component in the Java feature known as *reflection*. Just as functions are values in Scheme, types are values in Java, at least up to a point.)

Occasionally, you'll have a test that you suspect might occasionally run amok and take far too long (perhaps forever) to finish. Another parameter to `@Test` handles this:

[1]The `ucb.junit` package is part of our standard `ucb` package, installed on the instructional machines. To use it at home, copy the JAR file from `~cs61b/lib/ucb.jar` and arrange for it to be in your class path.

`void assertTrue (boolean condition);` `void assertTrue (String message, boolean condition);`	Succeeds iff `assertTrue`'s condition evaluates to `true`.

`void assertNull (Object obj);`
`void assertNull (String message, Object obj);`
`void assertNotNull (Object obj);`
`void assertNotNull (String message, Object obj);`
	Succeeds iff `assertNull`'s argument is null, or `assertNotNull`'s argument is non-null.

`void assertEquals (`T `expected,` T `actual);` `void assertEquals (String message,` T `expected,` T `actual);`	When T is a reference type, succeeds iff both expected and actual values are null or neither is null and `expected.equals(actual).` When T is a primitive, non-floating-point type, succeeds when `expected == actual`

`void assertEquals (double expected, double actual, double delta);`
`void assertEquals (String message, double expected, double actual, double delta);`
`void assertEquals (float expected, float actual, float delta);`
`void assertEquals (String message, float expected, float actual, float delta);`
	Succeeds when both expected and actual values are NaN (Not A Number), expected value is infinite, or the absolute difference between the expected and actual values is less than or equal to delta. Do not use this design as an example of how to deal with checking that floating-point values are "close enough;" it reflects the designers' ignorance of numerical issues, I'm afraid.

`void assertArrayEquals(T[] expected, T[] actual);` `void assertArrayEquals(String message, T[] expected, T[] actual);`	Succeeds when array arguments are both null or both arrays of the same length, with corresponding components equal as for assertEquals.
`void fail ();` `void fail (String message);`	Always fails. This is useful for indicating failure of a hand-written assertion-checking algorithm.

Table 1: Some useful assertion methods provided by the JUnit framework. The variants with a `message` argument use that argument in failure messages rather than their respective default failure messages.

```
@Test(timeout=1000) public void mightLoop () {
   ...
}
```

The JUnit framework will cause this test to fail after 1000 milliseconds.

4 Common code

You will sometimes find yourself writing a number of tests all of which rely on the same set of variables and objects—something like this, perhaps:

```
@Test public void test1 () {
    Graph G = new Graph ();
    for (int i = 0; i < 10; i += 1)
        G.makeVertex (i);
    G.makeEdge (1, 10);
    assertTrue (G.connected (1, 10));
}

@Test public void test2 () {
    Graph G = new Graph ();
    for (int i = 0; i < 10; i += 1)
        G.makeVertex (i);
    G.makeEdge (1, 5); G.makeEdge (5, 10);
    assertTrue (G.connected (1, 10));
}

...
```

Of course, you can create a method to handle the common setup lines:

```
Graph G;
public void setUp () {
    G = new Graph ();
    for (int i = 0; i < 10; i += 1)
        G.makeVertex (i);
}

@Test public void test1 () {
    setUp ();
    G.makeEdge (1, 10);
    assertTrue (G.connected (1, 10));
}

@Test public void test2 () {
    setUp ();
    G.makeEdge (1, 5); G.makeEdge (5, 10);
    assertTrue (G.connected (1, 10));
}

...
```

but to really show your intent, JUnit also allows you to mark `setUp` in such a way that it is executed before every test procedure:

```
Graph G;
@Before public void setUp () {
    G = new Graph ();
    for (int i = 0; i < 10; i += 1)
        G.makeVertex (i);
}

@Test public void test1 () {
    G.makeEdge (1, 10);
    assertTrue (G.connected (1, 10));
}

@Test public void test2 () {
    G.makeEdge (1, 5); G.makeEdge (5, 10);
    assertTrue (G.connected (1, 10));
}

...
```

so that explicit calls to `setUp` are unnecessary. There is also an `@After` annotation, which indicates a procedure to call after every method. This is a bit less common, since Java cleans most discarded data structures up automatically, but may be useful when dealing with files, for example.

A class containing tests can, like any class, be a subclass. In that case, the `@Before` methods of the superclass also get executed, before those of the subclass. Likewise, the `@After` methods of the superclass get executed after those of the subclass.

5 Access problems

You've probably noticed that all our test classes and annotated methods are marked '`public`'. JUnit is just a bunch of ordinary Java code, subject to the usual constraints. Since the code to be tested typically resides in a different package from the framework, these methods must normally be marked public to give JUnit the necessary access to find and run them. This is not generally a problem (although it should suggest to you that "in real life" it might be good practice to remove all these public tests classes from production versions of your system, lest you leave back doors).

A different problem is that you will generally want to test non-public methods. As long as your test classes are all in the same package as the code they are testing, this is not a problem, *unless* you want to test private methods. In that case, I suggest that you set up a kind of testing proxy:

- Add a static method with default access (thus, neither public nor private) to the class containing the private method, and put the actual testing code in that method.
- Add a public `@Test` method to one of your testing classes that simply calls this static method.