# Com S 228
# Spring 2011
# Quick JUnit summary and unit testing guidelines

**JUnit references**

**API documentation:** http://junit.sourceforge.net/javadoc/

**FAQ:** http://junit.sourceforge.net/doc/faq/faq.htm

**Unit testing guidelines**

The idea of unit testing is to write simple, fine-grained tests for the methods of one class to make sure it behaves according to its specification.

One approach to testing is to use System.out.println and examine the output, but it quickly becomes tedious to keep reading output to decide whether it is correct. Unit tests are most useful when they can be re-run with no effort. That is why we like JUnit.

The concept of an "assertion" is to state what *should be true* at a given point in the code. If the assertion fails, some output is produced, otherwise the test passes "silently".

Writing unit tests is a lot like buying a used car. Before you shell out $10,000 of your hard-earned money, you want to test drive the car and make sure everything works. You want to make sure that when you put the car in reverse, it goes backwards, and when you press the brake pedal the car stops. When you write unit tests, imagine that you are spending your own money on the software you're testing.

Many people start by writing a *test plan*. This is really just a collection of things to test. It usually comes from looking at each method and asking, "What should the object look like after I invoke this method? How can I check that it is working?" For example in the IntVector example, after you call the method set(i, 42), you know that get(i) should return 42. Write that down as a brief statement:

```
// Calling get(i) after set(i, value) should return the same value
```

Likewise if you call get(i) after set(j, value) where i is different from j, you know the value of get(i) should be unchanged:

```
// get(i) after set(j, value) should leave get(i) unchanged if i != j
```

And so on. It is useful to write such statements as comments in your test class, because then you can convert each one into a test case. Then make an empty test (that always fails) for each comment so you don't forget later to implement the test:

```
// Calling get(i) after set(i, value) should return the same value
@Test
public void testGetAfterSet1()
{
  // TODO
  fail();
}
```

Generally no further documentation is needed – we don't typically provide full javadoc for unit tests because they are not part of any public API.

**Using Junit with Eclipse**

JUnit is a test framework that makes it easy to create and run unit tests. The JUnit libraries are nicely integrated with Eclipse. To create a JUnit test, you just need to make the JUnit library available to your project.

**Add Junit 4 library to build path**

1. Right-click on project name in Package Explorer
2. Select Properties from context menu
3. Select Java Build Path in left pane
4. Select Libraries tab
5. Click the Add Library button
6. In the Add Library dialog highlight JUnit and click Next
7. From the drop-down box select JUnit4
8. Click Finish

**Creating a test class**

A JUnit test is really just an ordinary Java class. Each "test case" is just a public void method with no arguments, annotated with @Test. Each test case normally contains one or more assertions.

You will normally import several annotations and some static methods of the org.junit.Assert class. Here are the most useful:

```
import org.junit.Test;
import org.junit.Before;
import org.junit.After;
import static org.junit.Assert.assertTrue;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
```

The most useful assertions are:

```
assertTrue(condition)
assertFalse(condition)
assertEquals(expected value, actual value)
```

Here is an example of a simple test case (the sample vector v0 is being created in the @Before method, see below).

```
// Calling get(i) after set(i, value) should return the same value
@Test
public void testGetAfterSet1()
{
  for (int i = 0; i < v0.dimension(); ++i)
  {
    v0.set(i, 42);
    assertEquals(42, v0.get(i));
  }
}
```

**Running a JUnit test**

Right click on the test class and select Run As → JUnit test

In the "Failure Trace" window you can double click on any item to navigate to the assertion which failed.

**Testing exceptions**

When the correct behavior for the code is to throw a certain exception, use the "expected" option for the @Test annotation. It is followed by the class of the exception that should occur.

```
// method Foo.bar() should throw SomeException when called with argument 42
@Test(expected=some.package.SomeException.class)
public void myTest()
{
  Foo f = new Foo();
  f.bar(42);
}
```

(Note the ".class" after the exception type.) In general the method body for the test case will not contain any other assertions. The test succeeds if the correct exception is thrown, and fails otherwise.

**Debugging using JUnit**

JUnit tests can be run in Debug mode, and you can set breakpoints in the tests themselves as well as in any other code. When a test fails, it is often useful to set a breakpoint just before the failed assertion and re-run the test in Debug mode to have a look at what's happening just prior to the failure.

**@Before and @After**

It is common to have to create several objects at the beginning of each test case. You can put

this object creation in a “setup” method that is automatically run before each individual test case by annotating the method with “@Before”. Likewise there is an “@After” annotation for a method that should be run after every test case to perform cleanup such as closing sockets or files. A quick example:

```
private IntVector v0;
private IntVector v1;

@Before
public void setUp()
{
  // create a couple of test vectors
  v0 = new IntVector(4);
  v1 = new IntVector(4);
  for (int i = 0; i < 4; ++i)
  {
    v1.set(i, i);
  }
}
```

**Other assertions**

There are a number of other forms for assertions that can be seen in the javadoc for org.junit.Assert. For example, each method in Assert has an alternate form whose first argument is a String, which is a message that will be displayed upon failure, e.g.:

```
assertTrue(message, condition)
assertFalse(message, condition)
assertEquals(message, expected value, actual value)
```

Another particularly useful one to know is that for comparing expected and actual floating-point values, you can pass in a third argument to indicate “how close” the results have to be in order to be considered correct. (This is necessary because floating-point arithmetic often has round-off errors.) E.g. the assertion

```
assertEquals(98.6, result, 0.001);
```

will succeed if **`result`** is within .0001 of 98.6.