Introduction to testing Java programs with JUnit

JUnit is a framework for developing unit tests for Java classes. JUnit provides a base class called TestCase that can be extended to create a series of tests for the class you are creating, an assertion library used for evaluating the results of individual tests, and several applications that run the tests you create.

If you would like to download and install JUnit on your own computer, it can be downloaded here:

http://www.junit.org/index.htm

Further documentation is also available from the same source. I suggest reading the FAQ after you have read this introduction, then moving on to the other documents found there for descriptions of more sophisticated uses of JUnit.

JUnit should be in the default classpath on all of the CS lab computers, both Windows and Unix systems, so you shouldn’t need to do anything special to use JUnit when developing your programs there.

Creating a test class

Suppose you are working on a class called MathUtil, mainly because Java doesn’t provide a function to calculate base 2 logarithms. In order to use JUnit to test your methods, you will need to create a separate class, which we’ll call MathUtilTest. There are three things you must do as you set up this class in order to run your tests:

1. Import junit.framework.*.
2. Extend TestCase.
3. Provide a constructor with a single String parameter. This constructor should call theTestCase constructor with the parameter.

Here is a minimal class that demonstrates the structure of classes derived from TestCase:

```java
import junit.framework.*;

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

  public void testLog2()
  {
    // This version of the assertEquals method allows you to
    // specify a tolerance for equality of floating point
    // values.
    assertEquals(3.0, MathUtil.log2(8.0), 0.0000001);
    assertEquals(0.0, MathUtil.log2(1.0), 0.0000001);
    assertEquals(4.0, MathUtil.log2(16.0), 0.0000001);
  }
}
```
Note that if you don’t supply a constructor of the form indicated here, you’ll get error messages from javac complaining that TestCase( ) is not public in junit.framework.TestCase. The authors of JUnit did this to you on purpose to force you to write a constructor for your TestCase-derived class.

Running JUnit tests

One nice thing about JUnit is that you only have to write the tests, you don’t have to write the test driver. Two test drivers are supplied with JUnit, one character-based to run from the command line and one GUI-based. We’ll focus on the GUI test runner; see the JUnit documentation for details on the character-based driver.

The GUI test driver is called junit.swingui.TestRunner. It allows you to specify the test class to run, then when the “Run” button is clicked, it examines the class you’ve chosen and finds all methods whose names begin with “test.” (To get an idea of how JUnit does this, have a look at the documentation for java.lang.Class and find out about “reflection.”) It runs each of these methods, updating a progress bar based on how many tests have executed successfully.

If one of your tests fails, you will see a description of where the failure occurred, including line numbers for the method calls that were active at the time of the failure and any exceptions that were thrown.

Running tests on Windows

Many IDEs will work with JUnit. For example, NetBeans, an open source Java IDE, has built-in support for JUnit. It will generate a skeleton TestCase class from your class definition, and the error reporting is tied in with the included program editor.

If you are running from a command prompt, type a command that looks like this:

```
java junit.swingui.TestRunner MathUtilTest
```

where the name of your test class is substituted for MathUtilTest.

Running tests on Unix

On Unix, you will change directories to your source directory and execute a command that looks like this:

```
java junit.swingui.TestRunner MathUtilTest
```

where the name of your test class is substituted for MathUtilTest. This works just the way it would from a Windows command prompt.

Writing tests

Writing tests for JUnit involves using the methods of the junit.framework.assert class.
This is a reasonably small class, and you should read the javadoc material about it to get a complete view of its capabilities. Here we’ll just consider a few representative methods. Each of these methods checks some condition and reports back to the test runner whether the test failed or succeeded. The test runner uses the result to update the display. All of the methods return void (i.e. they are procedures).

```java
assertTrue(boolean)
You can supply any boolean expression to this method, and as long as the boolean expression evaluates to true, the method reports success. If the condition you want to check should be false, simply prepend the “!” negation operator to your parenthesized expression.

```java
assertTrue(String, boolean)
Same as the first form of assertTrue, except that the supplied String is printed if the assertion fails. Most methods have a version of this form with a message string.

```java
assertEquals(Object, Object)
Compares the two objects you pass in using the equals( ) method.

```java
assertNull(Object)
Succeeds if the Object reference passed in is null. There is also an assertNotNull method.

```java
fail(String)
Causes the test to fail, printing out the supplied String.
```

**Example: testing an absolute value method**

Suppose we have written an absoluteValue method for Integer objects in our MathUtil class. Here is a JUnit test for the method.

```java
public void testAbsoluteValue(){
    Integer i = new Integer(5);
    Integer j = new Integer(-3);
    Integer k = new Integer(0);
    Integer result = null;

    assertNotNull("Non-null test", MathUtil.absoluteValue(i));
    assertEquals("Positive value test", i, MathUtil.absoluteValue(i));
    assertEquals("Zero test", k, MathUtil.absoluteValue(k));
    assertTrue("Negative value test",
        (-j.intValue()) ==
        MathUtil.absoluteValue(j).intValue());
}
```
Appendix: MathUtil.java and MathUtilTest.java

/**
 * Class of static mathematics utility functions not provided by
 * java.Math
 * @author: Lewis Barnett
 * @version: 1.0
 */

import java.io.*;
import java.lang.Math.*;

public class MathUtil
{

/**
 * Multiply by this constant to convert from natural logarithm
 * (as supplied by java.lang.Math.log) to log base 2.
 * @see MathUtil#log2
 */
public static final double NATLOG_TO_LOG2 = 1.4426950408889634;

/**
 * Multiply by this constant to convert from log base 10 to
 * log base 2. Not too relevant here, since java.lang.Math.log()
 * gives you the natural log, but included for the sake of
 * completeness.
 */
public static final double LOG10_TO_LOG2 = 3.3219280948873626;

/**
 * Create a string containing the binary translation of a
 * non-negative integer. The string contains the binary
 * representation right-justified in a field of specified
 * width, filled with leading zeroes if necessary.
 * @param n The int to translate
 * @param width The width of the resulting string.
 * @return The string containing the binary representation.
 * Preconditions: n >= 0 and width > 0
 * Postconditions: none
 */
public static String intToBinaryString(int n, int width) {
    if (n < 0 || width <= 0) {
        return "";// Base case; no action necessary
    } else {
        String retStr = intToBinaryString(n/2, width - 1);
        if (isEven(n)) {
            retStr = retStr + "0";
        } else {
            retStr = retStr + "1";
        }
        return retStr;
    }
}
retStr = retStr + "1";
}
return retStr;

/**
  * log base 2 function.
  * @param x number to calculate log base 2 of.
  * @returns log base 2 of x as a double.
  *
  * Preconditions: x > 0.0
  * Postconditions: none
  */
public static double log2 (double x)
{
    return NATLOG_TO_LOG2 * Math.log(x);
}

/**
  * Print out the binary translation of a non-negative integer.
  *
  * @param n The number to translate
  * @param width The width of the field in which n should
  * be printed. The printed value is filled
  * to the left with 0.
  *
  * Preconditions: n >= 0 and width > 0
  * Postconditions: none
  */
public static void printBinary(int n, int width) {
    if (n < 0 || width <= 0) {
        return;// Base case; no action necessary
    } else {
        printBinary(n/2, width - 1);
        if (isEven(n)) {
            System.out.print(0);
        } else {
            System.out.print(1);
        }
    }
}

/**
  * Determine whether an integer is even.
  * @param n An integer.
  * @return <code>true</code> if n is even, <code>false</code>
  * if not.
  *
  * Pre/postconditions: none
  */
public static boolean isEven(int n) {
    return (n % 2 == 0);
}

/**
 * Determine whether an integer is odd.
 * @param n An integer.
 * @return <code>true</code> if n is odd, <code>false</code> if not.
 * @pre/postconditions: none
 */
public static boolean isOdd(int n) {
    return (n % 2 != 0);
}

import junit.framework.*;

public class MathUtilTest extends TestCase {
    /**
     * JUnit test cases must have a constructor with a string parameter, otherwise an error message about TestCase being private in the aren't class will be generated by javac. Calling the TestCase constructor is the only required action.
     */
    public MathUtilTest(String name) {
        super(name);
    }

    /**
     * Test the intToBinaryString method by comparing its result to the correct string representation.
     */
    public void testIntToBinaryString() {
        // Note that this class is for demonstration purposes and that this is a woefully inadequate set of tests for this method...
        assertEquals("0001", MathUtil.intToBinaryString(1, 4));
    }
}
/**
 * Test the log2 method by comparing its output to known correct values.
 */
public void testLog2()
{
    // This version of the assertEquals method allows you to specify a tolerance for equality of floating point values.
    assertEquals(3.0, MathUtil.log2(8.0), 0.0000001);
    assertEquals(0.0, MathUtil.log2(1.0), 0.0000001);
    assertEquals(4.0, MathUtil.log2(16.0), 0.0000001);
}

/**
 * Test the isEven method by trying it with various values.
 */
public void testIsEven() {
    assertTrue(MathUtil.isEven(2));
    assertTrue(MathUtil.isEven(0));
    assertTrue(MathUtil.isEven(43567342));
    assertTrue(!MathUtil.isEven(1));
    assertTrue(!MathUtil.isEven(989293847));
    assertTrue(!MathUtil.isEven(-1));
}

/**
 * Test the isOdd method by trying it with various values.
 */
public void testIsOdd() {
    assertTrue(!MathUtil.isOdd(2));
    assertTrue(!MathUtil.isOdd(0));
    assertTrue(!MathUtil.isOdd(43567342));
    assertTrue(MathUtil.isOdd(1));
    assertTrue(MathUtil.isOdd(989293847));
    assertTrue(MathUtil.isOdd(-1));
}