ITI 1120
Lab #6
Library Classes and JUnit Testing

Daniel Amyot, Diana Inkpen, Alan Williams

Agenda

- Topics in this lab:
  - Methods
  - Library classes
  - Testing with JUnit
- In this lab, you are going to create your own library class, and then learn how to create and run tests on the methods in your class.

A library class

- Create a class called **`MyMath`**, that implements the following methods, WITHOUT using the Math class from the Java software development kit.

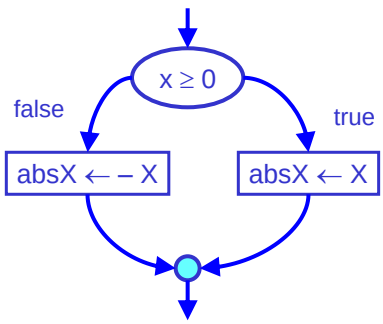
1. A method that returns the absolute value of an integer x.

2. A method that calculates x^y for integers x and y.
   - Use a loop the multiplies x by itself y times.
   - Assume $y \geq 0$

- No reading from the keyboard or printing to the screen!

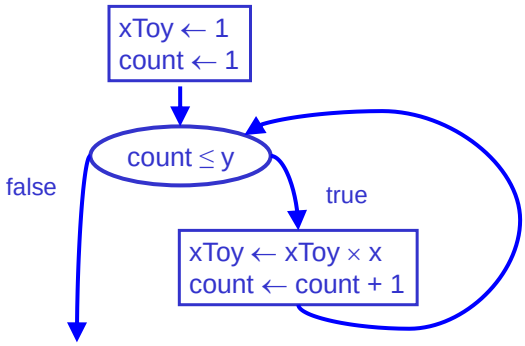
The absolute value algorithm

GIVENS: x (an integer)
RESULT: absX (the absolute value of x)
HEADER: absX ← abs(X)
BODY:



The exponentiation algorithm

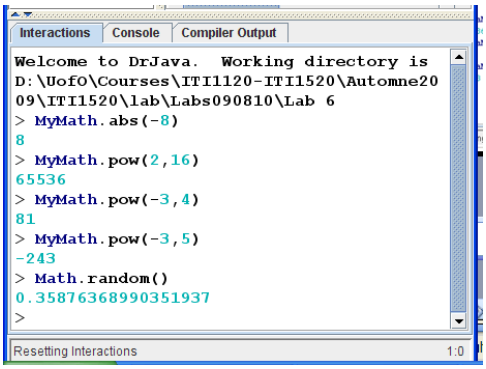
GIVENS:	x, y	(two integers)
RESULT:	xToy	(the value of x^y)
INTERMEDIATES:	count	(counts times x has been multiplied by itself)
HEADER:	xToy ← pow(x, y)	
BODY:		



Interacting with the Library Class

- Once you have successfully compiled your class, you can invoke the methods from other methods, including main() located in another class (where you may interact with the user).

- It is easier to call the methods created using the interaction window provided by Dr Java. Do not forget to specify the name of the class first:
  - E.g., MyMath.abs(-8)
- Try a few calls!
- You can do the same with Java classes:
  - E.g., Math.random()



Testing

- We're now going to do some testing of the methods in the class `MyMath`.

- When doing testing:
  1. Call a method with some test values as the parameters.
  2. Observe the results.
  3. See if the results are as expected.

- This means that you need to choose some test values for which you can determine the expected results using some other means.

Manual Testing

- Try testing your class by running the main method and typing various values for `x` and `y`.
  - Choose your values carefully, to try and take different paths through the code.

- For the absolute value method:
  - try values of `x` that are less than zero, equal to zero, and greater than zero.

- For the exponentiation method:
  - try `y = 0`, `y = 1`, `y > 1`
  - try `x < 0`, `x = 0`, `x = 1`, `x > 1`

    - What are some issues encountered when using this approach? (Especially when changing the code)

Testing with JUnit

- The interactions in the DrJava interface are entertaining but inefficient while we are dealing with complex methods. Many manual tests need to be performed each time we modify the method.
- JUnit is a set of classes that you can add to the Java software development kit to make testing easier.
- JUnit was coined from "Java unit".
  - Unit testing is when you testing parts of a complete application, such as methods in library classes.
- More information:
  - http://junit.org

Key JUnit concepts

- Test verdict: each test case can pass (green) or fail (red).
- Test case: an experiment to see if a method produces the correct result for a set of parameter values.
  - Consists of a method contained in a test class.
  - You can add as many methods as you wish in a test class.
- Test class: contains a set of test cases for a class.
  - Usually, there is a corresponding test class for each class you want to test.

Setting up a test class

- Be sure that your **MyMath** class is already loaded into Dr. Java
- In Dr. Java, from the 'File' menu, select "Create new JUnit test case…"
- You will be asked for the name of the test class. Enter **MyMathTest**.
  - It is important to include the word "Test" as part of the class name; this is how JUnit knows how to find test classes.
  - JUnit version 4 uses a different notation (@Test). We will be using version 3.
- The result should look similar to the code on the next slide.

The test class

```
import junit.framework.TestCase;

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called
 * when running the test with JUnit.
 */
public class MyMathTest extends TestCase
{
    /**
     * A test method.
     * (Replace "X" with a name describing the test.  You may
     * write as many "testSomething" methods in this class as
     * you wish, and each one will be called when running
     * JUnit over this class.)
     */
    public void testX()
    {
    }
}
```

Features of the test class

- `import junit.framework.TestCase;`
  - We will use the class TestCase from the JUnit collection of classes.
- After the name of the class, there is:
  `extends TestCase`
  - This says that the class is going to act like a test case, as defined by JUnit.
- An empty test method, which returns a `void` result and has no parameters.
  - JUnit will call this method as a test case.
  - The method should be renamed to `testXXX` where `XXX` gives some idea of the purpose of the test case.

Creating a test method

- Let's create a test for the absolute value method.
- The absolute value method takes one value as a parameter; we need to provide test data for that parameter.
  - Example: if we call `abs(-4)`, we should get `4` as a result.
  - Set up 3 values:
    - `testValueX`: the `-4` that will be the test data for our method.
    - `expected`: the result we expect: `4`
    - `actual`: the result that `abs(-4)` actually returns to us.

Checking the result

- An important part of testing is checking that the result you get matches what you expect.
- The result is a verdict: pass or fail.
- With JUnit, there is a method in the `Assert` class called `Assert.assertEquals( expected, actual )`
  - If the two values `expected` and `actual` are equal, the method will return and execution will continue.
  - If the values are not equal, the test case will be declared to have failed at this point. Execution of the test method stops.
- If you reach the end of a test method, and no failures have occurred, the test will be declared to have passed.

The `assertEquals` method

- There are several versions of the `assertEquals` method, so that you can test values of various types
  - The expected value is always the first parameter, and the actual value is always the second parameter

```
Assert.assertEquals( int, int )
Assert.assertEquals( char, char )
Assert.assertEquals( boolean, boolean )
Assert.assertEquals( String, String )
Assert.assertEquals( double, double, double )
```

  - Testing equality of double variables is a special case; remember that you should compare that they are "sufficiently close" to each other.
  - The third parameter specifies the maximum difference to accept as equal (a number like `0.00001`) (Section 6 of notes which talks about this subject).

Enter a test method

- Replace the empty `testX` method with the following:

```
public void testAbsNegative()
 {
    // Purpose:  test that abs() works for a value < 0.

    int testValueX; // Test data for calling method
    int expected;   // Value we expect to see as result
    int actual;     // Actual value that method returns

    //Given test value
    testValueX = -4;
    //Expected result and actual result
    expected = 4;
    actual = MyMath.abs( testValueX );
    //Verify if we have obtained expected result
    Assert.assertEquals( expected, actual );
 }
```

One more addition

- Add the following at the top of your test class:

```
import junit.framework.Assert;
```

- This is so we can use the **`assertEquals`** method from the JUnit class **`Assert`**.

Compiling the tests

- Be sure that you have both of the classes **`Math`** and **`MyMath`** loaded into Dr. Java.
- Click the 'Compile' button.
- After a few seconds, the 'Test' button should be enabled. (Unless there a problem arises during compilation…)

Potential problems with DrJava

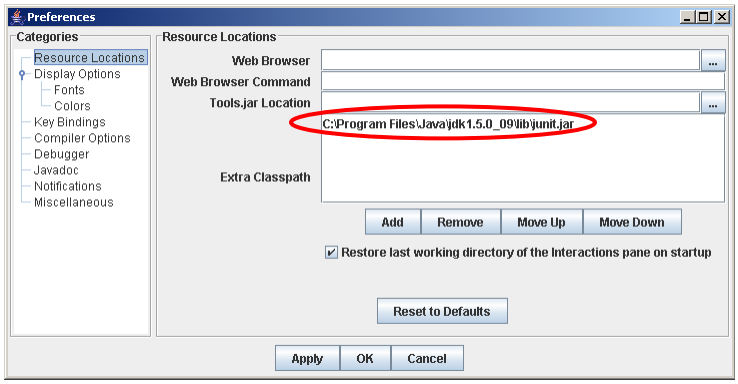
- If default installation does not work with the following error:

`Error: package junit.framework does not exist`

You will need to add the junit.jar to your Java installation and indicate where it is located in DrJava.

- Download junit.jar from the following link: (provide link)
- Find the **lib** folder of your Java installation (ex: **C:\Program Files\Java\jdk1.5.0_09\lib**)
- Move the junit.jar file into **lib** folder
- Start DrJava

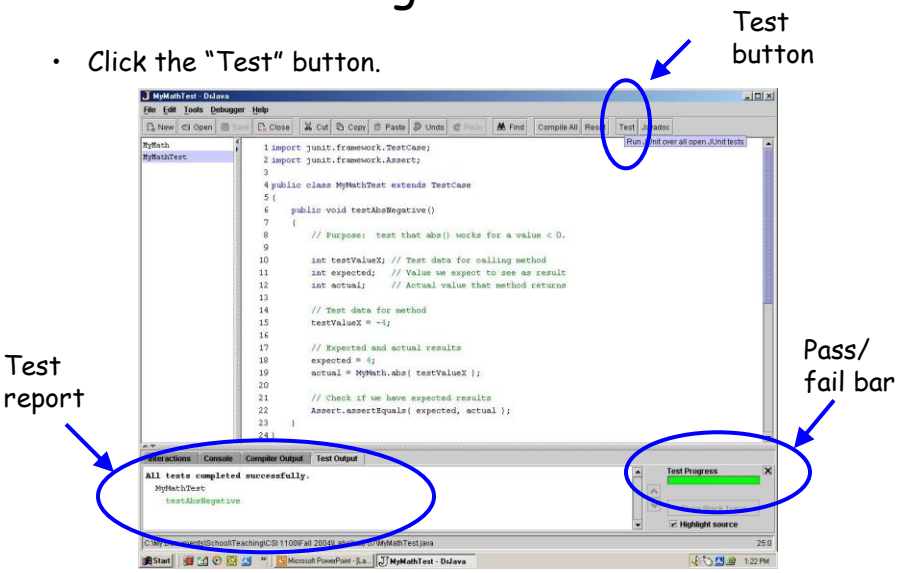
Potential problems with DrJava



Edit>Preferences> Click in the zone "Extra Classpath"(It should be located in the first tab, otherwise just select "Resource Locations" in the left panel)> Click on Add > Add junit.jar file located in your **lib** folder and click on "Apply"

Running the Tests

- Click the "Test" button.



Results from JUnit

- JUnit always shows a coloured bar after a test run:
  - Green: all tests passed
  - Red: at least one test failed.
  - [JUnit slogan: "**Keep the bar green to keep the code clean !**."]

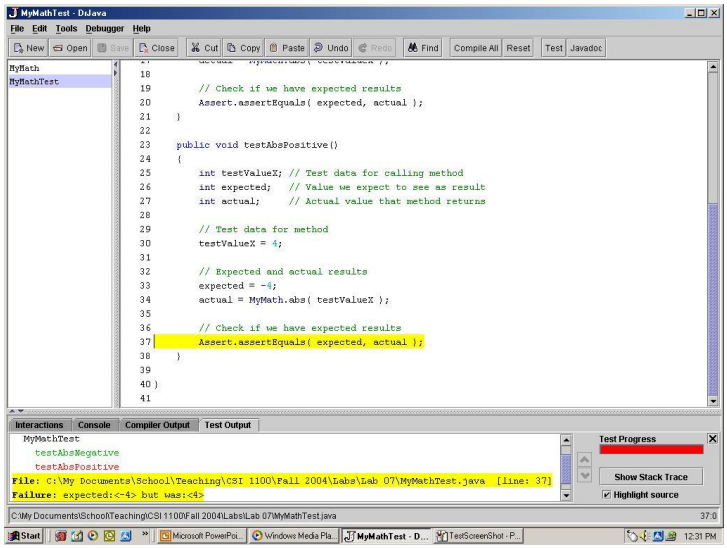
- You also get a test report that lists any failure messages.

Adding a test that will fail

- Create a second test by adding another method to the test class.

- This time, we deliberately want the test to fail, just to see what happens.
  - Call `abs(4)` and expect `-4` as a result (which is wrong!)

Result from a test failure



The test report

```
MyMathTest
    testAbsNegative
    testAbsPositive
File: C:\...\MyMathTest.java  [line: 39]
Failure: expected:<-4> but was:<4>
```

- A test name shown in green has passed; one shown in red has failed.
- For each failure, you get
  - The line number in the test case where the failure occurred.
  - A report from JUnit as to the result of the comparison
    - The values are enclosed in <> in case you need to check for extra spaces, etc.

Try creating the following test methods

- Absolute value
  - test data: 0
  - test data: >= 1
- Exponentiation:
  - x > 1, x = 1, x = 0, x < 0
  - y = 0, y = 1, y > 1, y odd/even when x<0

If time permits

- Add the following mathematical methods to your class `MyMath` and appropriate tests in `MyMathTest`
  - `square(int n)` – returns the square of n
    - Create tests with n=40000 and n=50000. What do you notice?
  - `round(double n)` – return the integer value rounded to n.
    - Assumption: the rounded value for n can be represented with an integer between about -2 billion to + 2 billion.
    - Note: `Math.floor(x)` returns a real number representing an integer less then or equal to the real x:
      - `Math.floor(3.6)` returns 3.0
      - `Math.floor(-3.2)` returns -4.0
    - Examples
      - `MyMath.round( 3.9 )` gives 4
      - `MyMath.round( 3.1 )` gives 3
      - `MyMath.round( 3.5 )` gives 4
      - `MyMath.round( -1.2 )` gives -1
      - `MyMath.round( -1.5 )` gives -1
      - `MyMath.round( -1.6 )` gives -2

`String` vs. `char[]`

- Similarities:
  - both are collections of characters
  - both indexed from `0` up to `length - 1`
  - both are reference variables
    - no `==` comparison!

`String` vs. `char[]`

- Differences:
  - Access of single character: `str.charAt(i)` vs `array[i]`
  - Strings cannot be modified internally once they are created
    - No equivalent of `array[i] = 'x'`
  - String variables can be assigned constant strings where using new is optional

```
String str;
str = "abc";
str = new String("def" );
```

  - Most operations on Strings are done with methods

```
array.length        // not a method call; no ( )
str.length( )       // method call; ( ) required
```

Conversions: `String` ↔ `char[]`

```
char[] array;
char[] array2;
...
// Create String from array
String str = new String( array );
// Create array from String
array2 = str.toCharArray( );
```

Common Methods of `String`

- Review the various methods available in the String class:

  http://java.sun.com/javase/6/docs/api/java/lang/String.html

  - `charAt(…)`, `indexOf(…)`, `length(…)`
  - `toCharArray(…)`
  - `equals(…)`, `compareTo(…)`
  - `concat(…)`, `substring(…)`,
  - `toLowerCase(…)`, `toUpperCase(…)`
  - …

Careful!

- For String character strings:
  - Use the double quotes (")
  - Do not use: ‟, ”, « or »

- For character arrays:
  - Use the single quote (')
  - Do not use: ‘ or ’

Exercises with String and char[]

- Using the Dr Java *Interactions* window
  - Using String constants of you choice, try to call common methods from the String class. For example:

```
"012345678".length()
"12345".charAt(4) and then "12345".indexOf('4')
"minuscule".toUpperCase()
"A".compareTo("a")   (et and the reverse?)
"happy".concat(" ").concat("holidays")

String s1="ab"; String s2="ab"; s1.equals(s2) (and s1==s2 ?)
String s3="thanks"; char[] tabS3= s3.toCharArray(); tabS3[0]
```

If you feel motivated …

- Develop and create a method **`countLowerCase`** which counts the number of lower case characters in a String object. Test it using the Interaction Window.
- (More advanced) Develop and write a method **`reverseUpperLower`** which receives a String object as a parameter and exchanges lower case characters to uppercase and vice versa.