Unit testing in Java

Using Junit 4.*

Or

Dilbert writes a test class

The Calculator

- The sample calculator is very simple, inefficient, and even has a few bugs;
- it only manipulates integers and stores the result in a static variable.
- Substract method does not return a valid result,
- multiply is not implemented yet,
- and it looks like there is a bug on the squareRoot method: It loops infinitely.
- You can switch this calculator on and off and you can clear the result.

These bugs will help illustrate the efficiency of the tests in JUnit 4.

Here is the offending code

```
public class Calculator {
private static int result;

public void add(int n) {
result = result + n;    }

public void substract(int n) {
 result = result - 1;}

public void multiply(int n) {}
```

where the result is stored

Bug : should be
result = result – n

Not implemented yet

```
public void divide(int n) {
result = result / n;    }

public void square(int n) {
result = n * n;    }

public void squareRoot(int n) {
for (; ;) ;}

public void clear() {
result = 0;    }
```

Bug :
loops indefinitely

Clears the result

```
 public void switchOn()
    {result = 0;}
```

Switch on the screen, display "hello", beep and do other things that calculator do nowadays

```
public void switchOff() { }
```

Display "bye bye", beep, switch off the screen

Not important

```
public int getResult() {
    return result;    }
}
```

Lets write that test class

- This test class also has some flaws☺
- It does not test all the methods and
- it looks like there is a bug in the testDivide method:

  (8/2 is not equal to 5)
- Because the implementation of multiply is not ready, its test is written but ignored.

Annotations in J2SE 5

- J2SE 5 introduces the Metadata feature (data about data)
- Annotations allow you to add decorations to your code
- Annotations are used for code documentation, compiler processing (@Deprecated), code generation, runtime processing

Dilbert’s test Code

```
package junit4;

import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;
```

Access
JUNIT 4.0
and our code

Assert.*
gives us access to
“First Order Predicate Logic”
OR
“Test Questions”

```
public class CalculatorTest {

private static Calculator calculator =
new Calculator();


@Before
  public void clearCalculator() {
    calculator.clear();
  }
```

@Before is an Annotation

Tells the “test executer” to execute this routine before any test routine

Fixtures

- Fixtures are methods to initialize and release any common objects during tests.
- JUnit 4 uses @Before and @After annotations.
- These methods can be called by any name (clearCalculator() in our example).

```
@Test
   public void add() {
     calculator.add(1);
     calculator.add(1);
    assertEquals
       (calculator.getResult(), 2);
   }

   @Test
   public void subtract() {
     calculator.add(10);
     calculator.subtract(2);
    assertEquals
       (calculator.getResult(), 8);
   }
```

@Test

says

"I am a test"

assertEquals

Asks

"Are two things equal?"

Remember @Before is execute before any test routine.

Tests

- Test methods use the same name as the method that they are testing.
- A test method must return void and have no parameters.
- With JUnit 4 this is controlled at runtime and throws an exception if not respected.

AssertEquals --Uses Autoboxing

Autoboxing: is the automatic conversion the Java compiler makes between the primitive (basic) types and their corresponding object wrapper classes (eg, int and Integer, double and Double, etc).

With Autoboxing

```
int i; Integer j;
i = 1; j = 2; i = j; j = i;
```

Without Autoboxing

```
int i; Integer j;
i = 1; j = new Integer(2);
i = j.intValue();
j = new Integer(i);
```

```
@Test
 public void divide() {
   calculator.add(8);
   calculator.divide(2);
   assert calculator.
        getResult() == 5;
 }
```

Dilbert writes a test …aaarrghh!

Assert keyword is just short-hand for assertEquals()

```
@Test(expected =
ArithmeticException.class)
 public void divideByZero()
{
   calculator.divide(0);
 }
```

Here we expect an exception

The @Test annotation supports the optional expected parameters.

It declares that a test method should throw an exception. If it doesn't or if it throws a different exception than the one declared, the test fails.

```
@Ignore("not ready yet")
@Test
  public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals
         (calculator.getResult
(), 100);
  }
}
```

Remember that the multiply method is not implemented.

@Ignore takes an optional parameter (a String) if you want to record why a test is being ignored.

Run the Tests

```
>javac Calculator.java CalculatorTest.java; java
    org.junit.runner.JUnitCore CalculatorTest
```

```
JUnit version 4.1
...E.EI                    (E is expected; I is ignored)
There were 2 failures:
1)  subtract(junit4.CalculatorTest)
java.lang.AssertionError: expected:<9> but was:<8>
    at org.junit.Assert.fail(Assert.java:69)
2) divide(junit4.CalculatorTest)
java.lang.AssertionError at
    junit4.CalculatorTest.divide(CalculatorTest.java:40) FAILURES!!!
Tests run: 4, Failures: 2
```