Unit Testing

Demo

Testing Strings

Testing Strings

This test **fails**!

- When Testing Strings:
  - **NEVER** use ==
  - This will be true for all non-primitive comparisons
- Using == checks if the two values store the same reference
- Strings can be.. weird.

```java
package week2;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Testing {

    @Test
    public void testStringsBadExample() {
        String str1 = "ab ".strip();
        String str2 = "ab ".strip();
        // Never use == to compare Strings
        assertTrue("strings equal?", str1 == str2);
    }

}
```

Testing Strings

- Test Strings using the equals method
  - Compares the *values* of the Strings

```java
package week2;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Testing {

    @Test
    public void testStringsGoodExample() {
        String str1 = "ab";
        String str2 = "ab";
        assertTrue("strings equal?", str1.equals(str2));
    }

}
```

Testing Strings

- In this example, we have 2 arguments for the assertTrue call
- If you pass a String and a boolean to assertTrue
  - The String will be printed *if* the test fails
- You can provide information here to help you debug the issue

```java
package week2;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Testing {

    @Test
    public void testStringsGoodExample() {
        String str1 = "ab";
        String str2 = "ab";
        assertTrue("strings equal?", str1.equals(str2));
    }

}
```

```java
package week2;

public class PlusMinus {
    public static String letter(int score){
        int tens=score/10;
        if (tens>=9){
            return "A";
        } else if(tens>=8){
            return "B";
        } else if(tens>=7){
            return "C";
        } else if(tens>=6){
            return "D";
        } else {
            return "F";
        }
    }
    public static String plusMinus(int score){
        int ones=score%10;
        if (ones>=7){
            return "+";
        } else if (ones>2){
            return "";
        } else {
            return "-";
        }
    }
    public static void main(String[] args) {
        System.out.println(letter(95));
        System.out.println(letter(78));
        System.out.println(letter(51));
    }

}
```

Testing Strings

- Let's expand our letter grade example to include plusses and minuses
- The plusMinus method should return the appropriate value "+", "-", or "" for the input
  - 87-89 -> B+
  - 83-86 -> B
  - 80-82 -> B-

Testing Strings

- To test the plusMinus method, we'll write a test class
- This is a good start with 3 test cases (We would write a lot more for true testing)
- Using the equals method to compare our Strings
- We run the test and our code passes! 😀

```
public class PlusMinusTests {

    @Test
    public void testPlusMinus() {
        String pm = PlusMinus.plusMinus(95);
        assertTrue("95 There should be no +-, got: " + pm, pm.equals(""));
        pm = PlusMinus.plusMinus(78);
        assertTrue("78 It should be +, got: " + pm, pm.equals("+"));
        pm = PlusMinus.plusMinus(51);
        assertTrue("51 It should be no -, got: " + pm, pm.equals("-"));
    }
}
```

Testing Strings

- Let's add one more test to be sure. We'll check the edge case of 100
- Oh no, the test fails! 😭
- The poor student with 100 was given an A-!! We have a bug!
- We passed 3/4 test but unit testing, and the student with an A-, demand perfection

```
public class PlusMinusTests {

    @Test
    public void testPlusMinus() {
        String pm = PlusMinus.plusMinus(95);
        assertTrue("95 There should be no +-, got: " + pm, pm.equals(""));
        pm = PlusMinus.plusMinus(78);
        assertTrue("78 It should be +, got: " + pm, pm.equals("+"));
        pm = PlusMinus.plusMinus(51);
        assertTrue("51 It should be -, got: " + pm, pm.equals("-"));
        pm = PlusMinus.plusMinus(100);
        assertTrue("100 It should be +, got: " + pm, pm.equals("+"));
    }
}
```

Testing Strings

- The goal of unit testing is to expose any bugs that exist
- This unit test did a great job exposing a bug
- Write unit tests for every possible bug you can think of
- Edit your code until it passes all your tests

```
public class PlusMinusTests {

    @Test
    public void testPlusMinus() {
        String pm = PlusMinus.plusMinus(95);
        assertTrue("95 There should be no +-, got: " + pm, pm.equals(""));
        pm = PlusMinus.plusMinus(78);
        assertTrue("78 It should be +, got: " + pm, pm.equals("+"));
        pm = PlusMinus.plusMinus(51);
        assertTrue("51 It should be -, got: " + pm, pm.equals("-"));
        pm = PlusMinus.plusMinus(100);
        assertTrue("100 It should be +, got: " + pm, pm.equals("+"));
    }
}
```

Testing Strings

```
public static String plusMinus(int score){
    if(score==100){
        return "+";
    }
    int ones=score%10;
    if (ones>=7){
        return "+";
    } else if (ones>2){
        return "";
    } else {
        return "-";
    }
}
```

- That's better
- Edge cases will often have special conditions in your code
- This code passes our test cases and the student gets the A+ they've earned

Testing Doubles

Testing Doubles

- This test fails.
- Why??

```
package week2;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Testing {

    @Test
    public void testDoublesBad() {
       assertTrue(0.3 == 0.1 * 3.0);
    }

}
```

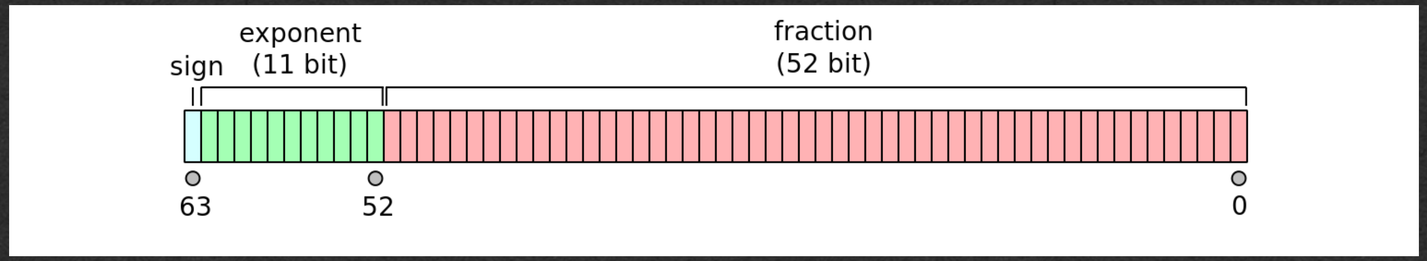
Testing Doubles

- If we print 0.1*3.0
- We get 0.30000000000000004
- Which is not == 0.3

```java
package week2;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

public class Testing {

    @Test
    public void testDoublesBad() {
       assertTrue(0.3 == 0.1 * 3.0);
    }

}
```

Testing Doubles

- A double is stored using a 64 bit representation
- If the number doesn't fit in those 64 bits, it must be truncated
- We lose precision when 64 bits is not enough



https://en.wikipedia.org/wiki/Double-precision_floating-point_format

Testing Doubles

- Decimal values are represented in binary as fractions of powers of 2
  - Ex. 0.11 == 1/2 + 1/4 == 3/4
- In decimal we have values that cannot be stored without truncation
  - Ex. 1/3 != 0.33333333333333333333333333333333333333333333333333
- Values such as 0.1 cannot be represented as a sum of powers of 2
  - 0.1 (base 10) != 0.0001100110011001100110011001100110011001100110011001 (base 2)
  - 64 bits is not enough to store an infinitely repeating decimal. We must truncate

Testing Doubles

- The solution?
  - Allow for some tolerance to accept doubles that are within truncations errors of each other
- Check that the difference between the doubles is less than a small number

```
public class Testing {

    private final double EPSILON = 0.001;

    public void compareDoubles(double d1, double d2) {
       assertTrue(Math.abs(d1 - d2) < EPSILON);
    }

    @Test
    public void testDoubles() {
        compareDoubles(1.0, 1.0);
        compareDoubles(0.3, 0.1 * 3.0);
    }

}
```

Testing Doubles

- We define the small number as a constant using the final keyword
  - Constants should be named with all capital letters
- This is our first private variable - it cannot be used outside of this class

```
public class Testing {

    private final double EPSILON = 0.001;

    public void compareDoubles(double d1, double d2) {
       assertTrue(Math.abs(d1 - d2) < EPSILON);
    }

    @Test
    public void testDoubles() {
        compareDoubles(1.0, 1.0);
        compareDoubles(0.3, 0.1 * 3.0);
    }

}
```

Testing Doubles

- Choose a small number that is:
  - Large enough to allow for truncation errors
  - Small enough to not interfere with the test (eg. 10.0 will pass code that is off by 9.9)
  - Can be different for different applications

```
public class Testing {

    private final double EPSILON = 0.001;

    public void compareDoubles(double d1, double d2) {
       assertTrue(Math.abs(d1 - d2) < EPSILON);
    }

    @Test
    public void testDoubles() {
        compareDoubles(1.0, 1.0);
        compareDoubles(0.3, 0.1 * 3.0);
    }

}
```

Testing Doubles

- Be sure to take the **absolute value** of the difference
  - If d1 is 5.0 and d2 is 1000000.0
  - Te difference is -999995.0 which is less than 0.001!

```
public class Testing {

    private final double EPSILON = 0.001;

    public void compareDoubles(double d1, double d2) {
       assertTrue(Math.abs(d1 - d2) < EPSILON);
    }

    @Test
    public void testDoubles() {
        compareDoubles(1.0, 1.0);
        compareDoubles(0.3, 0.1 * 3.0);
    }

}
```