

## Lecture 10 - Oct. 8

### TDD with JUnit, Object Equality.

***JUnit Test: Exception Expected vs. Not  
Using Loops in JUnit Test Methods***

## Announcements/Reminders

- **ProgTest1** tomorrow
- **ProgTest1** review session materials released
- **Written Test 1** results released
- **Lab1** solution released
- **Lab2** released

## A Default Test Case that **Fails**

The result of running a test is considered:

- **Failure** if either
  - an **assertion failure** (e.g., caused by `fail`, `assertTrue`, `assertEquals`) occurs
  - an unexpected **exception** (e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`) thrown
- **Success** if neither **assertion failures** nor (unexpected) **exceptions** occur.

```
TestCounter.java ✕
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7         //fail("Not yet implemented");
8     }
9 }
10
```

Q: What is the easiest way to making this test **pass**?

## Examples: JUnit Assertions (1)

Consider the following class:

```
public class Point {  
    private int x; private int y;  
    public Point(int x, int y) { this.x = x; this.y = y; }  
    public int getX() { return this.x; }  
    public int getY() { return this.y; }  
}
```

Then consider these assertions. Do they *pass* or *fail*?

```
Point p;  
assertNull(p);  
assertTrue(p == null);  
assertFalse(p != null);  
assertEquals(3, p.getX());  
p = new Point(3, 4);  
assertNull(p);  
assertTrue(p == null);  
assertFalse(p != null);  
assertEquals(3, p.getX());  
assertTrue(p.getX() == 3 && p.getY() == 4);
```

## Examples: JUnit Assertions (2)

Consider the following class:

```
class Circle {  
    double radius;  
    Circle(double radius) { this.radius = radius; }  
    int getArea() { return 3.14 * radius * radius; }  
}
```

Then consider these assertions. Do they **pass** or **fail**?

```
Circle c = new Circle(3.4);  
assertEquals(36.2984, c.getArea(), 0.01);
```

$$\text{expected} - \epsilon \leq \text{actual} \leq \text{expected} + \epsilon$$

public void increment() throws VTL --.

```
1  @Test
2  public void testIncAfterCreation() {
3      Counter c = new Counter();
4      assertEquals(Counter.MIN_VALUE, c.getValue());
5      try { c.v == 0;
6          c.increment();
7          assertEquals(1, c.getValue());
8      }
9      catch (ValueTooLargeException e) {
10         /* Exception is not expected to be thrown. */
11         fail("ValueTooLargeException is not expected.");
12     }
13 }
```

Annotations in the code:  
- Line 5: *C.v == 0* (handwritten)  
- Line 6: *c.increment()* is circled in blue. A red arrow points to it with the text "wrong: throw VTL unexpectedly".  
- Line 7: *assertEquals(1, c.getValue());* is underlined in purple.  
- Line 9: *catch (ValueTooLargeException e)* is boxed in red.  
- Line 11: *fail("ValueTooLargeException is not expected.");* is boxed in yellow.  
- A red arrow points from the red box on line 9 to the text "replace by VTLSE" on the right.  
- A red arrow points from the yellow box on line 11 to the text "L compilation error" on the right.  
- A purple arrow points from the *fail* method call to the text "no assertion fail. no unexpected." at the bottom.

replace by  
VTLSE

↳ compilation  
error

① No VTLSE occurred → expected  
→ test value is incremented

# JUnit: An Exception Expected

```
1 @Test
2 public void testDecFromMinValue() {
3     Counter c = new Counter();
4     assertEquals(Counter.MIN_VALUE, c.getValue());
5     try { c.v -= 0
6         ① c.decrement();
7         ② fail("ValueTooSmallException is expected.");
8     }
9     catch (ValueTooSmallException e) {
10         /* Exception is expected to be thrown. */
11     }
12 }
```

*Expect. VTSE should occur*

*pass*

*\* being able to move to next line means the expected VTSE did not occ.*

What if increment is implemented correctly?

## Expected Behaviour:

Calling `c.decrement()` when `c.value` is 0 should trigger a `ValueTooSmallException`.

```
1 @Test
2 public void testDecFromMinValue() {
3     Counter c = new Counter();
4     assertEquals(Counter.MIN_VALUE, c.getValue());
5     try {
6         c.decrement();
7         fail("ValueTooSmallException is expected.");
8     }
9     catch (ValueTooSmallException e) {
10         /* Exception is expected to be thrown. */
11     }
12 }
```

What if increment is implemented incorrectly?

e.g., It only throws VTSE when `c.value < Counter.MIN_VALUE`

# Running JUnit Test 2 on Correct Implementation

```
public void decrement() throws ValueTooSmallException {  
    ⑤ if (value == Counter.MIN_VALUE) {  
        ⑥ throw new ValueTooSmallException("counter value is " + value);  
    }  
    X else { value --; }  
}
```

```
1  @Test  
2  public void testDecFromMinValue() {  
3  ① Counter c = new Counter(); | c.v == 0  
4  ② assertEquals(Counter.MIN_VALUE, c.getValue());  
5  ③ try {  
6  ④ c.decrement();  
7  X fail("ValueTooSmallException is expected.");  
8  }  
9  ⑦ catch (ValueTooSmallException e) {  
10  → /* Exception is expected to be thrown. */  
11  }  
12 }
```

NTSE occurred  
↳ expc flow disrupted  
↳ pass!

pass.

←  
—  
—  
—



# Running JUnit Test 2 on Incorrect Implementation



```
public void decrement() throws ValueTooSmallException {  
    if (value < Counter.MIN_VALUE) {  
        X throw new ValueTooSmallException("counter value is " + value);  
    }  
    else { value --; }  
}
```

*Handwritten notes:*  
- Red arrow points to `if` condition.  
- Red arrow points to `else` block.  
- `0 → -1` written below the `value --;` line.

*\* exec flow normal: no exception occurred.*

```
1 @Test  
2 public void testDecFromMinValue() {  
3     ① Counter c = new Counter();  
4     ② assertEquals(Counter.MIN_VALUE, c.getValue());  
5     ③ try {  
6         ④ c.decrement();  
7         * fail("ValueTooSmallException is expected.");  
8     }  
9     X catch (ValueTooSmallException e) {  
10        /* Exception is expected to be thrown. */  
11    }  
12 }
```

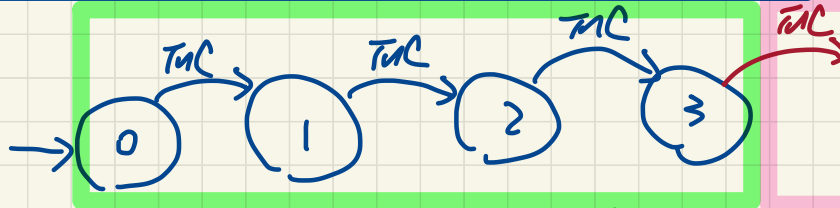
*Handwritten notes:*  
- Red arrow points to `fail` method call.  
- Red arrow points to `catch` block.  
- Red arrow points to `c.decrement();` line.  
- Red arrow points to `try` block.  
- Red arrow points to `assertEquals` method call.  
- Red arrow points to `new Counter()` line.  
- Red arrow points to `Counter c` line.  
- Red arrow points to `testDecFromMinValue()` line.  
- Red arrow points to `@Test` line.  
- Red arrow points to `fail` method call.  
- Red arrow points to `catch` block.  
- Red arrow points to `c.decrement();` line.  
- Red arrow points to `try` block.  
- Red arrow points to `assertEquals` method call.  
- Red arrow points to `new Counter()` line.  
- Red arrow points to `Counter c` line.  
- Red arrow points to `testDecFromMinValue()` line.  
- Red arrow points to `@Test` line.

# JUnit: Exception Sometimes Expected, Sometimes **Not**

```
1 @Test
2 public void testIncFromMaxValue() {
3     Counter c = new Counter();
4     try {
5         c.increment(); c.increment(); c.increment();
6     }
7     catch (ValueTooLargeException e) {
8         fail("ValueTooLargeException was thrown unexpectedly.");
9     }
10    assertEquals(Counter.MAX_VALUE, c.getValue());
11    try {
12        c.increment();
13        fail("ValueTooLargeException was NOT thrown as expected.");
14    }
15    catch (ValueTooLargeException e) {
16        /* Do nothing: ValueTooLargeException thrown as expected. */
17    }
18 }
```

*NtLE not expected*

*NtLE expected*



*NtLE not expected*

*NtLE expected*

## Expected Behaviour:

Calling c.increment()

3 times to reach c's max **should not** trigger any ValueTooLargeException.

Calling c.increment()

when c is already at its max **should** trigger a ValueTooLargeException

## Running JUnit Test 3 on Correct Implementation

```
public void increment() throws ValueTooLargeException {
    if (value == Counter.MAX_VALUE) {
        throw new ValueTooLargeException("counter value is " + value);
    }
    else { value++; }
}
```

```

1 @Test
2 public void testIncFromMaxValue() {
3     Counter c = new Counter();
4     try {
5         c.increment(); c.increment(); c.increment();
6     }
7     catch (ValueTooLargeException e) {
8         fail("ValueTooLargeException was thrown unexpectedly.");
9     }
10    assertEquals(Counter.MAX_VALUE, c.getValue());
11    try {
12        c.increment();
13    }
14    catch (ValueTooLargeException e) {
15        // Do nothing: ValueTooLargeException thrown as expected. */
16    }
17 }

```

# Running JUnit Test 3 on Incorrect Implementation

```
public void increment() throws ValueTooLargeException {  
    ④ if (value == Counter.MAX_VALUE) {  
        ⑤ throw new ValueTooLargeException("counter value is " + value);  
    }  
    else { value++; }  
}
```

```
1  @Test  
2  public void testIncFromMaxValue() {  
3      Counter c = new Counter();  
4      ① try { V==0 VLTE thrown unexpectedly  
5          ② c.increment(); X c.increment(); c.increment();  
6      }  
7      ③ catch (ValueTooLargeException e) {  
8          ④ fail "ValueTooLargeException was thrown unexpectedly.";   
9      }  
10     assertEquals(Counter.MAX_VALUE, c.getValue());  
11     try {  
12         c.increment();  
13         fail("ValueTooLargeException was NOT thrown as expected.");  
14     }  
15     catch (ValueTooLargeException e) {  
16         /* Do nothing: ValueTooLargeException thrown as expected. */  
17     }  
18 }
```

# Running JUnit Test 3 on Incorrect Implementation



```
public void increment() throws ValueTooLargeException {  
    if (value is Counter.MAX_VALUE) {  
        throw new ValueTooLargeException("counter value is " + value);  
    }  
    else { value ++; }  
}
```

```
1  @Test  
2  public void testIncFromMaxValue() {  
3      Counter c = new Counter();  
4      try {  
5          c.increment(); c.increment(); c.increment();  
6      }  
7      catch (ValueTooLargeException e) {  
8          fail("ValueTooLargeException was thrown unexpectedly.");  
9      }  
10     assertEquals(Counter.MAX_VALUE, c.getValue());  
11     try {  
12         c.increment();  
13         fail("ValueTooLargeException was NOT thrown as expected.");  
14     }  
15     catch (ValueTooLargeException e) {  
16         /* Do nothing: ValueTooLargeException thrown as expected. */  
17     }  
18 }
```

Handwritten annotations on the test code:

- Line 3:  $c.v = 0$  ✓
- Line 5:  $c.v == 2$  ✓
- Line 6:  $c.v == 3$  ✓
- Line 11:  $c.v == 3$  ✓
- Line 12:  $c.v == 3$  ✓
- Line 13:  $c.v == 3$  ✓
- Line 13: "no VTLException thrown as expected" (highlighted)

# Exercise: Console Tester vs. JUnit Test

Q. Can this **console tester** work like the **JUnit test** testIncFromMaxValue does?

```
1 public class CounterTester {
2     public static void main(String[] args) {
3         Counter c = new Counter();
4         println("Current val: " + c.getValue());
5         try {
6             c.increment(); c.increment(); c.increment();
7             println("Current val: " + c.getValue());
8         }
9         catch (ValueTooLargeException e) {
10            println("Error: ValueTooLargeException thrown unexpectedly.");
11        }
12        try {
13            c.increment();
14            println("Error: ValueTooLargeException NOT thrown.");
15        } /* end of inner try */
16        catch (ValueTooLargeException e) {
17            println("Success: ValueTooLargeException thrown.");
18        }
19    } /* end of main method */
20 } /* end of CounterTester class */
```

*scry VTLException occurred unexpectedly rather than fail();*

*not appropriate to continue the test after we knew it failed*

Hint: What if one of the first 3 c.increment() **mistakenly** throws a **ValueTooLargeException**?

## Exercise: Combining catch Blocks?

Q: Can we rewrite `testIncFromMaxValue` to:

```
1  @Test
2  public void testIncFromMaxValue() {
3      Counter c = new Counter();
4      try {
5          c.increment();
6          c.increment();
7          c.increment();
8          assertEquals(Counter.MAX_VALUE, c.getValue());
9          c.increment();
10         fail("ValueTooLargeException was NOT thrown as expected.");
11     }
12     catch (ValueTooLargeException e) { }
13 }
```

*Handwritten notes:*

- Lines 5-7: `c.increment();` are boxed in green. Next to them is the note "NLE not expected".
- Line 9: `c.increment();` is boxed in red. Next to it is the note "NLE expected".
- Line 12: The `catch` block is circled in purple. An arrow points from the circle to the handwritten text "pass? fail?".
- Below the code, the note "NLE occurred" is written in purple.

Hint: Say Line 12 is executed,

is it clear if that `ValueTooLargeException` was thrown as expected?



# Testing Many Values in a Single Test

Loops can make it effective on generating test cases:

```
1  @Test
2  public void testIncDecFromMiddleValues() {
3      Counter c = new Counter();
4      try {
5          for(int i = Counter.MIN_VALUE; i < Counter.MAX_VALUE; i++) {
6              int currentValue = c.getValue();
7              c.increment();
8              assertEquals(currentValue + 1, c.getValue());
9          }
10         for(int i = Counter.MAX_VALUE; i > Counter.MIN_VALUE; i--) {
11             int currentValue = c.getValue();
12             c.decrement();
13             assertEquals(currentValue - 1, c.getValue());
14         }
15     }
16     catch (ValueTooLargeException e) {
17         fail("ValueTooLargeException is thrown unexpectedly");
18     }
19     catch (ValueTooSmallException e) {
20         fail("ValueTooSmallException is thrown unexpectedly");
21     }
22 }
```

0 1 2 ] 3 times

3 2 1 ] 3 times

