

Test-Driven Development (TDD) with JUnit



EECS2030 E: Advanced
Object Oriented Programming
Summer 2025

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- **Testing** the Solution to a Bounded Counter Problem
- Deriving **Test Cases** for a Bounded Variable
- Application of **Normal** vs. **Disrupted Execution Flows**
- **Intention** of a Test: **Exceptions Expected** vs. **Not Expected**
- **Test Driven Development (TDD)** via **Regression Testing**

Motivating Example: Two Types of Errors (1)

Consider two kinds of exceptions for a counter:

```
public class ValueTooLargeException extends Exception {  
    ValueTooLargeException(String s) { super(s); }  
}  
public class ValueTooSmallException extends Exception {  
    ValueTooSmallException(String s) { super(s); }  
}
```

Any thrown object instantiated from these two exception classes must be handled (**catch-or-specify requirement**):

- Either **specify** `throws ...` in the method header/API (i.e., **propagate** it to the immediate **caller** in the **call stack**)
- Or **handle** it in a `try-catch` block

Motivating Example: Two Types of Errors (2)

Approach 1 – *Specify*: Indicate in the method header/API that a specific exception might be thrown.

Example 1: Method that throws the exception

```
class C1 {  
    void m1(int x) throws ValueTooSmallException {  
        if(x < 0) {  
            throw new ValueTooSmallException("val " + x);  
        }  
    }  
}
```

Example 2: Method that calls another which throws the exception

```
class C2 {  
    C1 c1;  
    void m2(int x) throws ValueTooSmallException {  
        c1.m1(x);  
    }  
}
```

Motivating Example: Two Types of Errors (3)

Approach 2 – *Catch*: Handle the thrown exception(s) in a try-catch block.

```
class C3 {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        int x = input.nextInt();  
        C2 c2 = new c2();  
        try {  
            c2.m2(x);  
        }  
        catch (ValueTooSmallException e) { ... }  
    }  
}
```

A Simple Counter (1)

Consider a class for keeping track of an integer counter value:

```
public class Counter {  
    public final static int MAX_VALUE = 3;  
    public final static int MIN_VALUE = 0;  
    private int value;  
    public Counter() {  
        this.value = Counter.MIN_VALUE;  
    }  
    public int getValue() {  
        return value;  
    }  
    ... /* more later! */  
}
```

- Access **private** attribute value using **public** accessor `getValue`.
- Two class-wide (i.e., `static`) constants (i.e., `final`) for lower and upper bounds of the counter value.
- Initialize the counter value to its lower bound.
- **Requirement** :

The counter value must be within its lower and upper bounds.

Exceptional Scenarios

- **Sound** Software Engineering Practice:
Design a **test strategy** even **before** code is completed.
- **Q**: Possible exceptional scenarios for such a counter?
 - An attempt to increment **above** the counter's upper bound.
 - An attempt to decrement **below** the counter's lower bound.

A Simple Counter (2)

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

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

- Change the counter value via two mutator methods.
- Changes on the counter value may **trigger an exception**:
 - Attempt to **increment** when counter already reaches its **maximum**.
 - Attempt to **decrement** when counter already reaches its **minimum**.

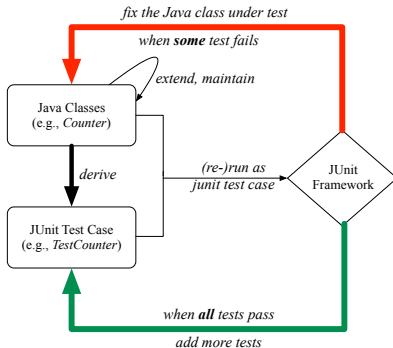
Components of a Test

- Manipulate the relevant object(s).
 - e.g., *Initialize a counter object c, then call c.increment().*
 - e.g., *Initialize a counter object c, then call c.decrement().*
- What do you **expect to happen**?
 - e.g., *value of counter is such that Counter.MIN_VALUE + 1*
 - e.g., *ValueTooSmallException is thrown*
- What does your program **actually produce**?
 - e.g., *call c.getValue() to find out.*
 - e.g., *Use a try-catch block to find out* (to be discussed!).
- A test:
 - **Passes** if *expected outcome* occurs.
 - **Fails** if *expected outcome* does not occur.

Why JUnit?

- **Automate** the *testing of correctness* of your Java classes.
- **Derive** the list of tests. **Transform** it into a **JUnit Test Class**.
- JUnit tests are **callers/clients** of your classes. Each **test** may:
 - Either attempt to use a method in a *legal* way (i.e., *satisfying* its precondition), and report:
 - **Success** if the result is as expected
 - **Failure** if the result is **not** as expected
 - Or attempt to use a method in an *illegal* way (i.e., *not satisfying* its precondition), and report:
 - **Success** if the expected exception (e.g., `ValueTooSmallException`) occurs.
 - **Failure** if the expected exception does **not** occur.
- **Regression Testing**: Any **change** introduced to your software *must not compromise* its established **correctness**.

Test-Driven Development (TDD)



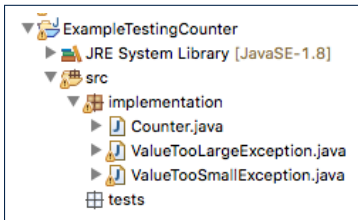
Maintain a collection of tests which define the **correctness** of your Java class under development (CUD):

- Derive and run tests as soon as your CUD is **testable**.
i.e., A Java class is testable when defined with method signatures.
- **Red** bar reported: Fix the class under test (CUT) until **green** bar.
- **Green** bar reported: Add more tests and Fix CUT when necessary.

How to Use JUnit: Packages

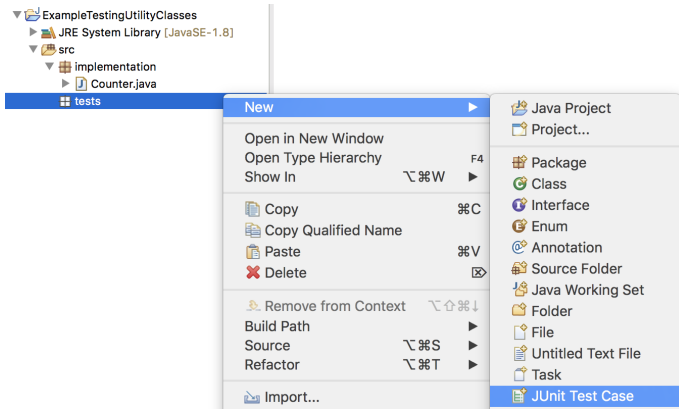
Step 1:

- In Eclipse, create a Java project `ExampleTestingCounter`
- *Separation of concerns* :
 - Group classes for *implementation* (i.e., `Counter`) into package `implementation`.
 - Group classes for *testing* (to be created) into package `tests`.



How to Use JUnit: New JUnit Test Case (1)

Step 2: Create a new *JUnit Test Case* in `tests` package.

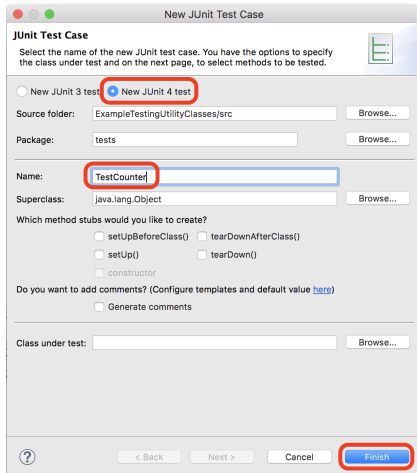


Create one JUnit Test Case to test one Java class only.

⇒ If you have *n Java classes to test*, create *n JUnit test cases*.

How to Use JUnit: New JUnit Test Case (2)

Step 3: Select the version of JUnit (JUnit 4); Enter the name of test case (`TestCounter`); Finish creating the new test case.



New JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☒ **New JUnit 4 test**

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

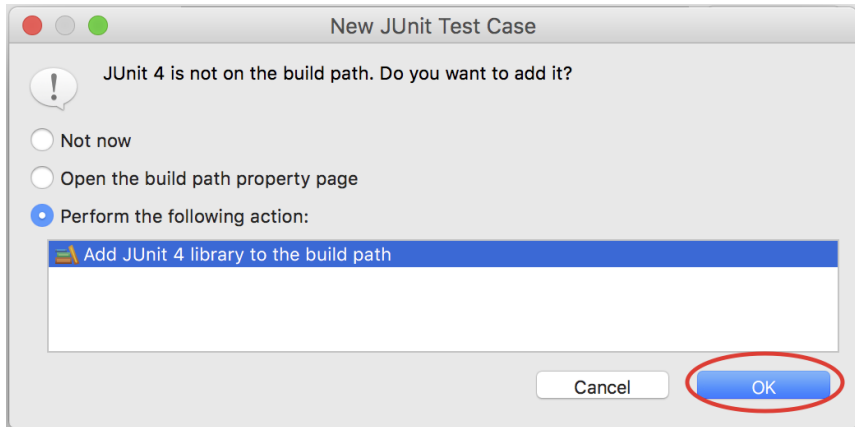
Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

How to Use JUnit: Adding JUnit Library

Upon creating the very first test case, you will be prompted to add the JUnit library to your project's build path.



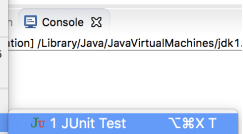
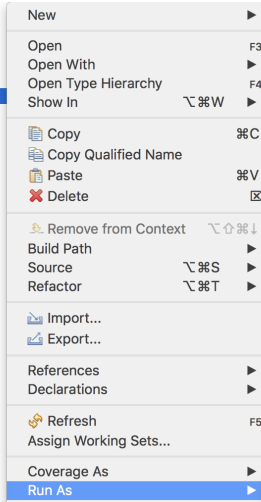
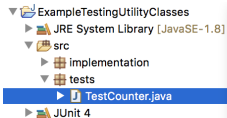
How to Use JUnit: Generated Test Case

```
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 }
```

- **Lines 6 – 8:** `test` is just an *ordinary mutator method* that has a one-line implementation body.
- **Line 5** is critical: Prepend the tag `@Test` verbatim, requiring that *the method is to be treated as a JUnit test*.
⇒ When `TestCounter` is run as a JUnit Test Case, only *those methods prepended by the @Test tags* will be run and reported.
- **Line 7:** By default, we deliberately fail the test with a message “Not yet implemented”.

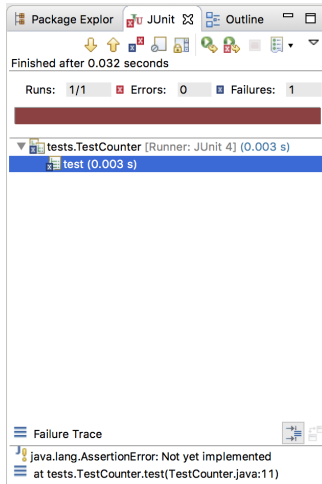
How to Use JUnit: Running Test Case

Step 4: Run the `TestCounter` class as a JUnit Test.



How to Use JUnit: Generating Test Report

A **report** is generated after running all tests (i.e., methods prepended with **@Test**) in `TestCounter`.



How to Use JUnit: Interpreting Test Report

- A **test** is a method prepended with the **@Test** tag.
- 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.
- After running all tests:
 - A **green** bar means that **all** tests succeed.
⇒ Keep challenging yourself if **more tests** may be added.
 - A **red** bar means that **at least one** test fails.
⇒ Keep fixing the class under test and re-running all tests, until you receive a **green** bar.
- **Question:** What is the easiest way to making test a **success**?
Answer: Delete the call `fail("Not yet implemented")`.

How to Use JUnit: Revising Test Case

```
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 }
```

Now, the body of `test` simply does nothing.

⇒ Neither assertion failures nor exceptions will occur.

⇒ The execution of `test` will be considered as a **success**.

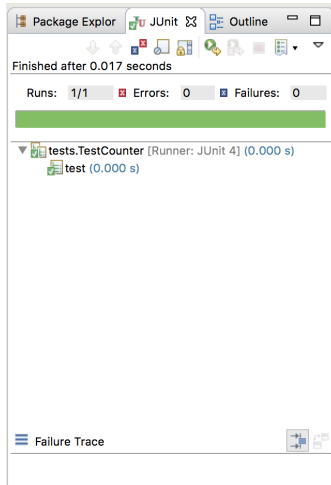
∴ There is currently only one test in `TestCounter`.

∴ We will receive a **green** bar!

Caution: `test` which passes at the moment is **not useful** at all!

How to Use JUnit: Re-Running Test Case

A new report is generated after re-running all tests (i.e., methods prepended with `@Test`) in `TestCounter`.



How to Use JUnit: Commons Assertions

- `void assertNull(Object o)`
- `void assertEquals(int expected, int actual)`
- `void assertEquals(double exp, double act, double epsilon)`
- `void assertEquals(expected, actuals)`
- `void assertTrue(boolean condition)`
- `void fail(String message)`

JUnit Assertions: Examples (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());  ✗ /* NullPointerException */  
p = new Point(3, 4);  
assertNull(p);      ✗  
assertTrue(p == null);  ✗  
assertFalse(p != null); ✗  
assertEquals(3, p.getX());  ✓  
assertTrue(p.getX() == 3 && p.getY() == 4);  ✓
```

JUnit Assertions: Examples (2)

- Consider the following class:

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

- How do we test `c.getArea()`?
 - Mathematically: $3.4 \times 3.4 \times 3.14 = 36.2984$
 - However, base-10 numbers **cannot** be represented perfectly in the binary format.
 - When comparing fractional numbers, allow some **tolerance**:

$$36.2984 - 0.01 \leq c.getArea() \leq 36.2984 + 0.01$$

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

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


More JUnit Assertion Methods

method name / parameters	description
<code>assertTrue(test)</code> <code>assertTrue("message", test)</code>	Causes this test method to fail if the given boolean test is not <code>true</code> .
<code>assertFalse(test)</code> <code>assertFalse("message", test)</code>	Causes this test method to fail if the given boolean test is not <code>false</code> .
<code>assertEquals(expectedValue, value)</code> <code>assertEquals("message", expectedValue, value)</code>	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the <code>equals</code> method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
<code>assertNotEquals(value1, value2)</code> <code>assertNotEquals("message", value1, value2)</code>	Causes this test method to fail if the given two values <i>are</i> equal to each other. (For objects, it uses the <code>equals</code> method to compare them.)
<code>assertNull(value)</code> <code>assertNull("message", value)</code>	Causes this test method to fail if the given value is not <code>null</code> .
<code>assertNotNull(value)</code> <code>assertNotNull("message", value)</code>	Causes this test method to fail if the given value <i>is</i> <code>null</code> .
<code>assertSame(expectedValue, value)</code> <code>assertSame("message", expectedValue, value)</code> <code>assertNotSame(value1, value2)</code> <code>assertNotSame("message", value1, value2)</code>	Identical to <code>assertEquals</code> and <code>assertNotEquals</code> respectively, except that for objects, it uses the <code>==</code> operator rather than the <code>equals</code> method to compare them. (The difference is that two objects that have the same state might be equal to each other, but not <code>==</code> to each other. An object is only <code>==</code> to itself.)
<code>fail()</code> <code>fail("message")</code>	Causes this test method to fail.

- What is the complete list of cases for testing `Counter`?

<code>c.getValue()</code>	<code>c.increment()</code>	<code>c.decrement()</code>
0	1	ValueTooSmall
1	2	0
2	3	1
3	ValueTooLarge	2

- Let's turn the two cases in the 1st row into two JUnit tests:
 - Test for the *green* cell *succeeds* if:
 - No failures and exceptions occur; and
 - The new counter value is 1.
 - Tests for *red* cells *succeed* if the *expected exceptions* occur (`ValueTooSmallException` & `ValueTooLargeException`).

Testing: Correct vs. Incorrect Imp.

- The real value of a **test** is:
 - Not only to **reaffirm** when your implementation is **correct**,
 - But also to **reject** when your implementation is **incorrect**.
- What if the method `decrement` was implemented **incorrectly**?

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

- A “good” test should **reject** such an **incorrect** implementation.

Test Case 1: Increment from Min (1)

```

1  @Test
2  public void testIncAfterCreation() {
3      Counter c = new Counter();
4      assertEquals(Counter.MIN_VALUE, c.getValue());
5      try {
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 }
  
```

- **L3** sets `c.value` to 0.
- **Line 6** requires a try-catch block \because potential `ValueTooLargeException`
- **Lines 4, 7 11** are all assertions:
 - **Lines 4 & 7** assert that `c.getValue()` returns the expected values.
 - **Line 11:** an **assertion failure** \because unexpected `ValueTooLargeException`
- **Line 7** can be rewritten as `assertTrue(1 == c.getValue())`.

Test Case 1: Increment from Min (2)

```
1  @Test
2  public void testIncAfterCreation() {
3      Counter c = new Counter();
4      assertEquals(Counter.MIN_VALUE, c.getValue());
5      try {
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 }
```

At L6, if method decrement is implemented:

- **Correctly** ⇒ a ValueTooLargeException does not occur.
⇒ Execution continues to L7, L8, L13, then the program terminates.
- **Incorrectly** ⇒ an unexpected ValueTooLargeException occurs.
⇒ Execution jumps to L9, L10 – L11, then the test program terminates.

Test Case 2: Decrement from Min (1)

```
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 }
```

- L3 sets `c.value` to 0.
- Line 6 requires a try-catch block \therefore potential `ValueTooSmallException`
- Lines 4 & 7 are both assertions:
 - Lines 4 asserts that `c.getValue()` returns the expected value (i.e., `Counter.MIN_VALUE`).
 - Line 7: an **assertion failure** \therefore expected `ValueTooSmallException` not thrown

Test Case 2: Decrement from Min (2)

```
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 }
```

At L6, if method `decrement` is implemented:

- **Correctly** ⇒ a `ValueTooLargeException` occurs.
⇒ Execution jumps to L9, L10 – L12, then the program terminates.
- **Incorrectly** ⇒ expected `ValueTooLargeException` does not occur.
⇒ Execution continues to L7, then the test program terminates.

Test Case 3: Increment from Max

```
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 }
```

- L4 – L9: a VTLE **is not** expected; L11 – 17: a VTLE **is** expected.

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 */

```

- A. Say one of the first 3 `c.increment()` **mistakenly** throws VTLE.
- After L10 is executed, flow of execution **still continues** to L12.
 - This allows the 4th `c.increment` to be executed!

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 }
```

No!

At **Line 12**, we would not know which line throws the VTLE:

- If it was any of the calls in **L5 – L7**, then it's **not right**.
- If it was **L9**, then it's **right**.

Using Loops in JUnit Test Cases

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 }
```

1. Run all 8 tests and make sure you receive a *green* bar.
2. Now, introduction an error to the implementation: Change the line `value ++` in `Counter.increment` to `--`.
 - Re-run all 8 tests and you should receive a *red* bar. [Why?]
 - Undo *error injections* & Re-Run all 8 tests. [What happens?]

- Official Site of JUnit 4:

<http://junit.org/junit4/>

- API of JUnit assertions:

<http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

- Another JUnit Tutorial example:

<https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml>

Beyond this lecture...

Play with the source code `ExampleTestingCounter.zip`

Tip. Change input values so as to explore, in Eclipse *debugger*, possible (*normal* vs. *abnormal*) **execution paths**.

Index (1)

Learning Outcomes

Motivating Example: Two Types of Errors (1)

Motivating Example: Two Types of Errors (2)

Motivating Example: Two Types of Errors (3)

A Simple Counter (1)

Exceptional Scenarios

A Simple Counter (2)

Components of a Test

Why JUnit?

Test-Driven Development (TDD)

How to Use JUnit: Packages

Index (2)

How to Use JUnit: New JUnit Test Case (1)

How to Use JUnit: New JUnit Test Case (2)

How to Use JUnit: Adding JUnit Library

How to Use JUnit: Generated Test Case

How to Use JUnit: Running Test Case

How to Use JUnit: Generating Test Report

How to Use JUnit: Interpreting Test Report

How to Use JUnit: Revising Test Case

How to Use JUnit: Re-Running Test Case

How to Use JUnit: Common Assertions

JUnit Assertions: Examples (1)

Index (3)

JUnit Assertions: Examples (2)

More JUnit Assertion Methods

Testing Strategy

Testing: Correct vs. Incorrect Imp.

Test Case 1: Increment from Min (1)

Test Case 1: Increment from Min (2)

Test Case 2: Decrement from Min (1)

Test Case 2: Decrement from Min (2)

Test Case 3: Increment from Max

Exercise: Console Tester vs. JUnit Test

Exercise: Combining `catch` Blocks?

Index (4)

Using Loops in JUnit Test Cases

Exercises

Resources

Beyond this lecture...