Unit and Regression Testing using JUnit

EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

A Simple Counter (1)

Consider a *utility class* (where attributes and methods are **static**) for keeping track of an integer counter value:

```
public class Counter {
  public final static int MAX_COUNTER_VALUE = 3;
  public final static int MIN_COUNTER_VALUE = 0;
  public static int value = MIN_COUNTER_VALUE;
  ... /* more code later! */
```

- When attempting to access the **static** attribute `value` ***outside*** the `Counter` class, write `Counter.value`.
- Two 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 between its lower and upper bounds.

Encode Precondition Violation as `IllegalArgumentException`

Consider two possible scenarios of *Precondition Violations* (i.e., scenarios of throwing `IllegalArgumentException`):

- When the counter value is attempted (but not yet) to be updated **above** its upper bound.
- When the counter value is attempted (but not yet) to be updated **below** its upper bound.

A Simple Counter (2)

```
public static void increment() {
  if( value == Counter.MAX_COUNTER_VALUE ) {
    /* Precondition Violation */
    throw new IllegalArgumentException("Too large to increment");
  }
  else { value ++; }
}
public static void decrement() {
  if( value == Counter.MIN_COUNTER_VALUE ) {
    /* Precondition Violation */
    throw new IllegalArgumentException("Too small to decrement");
  }
  else { value --; }
}
```

- Change the counter value via two mutator methods.
- Changes on the counter value may ***violate a precondition***:
  - Attempt to **increment** when counter value reaches its **maximum**.
  - Attempt to **decrement** when counter value reaches its **minimum**.

Testing the Counter Class from Console: Test Case 1

Consider a class for testing the `Counter` class:

```
public class CounterTester1 {
  public static void main(String[] args) {
    System.out.println("Init val: " + Counter.value);
    System.out.println("Attempt to decrement:");
    /* Right before calling the decrement mutator,
     * Counter.value is 0 and too small to be decremented.
     */
    Counter.decrement();
  }
}
```

Executing it as Java Application gives this Console Output:

```
Init val: 0
Attempt to decrement:
Exception in thread "main"
  java.lang.IllegalArgumentException: Too small to decrement
```

Testing the Counter Class from Console: Test Case 2

Consider **another** class for testing the `Counter` class:

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter.increment(); Counter.increment(); Counter.increment();
    System.out.println("Current val: " + Counter.value);
    System.out.println("Attempt to increment:");
    /* Right before calling the increment mutator,
     * Counter.value is 3 and too large to be incremented.
     */
    Counter.increment();
  }
}
```

Executing it as Java Application gives this Console Output:

```
Current val: 3
Attempt to increment:
Exception in thread "main"
  java.lang.IllegalArgumentException: Too large to increment
```

Limitations of Testing from the Console

- Do **Test Cases** 1 & 2 suffice to test `Counter`'s *correctness*?
  - Is it plausible to claim that the implementation of `Counter` is *correct* because it passes the two test cases?
- What other test cases can you think of?

`Counter.value`	`Counter.increment()`	`Counter.decrement()`
0	1	`ValueTooSmall`
1	2	0
2	3	1
3	`ValueTooBig`	2

- So in total we need 8 test cases.

  ⇒ 6 more separate `CounterTester` classes to create!
- Problems? It is inconvenient to:
  - Run each TC by executing `main` of a `CounterTester` and comparing console outputs ***with your eyes***.
  - Re-run ***manually*** all TCs whenever `Counter` is changed.

    Principle: Any **change** introduced to your software *must not compromise* its established **correctness**.

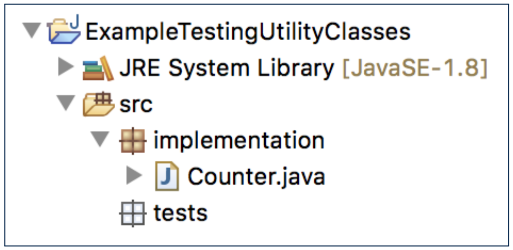
Why JUnit?

- *Automate* the ***testing of correctness*** of your Java classes.
- Once you derive the list of tests, translate it into a JUnit test case, which is just a Java class that you can execute upon.
- JUnit tests are *helpful clients* of your classes, where 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 precondition violation (i.e., `IllegalArgumentException`) occurs.
    - ***Failure*** if precondition violation (i.e., `IllegalArgumentException`) does ***not*** occur.

How to Use JUnit: Packages

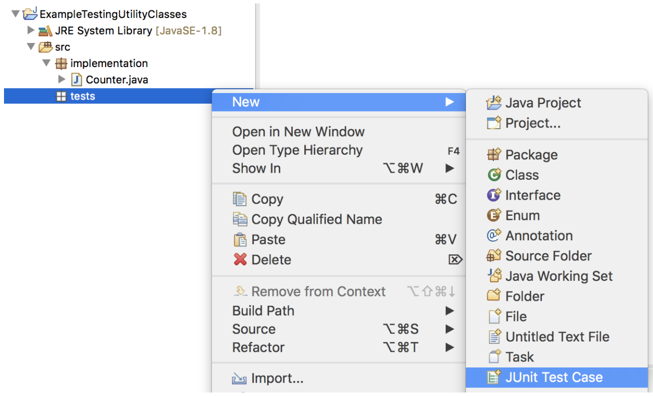
Step 1:

- In Eclipse, create a Java project `ExampleTestingUtilityClasses`
- *Separation of concerns* :
  - Group classes for ***implementation*** (i.e., `Counter`) into package `implementation`.
  - Group classes 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.

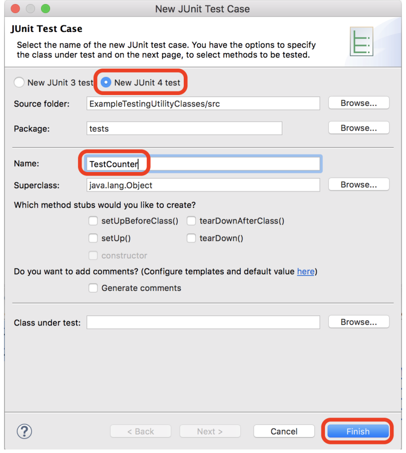


Create one JUnit Test Case to test one Java class only.

⇒ 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

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: ExampleTestingUtilityClasses/src

Package: tests

Name: TestCounter

Superclass: java.lang.Object

Which method stubs would you like to create?

□ setUpBeforeClass() □ tearDownAfterClass()
□ setUp() □ tearDown()
□ constructor

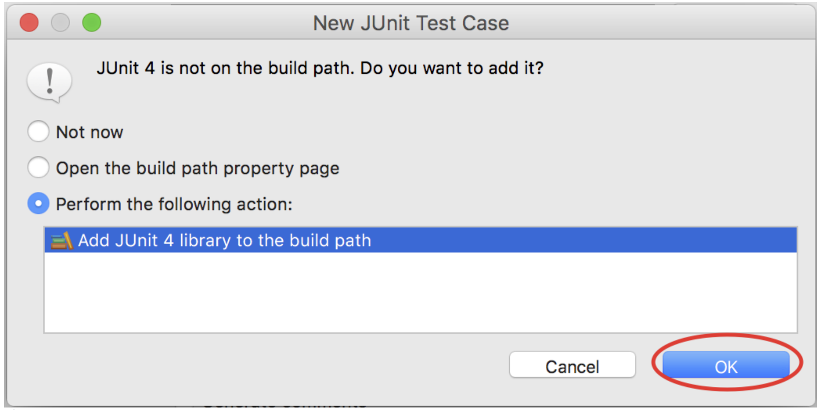
Do you want to add comments? (Configure templates and default value here)

□ 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.



New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

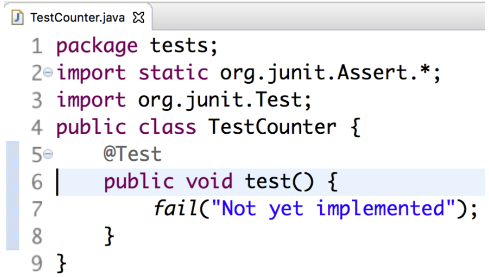
○ Not now

○ Open the build path property page

● Perform the following action:

Add JUnit 4 library to the build path

How to Use JUnit: Generated Test Case

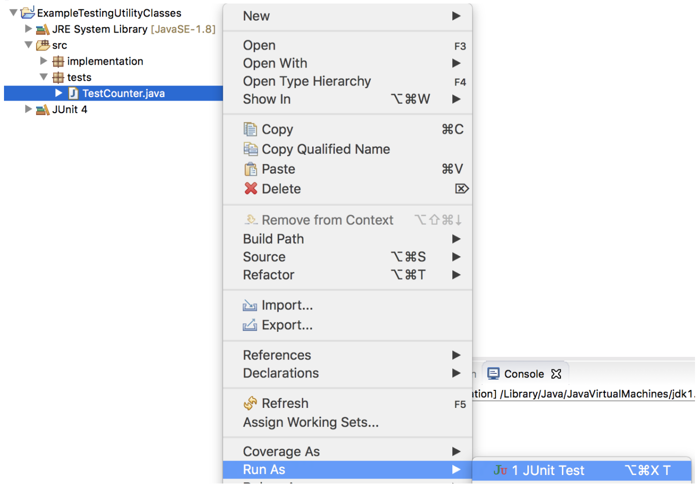


```
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7         fail("Not yet implemented");
8     }
9 }
```

- **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".

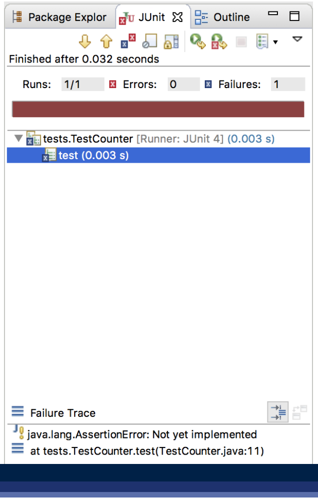
How to Use JUnit: Running Test Case

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`.

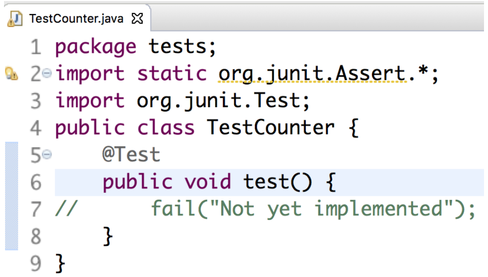


```
Finished after 0.032 seconds
Runs: 1/1   Errors: 0   Failures: 1
tests.TestCounter [Runner: JUnit 4] (0.003 s)
    test (0.003 s)
java.lang.AssertionError: Not yet implemented
at tests.TestCounter.test(TestCounter.java:11)
```

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; or
    - an *unexpected* exception (e.g., `NullPointerException`, `ArrayIndexOutOfBoundException`) is 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-runing 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



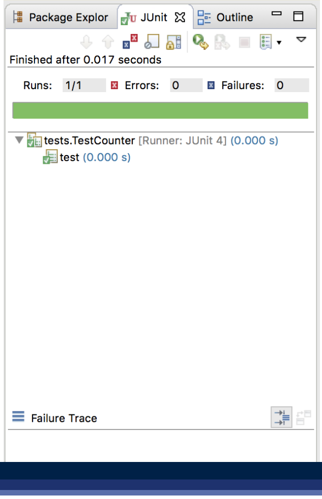
```
package tests;
import static org.junit.Assert.*;
import org.junit.Test;
public class TestCounter {
    @Test
    public void test() {
//      fail("Not yet implemented");
    }
}
```

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`.



Finished after 0.017 seconds

Runs: 1/1 Errors: 0 Failures: 0

tests.TestCounter [Runner: JUnit 4] (0.000 s)
test (0.000 s)

How to Use JUnit: Adding More Tests (1)

- Recall the complete list of cases for testing `Counter`:

`c.getValue()`	`c.increment()`	`c.decrement()`
0	1	`ValueTooSmall`
1	2	0
2	3	1
3	`ValueTooBig`	2

- Let's turn the two cases in the 1st row into two JUnit tests:
  - Test for left cell *succeeds* if:
    - No failures and exceptions occur; and
    - The new counter value is `1`.
  - Test for right cell *succeeds* if the *expected precondition violation* occurs (`IllegalArgumentException` is thrown).
- Common JUnit assertion methods (complete list in next slide):
  - `void assertNull(Object o)`
  - `void assertEquals(expected, actual)`
  - `void assertTrue(boolean condition)`
  - `void fail(String message)`

How to Use JUnit: Assertion Methods

method name / parameters	description

`assertTrue(`***test***`)`
`assertTrue("`***message***`",` ***test***`)`
	Causes this test method to fail if the given `boolean` test is not `true`.

`assertFalse(`***test***`)`
`assertFalse("`***message***`",` ***test***`)`
	Causes this test method to fail if the given `boolean` test is not `false`.

`assertEquals(`***expectedValue***`,` ***value***`)`
`assertEquals("`***message***`",` ***expectedValue***`,` ***value***`)`
	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the `equals` 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.

`assertNotEquals(`***value1***`,` ***value2***`)`
`assertNotEquals("`***message***`",` ***value1***`,` ***value2***`)`
	Causes this test method to fail if the given two values *are* equal to each other. (For objects, it uses the `equals` method to compare them.)

`assertNull(`***value***`)`
`assertNull("`***message***`",` ***value***`)`
	Causes this test method to fail if the given value is not `null`.

`assertNotNull(`***value***`)`
`assertNotNull("`***message***`",` ***value***`)`
	Causes this test method to fail if the given value *is* `null`.

`assertSame(`***expectedValue***`,` ***value***`)`
`assertSame("`***message***`",` ***expectedValue***`,` ***value***`)`
`assertNotSame(`***value1***`,` ***value2***`)`
`assertNotSame("`***message***`",` ***value1***`,` ***value2***`)`
	Identical to `assertEquals` and `assertNotEquals` respectively, except that for objects, it uses the `==` operator rather than the `equals` method to compare them. (The difference is that two objects that have the same state might be `equals` to each other, but not `==` to each other. An object is only `==` to itself.)

`fail()`
`fail("`***message***`")`
	Causes this test method to fail.

How to Use JUnit: Adding More Tests (2.1)

```
1  @Test
2  public void testIncAfterCreation() {
3    /* Assert that initial value of counter is correct. */
4    assertEquals(Counter.MIN_COUNTER_VALUE, Counter.value);
5    /* Attempt to increment the counter value,
6     * which is expected to succeed.
7     */
8    Counter.increment();
9    /* Assert that the updated counter value is correct. */
10   assertEquals(1, Counter.value);
11 }
```

- **L4**: Alternatively, you can write:

```
assertTrue(Counter.MIN_COUNTER_VALUE == Counter.value);
```

- **L10**: Alternatively, you can write:

```
assertTrue(1 == Counter.value);
```

How to Use JUnit: Adding More Tests (2.2)

- Don't lose the big picture!
- The JUnit test in the previous slide automates the following console tester which requires interaction with the external user:

```
public class CounterTester1 {
  public static void main(String[] args) {
    System.out.println("Init val: " + Counter.value);
    System.out.println("Attempt to decrement:");
    /* Right before calling the decrement mutator,
     * Counter.value is 0 and too small to be decremented.
     */
    Counter.decrement();
  }
}
```

- ***Automation** is exactly rationale behind using JUnit!*

How to Use JUnit: Adding More Tests (3.1)

```
1  @Test
2  public void testDecAfterCreation() {
3    assertTrue(Counter.MIN_COUNTER_VALUE == Counter.value);
4    try {
5      Counter.decrement();
6      /* Reaching this line means
7       * IllegalArgumentException not thrown! */
8      fail("Expected Precondition Violation Did Not Occur!");
9    }
10   catch(IllegalArgumentException e) {
11     /* Precondition Violated Occurred as Expected. */
12   } }
```

- **Lines 4 & 10**: We need a try-catch block because of **Line 5**.
  - Method `decrement` from class `Counter` is expected to throw the `IllegalArgumentException` because of a *precondition violation*.
- **Lines 3 & 8** are both assertions:
  - **Lines 3** asserts that `Counter.value` returns the expected value (`Counter.MIN_COUNTER_VALUE`).
  - **Line 8**: an assertion failure ∵ expected `IllegalArgumentException` not thrown

How to Use JUnit: Adding More Tests (3.2)

- Again, don't lose the big picture!
- The JUnit test in the previous slide automates the following console tester which requires interaction with the external user:

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter.increment(); Counter.increment(); Counter.increment();
    System.out.println("Current val: " + Counter.value);
    System.out.println("Attempt to increment:");
    /* Right before calling the increment mutator,
     * Counter.value is 3 and too large to be incremented.
     */
    Counter.increment();
  }
}
```

- Again, ***automation** is exactly rationale behind using JUnit!*

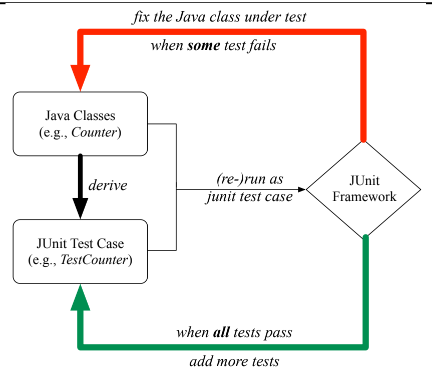
Exercises

1. Convert the rest of the cells into JUnit tests:

`c.getValue()`	`c.increment()`	`c.decrement()`
0	1	`ValueTooSmall`
1	2	0
2	3	1
3	`ValueTooBig`	2

2. Run all 8 tests and make sure you receive a *green* bar.
3. 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 the error injection, and re-run all 8 tests. [What happens?]

Regression Testing



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.

Resources

- 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`

Index (1)

Index (2)