Test-Driven Development (TDD) with JUnit

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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 classes must be handled (*catch-specify requirement*):

- Either *specify* `throws ...` in the method signature
  (i.e., propagating it to other caller)
- Or *handle* it in a `try-catch` block

Motivating Example: Two Types of Errors (2)

Approach 1 – Specify: Indicate in the method signature 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 between its lower and upper bounds.

Exceptional Scenarios

Consider the two possible exceptional scenarios:

- 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("counter value is " + value);
   }
   else { value ++; }
 }

 public void decrement() throws ValueTooSmallException {
   if(value == Counter.MIN_VALUE) {
     throw new ValueTooSmallException("counter 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()`.
- What do you *expect to happen*?
  e.g., value of counter is such that `Counter.MIN_VALUE + 1`
- What does your program *actually produce*?
  e.g., call `c.getValue` to find out.
- A test:
  - ***Passes*** if expected value *matches* actual value
  - ***Fails*** if expected value *does not match* actual value
- So far, you ran tests via a tester class with the `main` method.

Testing Counter from Console (V1): Case 1

Consider a class for testing the `Counter` class:

```
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("ValueTooSmallException NOT thrown as expected.");
    }
    catch (ValueTooSmallException e) {
      println("ValueTooSmallException thrown as expected.");
    } } }
```

Executing it as Java Application gives this Console Output:

```
Init val: 0
ValueTooSmallException thrown as expected.
```


Testing Counter from Console (V1): Case 2

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

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try { c.increment(); c.increment(); c.increment(); }
    catch (ValueTooLargeException e) {
      println("ValueTooLargeException thrown unexpectedly."); }
    println("Current val: " + c.getValue());
    try {
      c.increment();
      println("ValueTooLargeException NOT thrown as expected."); }
    catch (ValueTooLargeException e) {
      println("ValueTooLargeException thrown as expected."); } } }
```

Executing it as Java Application gives this Console Output:

```
Current val: 0
Current val: 3
ValueTooLargeException thrown as expected.
```


Testing Counter from Console (V2)

Consider a different class for testing the `Counter` class:

```
import java.util.Scanner;
public class CounterTester3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String cmd = null; Counter c = new Counter();
    boolean userWantsToContinue = true;
    while(userWantsToContinue) {
      println("Enter \"inc\", \"dec\", or \"val\":");
      cmd = input.nextLine();
      try {
        if(cmd.equals("inc")) { c.increment(); }
        else if(cmd.equals("dec")) { c.decrement(); }
        else if(cmd.equals("val")) { println( c.getValue() ); }
        else { userWantsToContinue = false; println("Bye!"); }
      }
      catch(ValueTooLargeException e){ println("Value too big!"); }
      catch(ValueTooSmallException e){ println("Value too small!"); }
    } } }
```


Testing Counter from Console (V2): Test 1

Test Case 1: Decrement when the counter value is too small.

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
dec
Value too small!
Enter "inc", "dec", or "val":
exit
Bye!
```

Testing Counter from Console (V2): Test 2

Test Case 2: Increment when the counter value is too big.

```
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
3
Enter "inc", "dec", or "val":
inc
Value too big!
Enter "inc", "dec", or "val":
exit
Bye!
```


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?

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

- So in total we need 8 test cases. ⇒ 6 more separate
  - `CounterTester` classes to create (like `CounterTester1`)!
  - Console interactions with `CounterTester3`!
- 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.

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

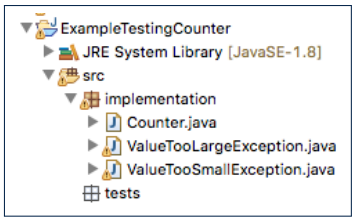
Why JUnit?

- *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 callers/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 the expected exception (e.g., `ValueTooSmallException`) occurs.
    - ***Failure*** if the expected exception does ***not*** occur.

How to Use JUnit: Packages

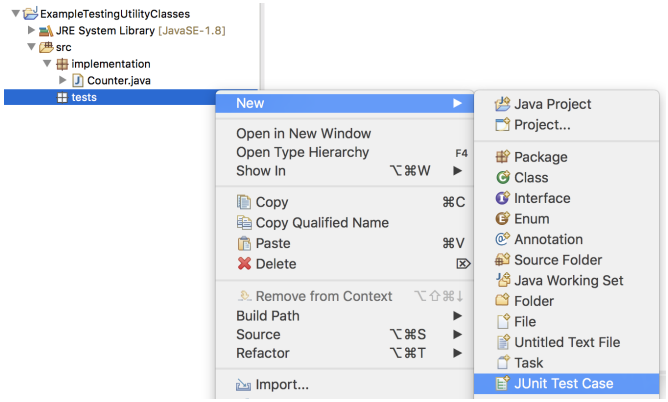
Step 1:

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

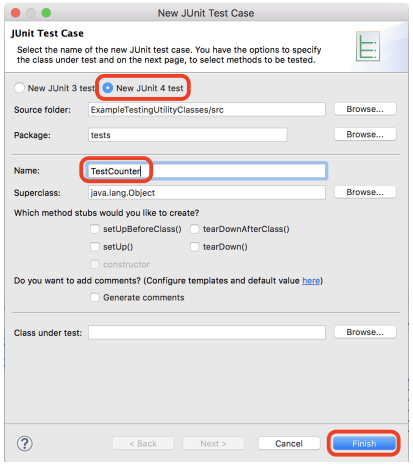


Create one JUnit Test Case to test one Java class only.

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

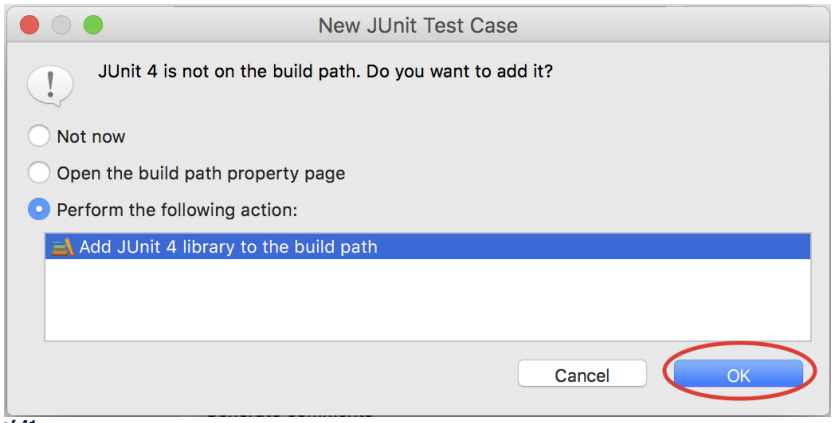
How to Use JUnit: New JUnit Test Case (2)

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



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

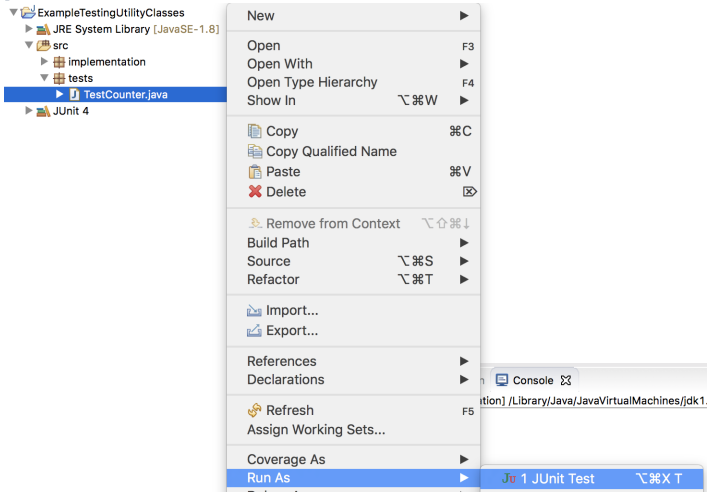
TestCounter.java

```
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.

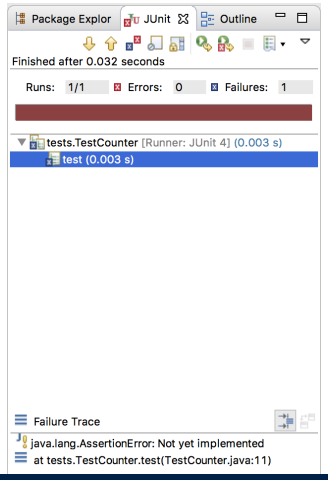


How to Use JUnit: Interpreting Test Report

- 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: Generating Test Report

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



How to Use JUnit: Revising Test Case

TestCounter.java

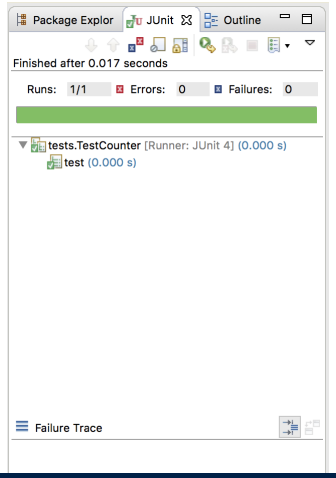
```
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7 //      fail("Not yet implemented");
8     }
9 }
```

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: 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	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`).
- Common JUnit assertion methods:
  - `void assertNull(Object o)`
  - `void assertEquals(expected, actual)`
  - `void assertArrayEquals(expecteds, actuals)`
  - `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    Counter c = new Counter();
4    assertEquals(Counter.MIN_VALUE, c.getValue());
5    try {
6      c.increment();
7      assertEquals(1, c.getValue());
8    } catch(ValueTooBigException e) {
9      /* Exception is not expected to be thrown. */
10     fail("ValueTooBigException is not expected."); } }
```

- **Lines 5 & 8**: We need a try-catch block because of **Line 6**.
  Method `increment` from class `Counter` may throw the `ValueTooBigException`.
- **Lines 4, 7 & 10** are all assertions:
  - **Lines 4 & 7** assert that `c.getValue()` returns the expected values.
  - **Line 10**: an assertion failure ∵ unexpected `ValueTooBigException`
- **Line 7** can be rewritten as `assertTrue(1 == c.getValue())`.

How to Use JUnit: Adding More Tests (2.2)

- Don't lose the big picture!
- JUnit test in previous slide automates this console interaction:

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
1
Enter "inc", "dec", or "val":
exit
Bye!
```

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

How to Use JUnit: Adding More Tests (3.1)

```
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    catch(ValueTooSmallException e) {
9      /* Exception is expected to be thrown. */ } }
```

- **Lines 5 & 8**: We need a try-catch block because of **Line 6**.
  Method `decrement` from class `Counter` may throw the `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 ∴ expected `ValueTooSmallException` not thrown

How to Use JUnit: Adding More Tests (3.2)

- Again, don't lose the big picture!
- JUnit test in previous slide automates `CounterTester1` and the following console interaction for `CounterTester3`:

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
dec
Value too small!
Enter "inc", "dec", or "val":
exit
Bye!
```

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

How to Use JUnit: Adding More Tests (4.1)

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

- **Lines 4 – 8**:
  We use a try-catch block to express that a VTLE *is not* expected.
- **Lines 9 – 15**:
  We use a try-catch block to express that a VTLE *is* expected.

How to Use JUnit: Adding More Tests (4.2)

- JUnit test in previous slide *automates* `CounterTester2` and the following console interaction for `CounterTester3`:

```
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
3
Enter "inc", "dec", or "val":
inc
Value too big!
Enter "inc", "dec", or "val":
exit
Bye!
```


How to Use JUnit: Adding More Tests (4.3)

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   } catch (ValueTooLargeException e) { }
12 }
```

No!

At **Line 9**, 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*.

How to Use JUnit: Adding More Tests (5)

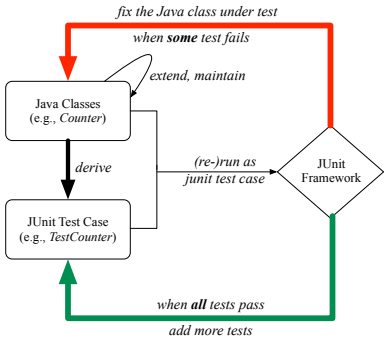
Loops can make it effective on generating test cases:

```
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   } catch(ValueTooLargeException e) {
16     fail("ValueTooLargeException is thrown unexpectedly");
17   } catch(ValueTooSmallException e) {
18     fail("ValueTooSmallException is thrown unexpectedly");
19   } }
```


Exercises

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 the error injection, and re-run all 8 tests. [What happens?]

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.

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)

Index (3)