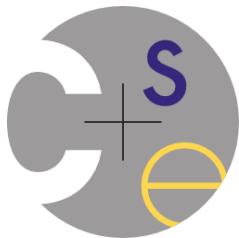# **Unit Testing with JUnit**
and other random notes

Posted 2015-07-13

Not required for HW, just
useful if you want to master java

Adapted from slides by Hye In Kim 2013

Contents

- Junit
- Generics
- Inheritance
- Comparators
- Iterators
- Commenting

Unit Testing with JUnit

Unit Testing

- Looking for errors in a subsystem in isolation
- Test one behavior at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails halts the test
  - You won't know whether a later assertion would have failed as well
- Tests should minimize logic - Bug in test code is hard to debug!
  - minimize use of if/else, loops, switch, etc.
- Torture tests are okay, but only *in addition* to simple tests

Junit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project → Properties → Build Path → Libraries → Add Library... → Junit → JUnit 4 → Finish**

- To create a test case:
  - right-click a file and choose:
    New → Test Case
  - Or click:
    File → New → JUnit Test
  - Eclipse can create method stubs
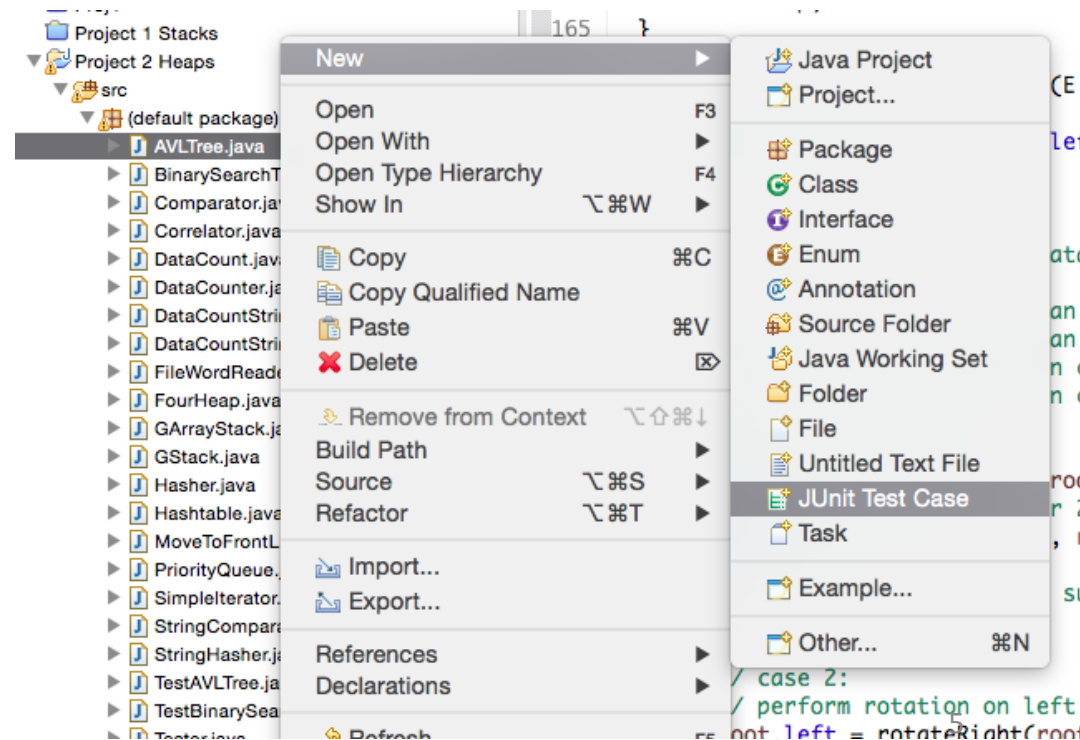
A JUnit Test Class

```java
import org.junit.*;
import static org.junit.Assert.*;

public class TestClassName {
    ...
    @Test
    public void testName() { // a test case method
        ...
    }
}
```

- A method with @Test is flagged as a JUnit test case
- All @Test methods run when JUnit runs your test class

JUnit Assertion Methods

Assertion	Description
`assertTrue(test)`	fails if the boolean test is false
`assertFalse(test)`	fails if the boolean test is true
`assertEquals(expected, actual)`	fails if the values are not equal
`assertSame(expected, actual)`	fails if the values are not the same (by ==)
`assertNotSame(expected, actual)`	fails if the values are the same (by ==)
`assertNull(value)`	fails if the given value is not null
`assertNotNull(value)`	fails if the given value is null
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails
  - `assertEquals("message", expected, actual)`

Good Testing Practices

```java
public class DateTest {
    // Give test case methods really long descriptive names
    @Test
    public void test_addDays_withinSameMonth() { ... }

    @Test
    public void  test_addDays_wrapToNextMonth() { ... }

    // Expected value should be at LEFT
    // Give messages explaining what is being checked
    @Test
    public void test_add_14_days() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }
}
```

Good Assertion Messages

```java
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

- JUnit will already show the expected and actual values
- Not needed in your assertion messages

Tests With a Timeout

- This test will fail if it doesn't finish running within 5000 ms

```
@Test(timeout = 5000)
public void name() { ... }
```

- Times out / fails after 2000 ms

```
Private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

Testing for Exceptions

```java
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it does throw the given exception
  - If the exception is not thrown, the test fails
  - Use this to test for expected errors

```java
@Test(expected = IndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);  // should throw exception
}
```

Setup and Teardown

- Create methods that run before or after each test case method is called

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- Create methods to run once before or after the entire test class runs

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```
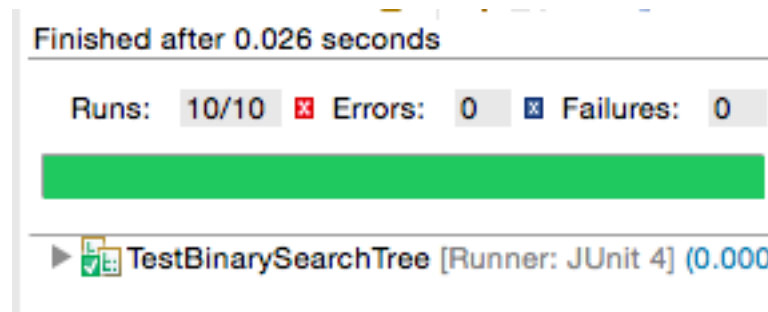
Test Case "Smells"

- Tests should be self-contained and not depend on each other
- "Smells" (bad things to avoid) in tests:
  - Constrained test order: Test A must run before Test B (usually a misguided attempt to test order/flow)

  - Tests that call each other: Test A calls Test B (calling a shared helper is OK, though)

  - Mutable shared state: Tests A and B both use a shared object (If A breaks it, what happens to B?)

Running a Test

- Right click the test class in the Eclipse Package Explorer and choose: **Run As → JUnit Test**
- The JUnit bar will show green if all tests pass, red if any fail
- The Failure Trace shows which tests failed, if any, and why

Finished after 0.026 seconds

Runs: 10/10 ☒ Errors: 0 ☒ Failures: 0

▶ TestBinarySearchTree [Runner: JUnit 4] (0.000

Generics

Generic Arrays

- Field & variable can have generic array type

  ```
  E[] elemArray;
  ```

- Cannot create new generic array
  - Arrays need to "know their element type"
  - Type "E" is unknown type

  ```
  E[] myArray = new E[INITIAL_SIZE]; //Error
  ```

- Workaround
  - Unavoidable warning, OK to suppress

  ```
  @SuppressWarnings("unchecked")
  E[] myArray = (E[]) new
  Object[INITIAL_SIZE]; //OK
  ```

Array of Parameterized Type

- Cannot create array of parameterized type

```
DataCount <E>[] dCount =
                    new DataCount<E>[SIZE]; // Error
```

- Object[] does not work - ClassCastException
  - Arrays need to "know their element type"
  - Object not guaranteed to be DataCount

```
DataCount <E>[] dCount =
        (DataCount<E>[]) new Object[SIZE]; // Error
```

- Specify it will always hold "DataCount"

```
DataCount<E>[] dCount =
        (DataCount<E>[]) new DataCount[SIZE]; // OK
```

Generics & Inner Classes

- Do not re-define the type parameter

```
class OuterClass<E> {          class OuterClass<E> {
    class InnerClass<E> {}         class InnerClass {}
}  // No ☹                     }  // Yes ☺
```

- Works, but not what you want!!
- Analogous of a local variable shadowing a field of the same name

```
class SomeClass {              class OuterClass<E> {
    int myInt;                     E myField;
    void someMethod() {            class InnerClass<E> {
        int myInt= 3;                  ...
        myInt++;                       E data = myField;
    }                              }
} // Not class field           } // Not the same type
```

Generic Methods

- A method can be generic when the class is not
  - Define the type variable at the method

  ```
  public static <E> void insertionSort
  (E[] array, Comparator<E> comparator);
  ```

- More generics
  http://docs.oracle.com/javase/tutorial/java/generics/index.html

Generic Wildcards

- Used to denote super/subtype of type parameter
- Upper bounded wildcard: `<? extends E>`
  - E and every subtype (subclass) of E
- Lower bounded wildcard: `<? super E>`
  - E and every supertype (superclass) of E
- Consider `<? extends E>` for parameters and `<? super E>` for return types
  - The only use in Project 2 is with the comparator

Inheritance

Interface & Inheritance

- Interface provides list of methods a class promises to implement
  - <u>Inheritance</u>: is-a relationship and code sharing
  - <u>Interfaces</u>: is-a relationship without code sharing
- Inheritance provides code reuse **Style Points!!**
  - Take advantage of inherited methods
  - Do not reimplement already provided functionality
  - Override only when it is necessary

Comparators

Comparing Objects

- Less-than (<) and greater-than (>) operators do not work with objects in Java

- Two ways of comparing:

  1. Implement **<u>Comparable</u>** interface

     - Natural ordering: 1, 2, 3, 4 ...

     - One way of ordering

  2. Use **<u>Comparator</u>** **<- Project 2**

     - Many ways of ordering

Comparable Interface

```
public interface Comparable<T> {
  public int compareTo(T other);
}
```

- A call of **A.**`compareTo`(**B**) should return:
  - a value < 0 if **A** comes "before" **B** in the ordering
  - a value > 0 if **A** comes "after" **B** in the ordering
  - or exactly 0 if **A** and **B** are considered "equal" in the ordering

What's the "natural" order?

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?
- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles a second way?

Comparator Interface

```
public interface Comparator<T> {
  public int compare(T first, T second);
}
```

- Interface Comparator:
  - External object specifies comparison function
  - Can define multiple orderings

Comparator Examples

```java
public class RectangleAreaComparator
                    implements Comparator<Rectangle>{
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
public class RectangleXYComparator
                    implements Comparator<Rectangle>{
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Using Comparators

- TreeSet and TreeMap can accept a Comparator parameter

```
Comparator<Rectangle> comp = new RectangleAreaComparator();
   Set<Rectangle> set = new TreeSet<Rectangle>(comp);
```

- Searching and sorting methods can accept comparators.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

- Methods are provided to reverse a comparator's ordering:

```
Collections.reverseOrder()
Collections.reverseOrder(comparator)
```

Iterators

Iterator

- Object that allows traverse elements of collection
  - Anonymous Class: Combined class declaration and instantiation.

```
public SimpleIterator<DataCount<E>> getIterator() {
    return new SimpleIterator <DataCount<E>>() {
        // Returns true if there are more elements to examine
        public boolean hasNext() {
            ...
        }
        // Returns the next element from the collection
        public DataCount<E> next() {
            if(!hasNext()) {
                throw new NoSuchElementException();
            }
            ...
        }
    }; // ← Notice the semicolon here!
}
```

Commenting

Commenting - Preconditions

- Precondition: Something assumed to be true at the start of a method call.

```java
// Returns the element at the given index.
// Precondition: 0 <= index < size
public int get(int index) {
    return elementData[index];
}
```

Index	0	1	2	3	4	5	6	7	8	9
Value	3	8	9	7	5	12	0	0	0	0

Size   6

- Stating a precondition doesn't "solve" the problem of users passing improper indexes, but it at least documents our decision and warns the client what not to do

Commenting - Postconditions

- Postcondition: Something your method promises will be true at the end of its execution, if all preconditions were true at the start

```
// Makes sure that this list's internal array is large
// enough to store the given number of elements.
// Precondition: capacity >= 0
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    while (capacity > elementData.length) {
        elementData = Arrays.copyOf(elementData,
            2 * elementData.length);
    }
}
```

- If your method states a postcondition, clients should be able to rely on that statement being true after they call the method

Javadoc Comments

- Put on all class headers, public methods and constructors

- Eclipse and other editors have useful built-in Javadoc support

```
/**
 * Description of class/method/field/etc.
 *
 * @tag attributes
 * @tag attributes
 * ...
 * @tag attributes
 */
```

Javadoc Tags

On a class header

Tag	Description
@author *name*	author of a class
@version *number*	class version number in any format

On a method or constructor

Tag	Description
@param *name description*	describes a parameter
@return *description*	describes what value will be returned
@throws *ExceptionType reason*	describes an exception that may be thrown and what would cause it

Javadoc Example

```java
/**
 * Each BankAccount object models the account information
 * for a single user of Fells Wargo bank.
 * @author James T. Kirk
 * @version 1.4 (Aug 9 2008)
 */
public class BankAccount {
    /** The standard interest rate on all accounts. */
    public static final double INTEREST_RATE = 0.03;
    ...
    /**
     * Deducts the given amount of money from this account's
     * balance, if possible, and returns whether the money was
     * deducted successfully (true if so, false if not).
     * If the account does not contain sufficient funds to
     * make this withdrawal, no funds are withdrawn.
     *
     * @param amount the amount of money to be withdrawn
     * @return true if amount was withdrawn, else false
     * @throws IllegalArgumentException if amount is negative
     */
    public boolean withdraw(double amount) {...}
}
```

Javadoc Output as HTML

- Java includes tools to convert Javadoc comments into web pages
    - In terminal: **`javadoc -d doc/ *.java`**
    - In Eclipse: **Project → Generate Javadoc…**
- The Java API webpages are generated from Sun's Javadoc comments on the actual source code.

Comments - Clear and Helpful

```
/** Takes an index and element and adds the element there.
* @param index index to use
* @param element element to add
*/
public boolean add(int index, E element) { ...
```

Instead…

```
/** Inserts the specified element at the specified position in
* this list. Shifts the element currently at that position (if
* any) and any subsequent elements to the right (adds one to
* their indices). Returns whether the add was successful.
* @param index index at which the element is to be inserted
* @param element element to be inserted at the given index
* @return true if added successfully; false if not
* @throwsIndexOutOfBoundsExceptionif index out of range
* ({@code index < 0 || index > size()})
*/
publicbooleanadd(intindex, E element) { ...
```

Javadoc and private

- Private internal methods do not need Javadoc comments

- Private members do not appear in the generated HTML pages

```
/** ... a Javadoc comment ... */
public void remove(int index) { ... }
// Helper does the real work of removing
// the item at the given index.
private void removeHelper(int index) {
    for (int i = index; i < size - 1; i++) {
        elementData[i] = elementData[i + 1];
    }
    elementData[size - 1] = 0;
    size--;
}
```

Custom Javadoc Tags

- Javadoc doesn't have tags for pre/post, but you can add them
  - By default, these tags wont appear in the generated HTML but...

Tag	Description
@pre *condition* *(or @precondition)*	Notes a precondition in the API documentation; describes a condition that must be true for the method to perform it's functionality
@post *condition* *(or @postcondition)*	Notes a postcondition in API documentation; describes a condition that is guaranteed to be true at the *end* of the method's functionality, so long as all preconditions were true at the *start* of the method.

Apply Custom Javadoc Tags

- In terminal:

  ```
  javadoc -d doc/
  -tag pre:cm:"Precondition:"
  -tag post:cm:"Postcondition:" *.java
  ```

- In Eclipse:
  Project → Generate Javadoc… → Next → Next →
  in the "Extra Javadoc options" box,
  ```
  -tag pre:cm:"Precondition:" –tag post:cm:"Postcondition:"
  ```

- The generated webpages will now display pre and post tags properly!