

UNIT TESTING

Krysta Yousoufian
CSE 331 Section
April 19, 2012

With material from Marty Stepp, David Notkin, and *The Pragmatic Programmer*



JUnit Semantics

How to write a technically correct JUnit test

A JUnit test class

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

public class name {
    ...

    @Test
    public void name() { // 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.

Verifying Behavior with Assertions

- Assertions: special JUnit methods
- Verifies that a value matches expectations

```
assertEquals(42, meaningOfLife()); ← fails if meaningOfLife() != 42
```

```
assertTrue(list.isEmpty()); ← fails if list.isEmpty() is false
```

- If the value isn't what it should be, the test fails
 - Test immediately terminates
 - Other tests in the test class are still run as normal
 - Results show details of failed tests

Using Assertions

<code>assertTrue (test)</code>	fails if the boolean test is <code>false</code>
<code>assertFalse (test)</code>	fails if the boolean test is <code>true</code>
<code>assertEquals (expected, actual)</code>	fails if the values are not equal
<code>assertSame (expected, actual)</code>	fails if the values are not the same (by <code>==</code>)
<code>assertNotSame (expected, actual)</code>	fails if the values <i>are</i> the same (by <code>==</code>)
<code>assertNotNull (value)</code>	fails if the given value is <i>not</i> <code>null</code>
<code>assertNotNull (value)</code>	fails if the given value is <code>null</code>

- And others: <http://www.junit.org/apidocs/org/junit/Assert.html>
- Each method can also be passed a string to display if it fails:
 - e.g. `assertEquals ("message", expected, actual)`

Checking for Exceptions

- Verify that a method throws an exception when it should
- Place above method:
`@Test (expected=IllegalArgumentException.class)`
- Test passes if specified exception is thrown, fails otherwise
- *Only time it's OK to write a test with no asserts!*

```
// Try to access the first item in an empty ArrayList
@Test (expected=IndexOutOfBoundsException.class)
public void test() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}
```

Setup and Teardown

- Methods to run before/after each test case method is called:

@Before

```
public void name() { ... }
```

@After

```
public void name() { ... }
```

- Methods to run once before/after the entire test class runs:

@BeforeClass

```
public static void name() { ... }
```

@AfterClass

```
public static void name() { ... }
```

Don't Repeat Yourself (let me repeat that...)

- JUnit tests are just regular Java code!
- Can declare fields for frequently-used values or constants

```
private static final String DEFAULT_NAME = "MickeyMouse";  
private static final User DEFAULT_USER =  
    new User("lazowska", "Ed", "Lazowska");
```

- Can write helper methods, etc.

```
private void eq(RatNum ratNum, String rep) {  
    assertEquals(rep, ratNum.toString());  
}
```

```
private BinaryTree getTree(int[] items) {  
    // construct BinaryTree and add each element in items  
}
```


Unit Test Best Practices

How to craft well-written JUnit tests

#1: Use descriptive asserts, test names

- When a test fails, JUnit tells you:
 - Name of test method
 - Message passed into failed assertion
 - Expected and actual values of failed assertion
- The more descriptive this information is, the easier it is to diagnose failures
- Avoid `System.out.println()`
 - Want any diagnostic info to be captured by JUnit and associated with that test method

#1: Use descriptive asserts, test names

- **Test name:** describe what's being tested
 - Good: “testAddDaysWithinMonth,” ...
 - Not so good: “testAddDays1,” “testAddDays2,” ...
 - Useless: “test1,” “test2,” ...
 - Overkill:
“testAddDaysOneDayAndThenFiveDaysThenNegativeFourDaysStartingOnJanuaryTwentySeventhAndMakeSureItRollsBackToJanuaryAfterRollingToFebruary()”

#1: Use descriptive asserts, test names

- **Assertions:** take advantage of expected & actual values
- Make sure you have the right order:

```
assertEquals(message, expected, actual)
```

- Use the right assert for the occasion:

```
assertEquals(expected, actual) instead of  
assertTrue(expected.equals(actual))  
(why?)
```

```
assertTrue(b) instead of assertEquals(true, b)  
(why?)
```

#1: Use descriptive asserts, test names

- **Assertion message:** contribute new information
 - No need to repeat expected/actual values or info in test name
 - e.g. details of what happened before the failure

Example:

```
@Test
public void test_addDays_wrapToNextMonth() {
    Date actual = new Date(2050, 2, 15);
    actual.addDays(14);
    Date expected = new Date(2050, 3, 1);
    assertEquals("date after +14 days", expected, actual);
}
```

Let's put it all together!

```
public class DateTest {  
  
    ...  
  
    // Test addDays when it causes a rollover between months  
    @Test  
    public void testAddDaysWrapToNextMonth() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected,  
            actual);  
    }  
}
```

Let's put it all together!

```
public class DateTest {
```

```
...
```

```
// Test addDays when it causes a rollover
```

```
@Test
```

```
public void testAddDaysWrapToNextMonth() {
```

```
    Date actual = new Date(2050, 2, 15);
```

```
    actual.addDays(14);
```

```
    Date expected = new Date(2050, 3, 1);
```

```
    assertEquals("date after +14 days", expected,  
                actual);
```

```
}
```

Descriptive method
name

Let's put it all together!

```
public class DateTest {
```

```
...
```

```
// Test that Date.addDays() uses a rollover between months
```

```
@Test
```

Tells JUnit that this method is a test to run

```
public void testAddDaysWrapToNextMonth() {
```

```
    Date actual = new Date(2050, 2, 15);
```

```
    actual.addDays(14);
```

```
    Date expected = new Date(2050, 3, 1);
```

```
    assertEquals("date after +14 days", expected,  
                actual);
```

```
}
```


Let's put it all together!

```
public class DateTest {
```

```
...
```

```
// Test addDays when it causes a rollover between months  
@Test
```

```
public void testAddDaysWrapToNextMonth() {
```

```
    Date actual = new Date(2050, 2, 15);
```

```
    actual.addDays(14);
```

```
    Date expected = new Date(2050, 3, 1);
```

```
    assertEquals("date after +14 days", expected,  
                actual);
```

```
}
```

Method names describe
function of each object

Let's put it all together!

```
public class DateTest {  
  
    ...  
  
    // Test addDays when it causes a rollover between months  
    @Test  
    public void testAddDaysWrapToNextMonth() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("addDays(14) should wrap to next month", expected,  
            actual);  
    }  
}
```

Use assertion to check
expected results

Let's put it all together!

```
public class DateTest {
```

```
...
```

```
// Test addDays when it causes a rollover between months  
@Test
```

```
public void testAddDaysWrapToNextMonth() {
```

```
    Date actual = new Date(2050, 2, 15);
```

```
    actual.addDays(14);
```

```
    Date expected = new Date(2050, 3, 1);
```

```
    assertEquals("date after +14 days", expected,  
                actual);
```

```
}
```

Message gives details about the test in case of failure

Let's put it all together!

```
public class DateTest {
```

```
...
```

```
// Test addDays when it causes a rollover between months  
@Test
```

```
public void testAddDaysWrapToNextMonth() {
```

```
    Date actual = new Date(2050, 2, 15);
```

```
    actual.addDays(14);
```

```
    Date expected = new Date
```

Expected value first,
actual value second

```
    assertEquals("date after +14 days", expected,  
                actual);
```

```
}
```

Let's put it all together!

```
public class DateTest {  
  
    ...  
  
    // Test addDays when it causes a rollover between months  
    @Test  
    public void testAddDaysWrapToNextMonth() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(14);  
        Date expected = new Date(2050, 3, 1);  
        assertEquals("date after +14 days", expected,  
            actual);  
    }  
}
```

That's it! Test is
short & sweet

#2: Keep tests small

- Ideally, each test only tests one “thing”
 - One “thing” usually means one method under one input condition
- Low-granularity tests help you isolate bugs
 - Tell you exactly what failed and what didn’t
- Only a few (likely one) assert statements per test
 - Test halts after first failed assertion
 - Don’t know whether later assertions would have failed
- Where possible, only test one method at a time
 - Not always possible – but if you test `x()` using `y()`, try to test `y()` in isolation in another test
 - E.g. if you test `add()` using `contains()`, separately test `contains()` before any items are added

What NOT to do

- [IntArrayTest](#)
- What's wrong?

What NOT to do

- [IntArrayTest](#)
- What's wrong?
- testIntArray tests way too many things
 - Too many methods, array states
- Solution: break down by method being tested and/or state of array
- [IntArrayTestBetter](#)

#3: Choose the right tests

- Given a finite number of tests, want reasonable confidence in an infinite number of inputs
- Input = initial state of object + method arguments + ...

#3: Choose the right tests

- For each method, ask: what are the equivalence classes?
 - Items in a collection: none, one, many
- Write a test for each equivalence class
- Consider common input categories
 - `Math.abs()`: negative, zero, positive values
- Consider boundary cases
 - Inputs on the boundary between equivalence classes
 - `Person.isMinor()`: `age < 18`, **`age == 18`**, `age > 18`
- Consider edge cases
 - `-1`, `0`, `1`, empty list, `arr.length`, `arr.length-1`
- Consider error cases
 - Empty list, null object

Other guidelines

- Test all methods
 - Caveat: constructors don't necessarily need explicit testing
- Keep tests simple – avoid complicated logic
 - minimize `if/else`, `loops`, `switch`, `etc.`
 - Don't want to debug your tests!
- Tests should always have at least one assert
 - *Unless* testing that an exception is thrown
 - Simply testing that an exception is *not* thrown is not necessary
 - `assertTrue(true)` ; doesn't count!

Other guidelines

- Tests should be *isolated*
 - Not dependent on side effects of other tests
 - Should be able to run in any order
- Use helper methods to factor out common operations
 - E.g. setting up initial state of an object

Example: Date

- `public Date(int year, int month, int day)`
 - `public Date() // today`
 - `public int getDay(), getMonth(), getYear()`
 - `public void addDays(int days) // advances by days`
 - `public int daysInMonth()`
 - `public String dayOfWeek() // e.g. "Sunday"`
 - `public boolean equals(Object o)`
 - `public boolean isLeapYear()`
 - `public void nextDay() // advances by 1 day`
 - `public String toString()`
- **Come up with unit tests to check the following:**
 - That no `Date` object can ever get into an invalid state.
 - That the `addDays` method works properly.
 - It should be efficient enough to add 1,000,000 days in a call.

Example: IntStack

- What tests should we write?

More examples

- How would we test the following Collections interface methods:
- [Collections.binarySearch](#)
- [Collections.sort](#)
- ...
- (Assume the `List` we pass in has already been tested)

JUnit Summary

- Tests need *failure atomicity* (ability to know exactly what failed).
 - Each test should have a descriptive name.
 - Assertions should have clear messages to know what failed.
 - Write many small tests, not one big test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always `assertTrue`.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use helpers, `@Before` to reduce redundancy between tests.