# Eclipse & Java Tools

Krysta Yousoufian

CSE 331 Section

January 12, 2012

# Homework 2

- Due Wednesday 1/18
- Part 1: written problems
- Part 2: Eclipse/Java setup
- Today: introduce a lot of part 2 stuff

# Javadoc

# Javadoc

- Standard format for documenting Java code
- Defines a specific structure for comments
  - Free-form descriptions + `@tags`
- Used mainly for *public* documentation
  - Not internal details
- Auto-generate HTML page for a class
  - Official Java API specs are all javadoc!
- Uses:
  - Publish javadoc pages (HTML)
  - Eclipse auto-complete
  - Reading source code

# Writing Javadoc

*A regular comment*

```
/* some stuff */
```

*A Javadoc comment*

```
/** some stuff */
```

Or

```
/**
    some stuff
    @someTag more stuff
*/
```

**Notice the extra asterisk**

# Writing Javadoc

- Use to describe public interface:
  - Methods
  - Classes
  - Constants an enums
- Don't use for:
  - Describing implementation details
  - Internal comments within methods
- Comment goes immediately above corresponding field/method/class

# Common Javadoc tags

@param name description
    Describes a parameter
@return description
    Describes return value
@throws ExceptionType reason
    Describes when an exception may be thrown

**Method**

@author
    Author of a class
@version
    Class's version number (any format)

**Class**

# Example

```
/**
* 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   * make this withdrawal, no funds are withdrawn.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   * @param amount the amount of money to be withdrawn
   * @return true if amount was withdrawn, else false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) {
   ...
```

# Example

```
/**
* An instrument section of a symphony orchestra.
* @author John Williams
*/
public enum OrchestraSection {
   /** Woodwinds, such as flute, clarinet, and oboe. */
   WOODWIND,
   /** Brass instruments, such as trumpet. */
   BRASS,
   /** Percussion instruments, such as cymbals. */
   PERCUSSION,
   /** Stringed instruments, such as violin and cello. */
   STRING;
}
```

# What goes in @param/return

- Don't repeat yourself or write vacuous comments.

```
/** 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) { ...
```

- better:

```
/** 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
* @throws IndexOutOfBoundsException if index out of range
* ({@code index < 0 || index > size()})
*/
public boolean add(int index, E element) { ...
```

# Javadoc is extensible

- In 331, we add new tags:
- `requires`: precondition
- `modifies, effects`: postcondition
  - `modifies`: any objects that may be affected by method
    - Parameters or `this`; **not** internal fields (why?)
  - `effects`: what this method changes/causes

# Example

```
/**
@requires lst1 and lst2 are non-null
          lst1 and lst2 are the same size
@modifies lst1
@effects ith element of lst2 is added to ith element
         of lst1
*/
static void listAdd(List<Integer> lst1,
       List<Integer> lst2) {
```

# JUnit

# Testing

- Need to ensure code works correctly
- Manual testing is slow and tedious
- Testing process:
  - Run a method
  - Determine what we expect it to do
  - Verify that it did that
  - Repeat for other methods, inputs until satisfied the program is correct
- Want to automate this process

# Unit Testing

- Test small components of a program in isolation (why?)
- JUnit: framework for unit testing in Java
- How it works:
  - For each class `Foo`, create a test class `FooTest`
  - Write test methods, each testing one thing
  - Run `FooTest` to find out whether the right thing happened in each test

# Verifying behavior: assertions

- Assertions: collection of JUnit methods
  - assertEquals, assertTrue, assertNotNull, …
- Verifies that a value matches expectation

```
assertEquals(42, meaningOfLife());
assertFalse(list.isEmpty());
```

- If an assertion fails, test method immediately fails and terminates
- Afterward, JUnit shows details about the failure (e.g. expected & actual values)

# Verifying behavior: fail()

- fail()
  - Causes test to immediately fail

```
if ( result != 0)
    fail();
```

  - **Not recommended!** – tells you nothing about the failure afterward
  - assertEquals(0, result) would show the actual value of result if it failed

# Verify behavior: exceptions

- Can make sure a method throws an exception
- To test for an IllegalArgumentException:

```
@Test(expected=IllegalArgumentException.class)
public void testFooWithNullArg() {
    foo(null);
 }
```

- Test fails if exception is not thrown

# Example

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
      IntArray arr = new IntArray();
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

# Example

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
        IntArray arr = new IntArray();
        arr.add(10);
        arr.add(20);
        arr.add(30);
        arr.clear();
        assertTrue(arr.isEmpty());
}
```

Tells JUnit that this method is a test to run

# Example

Comment describes what's being tested

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
      IntArray arr = new IntArray();
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

# Example

```
// Tests that clear() remove
@Test
public void testClearSeveral
      IntArray arr = new Int
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

Method name describes what is being tested, too (useful when reading list of test results)

# Example

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
      IntArray arr = new IntArray();
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

Use assertion to check behavior of clear()

# Example

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
      IntArray arr = new IntArray();
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

Use assertion to check expected results

# Example

```
// Tests that clear() removes all elements
@Test
public void testClearSeveralElements() {
      IntArray arr = new IntArray();
      arr.add(10);
      arr.add(20);
      arr.add(30);
      arr.clear();
      assertTrue(arr.isEmpty());
}
```

That's it! Test is short & sweet

# Keep unit tests small

- Ideally, each method only tests one thing
  - Test one method for one input condition
  - One assert statement
- If a test fails, want to know exactly what failed
- If program has bug, want low-granularity tests to help isolate it

# What NOT to do!

```
// Tests that items can be added and removed
@Test
public void testAddClear() {
	IntArray arr = new IntArray();
	arr.add(10);
	assertTrue(arr.contains(10));
	arr.add(20);
	assertTrue(arr.contains(20));
	arr.add(30);
	assertTrue(arr.contains(10));
	assertTrue(arr.contains(20));
	assertTrue(arr.contains(30));

	arr.remove(30);
	assertTrue(arr.contains(10));
	assertTrue(arr.contains(20));
	assertFalse(arr.contains(30));
	arr.remove(20);
	assertFalse(arr.contains(20));
	arr.remove(10);
	assertFalse(arr.contains(10));
}
```

# What NOT to do

```
// Tests that items can be added and removed
@Test
public void testAddClear() {
	IntArray arr = new IntArray();
	arr.add(10);
	assertTrue(arr.contains(10));
	arr.add(20);
	assertTrue(arr.contains(20));
	arr.add(30);
	assertTrue(arr.contains(10));
	assertTrue(arr.contains(20));
	assertTrue(arr.contains(30));

	arr.remove(30);
	assertTrue(arr.contains(10));
	assertTrue(arr.contains(20));
	assertFalse(arr.contains(30));
	arr.remove(20);
	assertFalse(arr.contains(20));
	arr.remove(10);
	assertFalse(arr.contains(10));
}
```

Too long!

# More guidelines

- One test class per regular class
- Keep tests simple to avoid bugs
- Comment each test
- Descriptive method names (within reason)
  - No “testGetItem1,” “testGetItem2,” …
- Test all methods
  - Constructors don’t necessarily need explicit testing
- Test both common cases and edge cases

# More guidelines: assertions

- Don't use System.out.println() (why?)
- Always have at least one assert
  - *Unless* checking that an exception is thrown
  - Trivial asserts don't count: assertTrue(true);
- Use assert() rather than fail()
- Use the right assert for the occasion
  - assertTrue(b) instead of assertEquals(true, b)
  - assertEquals(expected, actual) instead of assertTrue(expected == actual)