



CSE332: Data Abstractions

Section 3

Hyeln Kim

CSE 331 Slides (JUnit/Javadoc)

Spring2014

Section Agenda

- **Project 2: Shake-n-Bacon**
 - More Generics
 - Comparator
 - Inheritance review
 - Iterator / Anonymous class
 - Javadoc Commenting
 - JUnit Testing
- **HW Q & A**

Generics

Generic Arrays & Wildcard

Generic Arrays

- **Field & variable can have generic array type**

```
E[] elemArray;
```

- **Cannot create new generic array**

```
E[] elemArray = new E[INITIAL_CAPACITY]; // Error
```

- Arrays need to “know their element type”
- Type “E” is unknown type

- **Workaround with Object[] - Unavoidable warning**

```
E[] elemArray = (E[]) new Object[INITIAL_CAPACITY];  
// Generates warning, but ok
```

Array of Parameterized type

- **Cannot create array of parameterized type**

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

- **Object[] does not work - ClassCastException**

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

- Arrays need to “know their element type”
- Object not guaranteed to be DataCount

- **Specify it will always hold “DataCount”**

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

Generics & Inner Class

- **Do not re-define type parameter**

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

```
class OuterClass<E> {  
    class InnerClass {}  
} // Yes 😊
```

- Works, but not what you want!!

- Analogous of local variable shading field

```
class SomeClass {  
    int myInt;  
    void someMethod() {  
        int myInt = 3;  
        myInt ++;  
    } // Not the field  
}
```

```
class OuterClass<E> {  
    E myField;  
    class InnerClass<E> {  
        ...  
        E data = myField;  
    } // Not the same type!!  
}
```

Generic Methods

- **A method can be generic when the class is not**

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

- Define the type variable at the method

- **More generics**

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

Wildcard

- **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,**
`<? super E>` **for return type**
 - The only use in this project is with comparator

```
public BinarySearchTree(Comparator<? super E> c);
```

Inheritance

Superclass & Interface

Interface & Inheritance

- **Interface provides list of methods a class promise to implement**
 - Inheritance: is-a relationship *and* code sharing.
 - `AVLTree` can be treated as `BinarySearchTree` and inherits code.
 - Interfaces: is-a relationship *without* code sharing.
 - `FourHeap` can be treated as `Heap` but inherits no code.
- **Inheritance provides code reuse** **Style Points!!**
 - Take advantage of inherited methods
 - Do not re-implement already provided functionality
 - Override only when it is necessary

Comparing Objects

Comparable & Comparator

Comparing objects

- Operators $<$, $>$ do not work with objects in Java
- **Two ways of comparing:**
 1. Implement Comparable Interface
 - Natural Ordering: 1, 2, 3, 4 ...
 - One way of ordering
 2. Use Comparator **<- Project 2**
 - Many ways of ordering

The 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?

```
public class Rectangle implements Comparable<Rectangle> {  
    private int x, y, width, height;  
  
    public int compareTo(Rectangle other) {  
        // ...?  
    }  
}
```

- 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 into some order anyway?

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

```
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)
```

Iterator

objects that traverse collections

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();  
            }  
            ...  
        }  
    };  
}
```

Project 2 Tips

Style Guide

Project 2 Tips

- Take advantage of superclass's implementation when writing subclass
- Minimize casting
 - Remember **AVLNode is-a BSTNode**
 - AVLNode can be treated as BSTNode, **only except** when accessing its **height** information
 - Consider some private function like (**only cast in this function**):
private int height(BSTNode node) { ... }
private void updateHeight(BSTNode node) {..}

Commenting

Javadoc

Preconditions

- **precondition:** Something *assumed to be true* at the start of a call.

```
// Returns the element at the given index.
```

```
// Precondition: 0 <= index < size
```

```
public int get(int index) {  
    return elementData[index];  
}
```

<i>index</i>	0	1	2	3	4	5	6	7	8	9
<i>value</i>	3	8	9	7	5	12	0	0	0	0
<i>size</i>	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.

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

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

- **Javadoc comments:**

- Put on all class headers, public methods and constructors
- Eclipse and other editors have useful built-in Javadoc support

Javadoc tags

- on a method or constructor:

tag	description
@param <i>name</i> <i>description</i>	describes a parameter
@return <i>description</i>	describes what value will be returned
@throws <i>ExceptionType</i> <i>reason</i>	describes an exception that may be thrown (and what would cause it to be thrown)

- on a class header:

tag	description
@author <i>name</i>	author of a class
@version <i>number</i>	class's version number, in any format

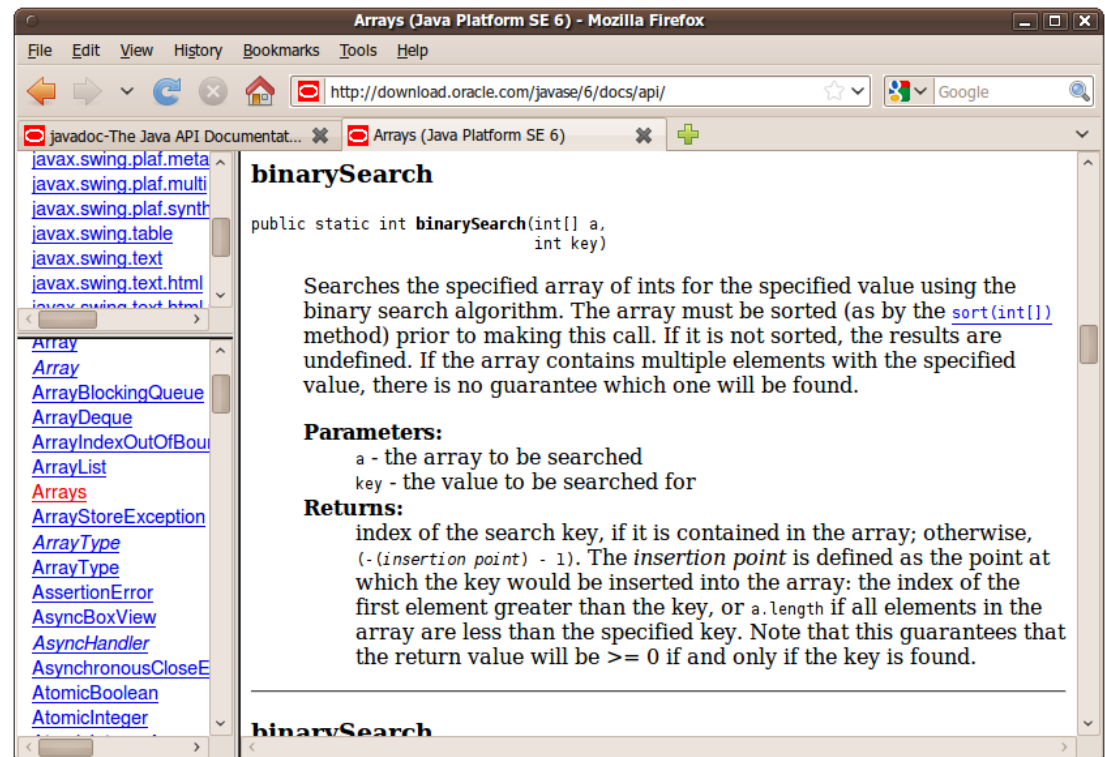
Javadoc example

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

Javadoc output as HTML

- Java includes tools to convert Javadoc comments into web pages
 - from Terminal: `javadoc -d doc/ *.java`
 - Eclipse has this built in: Project → Generate Javadoc...
- The actual Java API spec web pages are generated from Sun's Javadoc comments on their own source code:



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 and private

- Private internal methods do not need Javadoc comments:

```
/** ... 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--;
}
```

- Private members do not appear in the generated HTML pages.

Custom Javadoc tags

- Javadoc doesn't have tags for pre/post, but you can add them:

tag	description
@pre <i>condition</i> (or @precondition)	notes a precondition in API documentation; describes a condition that must be true for the method to perform its functionality
@post <i>condition</i> (or @postcondition)	notes a postcondition in API documentation; describes a condition that is guaranteed to be true at the <i>end</i> of the method's functionality, so long as all preconditions were true at the <i>start</i> of the method

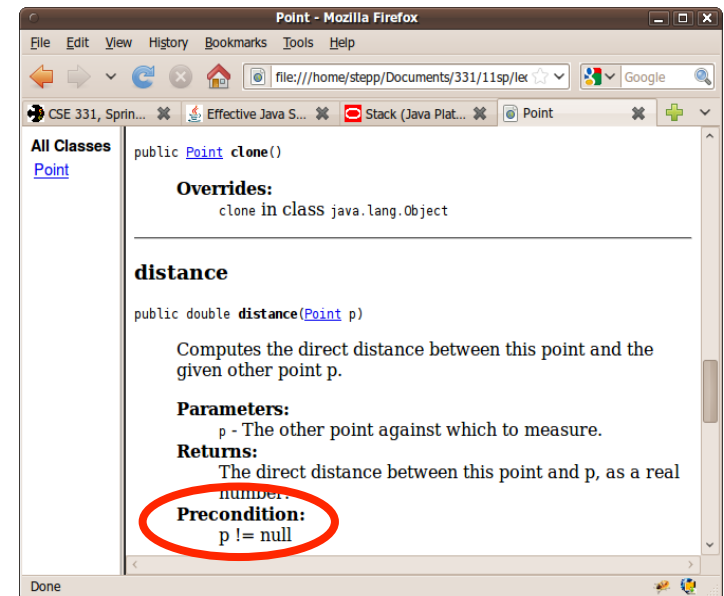
- By default, these tags won't show up in the generated HTML. But...

Applying custom Javadoc tags

- from 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, type:

`-tag pre:cm:"Precondition:" -tag post:cm:"Postcondition:"`

- The generated Java API web pages will now be able to display `pre` and `post` tags properly!



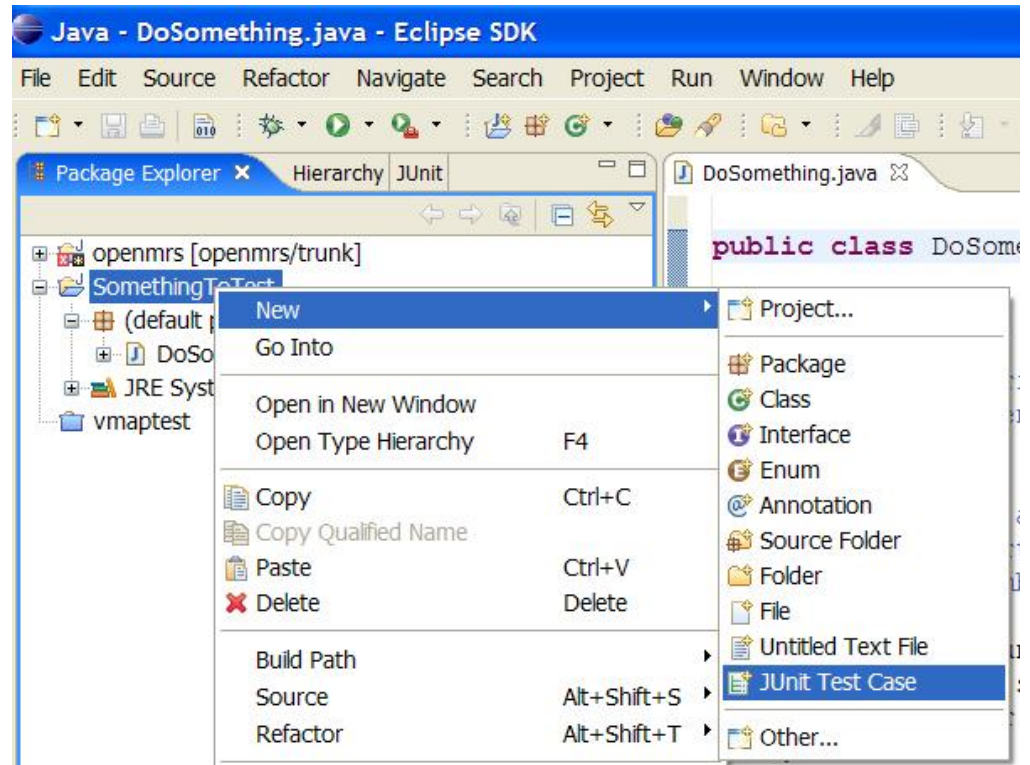
JUnit

Unit testing: Looking for errors in a subsystem in isolation

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 Case**
 - Eclipse can create stubs of method tests for you.



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.

JUnit assertion methods

<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>
<code>fail ()</code>	causes current test to immediately fail

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

Trustworthy tests

- Test **one thing 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 stops the test.
 - You won't know whether a later assertion would have failed.
- Tests should minimize logic – Bug in test code is hard to debug!
 - minimize `if/else`, `loops`, `switch`, etc.
- Torture tests are okay, but only *in addition to* simple tests.

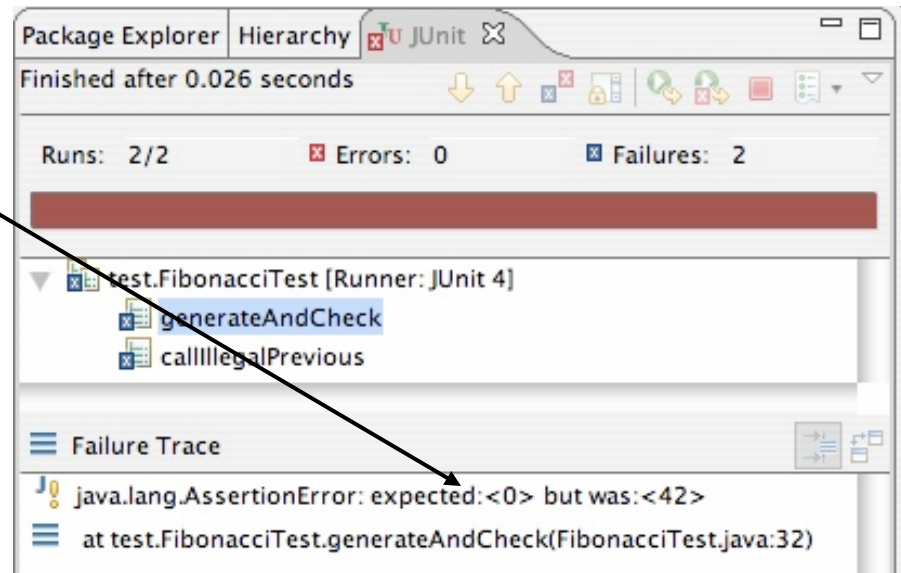
Good Practices

```
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

```
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 in its output;
//
// don't need to repeat them
// in the assertion message
```



Tests with a timeout

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

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...
```

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

- Times out / fails after 2000 ms

Testing for exceptions

```
@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.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayList list = new ArrayList();
    list.get(4);    // should fail
}
```

Setup and teardown

@Before

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

@After

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

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

@BeforeClass

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

@AfterClass

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

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

Flexible helpers

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addhelper(d, +32, 2050, 4, 2);
        addhelper(d, +98, 2050, 7, 9);
    }

    // Helpers can box you in; hard to test many calls/combine.
    // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                            int y2, int m2, int d2) {
        Date date = new Date(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    private void addHelper(Date date, int add,
                            int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect,
d);
    }
}
```

Test case "smells"

- Tests should be self-contained and not care about 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 call each other* : Test A calls Test B's method
(calling a shared helper is OK, though)
 - *Mutable shared state* : Tests A/B both use a shared object.
(If A breaks it, what happens to B?)



Running a test

- Right click it in the Eclipse Package Explorer at left; 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.

