CSE 143
Lecture 5

More `ArrayIntList`:

Pre/postconditions; exceptions; testing and JUnit

reading: 15.2 - 15.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Problem: size vs. capacity

- What happens if the client tries to access an element that is past the size but within the capacity (bounds) of the array?
  - Example: `list.get(7);` on a list of size 5 (capacity 10)

index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	0	0	0	0	0
size	5									

  - Answer: Currently the list allows this and returns 0.
    - Is this good or bad? What (if anything) should we do about it?

Preconditions

- **precondition**: Something your method *assumes is true* at the start of its execution.
  - Often documented as a comment on the method's header:

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

  - Stating a precondition doesn't "solve" the problem, but it at least documents our decision and warns the client what not to do.
  - *What should we do if the client violates the precondition?*

Throwing exceptions (4.5)

```
throw new ExceptionType();
throw new ExceptionType("message");
```

- Causes the program to immediately crash with an exception.

- Common exception types:
  - ArithmeticException, ArrayIndexOutOfBoundsException, FileNotFoundException, IllegalArgumentException, IllegalStateException, IOException, NoSuchElementException, NullPointerException, RuntimeException, UnsupportedOperationException

- Why would anyone ever *want* a program to crash?

Exception example

```
public void get(int index) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return elementData[index];
}
```

– Exercise: Modify the rest of `ArrayIntList` to state preconditions and throw exceptions as appropriate.

Private helper methods

```
private type name(type name, ..., type name) {
    statement(s);
}
```

- a **private method** can be seen/called only by its own class
  - your object can call the method on itself, but clients cannot call it
  - useful for "helper" methods that clients shouldn't directly touch

```
private void checkIndex(int index, int min, int max) {
    if (index < min || index > max) {
        throw new IndexOutOfBoundsException(index);
    }
}
```

Postconditions

- **postcondition**: Something your method *promises will be true* at the *end* of its execution.
  - Often documented as a comment on the method's header:

```
// Makes sure that this list's internal array is large
// enough to store the given number of elements.
// Postcondition: elementData.length >= capacity
public void ensureCapacity(int capacity) {
    // double in size until large enough
    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.

Thinking about testing

- If we wrote `ArrayIntList` and want to give it to others, we must make sure it works adequately well first.

- Some programs are written specifically to test other programs. We could write a client program to test our list.
  - Its `main` method could construct several lists, add elements to them, call the various other methods, etc.
  - We could run it and look at the output to see if it is correct.

  - *But that is tedious and error-prone; there is a better way.*

Testing with JUnit

(in brief)

Unit testing

- **unit testing**: Looking for errors in a subsystem in isolation.
  - generally a "subsystem" means a particular class or object
  - the Java library **JUnit** helps us to easily perform unit testing

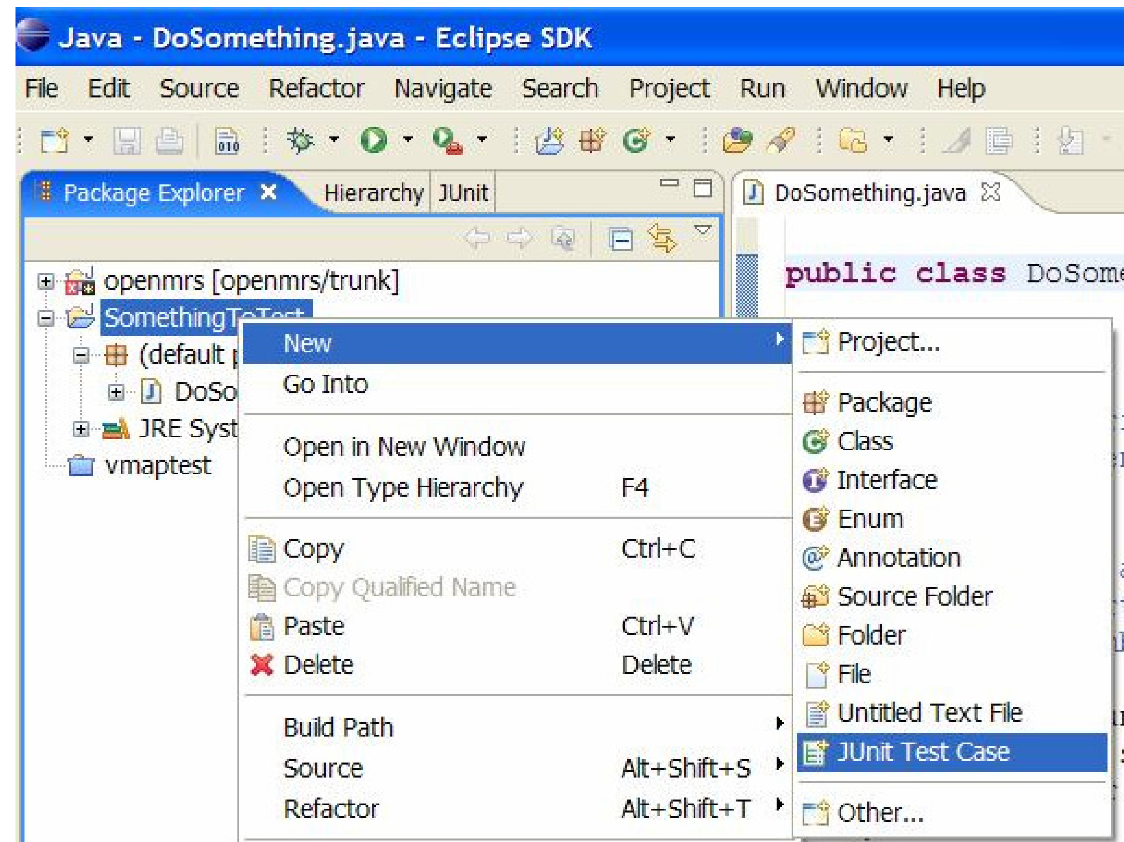
- the basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it that contains "test case" methods to run.
  - Each method looks for particular results and passes / fails.

- JUnit provides "**assert**" commands to help us write tests.

JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project** → **Properties** → **Build Path** → **Libraries** → **Add Library...** → **JUnit** → **JUnit 4** → **Finish**
  - *(see web site for jGRASP instructions)*

- To create a test case:
  - right-click a file and choose **New Test**
  - 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

`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 the same
`fail()`	immediately causes current test to fail
other assertion methods: `assertNull`, `assertNotNull`, `assertSame`, `assertNotSame`, `assertArrayEquals`	

- The idea: Put assertion calls in your `@Test` methods to check things you expect to be true. If they aren't, the test will fail.
  - Why is there no `pass` method?

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

ArrayIntList JUnit test

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

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
    }
    ...
}
```

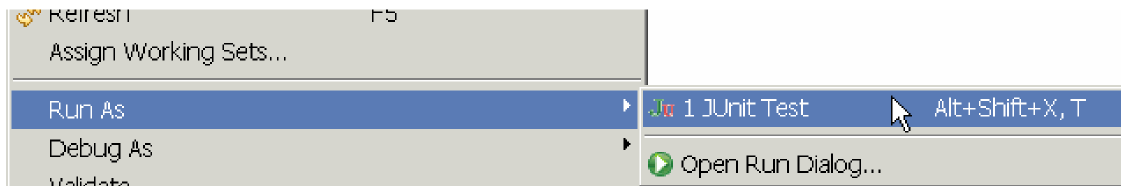
Running a test

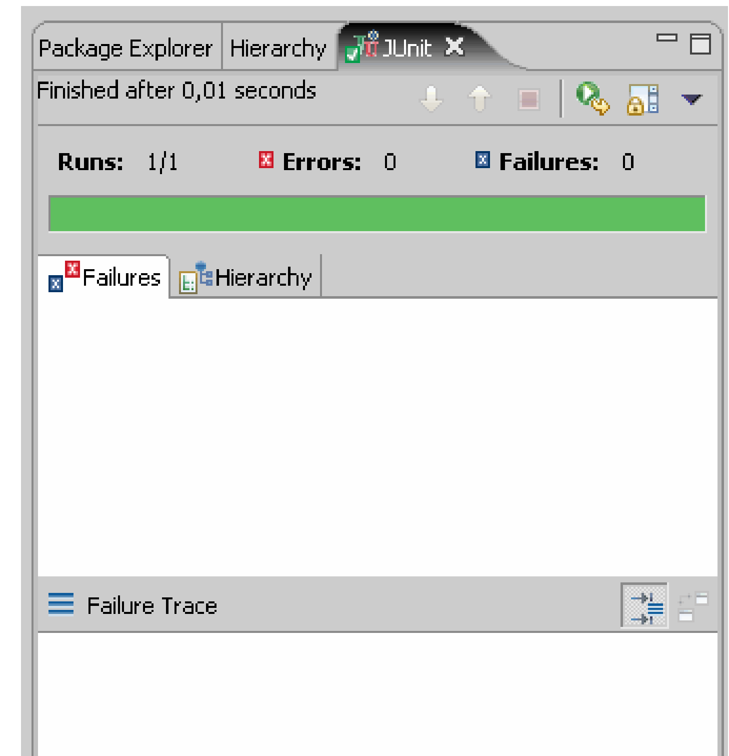
- Right click it in the Eclipse Package Explorer at left; choose:

  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





Testing for exceptions

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

- will pass if it *does* throw the given exception, and fail if not
  - use this to test for expected errors

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

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

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array

- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - make multiple calls; maybe `size` fails the second time only