OurArrayList & Automated Testing

4/1/26

(Including green slides based on lecture by Cynthia Lee)

Administrivia

- Due tonight:
  - Last codingbat problems
  - Clicker ID and essay
  - pre-test for course

- Extra credit:
  - Tomorrow (4/2) at 4pm in GDH 105: “We haven’t thought about that”: Religious Communities as Invisible Stakeholders in the Security Drone Robot Discourse (attend and submit a writeup)
  - Friday (4/3) at noon in SMC 203: “Opportunities for Math, Data Science, and Computer Science Students” (with Pizza!)
  - Parallel and distributed computing survey questions

Automated testing

- Nice feature of codingbat: it tells if your code is correct without a human looking at it
- Want same thing for large coding projects so you can tell if changes break part of the code
- Relies on thorough test cases
  - each is a call to your code and expected answer

Unit testing

- Whether you are doing black-box or clear-box testing, you should test every important unit of a software system
- What is a *unit*? In object-oriented programming, usually a software unit is taken to be a single method
- So: we should test every method of every class in the software
- **JUnit** is a widely used framework for unit testing of Java software…

JUnit basics

- To do unit testing in the Junit framework:
- Mark methods as test cases by putting @Test before them. Each "@Test" method should test a particular thing
- Optional:
  - Define instance variables that store the state of the "test fixture", i.e. the objects that will be tested
  - Initialize the fixture state with a method marked with @Before
  - Clean-up after a test with a method marked with @After

TestCase and test fixtures

- Import as appropriate
- Define instance variables that store the state of the "test fixture", i.e. the objects that will be tested

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

import org.junit.After;
import org.junit.Before;
import org.junit.Test;


public class RectangleTester {

  private Rectangle r1, r2; // test fixtures
```

setUp() and tearDown()

```
/* Called AUTOMATICALLY before each @Test method is run */
@Before
void setUp() {
    r1 = new Rectangle();
    r2 = new Rectangle(2.0,3.0);
}

/* Called AUTOMATICALLY after each @Test method is run */
@After
void tearDown() {
    r1 = null;
    r2 = null;
}
```

Make sure each test method starts with a clean copy of the test fixture.

```
@Before
@Test
@After
```

This is the sequence of calls the JUnit framework does for you *automatically* for *each* test method @Test that is invoked.

Coding a test case

```
/** Test case 2.1: verify that default constructor sets
    default instance variable values correctly
*/
@Test
public void testDefaultInstance() {
    assertEquals(1.0,r1.getLength());
    assertEquals(1.0,r1.getHeight());
}
```

Remember:
`@Before` will be called *prior* to each test method getting called, creating test object `r1` anew for each test

```
/** Test case 2.6: verify that mutator for length throws
 *  exception on illegal input.
*/
@Test (expected = IllegalArgumentException.class)
public void testMutatorIllegalInput() {
    r1.setLength(0); // 0 is illegal, should throw exception
}
```

More JUnit basics

- Test fixture instance variables are optional: you can use test method local variables instead
- When you run your TestCase class as a program, each @Test method is called automatically…
- If a JUnit assertion fails when a @Test method runs, that test **fail**s
- If @Test method throws an exception, that is considered an **error**, not a test failure!
- If a @Test method returns normally, that test **passes**

JUnit assertion methods

- The JUnit framework provides many useful assertion methods to use in your @Test methods

```
assertEquals(x,y) // fail if x is not equal to y
assertTrue(b) // fail if b has boolean value false
assertFalse(b) // fail if b has boolean value true
assertSame(x,y) // fail if x & y point to diff objs
assertNotSame(x,y) // fail if point to same obj
assertNull(x) // fail if x is not null
assertNotNull(x) // fail if x is null
fail() // always fails
```

- All assertion methods are overloaded with a version that takes an additional first argument of type String, for a message

Short version for our class

- Create a new class, normally called Test*Something*
- Give it the import statements
- Select the test tube and enable tests (select JUnit)
- Declare object references as attributes, create the objects (and set them up) in @Before
- Write testing methods (@Test) with assertions
  - I almost exclusively use assertEquals

What assertions should we use?

- What promises does method/class make?

  (promise = statement of method behavior on some input)

- Need assertions to
  - Check each promise (at least once)
  - Try edge cases when you barely hit a promise
  - Hit special cases: empty array, negative values, changing first/last in a list

Which of the following lines has an error?

```
public T remove() {
   //return arbitrary element (null if List is empty)
   if(num == 0)                                    //A
    return null;                                   //B
   num--;                                          //C
   T retVal = vals[num];                           //D
   return retVal;
}              //E: Not exactly one of the above
```

Which of the following lines has an error?

```
public T remove() {
   //return arbitrary element (null if List is empty)
   if(num == 0)                                    //A
    return null;                                   //B
   num--;                                          //C
   T retVal = vals[num];                           //D
   return retVal;
}                  //E: Not exactly one of the above (none)
```

Which of the following lines has a bug?

(E: not exactly one of the above)

```
public boolean remove(T val) {
    //removes val if it occurs (returns whether it did)
    for(int i=0; i < num; i++)              //A
        if(vals[i] == val) {                //B
            vals[i] = vals[num-1];          //C
            num--;                          //D
            return true;
        }

    return false;
}
```

Which of the following lines has a bug?

(E: not exactly one of the above) B and C

```
public boolean remove(T val) {
    //removes val if it occurs (returns whether it did)
    for(int i=0; i < num; i++)              //A
        if(vals[i] == val) {                //B
            vals[i] = vals[num-1];          //C
            num--;                          //D
            return true;
        }

    return false;
}
```

How many of the following correctly implement clear (empties list)?

I. vals.clear();

II. vals = (T[])new Object[3];

III. num = 0;

IV. for(i=0; i<vals.length; i++)
    vals[i] = null;

A. 0
B. 1
C. 2
D. 3
E. 4

How many of the following correctly implement clear (empties list)?

I. vals.clear();

II. vals = (T[])new Object[3];

III. num = 0;

IV. for(i=0; i<vals.length; i++)
    vals[i] = null;

A. 0
B. 1 (just III)
C. 2
D. 3
E. 4