Announcements

• Lab tomorrow
  – Exceptions and unit testing
• Quiz Thursday
• P3 due Friday
Follow-ups

• Exceptions and Try-catch
  – Using try-catch with loops
  – Comparison to switch vs. if-else if
  – Realistic examples
Unit testing

• Testing is a critical part of real-world software development

• “Unit testing”: running a series of automated tests on each conceptual “unit” of code
  – Each class might be treated as a unit
  – Individual methods might be treated as units

• Unit tests are written by developers (i.e. you!)
  – Going forward you may be expected to write your own tests on labs and projects

• Ideally every possible scenario gets tested
  – Each possible branch of execution
Unit testing

Example of “exhaustive” testing: myLine.toString() -> e.g. “3.0*x – 2.0”

Suppose our spec has the following requirements:
- If b is negative, no ‘+’ sign (not “3.0*x + –2.0”)
- If m or b is zero, don’t print

How many logical branches are there?
Unit testing

Example of “exhaustive” testing:
The pentium floating-point division bug


- Identified by a mathematician (Thomas R. Nicely) who was enumerating twin primes.
Unit testing

Corner cases
• Pathological cases that often get missed in a “first draft” implementation.
  – E.g. An implementation of list insertion may work correctly in the middle of the list, but not at the first or last position.

Error cases
• Testing cases where methods are “supposed” to fail (throw exceptions) on bad input.
The JUnit testing framework

• A popular framework for writing unit tests in Java.
  – Included with your Eclipse installation:
    File -> New -> Junit Test case

• Based around “assertions”
  – The typical unit test will initialize some objects, call some methods, and then make assertions about their various states
  – If the assertion is false the test will fail.
import static org.junit.Assert.*;
import org.junit.Test;

public class JUnit4Tests {

  @Test
  public void testMyObj() {
    MyObj obj1 = new MyObj(3);
    assertTrue(obj1.data == 3);
  }

  @Test
  public void testMyObjMethod1() {
    ...
  }

  ...
}
Testing Syntax (JUnit 4)

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

public class JUnit4Tests {

    @Test
    public void testMyObj() {
        MyObj obj1 = new MyObj(3);
        assertTrue(obj1.data == 3);
    }

    @Test
    public void testMyObjMethod1() {
        ...}

    ...

}
```
Testing Syntax (JUnit 4)

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

public class JUnit4Tests {

    @Test
    public void testMyObj() {
        MyObj obj1 = new MyObj(3);
        assertTrue(obj1.data == 3);
    }

    @Test
    public void testMyObjMethod1() {
        ...
    }

    ...
}
```

"@Test" before every unit test
import static org.junit.Assert.*;
import org.junit.Test;

public class JUnit4Tests {

    @Test
    public void testMyObj() {
        MyObj obj1 = new MyObj(3);
        assertTrue(obj1.data == 3);
    }

    @Test
    public void testMyObjMethod1() {
        ...
    }

    ...
}

Assertion of expected outcome
Other JUnit 4 syntax

• Other “Annotations”
  – @Before, @After: Can be used to do some set-up/clean-up, like initializing objects that other tests will use

• Other assertions
  – assertFalse(boolean expression)
  – assertSame(variable1, variable2): Compares variable1 and variable2 (either primitive or reference) with ==.
  – assertEquals(variable1, variable2): Compares using variable1.equals(variable2). Requires a special version of equals we have not fully covered yet.
    • When used with primitives, behaves according to ==.
JUnit 3

• An older version of JUnit, you may still see it used in some labs/projects for legacy reasons.
• Both versions 3 & 4 can be used in one project (but separate classes).
• JUnit 4 seems to be generally preferable.

http://sanjaal.com/java/981/java-unit-testing-and-junit/10-differences-between-junit-3-x-and-junit-4-x-and-why-you-should-move-to-junit-4-x-platform/
More on reference

“Reference” ↔ “Memory address”

• Terminology:
  – “Reference variable”: a variable that stores the memory address of an object (any variable that is not a primitive data type)
    e.g. MyObj obj1 = new MyObj();
  – “De-referencing”: Accessing data or calling methods via a reference variable
    e.g. obj1.getData();
More on reference

“Reference” ↔ “Memory address”

• Terminology:
  – “Call by value”: A method call where the parameters are assigned copies of the argument values
  – “Call by reference”: A method call where the parameters are assigned references to the arguments

• Java is often said to be “call by reference”, since an object is not copied in RAM every time its reference is passed to a method.
  – On the other hand, one might say that Java is call by value, but the “values” of reference variables are memory addresses.
Copy constructors

Sometimes we do want to copy objects: Allocating a new object in memory, but with identical fields to the original.

```java
MyObj obj2 = new MyObj();
obj2.a = obj1.a;
obj2.b = obj1.b;
obj2.c = obj1.c;
//OR
MyObj obj2 = new MyObj(obj1.a, obj1.b, ...);
```
Copy constructors

“Copy constructors” allow a more convenient usage.

//Usage:
MyObj obj2 = new MyObj(obj1);

//Implementation:
public MyObj(MyObj orig) {
    this.a = orig.a;
    this.b = orig.b;
    ...
}

Prep for Lab05

• BigInt implementation
  – Using and testing linked lists, throwing out of bounds exceptions
  – Copying linked lists
  – Algorithm for adding two BigInts