CS 61B: Lecture 11
Wednesday, February 12, 2014

Today's reading: Sierra & Bates, pp. 95-109, 662.

equals()

Every class has an equals() method. If you don't define one explictly, you inherit Object.equals(), for which "r1.equals(r2)" returns the same boolean value as "r1 == r2", where r1 and r2 are references. However, many classes override equals() to compare the _content_ of two objects.

Integer (in the java.lang library) is such a class; it stores one private int. Two distinct Integer objects are equals() if they contain the same int. In the following example, "i1 == i2" is false, but "i1.equals(i2)" is true. "i2 == i3" and "i2.equals(i3)" are both true.

```
        ---    -------            ---    -------    ---
     i1 |.+--->|  7  |         i2 |.+--->|  7  |<---+.| i3
        ---    -------            ---    -------    ---
```

IMPORTANT: r1.equals(r2) throws a run-time exception if r1 is null.

There are at least four different degrees of equality.
(1) Reference equality, ==. (The default inherited from the Object class.)
(2) Shallow structural equality: two objects are "equals" if all their fields are ==. For example, two SLists whose "size" fields are equal and whose "head" fields point to the same SListNode.
(3) Deep structural equality: two objects are "equals" if all their fields are "equals". For example, two SLists that represent the same sequence of items (though the SListNodes may be different).
(4) Logical equality. Two examples:
  (a) Two "Set" objects are "equals" if they contain the same elements, even if the underlying lists store the elements in different orders.
  (b) The Fractions 1/3 and 2/6 are "equals", even though their numerators and denominators are all different.

The equals() method for a particular class may test any of these four levels of equality, depending on what seems appropriate. Let's write an equals() method for SLists that tests for deep structural equality. The following method returns true only if the two lists represent identical sequences of items.

```
  public class SList {
    public boolean equals(Object other) {
      if (!(other instanceof SList)) {           // Reject non-SLists.
        return false;
      }

      SList o = (SList) other;
      if (size != o.size) {
        return false;
      }

      SListNode n1 = head;
      SListNode n2 = o.head;
      while (n1 != null) {
        if (!n1.item.equals(n2.item)) {          // Deep equality of the items.
          return false;
        }
        n1 = n1.next;
        n2 = n2.next;
      }
      return true;
    }
  }
```

Note that this implementation may fail if the SList invariants have been corrupted. (A wrong "size" field or a loop in an SList can make it fail.)

IMPORTANT: Overriding DOESN'T WORK if we change the signature of the original method, even just to change a parameter to a subclass. In the Object class, the signature is equals(Object), so in the code above, we must declare "other" to be an Object too. If we declare "other" to be an SList, the equals() method will compile but it will NOT override. That means the code

```
  Object s = new SList();
  s.equals(s);
```

will call Object.equals(), not SList.equals(). Dynamic method lookup won't care that s is an SList, because the equals() method above is not eligible to override Object.equals().

Therefore, if you want to override a method, make sure the signature is EXACTLY the same.

"for each" LOOPS

Java has a "for each" loop for iterating through the elements of an array.

```
  int[] array = {7, 12, 3, 8, 4, 9};

  for (int i : array) {
    System.out.print(i + " ");
  }
```

Note that i is _not_ iterating from 0 to 5; it's taking on the value of each array element in turn. You can iterate over arrays of any type this way.

```
  String concat = "";
  for (String s : stringArray) {
    concat = concat + s;
  }
```

For some reason, the type declaration _must_ be in the "for" statement. The compiler barfs if you try

```
  int i;
  for (i : array) { ... }
```

TESTING
=======
Complex software, like Project 1, is easier to debug if you write lots of test code. We'll consider three types of testing:

(1) Modular testing: testing each method and each class separately.
(2) Integration testing: testing a set of methods/classes together.
(3) Result verification: testing results for correctness, and testing data structures to ensure they still satisfy their invariants.

(1) Modular Testing

When you write a program and it fails, it can be quite difficult to determine which part of the code is responsible. Even experienced programmers often guess wrong. It's wise to test every method you write individually.

There are two types of test code for modular testing: test drivers and stubs.

(a) Test drivers are methods that call the code being tested, then check the results. In Lab 3 and Homework 3, you've seen test drivers in the SList class that check that your code is doing the right thing.

Both public and private methods should be tested. Hence, a test driver usually needs to be inside the class it tests. In a class intended for use by other classes, the obvious place to put a test driver is in the main() method, as we did in Lab 3 and Homework 3. However, if a class is the entry point for the program, you can't put your test driver in main(). Instead, put it in a method with a name like testDriver(), and then write _another_ class whose main() method calls your test driver.

(b) Stubs are small bits of code that are _called_ by the code being tested. They are often quite short. They serve three purposes.

(i) If you write a method that calls other methods that haven't yet been implemented, you can write simple stubs that fake the missing methods.
(ii) Suppose you are having difficulty determining whether a bug lies in a calling method, or a method it calls. You can temporarily replace the callee with a stub that returns controlled results to the caller, so you can see if the caller is responsible for the problem.
(iii) Stubs allow you to create repeatable test cases that might not arise often in practice. For instance, suppose a subroutine fetches and returns input from an airline database, and your code calls this subroutine. You might want to test whether your code operates correctly when ten airplanes depart at the same time. Such an event might be rare in practice, but you can replace the database access subroutine with a stub that feeds fake data to your code. There are two advantages:

- Stubs can produce test data that the real code rarely or never produces.
- Stubs produce _repeatable_ test data, so that bugs can be reproduced.

(2) Integration Testing

Integration testing is testing all the components together (preferably _after_ you have tested them in isolation). Sometimes bugs arise during integration because your test cases weren't thorough enough. Other times, they arise because of misunderstandings about how the components are supposed to interact with each other. Integration testing is harder than modular testing, because it's harder to determine where a bug is, or to identify your mistaken assumptions about how the components interact.

The most important task in avoiding these bugs is to define your interfaces well and unambiguously. There should be no ambiguity in the descriptions of the behavior of your methods, especially in unusual cases. We'll talk a lot more about this in later lectures.

The best advice I can give on integration testing: learn to use a debugger.

(3) Result Verification

A result verifier is a method that checks the results of other methods. There are at least two types of result verifiers you can write.

(a) Data structure integrity checkers. A method can inspect a data structure (like a list) and verify that all the invariants are satisfied. For Project 1, we are asking you to write a simple checker named "check()" that verifies the integrity of your run-length encodings.
(b) Algorithm result checkers. A method can inspect the output of another method for correctness. For example, if a method is supposed to sort an array of numbers, a result checker can walk through the output and check that each item really is less than or equal to its successor.

An _assertion_ is a piece of code that tests an invariant or a result. Java offers an "assert" keyword that tests whether an assertion evaluates to "true". If the assertion comes up "false", Java terminates the program with an "AssertionError" error message, a stack trace, and an optional message of your own choosing.

```
  assert x == 3;
  assert list.size == list.countLength() : "wrong SList size:  " + list.size;
```

At the end of each method that changes a data structure, add assertions (possibly a call to an integrity checker). At the end of each method that computes a result, add an assertion that calls a result checker.

Assertions are convenient because you can turn them on or off. To turn them on when you're testing your code, run your code with "java -ea" (for "enable assertions"). To turn them off for greater speed, run with "java -da" (for "disable assertions"). The default (if you specify no switch) is -da. WARNING: when assertions are turned off, the method "list.countLength()" above is never called. Good for speed, but countLength() must not perform a task that is necessary for your program's correctness.

Regression Testing

A _regression_test_ is a test suite can be re-run whenever changes are made to the code. Nearly every software company has reams of regression tests for each product. They run them again every time they fix a bug or add a feature.

Some principles of regression testing:

(a) All-paths testing: your test cases should try to test every path through the code. Test every method. For every "if" statement, you should try to write a test case for each of the two paths.
(b) "Boundary cases" should be tested, as well as non-boundary cases. For instance, if you write a binary search method, test it on arrays of lengths zero and one, as well as longer lengths. Test the cases where the item sought is the first element, the last element, in the middle, not present. For every loop in the code, try to test the cases where it iterates zero or one times, as well as the case where it iterates several times. Test the branch "if (x >= 1)" for x equal to 0, 1, and 2.
(c) Generally, methods can be divided into two types: extenders, which construct or change an object; and observers, which return information about an object. (Some methods do both, but you should always think hard about whether that's good design.) Ideally, your test cases should test every combination of extender and observer.

In real-world software development, the size of the test code is often larger than the size of the code being tested.