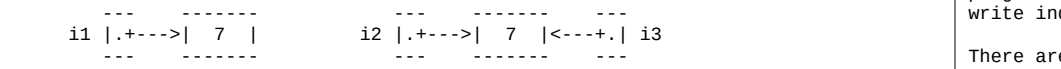
CS 61B: Lecture 10
Wednesday, February 13, 2013

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

equals()

Every class has an equals() method. If you don't define one explictly, then "r1.equals(r2)" returns the same boolean value as "r1 == r2", where r1 and r2 are references. However, many classes redefine 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.



There are at least four different degrees of equality.
(1) Reference equality, ==.
(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(SList other) {
      if (size != other.size) {
        return false;
      }
      SListNode n1 = head;
      SListNode n2 = other.head;
      while (n1 != null) {
        if (!n1.item.equals(n2.item)) {
          return false;
        }
        n1 = n1.next;
        n2 = n2.next;
      }
      return true;
    }
  }
```

Note that this implementation may fail if the SList invariants have been corrupted.

TESTING

Complex software, like Project 1, is easier to debug if you write test code to make sure that the methods you write are working correctly. 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 to do the right thing, it can be quite difficult to determine which lines of code are responsible. Even experienced programmers often guess wrong. It's wise to test each method and class 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.

In a class intended for use by other classes, the obvious place to put a test driver is in the main() method. That's what we did in Lab 3 and Homework 3. When you call SList methods from another program that has its own main() method, the main() method in SList is ignored. The test driver is run only when you request it by typing "java SList".

If a class is the entry point for the program, you can't put your test driver in the main() method. Instead, put it in a method with a name like testDriver(), and then write another class whose main() method calls your test driver.

```
  public class MyProgram {
    public static void testDriver() { ... }
  }

  public class TestMyProgram {
    public static void main(String[] args) {
      MyProgram.testDriver();
    }
  }
```

Run the test by typing "java TestMyProgram".

Both public and private methods should be tested. Hence, a test driver generally needs to be inside the class it tests.

(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 needs to call other methods that haven't yet been implemented, you can write stubs that fill in for the missing methods. The stubs are usually very simple, and do just enough work to make sure the calling method can be compiled and tested.
(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 subroutine rarely or never produces.
- Stubs produce _repeatable_ test data, so that bugs can be reproduced (usually a necessary first step to finding what causes them).

A stub often replaces a complex method with one that returns a fixed sequence of preprogrammed inputs. For example, you can easily write a method that returns 2 the first time it's called, 5 the second time, and so on.

(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, it happens because of misunderstandings about how the components are supposed to interact with each other. Integration testing is harder than modular testing, because it's often hard to determine where the bug is, or to identify your mistaken assumptions about how the modules 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.

When you start working with partners on programming projects, the importance of interfaces will increase. You'll need to negotiate interfaces with your project partners, so that one of you can implement a class while the other one implements a program that uses the class. An interface is a contract between you that allows you to do business in harmony. 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 supposed to be -da, but on the lab machines it seems to be -ea. 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 of their products.

You could write a test driver that lets you type in test cases to try out. However, if you change a routine and need to test it again, you may forget some of your earlier creative test cases. It's better to encode all your test cases right into your test driver, so that you can run them again every time you 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. (You can write a method that does both, but you should always think hard about whether it's good design.) Ideally, your test cases should test every combination of extender and observer. Try out each extender at least once, and after each call to an extender, call all the observers and make sure that they return the right results.

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