Index of /~cs61b/lectures/05

Name	Last modified	Size	Description
Parent Directory	24-Sep-98 21:22	-	

```
                        CS61b: Lecture 5
                      Wednesday, September 2
ANNOUNCEMENTS:
-------------
New labs have been added:
  Lab 23, 8-10pm on Tuesday, and Discussion 123, 2-3pm on Thursday
  Lab 24, 11am-1pm on Wednesday.  Not yet in Telebears, discussion TBD.

CONDITIONALS: if and if/else
------------
Simple if:  if (boolean-expr) { statement1; ... }
If-then-else: if (boolean-expr) { statement1; ... } else { statement2; ...}
     if (grade >= 75) {
        System.out.println("Pass");
     }
     else {
        System.out.println("Fail");
     }

Nested if-then-elses: if (boolean-expr1) {s1; ...} else if (boolean-expr2) {s2; ...}
You may omit the { }'s if there is only one statement, but shouldn't.  Why not?
Each statement and "else if (..)" on a separate line.
Put trailing }'s on a line by themselves and let emacs indent for you.

SWITCH
------
For a long nested if/then/else, if the boolean-expr is integer ==, use:

    switch (month) {
      case 2;
        daysIn = 28;
        break;
      case 9: case 4: case 6: case 11:
        daysIn = 30;
        break;
      default:
        daysIn = 31;
        break;
    }

As written, if month is 2 (February) daysIn will be 28.
What if we omit the first "break"?    Second "break"?    Both?

CONSTANTS IN CODE
-----------------
In general, avoid use of numeric constants the have other "meanings."

final int FEBRUARY = 2;  // final says that FEBRUARY's value will never change.
```

```
  ... case FEBRUARY ...

TESTING
-------
Two kinds of testing:
  Modular testing -- testing each function and each class separately.
  Integration testing -- testing a set of functions/classes together
Two kinds of test code for modular testing:
  Drivers -- code written for testing that calls the code being tested
  Stubs -- bits of code that are called by the code being tested
General principles of testing:
  * Many tests are not necessarily better than a few carefully chosen ones
  * First test the normal input, then test the boundary cases
        boundaries in the interface
        branches in the code
  * Test your own code and have someone else test it.

MODULAR TESTING EXAMPLE
-----------------------
Consider a List class with methods:
    List(), isEmpty(), length(), insertFront(), insertEnd(), toString().

First, write test cases based on the interface; divide methods into 4 categories:

  1) Constructors: constructors
       List()
  2) Extenders: methods that either modify or return objects of the given type.
       List(), insertFront(), insertEnd()
  3) Observers: methods that return a different type.
       isEmpty(), length(), toString()
  4) Static methods: methods with the keyword "static" which don't
     work on an object.  These are often helping functions.
       None in the List class (that we can see).

Basic plan: write at least one test case for every combination of
               constructor + observer
               extender + observer

  E.g., observers on constructor:
     List lst1 = new List();
     ...lst1.toString()...
     ...lst1.isEmpty()...
     ...lst1.length()...

The above strategy is for writing test cases based on the specifications.
One also wants to include test cases based on the program itself:
  1) exercise each path in the code
  2) test values at branch boudaries, e.g.,   if (x >= 1), test x = 0, 1, 2
Many of these will already be tested by test cases based on the specification.

REGRESSION TESTING
------------------
Regression tests: a test suite that is re-run after any changes to the code.

Why not just build a driver that lets the user type in test cases? E.g.,
    "What length?"  __l__
    "What method?"  __m__
     driver constructs a list of length l, calls m, and prints result

A lot of thought needs to go into the selection of test cases.
Size of test code my be larger than the size of the code being tested.
```

TESTING IN JAVA

The easiest way to set up test code in Java is:
  1) Define a "main" method for each class; have "main" do complete testing.
     Add more "static" methods to help "main" with pieces of testing, e.g.,
     testEmpty(), testAfterInser()
  2) These methods can and should test private as well as public methods.

The book (A&W) provides a small class for writing test code.

```
  public class TestHelper {
    /** Checks the given condition and prints the error messages
     * if it fails.  If testCondition is true, this method does
     *  nothing; if it is false, the given message is printed followed
     *  by a dump of the program call stack which terminates the program.
     */
    static void verify(boolean testCondition, String message) { ... }
  }
```