Index of /~cs61b/labs/lab2

```
   Name                    Last modified     Size  Description
```

Name	Last modified	Size	Description
Parent Directory	30-Aug-98 15:13	-	
Fraction.java	20-Aug-98 14:31	4k	

```
                              CS61B Lab 2
                   August 31 - September 2, 1998

The goal of this lab to gain experience with:
        1) defining new classes
        2) creating objects that are instances of those classes
        3) conditionals
        4) recursive functions

Getting started:  You will find the code for this lab in
$master/labs/02.  Start by copying it using the commands:
            mkdir lab2
            cd lab2
            cp $master/labs/lab2/* .

Part 0: Getting Started
-----------------------

Read the Fraction.java class into emacs and compile it using
C-x C-e, filling in the command javac -g Fraction.java.   The
program should compile without errors.  Now run the program using
"java Fraction" in a shell window (from your lab2 directory).
The program should run, although it will print fractions in
non-reduced form, e.g., 12/20.

Part I: Constructors (1 point)
------------------------------

Look at the main function in the Fraction class.  First, it declares
three variables as Fraction and initialized each to a Fraction object.
Three different constructors are used, each with a different number
of arguments.

    Fraction f0 = new Fraction();
    Fraction f1 = new Fraction(3);
    Fraction f2 = new Fraction(12, 20);

Look at the implementation of the constructors.  The first constructor
of 2 arguments is straightforward.  It assigns the numerator and
denominator fields to the given values.  The second 1-argument uses
some new syntax:

        this(n,1);

The effect of this statement is to call the 2-argument constructor,
passing n and 1 as arguments.  "this" is a keyword in Java, which
```

normally refers to the object on which a method is invoked. In the constructor, it can be used as above to invoke one of the other constructors.

We could have written the 1-argument constructor as simply:

```
        numerator = n;
        denominator = 1;
```

Why call the 2-argument constructor instead? The reason is one of good software engineering: by having two of the constructors call the third, we have reduced duplicate code. In this case, the code is very small (just the two assignment statements), but in general, the constructor might be more complicated. Not only is the program shorter (in this case very marginally), but more importantly, if we later find a bug in the constructor, it's likely the change can be made in only 1 place, the 2-argument constructor. This principle applies to other methods, not just constructors. If you find yourself copying several (or even a few) lines of code to use at another point in your program, chances are you should have defined another method that would contain the common piece of code.

The 0-argument constructor does not use the good style just described. Modify it to call the 2-argument constructor. You TA or LA will ask to see this when you get checked off, but you should continue on with the assignment for now.

Part II: Using Objects (1 point)

After declaring three variables and constructing an object for each, the main function prints the objects. The print code is:

```
    System.out.println("The fraction f0 is " + f0.toString());
    System.out.println("The fraction f1 is " + f1); // toString is implicit
    System.out.println("The fraction f2 is " + f2);
```

How does Java know what to do when printing the fractions f1 and f2? In the case of f0, we have invoked the toString method, and because Java knows how to concatenate (add together) strings and print them, the behavior of the first line should be clear. In the second and third lines, we are asking Java to concatenate a Fraction to a String and print the result, but these builtin Java functions do not know about Fractions. Instead, it implicitly looks for a method called toString to convert each of these objects to a string. Since we have defined such a method on Fractions, it will use that method. Thus, we could have omitted the toString() invocation the first line as well.

The next four lines shows how to invoke the evaluate() method and print the result. After that, there are some lines surrounded by comments. Remove the comments and fill in the two missing expressions so that sumOfTwo is the sum of f1 and f2, and someOfThree is the sum of f0, f1, and f2. (Do not add any additional lines, variables, or assignment statements.) You may add the fractions in any order, although your TA or LA may ask you which order the additions will performed.

Part III: Conditionals and Recursive Functions (1 point)

As we noted earlier, the toString method prints Fraction in unreduced form. Examine the code in the toString method. It is calling another function called gcd for computing the greatest common divisor (gcd) of two positive integers. If this function worked correctly, toString would print Fractions in reduced form; instead, it currently returns 1 (which is indeed a common divisor, but not the largest one). Replace the body of the gcd function so that it iss a recursive function that correctly computes the gcd. Recompile and run your program.

Hint: Here is a recursive gcd function written in Scheme from the 61A text. Note that a and b are assumed to be positive integers:

```
(define (gcd a b)
   (if (= b 0)
       a
       (gcd b (remainder a b))))
```

The "remainder" function in Java is called "mod" for "modular arithmetic" and is written "a % b".

Part IV: Representation Invariants (1 point)

Once we have decided to use reduced fraction for the toString version (which changes the behavior or "meaning" of the class), we can use three different strategies for doing reduction:

1) Always reduce Fractions when converting to a string, but leave the internal representation in unreduced form.
2) Change the toString function so that it modifies the numerator and denominator so they will be reduced, i.e., they will be relatively prime. The gcd function will still be called every time we print a string, but it may run a bit faster after the first invocation.
3) Change all of the methods that set or modify numerator and denominator to guarantee that they will always be reduced. We may then remove the gcd call from toString.

Note that all three strategies should give the same behavior (modulo some differences in when some numerical operations might overflow, for example, which we will ignore for the moment). The third strategy will give a faster program, assuming that toString is often called on a Fraction. There is now an invariant on the representation of all Fractions, namely:

```
gcd(numerator/denominator) == 1
```

It is the responsibility of anyone modifying this class to ensure that, if they add or change any methods (including constructor) that assign to numerator or denominator, this invariant will always hold when the method returns. That same implementor, when writing new methods that look at the values numerator and denominator, may assume that their gcd is 1. This "representation invariant" should be added as a comment near the declaration of these two fields.

Hint: before starting on this part, you may want to save a backup copy of the Fraction.java file, in case you don't finish this part. You can still checkoff parts 1-3 based on the old file.

Modify the Fraction class to use strategy three. You will need to do 2 things:

1) remove the call to gcd in the toString method
2) add a call to gcd in 2-argument constructor.

Why is it sufficient to add the gcd call in only one place, when all of the constructors, as well as add method, manipulate the internals of an object.

Check-off

When you're done with all 4 parts (or have finished 3 out of 4 and are running out of time), contact your TA or LA for checkoff. You will need to demonstrate working program with gcd's computed correctly and fractions in reduced form. You will also need to show them parts of your code (e.g., the constructors and toString methods).

Final Comments

One weakness in the design of the Fraction class is that it does not handle negative numerators or denominators. The implementation you saw here could easily be extended to handle negative fractions -- we decided, for the purpose of this assignment, to opt for simplicity over generality. The most unfortunate aspect of this decision is that the restriction on constructing fractions is not enforced by the program, but is left as a comment to the user of the class, known as a "precondition" on the invocation of the constructors. A careless user could pass a negative number into a constructor, which at some later point could be passed to the gcd function. (Be sure you understand what could go wrong in that case.) We could have made the program more robust by enforcing the requirement of non-negative n and d in the contructor, either by adding some tests in the constructor body or by changing the input types to "unsigned" which is a Java type similar to "int" but only handling non-negative numbers. This would, again, have complicated our program in certain ways that were not related to the point of this assignment. From a software engineering standpoint, the right solution is to allow negative input to the constructor, since fractions of negative numbers are perfectly reasonable mathematically.