CS61B Lab #2

Please finish this lab today, preferably during the lab period. It is due at midnight Friday.

1. Building a Class

Writing a substantial program consists of writing a set of definitions that build on each other. The things we define are those that our programming languages allow us to name: procedures, classes, variables, and so forth. In "object-based" programming, classes (the main structuring tool) tend to correspond to nouns, and procedures (methods) to verbs and adjectives. When we analyze a problem from this perspective, we try to identify the kinds of things we are going to manipulate (the nouns), what they can do or have done to them (the verbs), and the properties they have that we will need to inquire about (the adjectives).

Java's library includes a class java.lang.math.BigDecimal, which is capable of representing decimal numbers (with decimal points) of finite length. In this exercise, we'll design a simple class to represent general rational numbers ("fractions", if you prefer). It will be able to represent numbers that one cannot represent exactly in BigDecimal (such as 4/3), but for the purposes of this lab, we'll take a short cut and limit ourselves to rational numbers that are representable as ratios of java 'long' numbers (in the range -2**63..2**63-1, where the Fortran operator '**' means 'exponentiation').

Before building such a class, it's often useful to have a client in mind to give some idea of what you need out of it. (The term "client" is generic in the computing profession, referring to a program or module that uses a particular package or service of interest.) The files that come with the skeleton for this lab contain files Root1.java and Root2.java, which compute approximate Kth roots of numbers for non-negative integers K. As currently written, Root1.java does all arithmetic in "double-precision floating point", which basically means using something like scientific notation with results approximated to 53 significant bits (roughly 15 significant decimal digits). It's not particularly important that you understand the algorithm being used, but do try to understand what all the statements in these programs mean (and ask if you don't).

Root2.java is an incomplete version of Root1.java that uses rational rather than floating-point numbers. As you can see, it's not sufficient to change all the 'doubles' to 'Rationals' and I've had to comment out many of the statements from Root1 (and replace expressions with 'null' or 'false') in order to get the program to compile. So job #1 in this lab is to figure out an appropriate set of additional methods for Rational and that will allow you to rewrite Root2 so that it SHOULD work once the bodies of Rational's methods are filled in properly.

First, find and scan the Javadoc page for java.math.BigDecimal (see the class home page for a link to the Java library documentation). That should give you an idea of what methods might be appropriate. Of course, you'll need to do some adaptation. For example, all the MathContext arguments and rounding mode arguments are inappropriate, since Rational arithmetic is exact arithmetic.

Add skeletons for the methods you choose to arith/Rational.java, WITHOUT filling in their bodies (unless they are trivial and obvious). Instead, just leave "stubs" that return null or some other arbitrary value that is legal for the desired return type (see the current skeletons for 'frac', for example). No need to make this tedious; you can limit yourself to methods you'll use in Root2.java for now.

Likewise, modify Root2.java to use the appropriate methods in lieu of +, -, *, etc. Make sure everything compiles. Of course, at this point, Root2 is likely to blow up when run because of the missing implementation in Rational.

[A design question to think about: Instead of having clients create new Rational numbers in Root2 by writing, e.g.,

```
new Rational(1, 3)
```

I elected instead to have them write

```
frac(1, 3)     (or in full, arith.Rational.frac (1, 3))
```

What advantages do you suppose there might be in using a (static) method such as frac for this purpose rather than a constructor?]

2. Unit Testing

In Lab #1 and HW #1, we looked at some simple "black-box" testing of complete programs. The term "black-box" is supposed to suggest that one writes tests against the intended specifications of the program, treating the program itself as a black box that hides its interior details. In this lab, we'll look at "unit" testing, in which one tests internal parts of a program directly---using some knowledge, in other words, of the program's inner structure.

A framework called JUnit enjoys widespread use as a way of building unit tests in Java. Unfortunately, it is also one of the most poorly documented bunches of Java code I've seen; we'll try to compensate. The lab #2 skeleton contains a file arith/RationalTest.java with a couple of tests in it, and 'gmake check' in the arith subdirectory executes it. The JUnit framework provides a number of components:

 * A set of methods with names like 'assertTrue' and 'assertEquals' that perform some simple test and cause an error if it fails.
 * A number of 'annotations', such as @Test, which marks a method as being a unit test.
 * Various main testing routines, such as JUnitCore.main, that examine specified classes at execution time and call all of the methods in them that are annotated to be tests.

In our examples, you can see that each test is tiny, testing just a few related pieces of the Rational class. We've provided a couple of examples. The tests currently fail, since Rational is mostly unimplemented (or buggy), and naturally the goal is to get them to work (by fixing Rational, one hopes, rather than the tests!). JUnit comes from a philosophy in which the idea is to write tests BEFORE finishing (or even starting) the code, so that

 * You have set of reproducible tests included in your source code, and easily invoked with a simple command (gmake check in our case).
 * You can isolate errors quickly, rather than always having to search for the cause when they surface as more obscure errors in tests of the entire program.
 * You are not tempted to let knowledge of the implementation bias your testing.
 * You have a ready-made progress guage to allow you to see how much work remains and whether you have succeeded in implementing what you set out to do.
 * You have a set of regression or acceptance tests for your work.

For this lab,

1. Fill in or fix enough of Rational.java to get the handful of unit tests included in the skeleton to work.

2. Add at least one test of your own.

3. Turn in the your lab work

Use hw submit to commit and submit all your lab2 files. Check that the
submission worked (if you don't know how to do that (there are a
number of ways), the DBC rule suggests that now would be a good time
to find out!)