COLBY COLLEGE
Computer Science Department

CS115 **Allen B. Downey**
Spring 1999

Assignment 9: Homemade Objects

Due: Wednesday 10 November

Rational numbers

A rational number is a number that can be represented as the ratio of two integers. For example, 2/3 is a rational number, and you can think of 7 as a rational number with an implicit 1 in the denominator. For this assignment, you are going to write a class definition for rational numbers.

1. Go to `http://rocky.colby.edu/cs151/code/hw9/cs151.java`. It contains a class definition for complex numbers and a `main` method that tests the various functions. Print a copy and make sure you understand how it works. You should use this example as a template for what follows.

2. Create a new project called `Rational.prj`.

3. In addition to the class named `cs151`, add a second class named `Rational`. Remember that only `cs151` should be a `public` class.

4. Add two instance variables (integers) to the `Rational` class. These will contain the numerator and denominator of each Rational object.

5. Write a constructor that takes no arguments and that sets the two instance variables to zero.

6. Write a method called `printRational` that takes a Rational object as an argument and prints it in some reasonable format.

7. Write a `main` method that creates a new object with type Rational, sets its instance variables to some values, and prints the object.

8. At this stage, you have a minimal testable (debuggable) program. Debug it. Whenever I start writing a new class, I start with these three steps: instance variables, a simple constructor, and a print method. Then I add methods and test them one at a time.

Another constructor

1. Write a second constructor for your class that takes two arguments and that uses them to initalize the instance variables.

2. Add lines to `main` that test both constructors. You will probably want to add a print statement to the constructors so that you can tell exactly when each constructor gets invoked.

Modifers and functions

1. Write a new method called `negate` that reverses the sign of a rational number. This method should be a modifier, so it should return void. Add lines to `main` to test the new method.

2. Write a method called `invert` that inverts the number by swapping the numerator and denominator. Remember the swap pattern we have seen before. Add lines to `main` to test the new method.

3. Write a method called `toDouble` that converts the rational number to a double (floating-point number) and returns the result. This method is a pure function; it does not modify the object. As always, test the new method.

4. Write a modifier named `reduce` that reduces a rational number to its lowest terms by finding the GCD of the numerator and denominator and then dividing top and bottom by the GCD.

   You may want to write a method called `commonDivisor` that finds the greatest common divisor of the numerator and the denominator. The next section explains an algorithm for finding the GCD. This method should be a pure function; it should not modify the instance variables of the object on which it is invoked.

5. Write a method called `add` that takes two Rational numbers as arguments and returns a new Rational object. The return object, not surprisingly, should contain the sum of the arguments.

   There are several ways to add fractions. You can use any one you want, but you should make sure that the result of the operation is reduced so that the numerator and denominator have no common divisor (other than 1).

Calculating GCDs

(This section is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs.*)

The following algorithm is known as Euclid's Algorithm because it appears in Euclid's *Elements* (Book 7, ca. 300 B.C.). It may be the oldest nontrivial algorithm.

The algorithm is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are the same as the common divisors of b and r. Thus we can use the equation

$$GCD(a, b) = GCD(b, r)$$

to successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

$$GCD(36, 20) = GCD(20, 16) = GCD(16, 4) = GCD(4, 0) = 4$$

implies that the GCD of 36 and 20 is 4. It can be shown that for any two starting numbers, this repeated reduction eventually produces a pair where the second number is 0. Then the GCD is the other number in the pair.

The Card class

1. Create a new Java project called `Card.prj`. The structure of this project will be similar to the previous one—two class definitions, named `cs151` and `Card`. The Card class will define a new type of object that will represent a playing card. It will have two instance variables, `rank` and `suit`, that represent the rank and suit of the card.

   There are several choices for the type of `rank` and `suit`. They could be Strings or ints, or even doubles. For this project we will make them ints, so that they are easy to compare (among other things).

2. Write two constructors for the Card class. One should take no arguments and initialize both instance variables to 0. The other should take the rank and suit as arguments.

3. Write a `printCard` method. To do this, we need to choose a **mapping** from integers onto suits, and from integers onto ranks. For example, a sensible mapping for suits is:

   $$\begin{aligned} 0 &\mapsto Clubs \\ 1 &\mapsto Diamonds \\ 2 &\mapsto Hearts \\ 3 &\mapsto Spades \end{aligned}$$

   where the symbol $\mapsto$ is pronounced "maps to." This mapping makes sense because the suits are in increasing order, so a larger integer represents a superior suit.

   This week, you will probably need to use a chained conditional to print the suits and ranks, but next week we will see a better way to do it.

4. Write a `main` method that creates a new card using either constructor, and prints the card using `printCard`.

Compare cards

1. Write a method called `compareCards` that takes two cards as arguments and returns an integer with value -1 if the first card is less than the second, 1 if the first is greater than the second, and 0 if the two cards are the same. One card outranks another if its rank is higher or, if the two ranks are the same, if its suit is higher.

   At this point, you might want to give some thought to the mapping you chose for the ranks. An obvious choice is something like

   $$\begin{aligned} 1 &\mapsto Ace \\ 2 &\mapsto 2 \\ \ldots \\ 11 &\mapsto Jack \\ 12 &\mapsto Queen \\ 13 &\mapsto King \end{aligned}$$

   This mapping is convenient for printing, but not as convenient for comparing, since it (incorrectly) puts the Ace at the bottom, when the Ace should outrank the King. Think about the pros and cons of various solutions and implement whatever you think is the best choice.

2. Invoke your method from `main` and test that it works.