Problem Set 2

Assigned: Sept. 18
Due: Sept. 25

Problem 1

In the file Figures.java (linked from the notes on Abstract Methods), the bounding box is defined as a data field `boundingBox` which is initialized by the constructor for each concrete class of figure. This has the advantage that, for a circle for example, it is only necessary to calculate the bounding box once from the data fields `center` and `radius`. However, it has the disadvantage that the bounding box is calculated even if no one ever uses it. (Of course the calculation here is so trivial that it makes no difference; but imagine a value for which the calculation from the basic data fields is very laborious e.g. the volume of a three-dimensional polyhedron with 10,000 faces. This is not at all unrealistic in engineering applications.)

The area, on the other hand, is defined as a method `area()`. This has the advantage that if no one ever needs it, it never has to be executed. But it has the disadvantage that, if it is often needed, the same calculation has to be repeated every time it is needed.

Another problem with the way `boundingBox` is defined in `Circle` is that, if you change the radius after setting up the bounding box, the old bounding box is no longer valid. Whereas the method `area()` always uses the current value of the data fields.

Show how the definition of `Circle` can be rewritten to get the best of both worlds in both respects. That is, there should be a method `area()` such that,

- The first time `C.area()` is called for circle `C`, the method computes the area.
- On any later call to `C.area()`, if the radius has been changed since the previous call to `C.area()`, then the area is recomputed. Otherwise, `C.area()` just looks up the value in a data field. (Hint: you will need another data field to keep the value in. You may assume that the area of a figure is never negative.)

You may assume that the fields `center` and `radius` are declared `private` and that getters and setters are provided. You will need to describe the code for the methods `area()` and `setRadius(r)`.

For example with the following sequence of calls, the area is computed at the points shown.

```java
Circle C = new Circle(5.0, 2.0, 10.0);
double W = C.area(); // Area is computed.
double X = C.area(); // Area is looked up.
C.setCenter(0.0, 5.0);
double Y = C.area(); // Area is looked up.
C.setRadius(2.0);
C.setRadius(20.0);
double Z = C.area(); // Area is recomputed.
Y = C.area(); // Area is looked up
C.setRadius(10.0)
double V = C.area(); // Area is recomputed. (The program does not have to
                       // keep track of all earlier values.)
```

This idea of carrying out a calculation only when needed and no more than once appears often in computer science; it is used in techniques such as `caching results`, `dynamic programming` and `lazy evaluation`. Making sure that the answer returned corresponds to the current state of things is the problem of `cache currency`.

The answer to this problem is quite short, despite this long-winded problem statement.
Problem 2

Consider the following recursive method

```java
public static int f(int n) {
    if (n <= 1) return 1;
    int a = f(n/2);
    return a + n;
}
```

Show a trace of the execution of this method for \( f(10) \), similar to those in the notes on Recursion. Your trace should show the values of the parameter \( n \), the variable \( a \), and the value returned associated with each call to \( f \).

Problem 3

If you have a list of numbers \( L = [X_1, X_2, \ldots X_k] \) and you have a two place numerical function \( f(A, B) \), then the reduction of \( L \) by \( f \) is the value \( f(X_1, f(X_2, f(X_3, \ldots f(X_{k-1}, X_k) \ldots)) \).

For instance if \( L \) is the list \( [2,3,5,7] \) and \( p(A, B) = A + B \) then the reduction of \( L \) by \( p \) is \( 2 + (3 + (5 + 7)) = 17 \).

If \( q(A, B) = A * B \) then the reduction of \( L \) by \( q \) is \( 2 * (3 * (5 * 7)) = 210 \).

If \( r(A, B) = 2B - A \) then the reduction of \( L \) by \( r \) is \( r(2, r(3, r(5, 7))) = r(2, r(3, 9)) = r(2, 15) = 28 \).

Using the same technique used in the class `ApplierInt` from lecture 2, show how an interface can be written that allows you to write a general `Reduce` function to the values in a linked list of `ints`, as defined in the file `IntList.java`.

Specifically, you should:
- Write an interface `ReduceFun` that has one abstract method `int f(int a, int b)`.
- Write three classes that instantiate `ReduceFun` and that override `f` with methods that instantiate the above three functions \( p, q, r \).
- Write a static method `int reduceList(ReduceFun w, IntList l)` which returns the reduction of \( l \) by \( w \).
- Write a small main function which is a driver illustrating the use of all this.

Problem 4

Repeat problem 3 using an abstract class rather than an interface. Specifically, you should

1. Define an abstract class `Reducer` that has
   a. An abstract method `int f(int x, int y)`.
   b. A concrete method `int reduce(IntList l)` that reduces list \( l \) using the method \( f(x, y) \).
2. Three concrete classes that extend `Reducer` and that override `f` with methods instantiating the above three functions \( p, q, r \).
3. A small main function that is a driver.

Note: It is not required, but I strongly recommend that you actually get problems 3 and 4 running.