Problem 1

In the file Figures.java, 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is defined in `Circle` is that, if you change the radius after setting up the bounding box, the old bounding box is no longer valid.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.
```

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**

Using Prof. Korth’s definition of the Node class (singly linked list of ints), write a recursive static method `doubleList(L)` that non-destructively returns a list in which every element of L appears twice. For example:
- If L is the list [4,5,6] then `doubleList(L)` returns the list [4,4,5,5,6,6].
- If L is the list [5,5,3,3,3,1] then `doubleList(L)` returns the list [5,5,5,3,3,3,3,3,3,3,1,1].

**Honors Problem**

Do problems 1 and 2 above, and also the following.

Any program with a loop (iteration) can be rewritten as a recursive program. In fact, there are programming languages in which the only way to write a loop is using recursion.

The “hailstone procedure” is defined as follows. Start with a number X. If X is even, divide it by 2. If X is odd, compute 3X + 1. Keep doing this until you reach 1.

For example, if you start with X = 3, you go through the following steps:

Step 1. \( X = 3 \).
Step 2. \( X = 3 \cdot 3 + 1 = 10 \).
Step 3. \( X = 10/2 = 5 \).
Step 4. \( X = 3 \cdot 5 + 1 = 16 \).
Step 5. \( X = 16/2 = 8 \).
Step 6. \( X = 8/2 = 4 \).
Step 7. \( X = 4/2 = 2 \).
Step 8. \( X = 2/2 = 1 \).

If you start with \( X = 7 \), you go through the following sequence, of length 17: 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

Try what happens if you start with \( X = 27 \), but only if you have time to spare (it goes through 112 steps, and the intermediate values get as large as 9232). (This is not part of the assignment!)

The hailstone number of \( X \) is defined as the number of steps needed to get from \( X \) to 1, including both \( X \) and 1. For example, the hailstone number of 3 is 8, and the hailstone number of 7 is 17.

The hailstone procedure is the simplest badly-understood algorithm in computer science (look up “Hailstone number” in Wikipedia for details). In particular, it is not known whether it terminates for all starting values \( X \).

The following simple Java method computes the hailstone number.

```java
public static int HailstoneNumber(int X) {
    int count = 1;
    while (X != 1) {
        if (X % 2 == 1) X = 3*X + 1;
        else X = X/2;
        count++;
    }
    return count;
}
```

**Assignment**: Rewrite this as a recursive function. The answer is short, and actually simple, but does involve a little mind-bending.