Review and
Introduction to Recursion

- Turn in your Design Problem 3 solution

Review

- Object Oriented Design Principles
- Encapsulation
- Cohesion
- Coupling
- Review Solar System Problem
- Scope and Static Variables

Principles of Design (for CSSE220)

- Make sure your design **allows proper functionality**
  - Must be able to **store required information** (one/many to one/many relationships)
  - Must be able to **access the required information** to accomplish tasks
  - Data should **not be duplicated** (id/identifiers are OK!)
- Structure design **around the data** to be stored
  - **Nouns should become classes**
  - **Classes should have intelligent behaviors** (methods) **that may operate on their data**
- Functionality should be **distributed efficiently**
  - **No class/part should get too large**
  - **Each class should have a single responsibility** it accomplishes
- **Minimize dependencies** between objects when it does not disrupt usability or extendability
  - Tell don't ask
  - Don't have message chains
- **Don't duplicate** code
  - Similar "chunks" of code should be **unified into functions**
  - Classes with similar features should be given **common interfaces**
  - Classes with similar internals should be simplified using **inheritance**

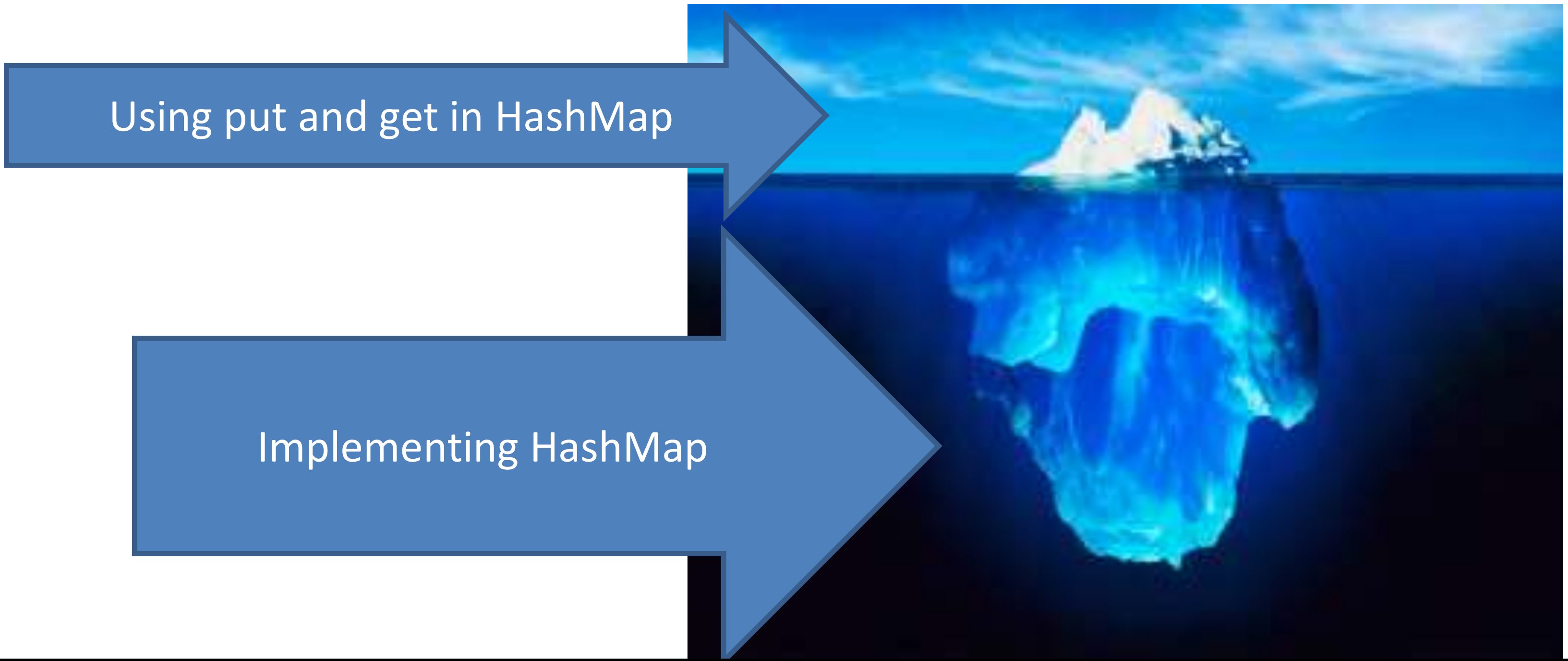
Encapsulation

- Makes your program easier to understand by
  - Grouping related stuff together

- Rather than passing around data, pass around objects that:
  - Provide a powerful set of operations on the data
  - Protect the data from being used incorrectly

Encapsulation

- Makes your program easier to understand by…
  - Saving you from having to think about how complicated things might be



Coupling and Cohesion

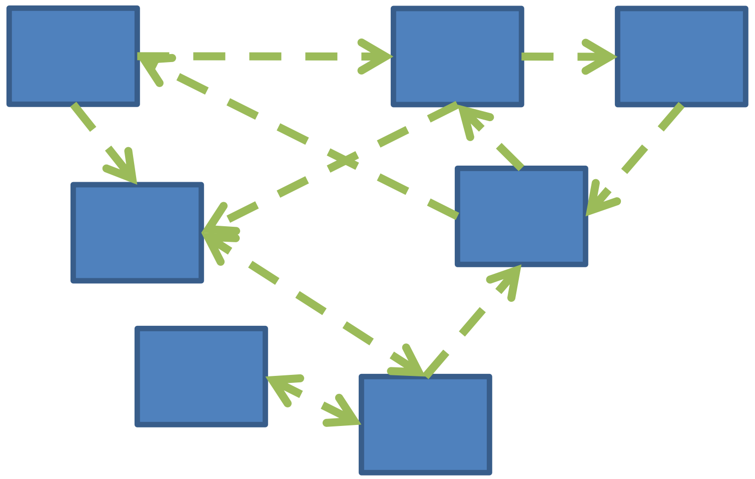
- Two terms you need to memorize
- Good designs have:
  - High coHesion
  - Low coupLing

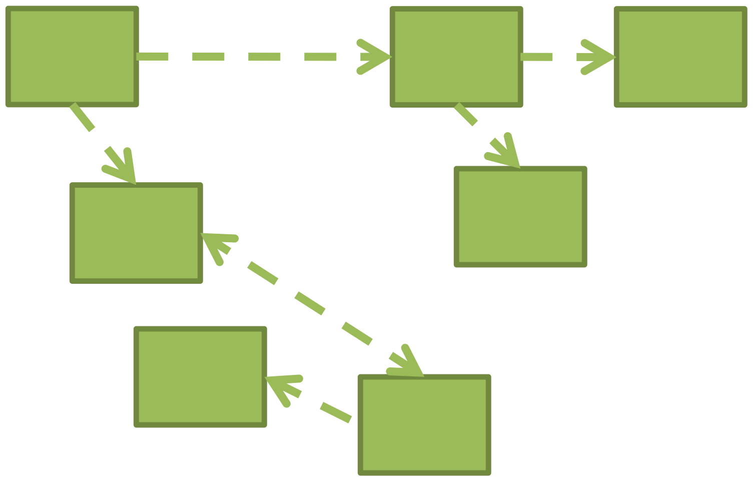
Consider the opposite:

- Low cohesion means that you have a small number of really large classes that do too much stuff (i.e., do more than one thing)
- High coupling means you have many classes that depend ("know") too much on each other

Coupling – UML Diagrams

- Lot's of dependencies ➔ high coupling
- Few dependencies ➔ low coupling



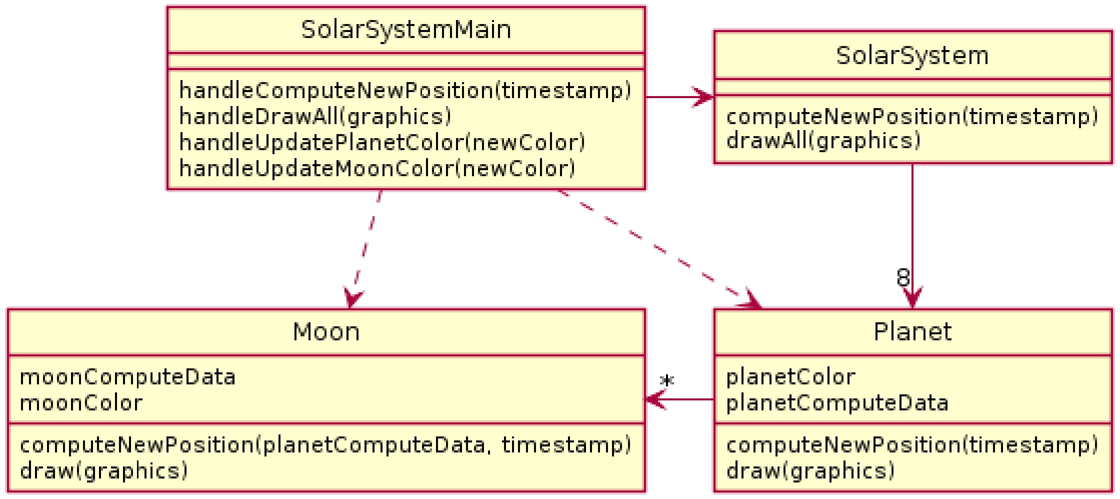


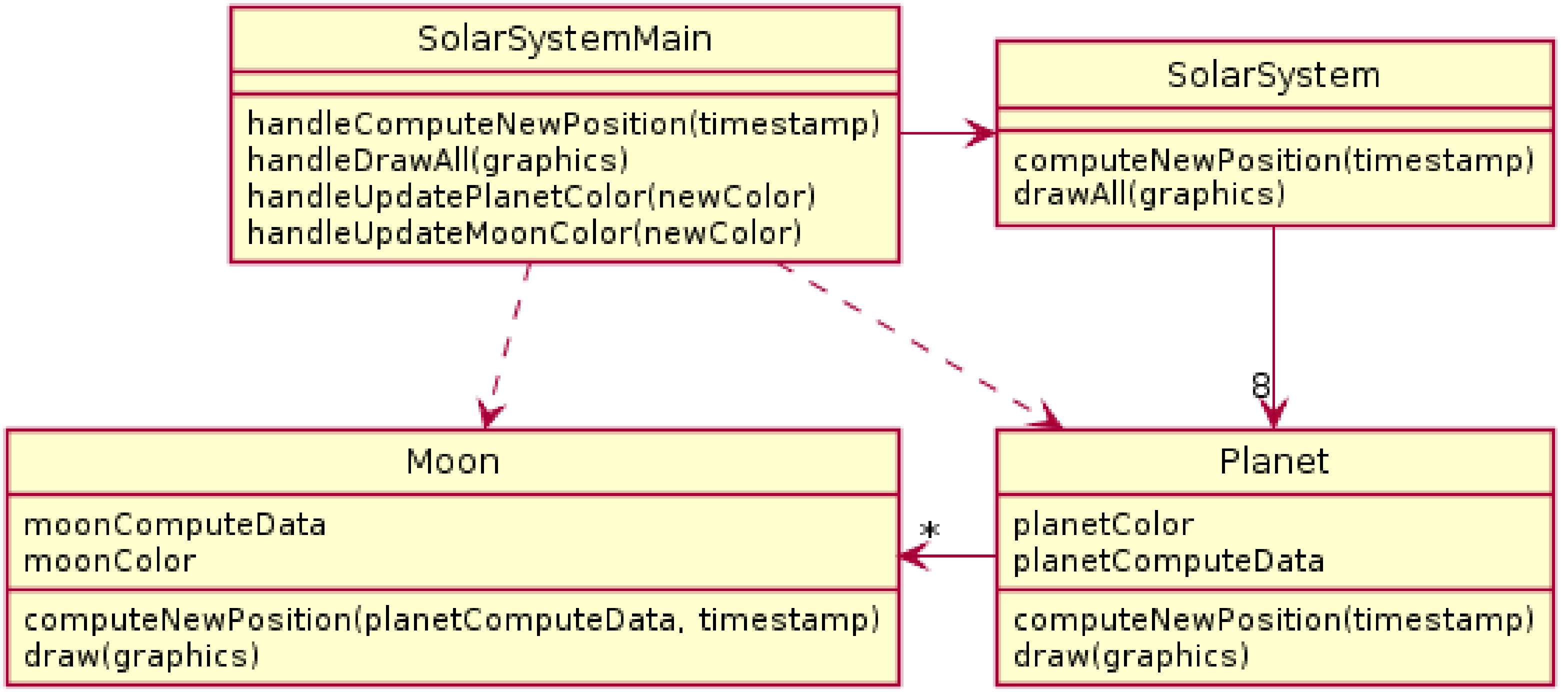
How hard will it be to change code with:

High coupling? Low coupling?

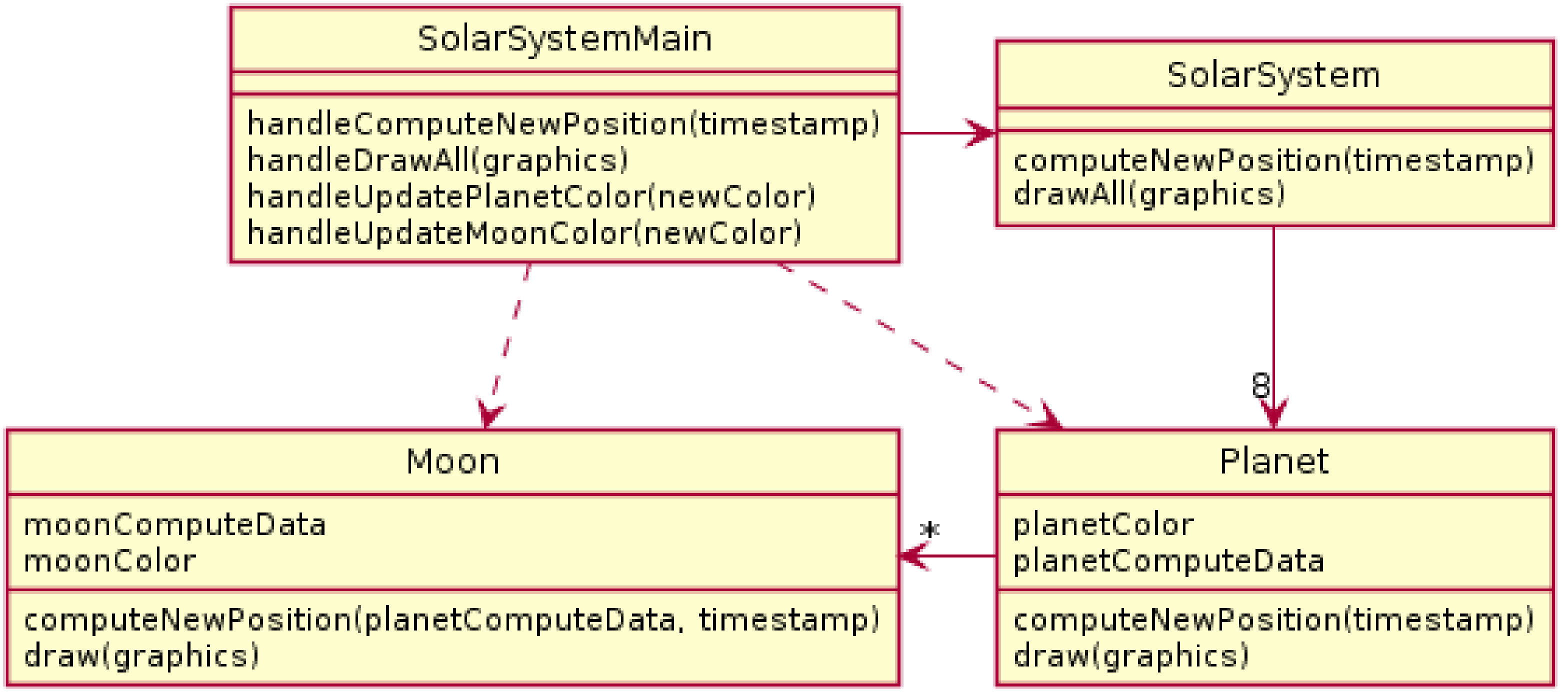
Review: Solar System Problem

A Java program draws a minute by minute updated diagram of the solar system including all planets and moons. To update the moon's position, the moon's calculations must have the updated position of the planet it is orbiting. The diagram is colored - all planets are drawn the same color and all moons are drawn the same color. However, it needs to be possible to reset the planet color or the moon color and the diagram should reflect that.


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

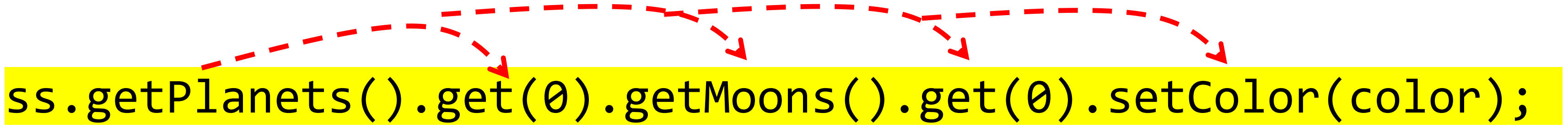

SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

- What is wrong here?


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

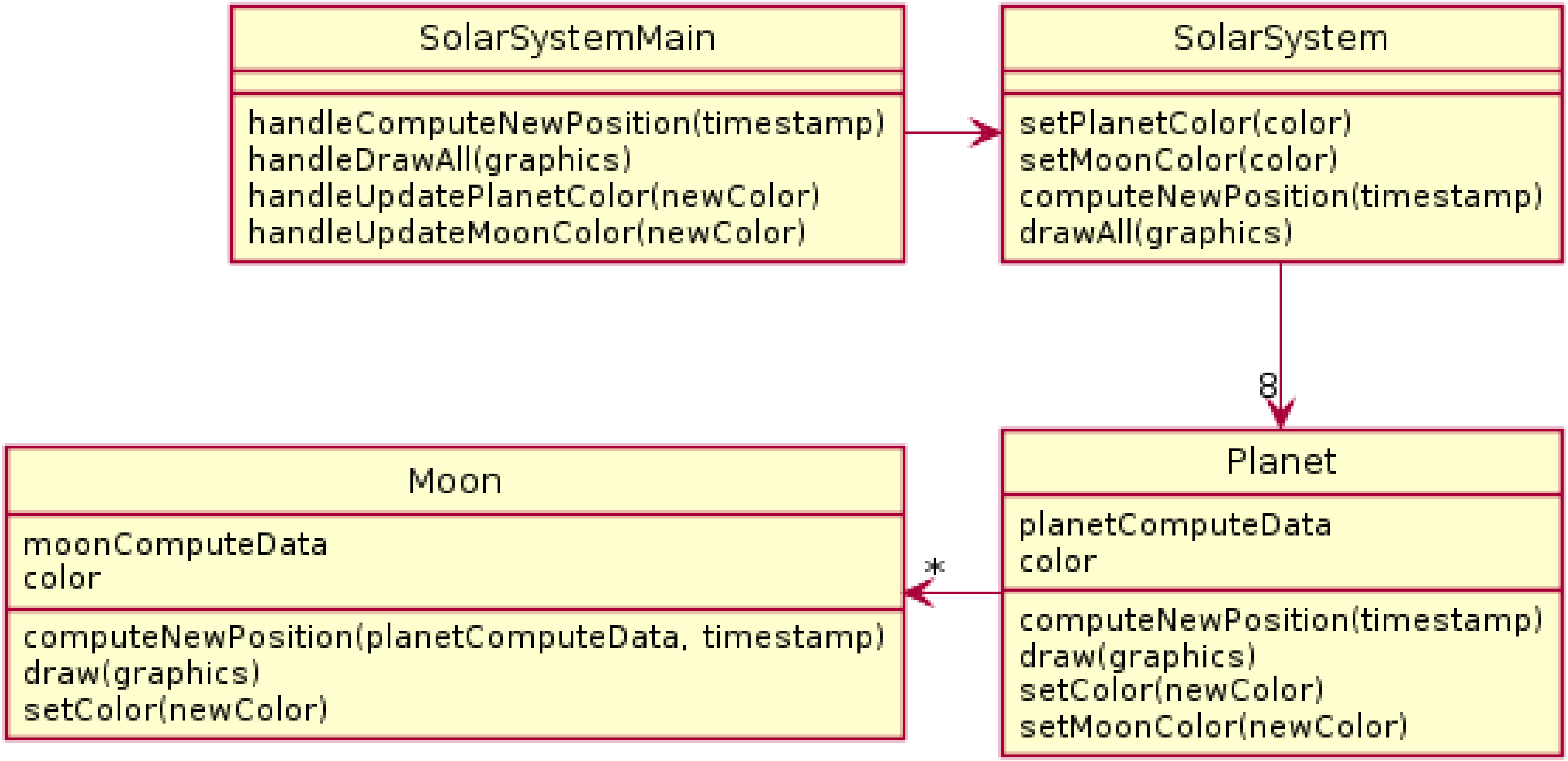
- What is wrong here?

4b. methodChain to update moon

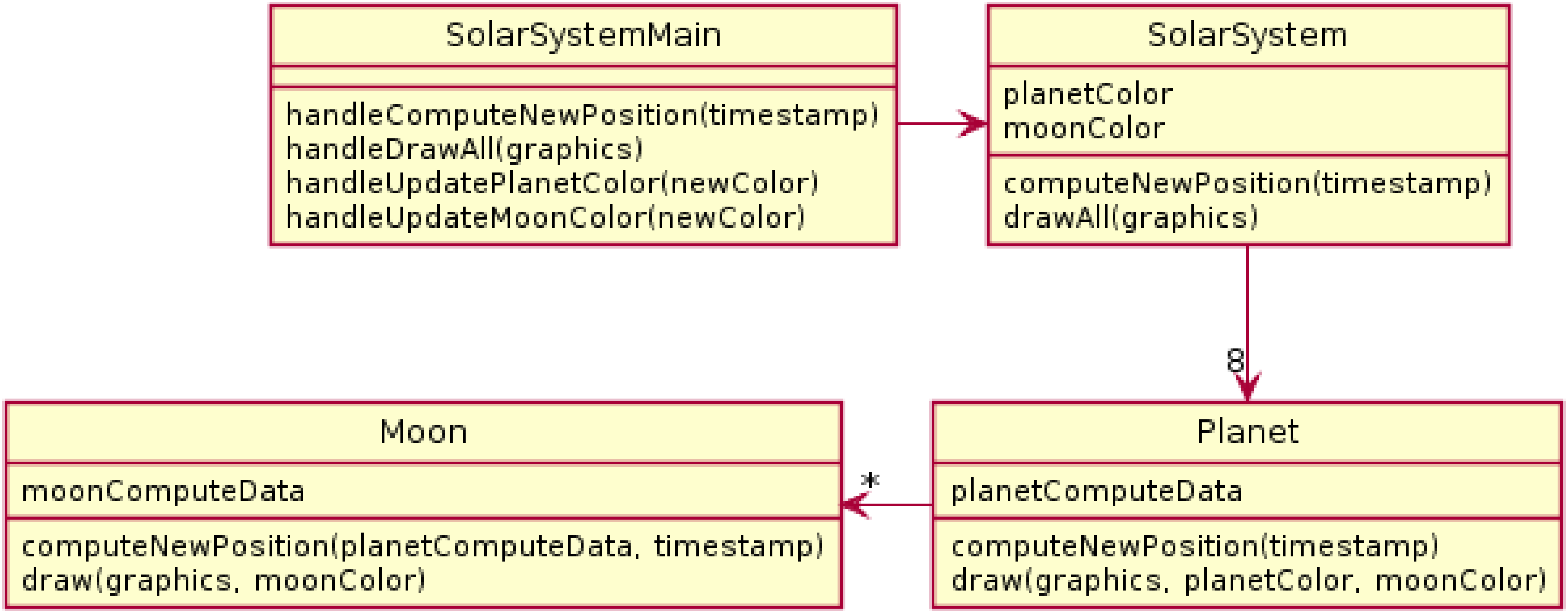


```
ss.getPlanets().get(0).getMoons().get(0).setColor(color);
```

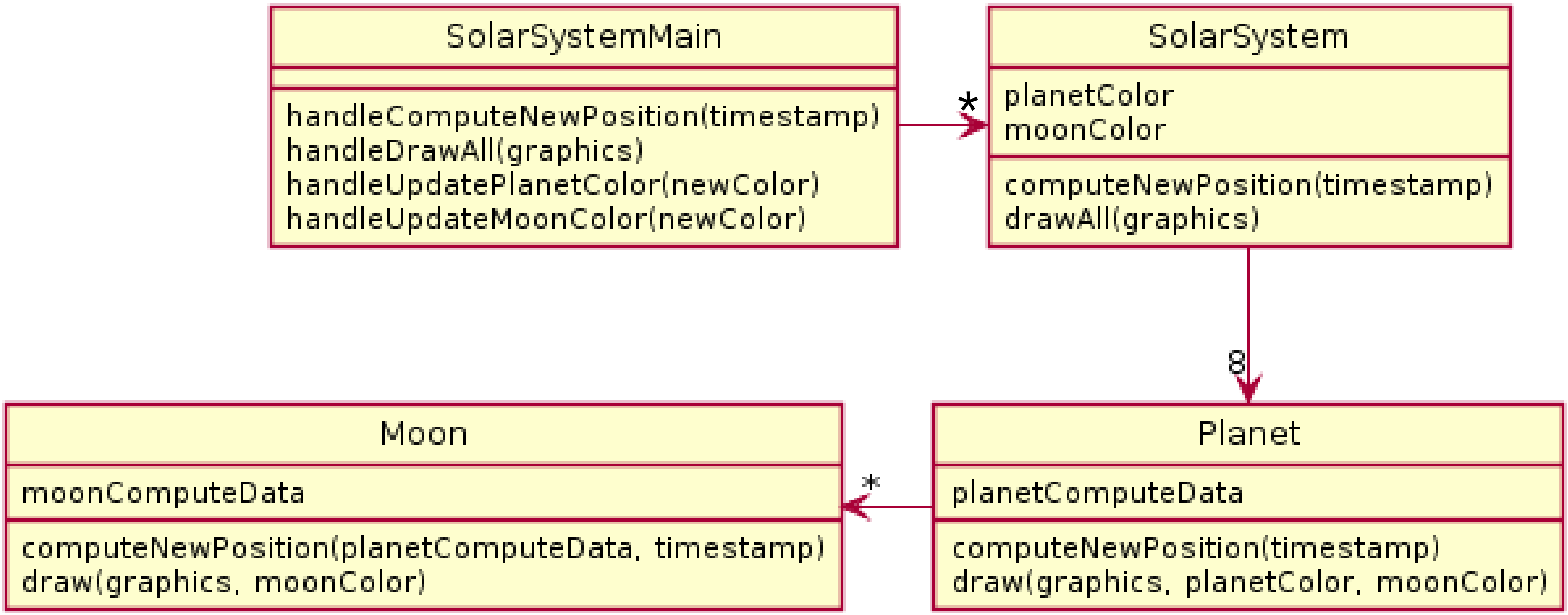
Partial Solution



Why not use static here?
All moons, planets have same color!


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
planetColor
moonColor
computeNewPosition(timestamp)
drawAll(graphics)
8
Planet
planetComputeData
computeNewPosition(timestamp)
draw(graphics, planetColor, moonColor)
*
Moon
moonComputeData
computeNewPosition(planetComputeData, timestamp)
draw(graphics, moonColor)

What if we had many solar systems? Would our design be easily extended?



SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
*
SolarSystem
planetColor
moonColor
computeNewPosition(timestamp)
drawAll(graphics)
8
Planet
planetComputeData
computeNewPosition(timestamp)
draw(graphics, planetColor, moonColor)
*
Moon
moonComputeData
computeNewPosition(planetComputeData, timestamp)
draw(graphics, moonColor)

Rule of Thumb: No Global Variables

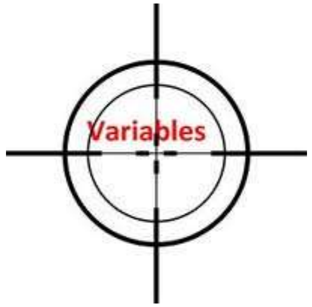
- Or static variables that are used like globals
- A static variable can be accessed/modified in any function at any time
- As a result many parts of the code can be coupled to a single class

Rule of Thumb: No Global Variables

- Or static variables that are used like globals
- A static variable can be accessed/modified in any function at any time
- As a result many parts of the code can be coupled to a single class

- Why?
- Increases coupling among all the clients that get or change value of the global variable

Variable Scope



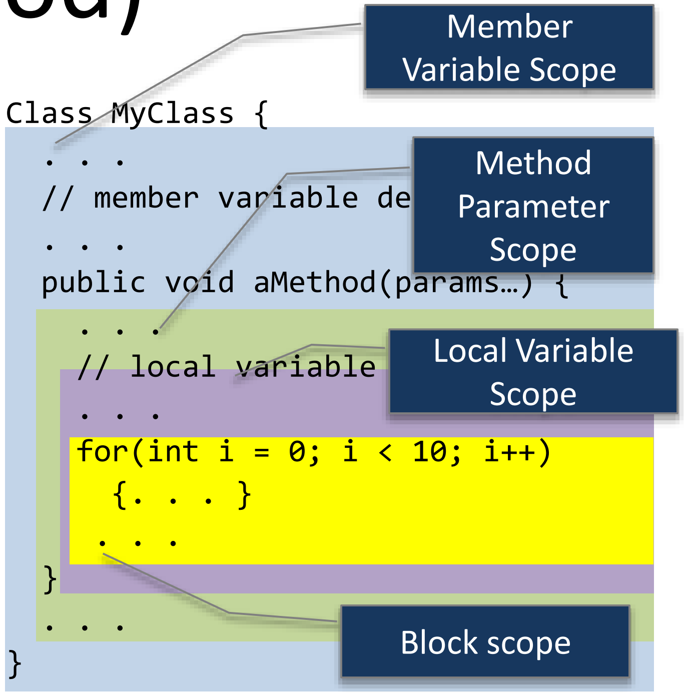
Scope is the region of a program in which a variable can be accessed

- *Parameter scope:* the whole method body
- *Local variable scope:* from declaration to block end

```
public double myMethod() {
   double sum = 0.0;
   Point2D prev = this.pts.get(this.pts.size() - 1);
   for (Point2D p : this.pts) {
     sum += prev.getX() * p.getY();
     sum -= prev.getY() * p.getX();
     prev = p;
   }
   return Math.abs(sum / 2.0);
}
```

Member Scope (Field or Method)

- *Member scope:* anywhere in the class, including *before* its declaration
  - Lets methods call other methods later in the class
- **`public static`** class members can be accessed from outside with "class qualified names"
  - `Math.sqrt()`
  - `System.in`



```
Class MyClass {
  . . .
  // member variable de
  . . .
  public void aMethod(params…) {
    . . .
    // local variable
    . . .
    for(int i = 0; i < 10; i++)
      {. . . }
     . . .
  }
  . . .
}
```

Overlapping Scope and Shadowing

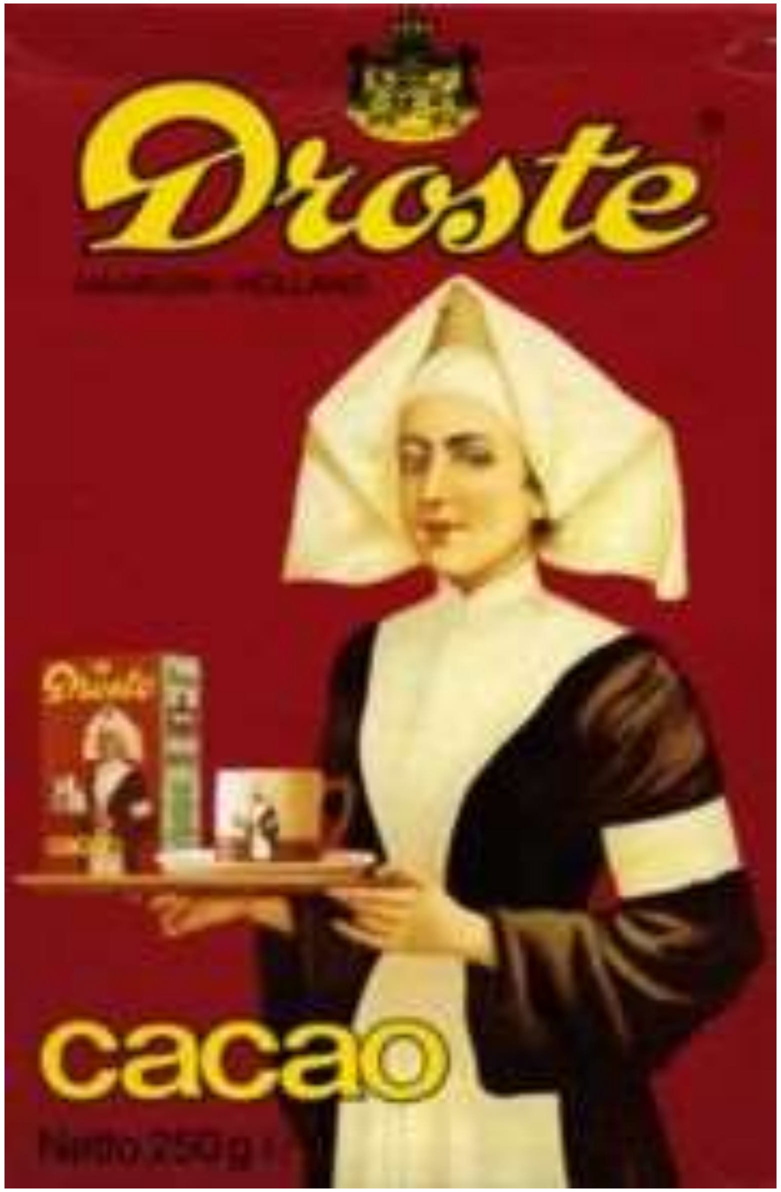
```
public class TempReading {
        private double temp;

        public void setTemp(double temp) {
                this.temp = temp;

        }
        // …
}
```

What does this "temp" refer to?

Always qualify field references with `this`. It prevents accidental shadowing.



CSSE 220: New Material

Recursion

Import *Recursion* project from the repo

Announcements

- The next 4 class days:
  - A new way to think: **Recursion**
  - A new way to break up and re-use code: **Interfaces**
    - Making interactive apps requires this

Recursion

- A solution technique where the same computation **occurs repeatedly** as the problem is solved

recurs

- Examples:
  - Sierpinski Triangle: https://en.wikipedia.org/wiki/Sierpinski_triangle
  - Towers of Hanoi: http://www.mathsisfun.com/games/towerofhanoi.html or search for Towers of Hanoi

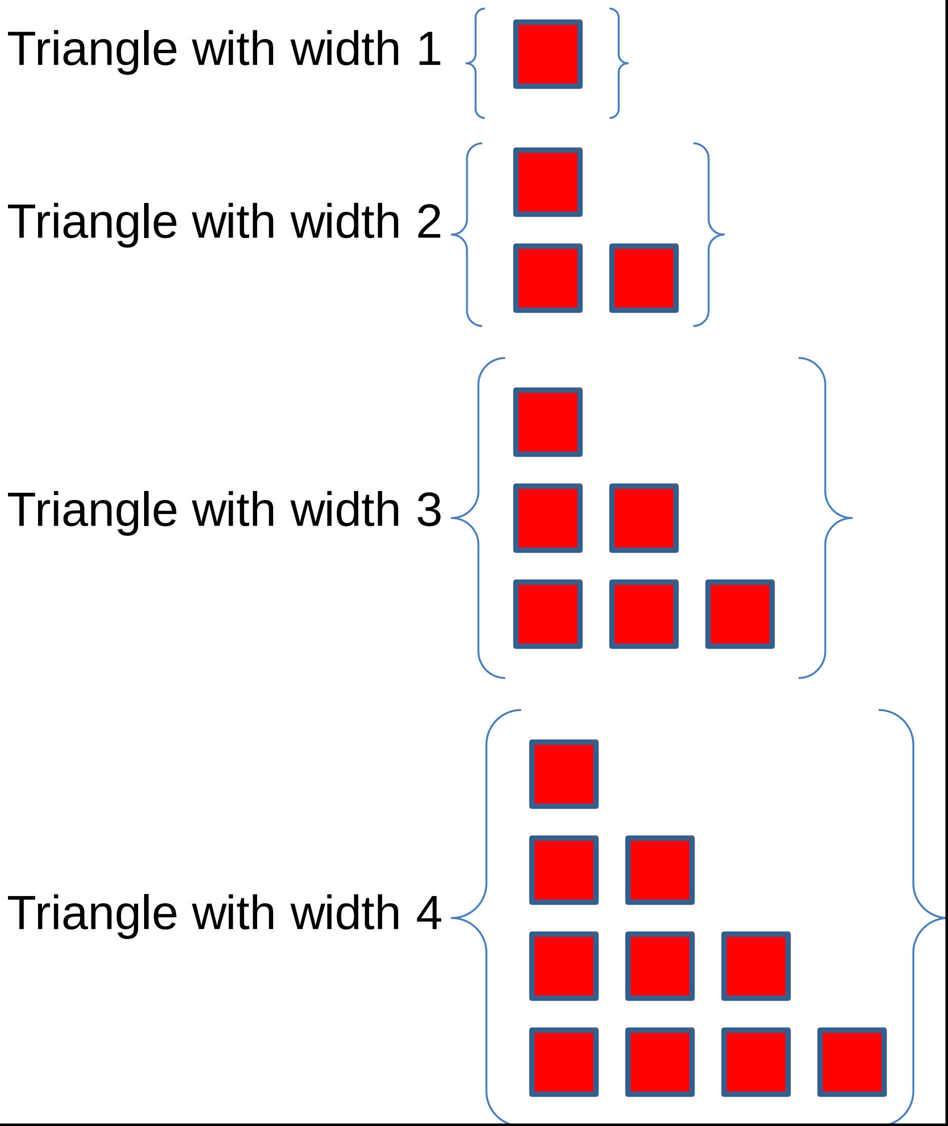
An example – Triangle Numbers

- If each red block has area 1, what is the ***area*** **A(n)** of the Triangle whose *width* is n?
  - Answer:

    $A(n) = n + A(n-1)$
- The above holds for which *n* ? What is the answer for other *n* ?
  - Answer: The recursive equation holds for

    n >= 1.

    For n = 0, the area is 0.



Key Rules to Using Recursion

- Always have a **base case** that **doesn't recurse**
- Make sure recursive case always **makes progress**, by **solving a smaller problem**
- **You gotta believe**
  - Trust in the recursive solution
  - Just consider one step at a time

Frames for Tracing Recursive Code

1. Draw box when method starts

2. Fill in name

3. List every parameter and its argument value.

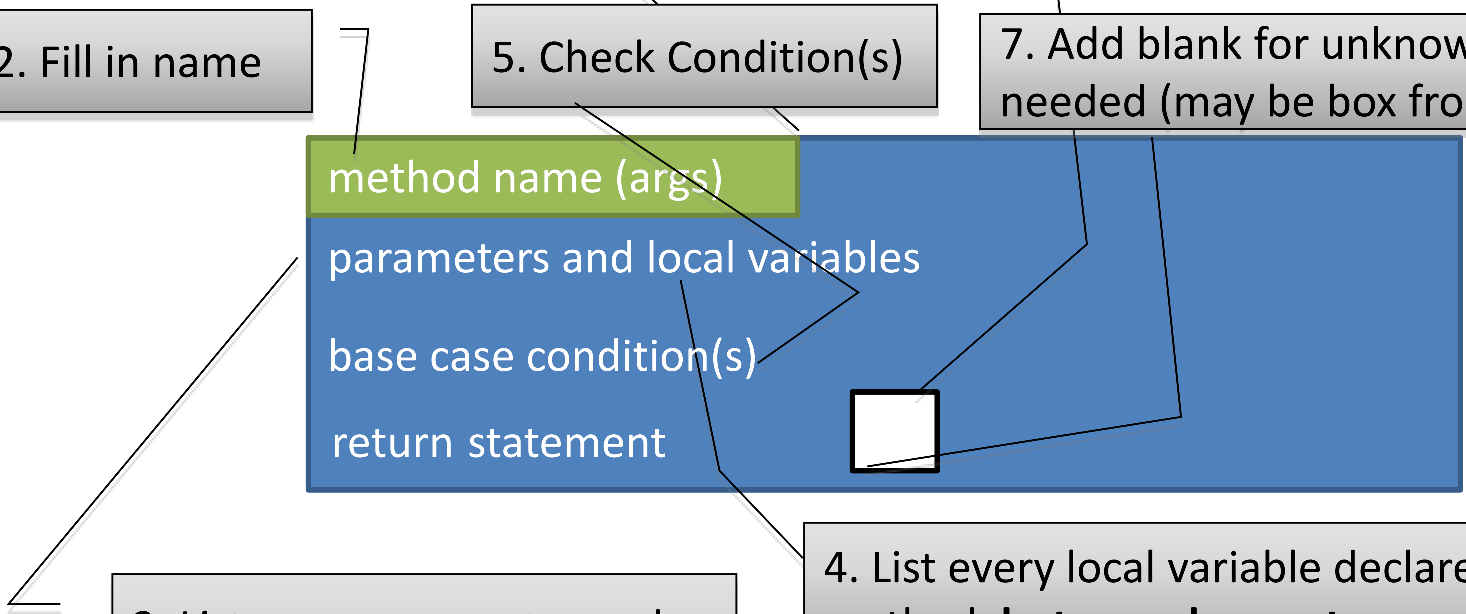
4. List every local variable declared in the method, **but no values yet**

5. Check Condition(s)

6. Add box for next recursive call frame. Add blank for unknown value

7. Add blank for unknown value, if needed (may be box from #6)

8. Step through the method, update variable values, draw new frame for new calls

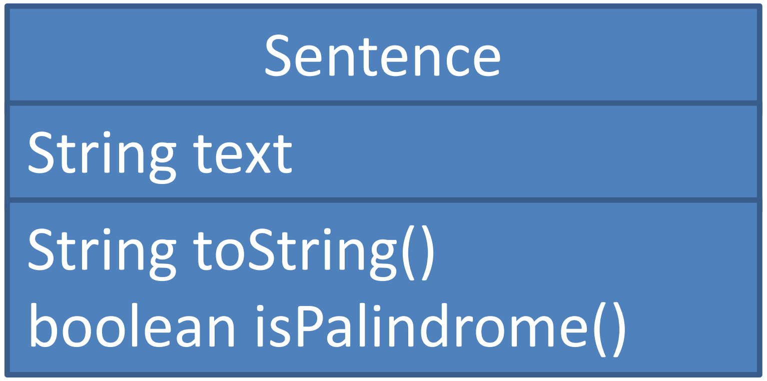


Thanks to David Gries for this technique

Q1-Q3

Programming Problem

- Add a recursive method to Sentence for computing whether Sentence is a palindrome



Practice Practice Practice

- Head to http://codingbat.com/java/Recursion-1 and solve 5 problems. I personally like bunnyEars, bunnyEars2, count7, fibonacci, and noX
- Get help from me if you get stuck
- Then take a look at the recursion homework