CSSE 220

Coupling and Cohesion
Scoping

Please sit with your Crazy Eight partner

Please turn in your assignment in the back

Review of Design Problems

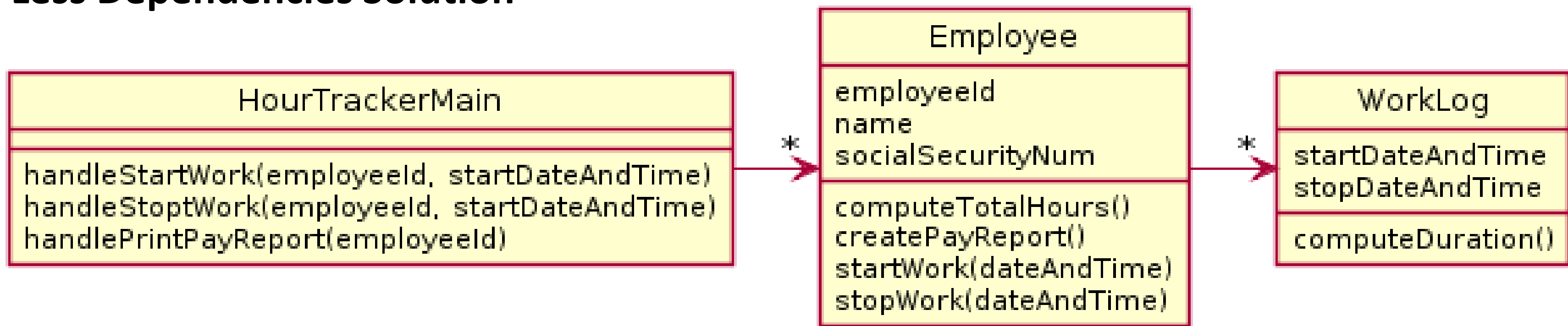
- Turn in your homework before we go over solution

Today’s topic

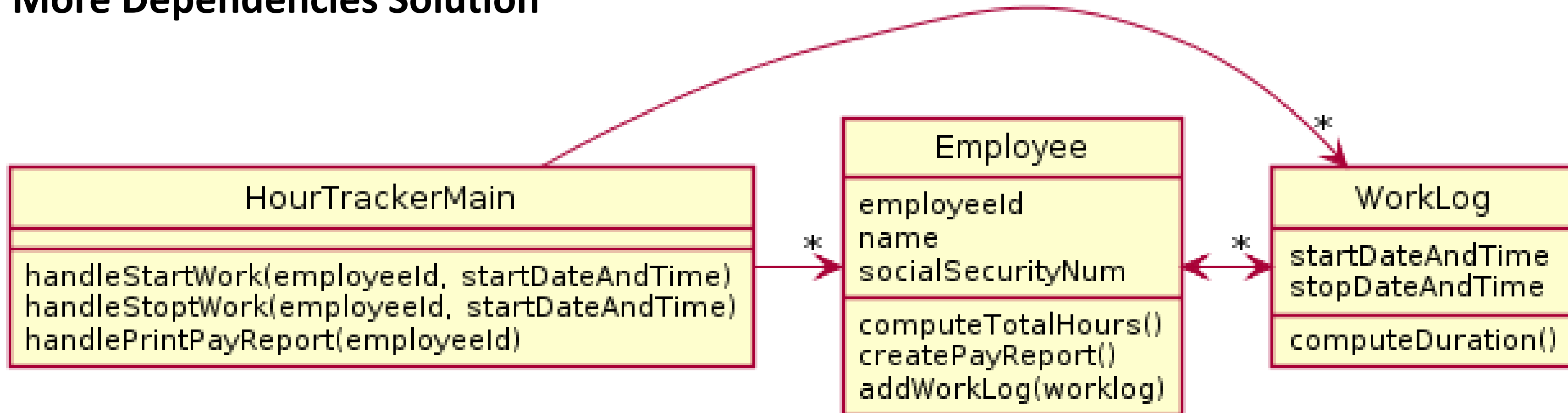
- **Minimize dependencies** between objects when you can
  - Tell don't ask
  - Don't have message chains

A system tracks employee hours at a particular company. Every time any employee starts work and stops work, the system must log it so the employee can be paid correctly and so management knows who was working when. The system must also print out a weekly pay report for each employee which includes total hours, the employee's name, social security number, and employee id.

Less Dependencies Solution



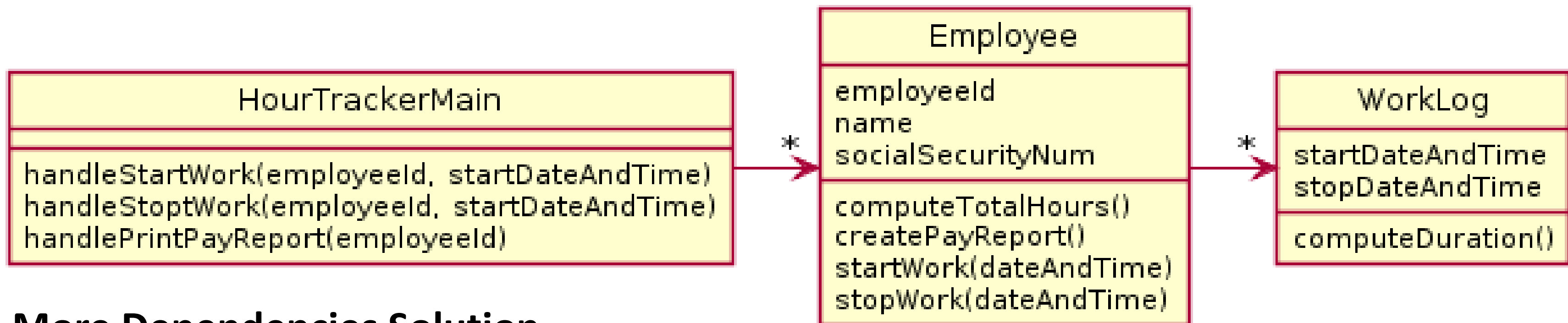
More Dependencies Solution



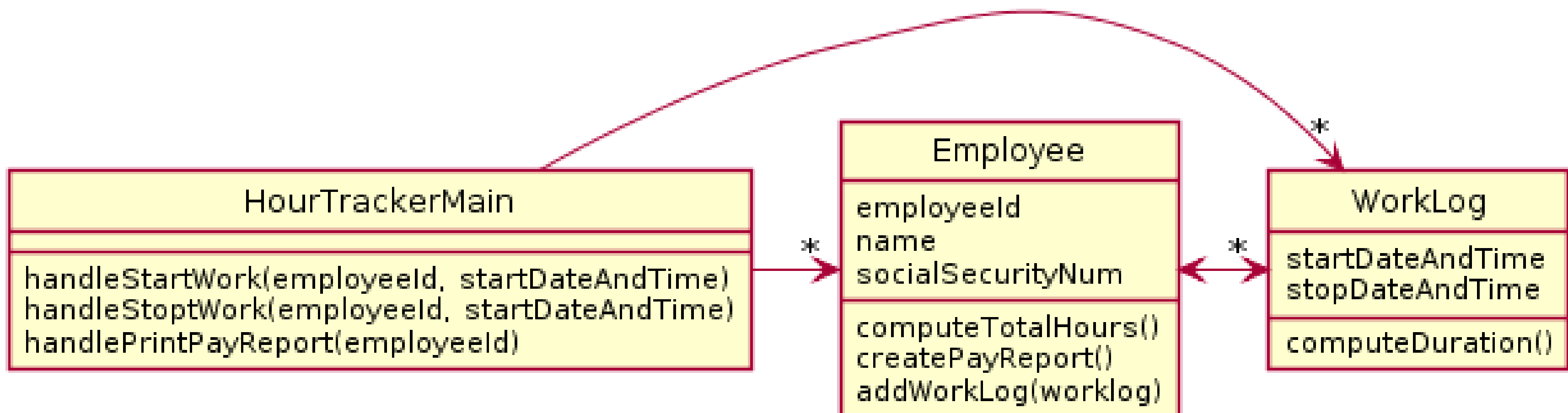
In less dependencies, Employee "insulates" HourTrackerMain from the existence of the WorkLog class. This means changes in the way WorkLog works cannot affect Employee. Similarly, changes in Employee cannot affect WorkLog.

The less dependencies solution is also simpler. Employee fully "owns" all it's own data. In more dependencies, the worklog is edited without employee's knowledge.

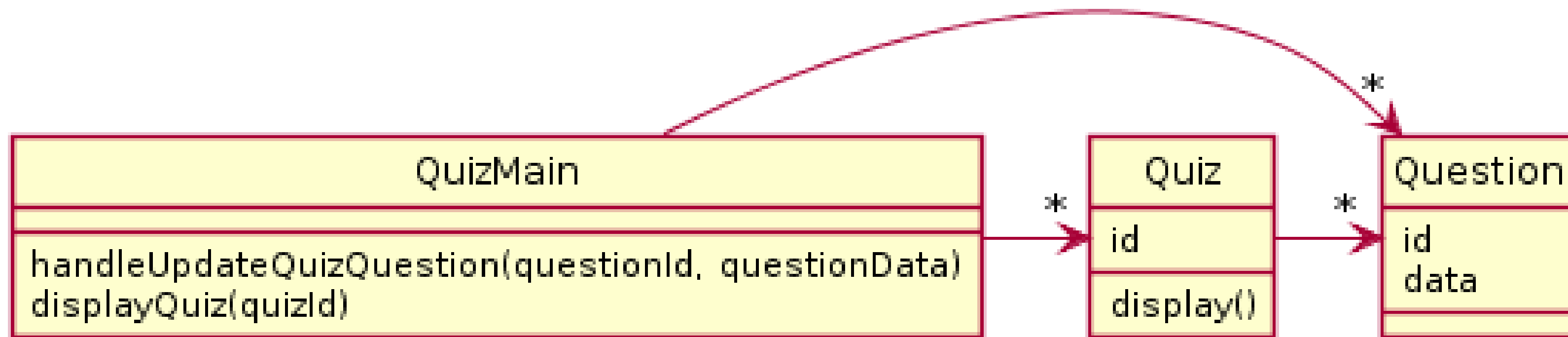
Less Dependencies Solution



More Dependencies Solution



Oftentimes you cannot remove dependencies without breaking functionality though.



Today’s topic

- **Minimize dependencies** between objects when you can
  - If you can see a simpler design that works use it
  - But if you can’t see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains

Tell Don't Ask

```
Point2D center1 = region1.getPosition();
Point2D center2 = region2.getPosition();
double dist = center1.distance(center2);
if(dist > region1.getRadius()) {
   region1.setIsOverlapping(true);
}
```

Sometimes you'll have code that calls a lot of getters on some other object. In essence, this code is **Asking** for a lot of information from the region object.

Note how much this code knows about the Region class. It knows about many of its fields. It has a very strong dependency on the Region class.

Tell Don't Ask

```
region1.flagOverlappingWith(region2);
```

In this code, we've moved the center point and distance calculations into the Region class. Now rather than **asking** the Region for all sorts of data we simply **tell** the region to handle the problem itself and rely on it to do it.

Now, because we rely on the Region object to handle its own data, we have a weaker dependence on the region object.

Tell Don't Ask

```java
public LogFramework getLogFramework() {
    return this.framework;
}
```

Asking is especially bad when you return some internal class that the caller would otherwise not know exists. Why does the caller want the framework? Maybe that should be a tell?

```java
public void activateVerboseLogging() {
    this.framework.setLevel(5);
}
```

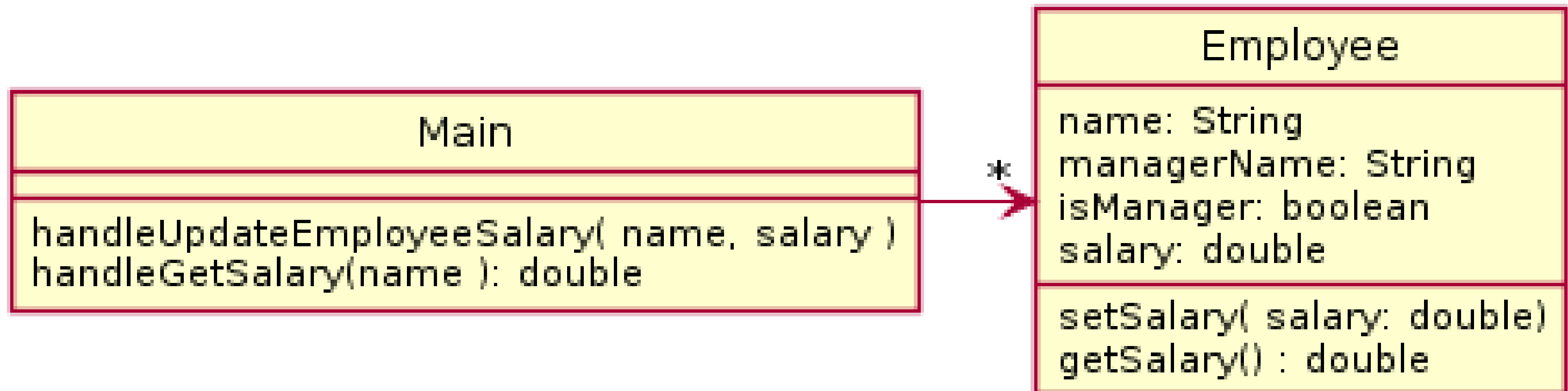
If the caller only needs to do one thing, just add a method to do that thing and insulate the caller from dependence on LogFramework.

Tell Don’t Ask

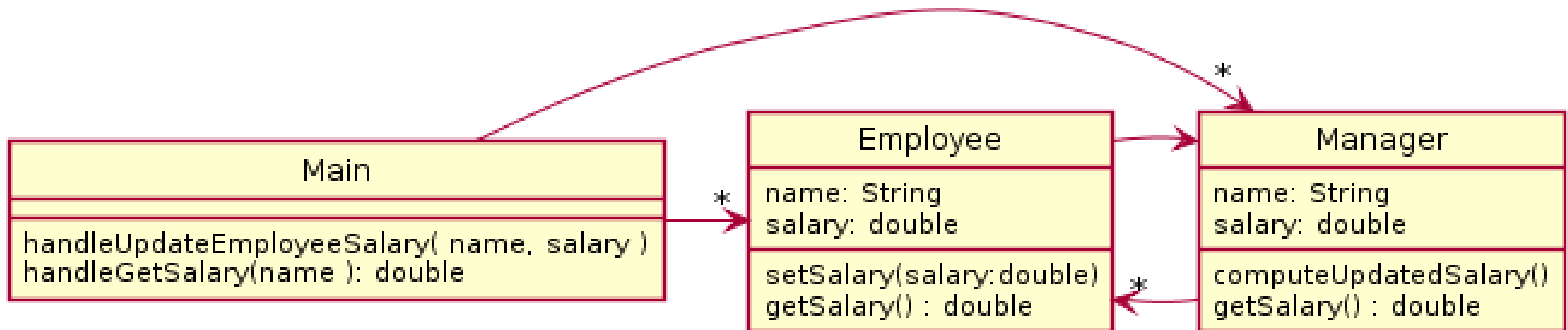
- Be wary getter methods
- Prefer methods that command a class to do something and be responsible for its own state and responsibilities
- If code accesses a lot of internal data of another class, consider if a tell method in that other class might improve the design

Employee Salary Problem

There is a company which has employees, each of which has a salary. There are managers which oversee other employees. Employees have salaries which can be updated from time to time. Unlike employees, a manager's salary is always 10% more than the salary of their top paid employee.

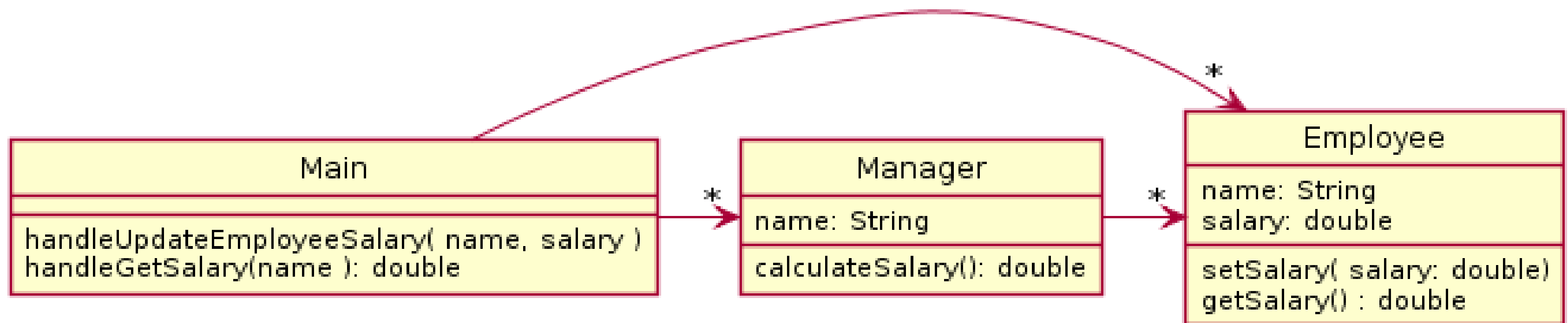


Better Solution



- Anything wrong?
- Room to improve?

Eliminate manager salary field!



Data is technically duplicated if manager contains its own salary field.
What if the two pieces of data were out of sync?
Works well to calculate the salary as needed since it depends upon other data.

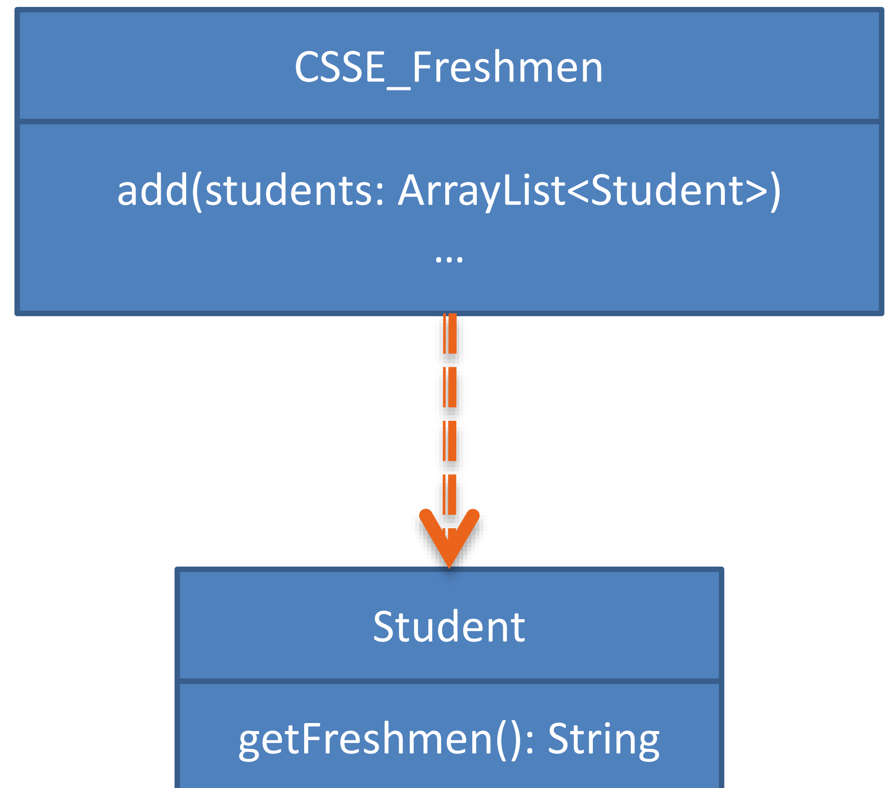
Today’s topic

- **Minimize dependencies** between objects when you can
  - If you can see a simpler design that works use it
  - But if you can’t see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains

UML Interlude: Dependency Relationship

- When one class requires another class to do its job, the first class depends on the second

- Shown on UML diagrams as:
  - dashed line
  - with open arrowhead



Message Chain

A message chain is code in the form:

```
someObject.someMethod().otherMethod().stillOtherMethod();
```

For example

```
myFrame.getBufferStrategy().getCapabilities().getFlip().wait(17);
```

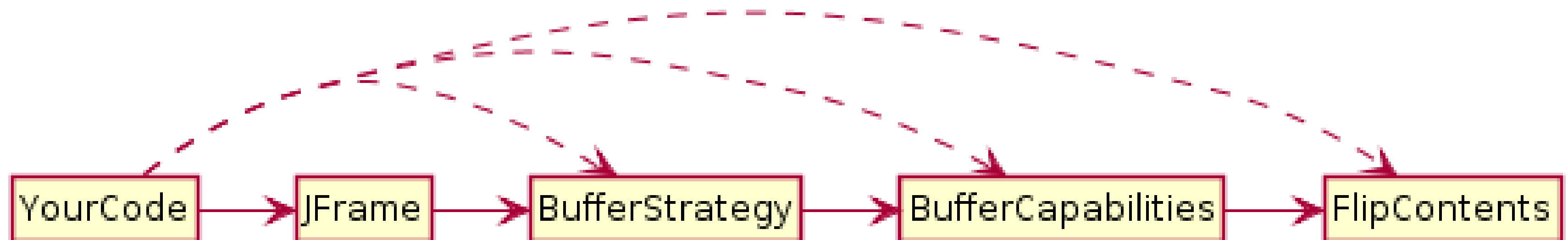
This is generally considered to a warning sign of excessive dependency and problems.

Message Chain

Message chains are not better if you space them across multiple lines, but it does make it more obvious what the problem is.

```
BufferStrategy strategy = myFrame.getBufferStrategy();
BufferCapabilities capabilities = strategy.getCapabilities();
FlipContents flip = capabilities.getFlipContents();
flip.wait(17);
```

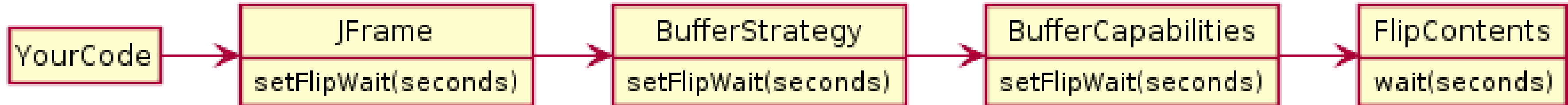
You are depending on internal classes deep within some other object's data



Message Chain: Solution

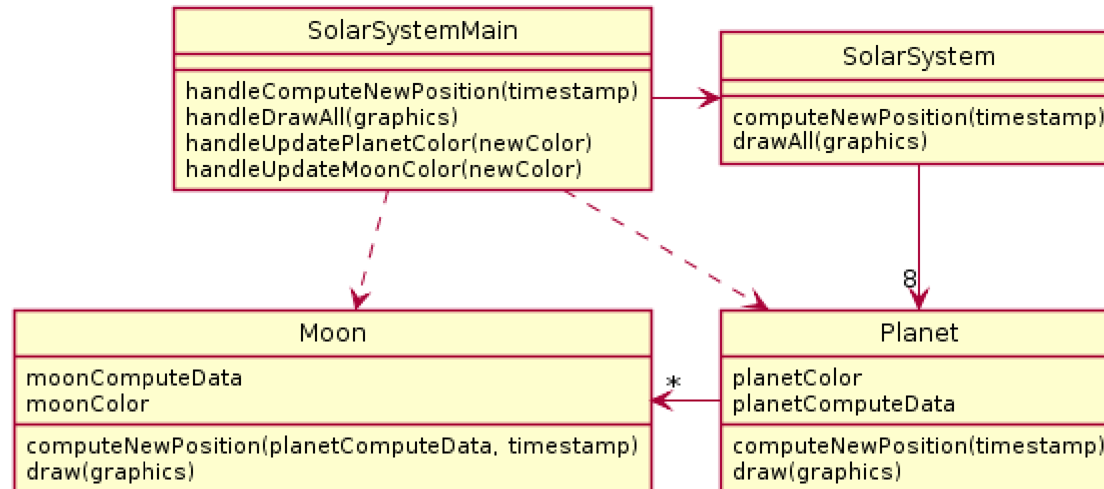
The solution is usually to embed the required feature in the first class in the chain. This insulates the caller from the inner classes. Then the first class might implement the feature itself OR if it still needs to rely on its internals repeat the message chain removal.

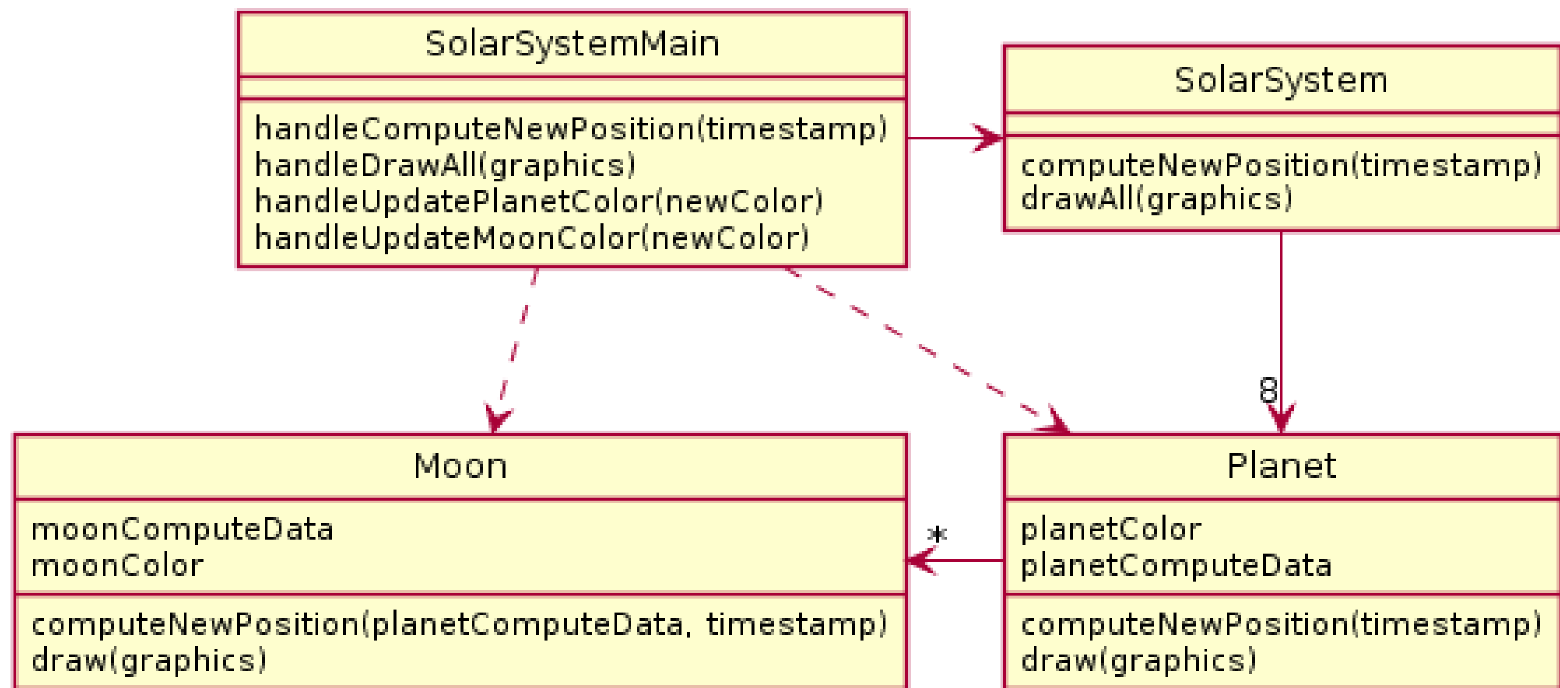
```
myFrame.setFlipWait(17);
```



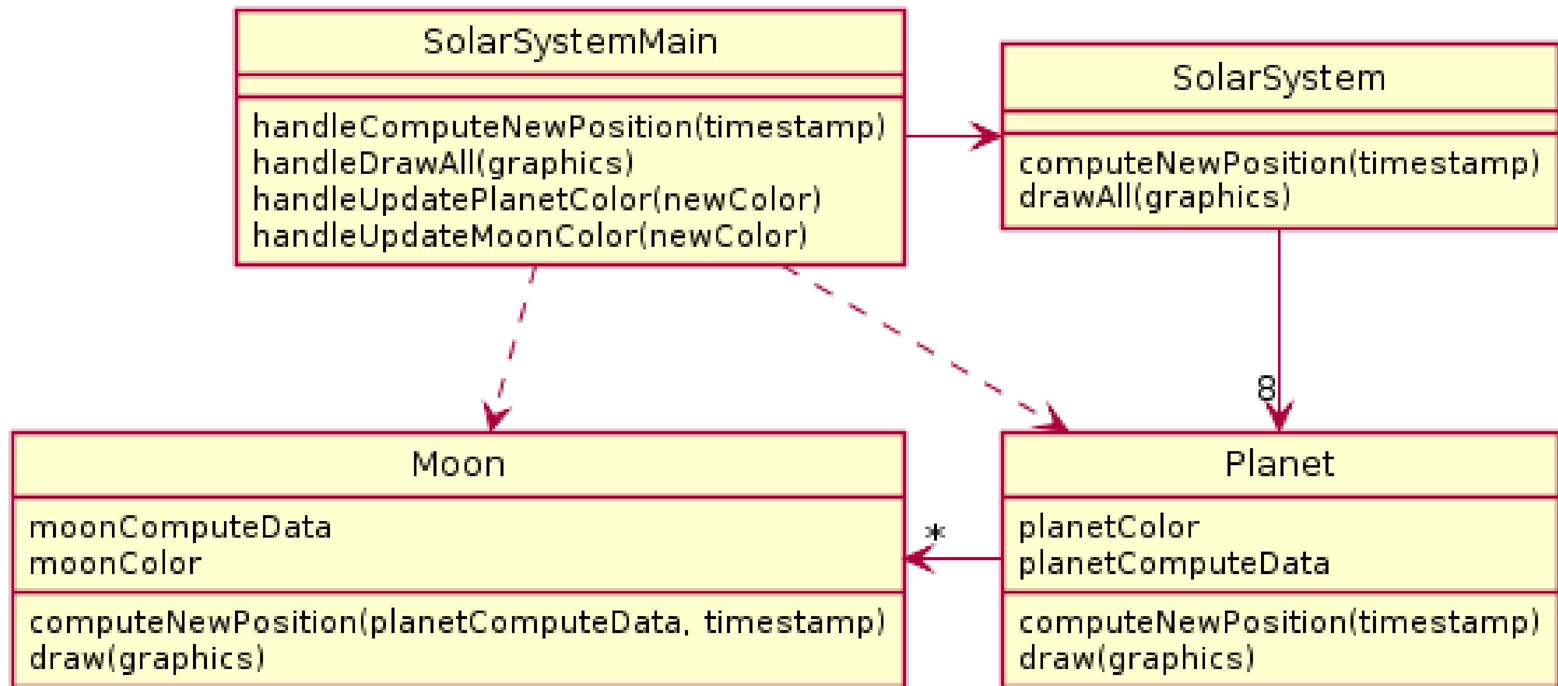
Solar System Problem

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.





- What is wrong here?

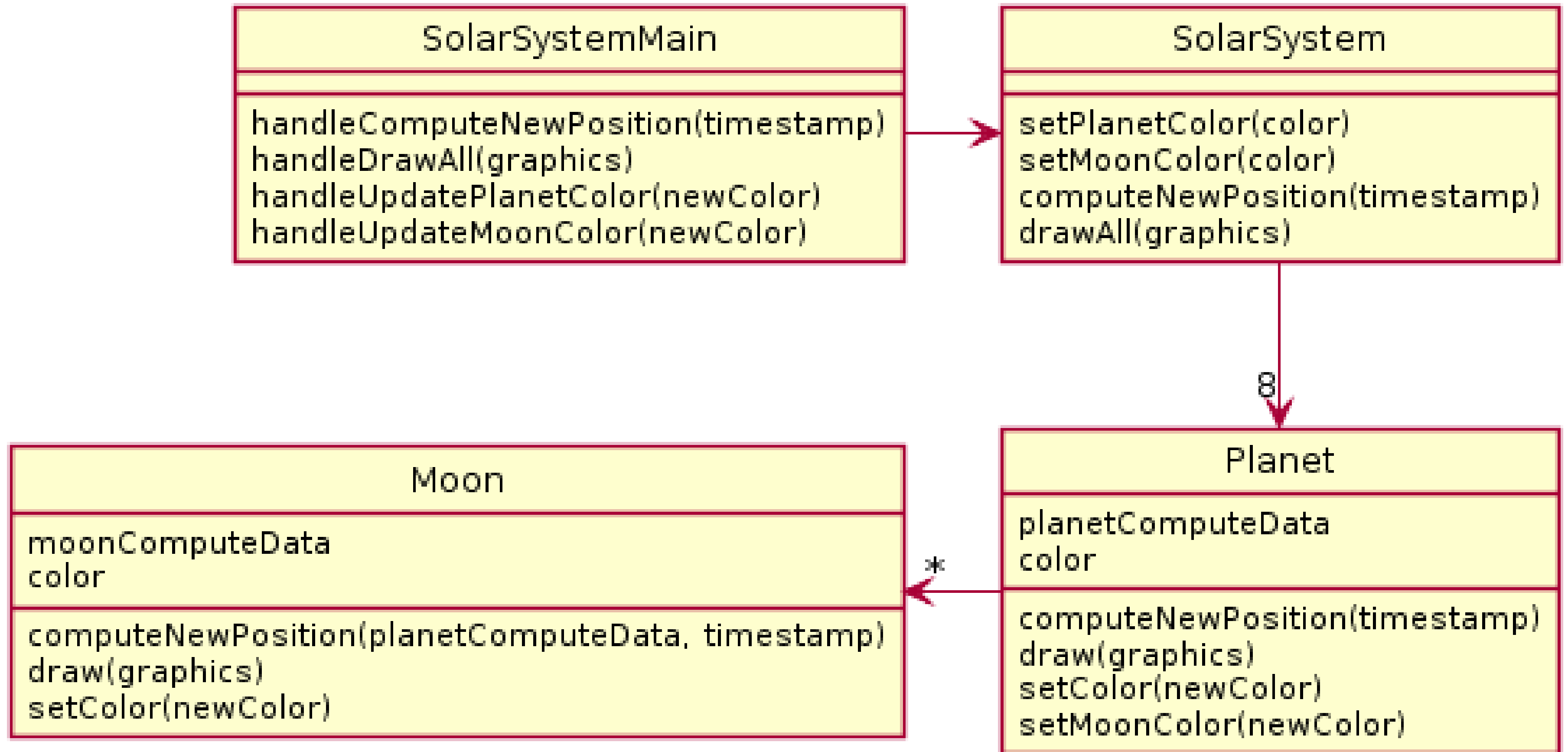


- What is wrong here?

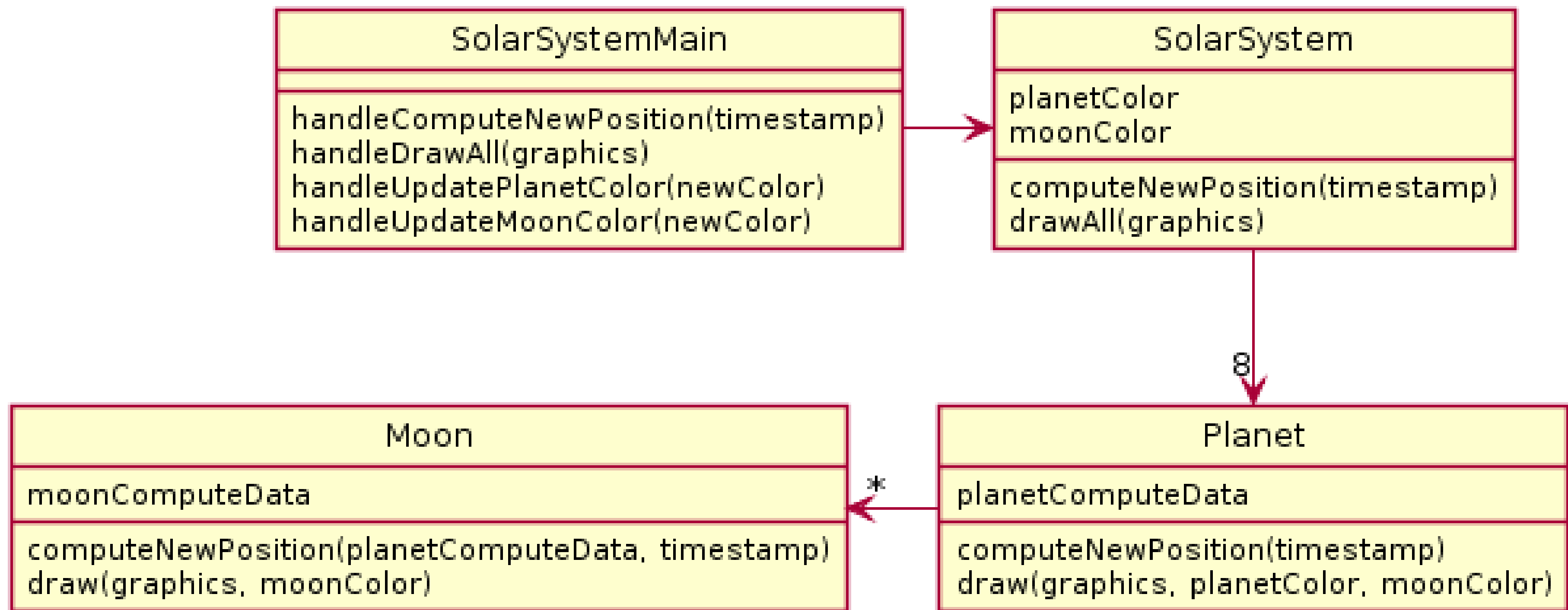
4b. methodChain to update moon (ss.getPlanets().get(0).getMoons().get(0).setColor(color)

2c. planet and moon color are duplicated

Partial Solution



Better Solution



Today’s topic

- **Minimize dependencies** between objects when you can
  - If you can see a simpler design that works use it
  - But if you can’t see a simpler design than the one that you have, at least ensure that you:
    - Tell don't ask
    - Don't have message chains
- Now two related terms: coupling and cohesion

The plan

- Learn 3 essential object oriented design terms:
  - Encapsulation (done)
  - Coupling
  - Cohesion
- Scope (if we have time)

Coupling and Cohesion

- Two terms you need to memorize
- Good designs have high cohesion and low coupling

Consider the opposite:

- Low cohesion means that you have a small number of really large classes that do too much stuff
- High coupling means you have many classes that depend too much on each other

Imagine I want to make a Video Game. Here are two classes in my design. Which is more cohesive?

GameRunner
main(args:String) loadLevel(levelName:String) moveEnemies() drawLevel(g:Graphics2D) computeScore():int computeEnemyDamage() handlePlayerInput() doPowerups(…) runCutscene(cutsceneName:String) //some more stuff

Image
loadImageFile(filename:String) setPosition(x:int,y:int) drawImage(g:Graphics2D)

*Note that in both these classes I've omitted the fields for clarity

Cohesion

- A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a class is said to be cohesive.

  - Your textbook

On to coupling...

Coupling

- Coupling is when one object depends strongly on another

```
//do setup must be called first
this.otherObject.doSetup(var1, var2, var3);

//now we compute the parameter
int var4 = computeForOtherObject(var1,var2);
this.otherObject.setAdditionalParameter(var4);

//finally we display
this.otherObject.doDisplay(this.var5, this.var6);
```

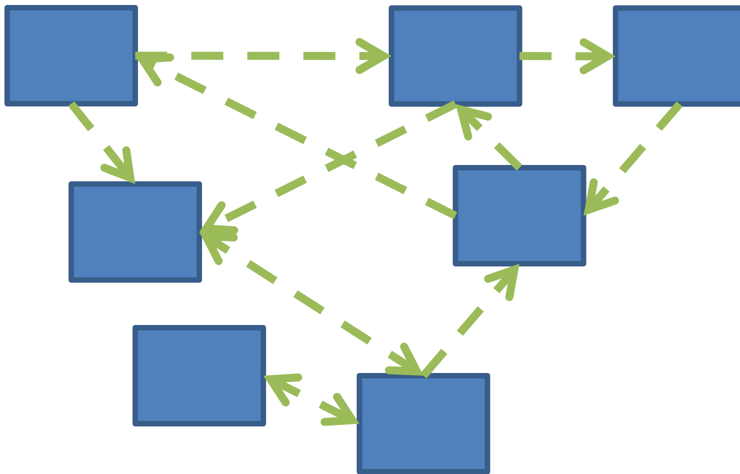
Note that in this design, GameRunner probably had many objects of the image class, but Image does not know the GameRunner class even exists. That's a sign of low coupling between Image and GameRunner.

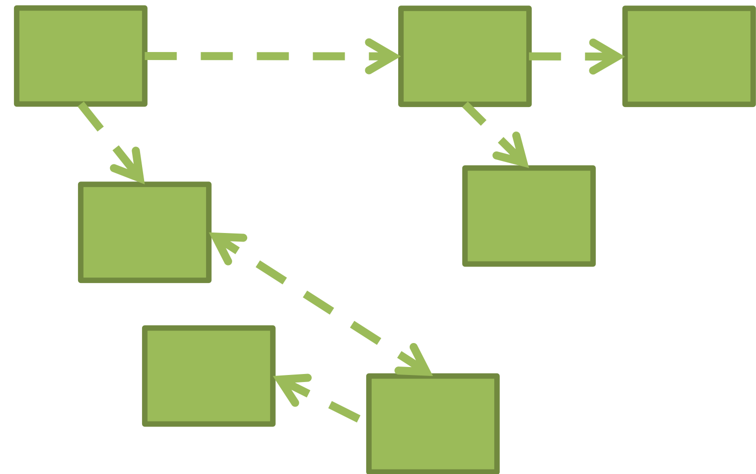
GameRunner
main(args:String) loadLevel(levelName:String) moveEnemies() drawLevel(g:Graphics2D) computeScore():int computeEnemyDamage() handlePlayerInput() doPowerups(…) runCutscene(cutsceneName:String) //some more stuff

Image
loadImageFile(filename:String) setPosition(x:int,y:int) drawImage(g:Graphics2D)

Coupling

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





If we do our design job carefully

- We will break our larger problem into several classes
- Each of these classes will do one kind of thing (i.e. they will have *high cohesion*)
- Our classes will only need to depend on each other in specific, highly limited ways (i.e. they will have *low coupling*). Many classes won't even be aware of most of the other classes in the system.

Imagine that you're writing code to manage a school's students

Things your design should accommodate:

- Handle adding or removing students from the school
- Setting the name, phone number, and GPA for a particular student
- Compute the average GPA of all the students in the school
- Sort the students by last name to print out a report of students and GPA

Discuss and come up with a design with those near you. How many classes does your system need?

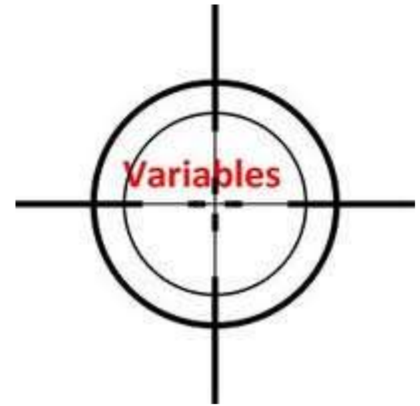
Note that

- Cohesion makes us want:
  - Many smaller classes
  - Classes do only one thing
- If classes are too small
  - Tend to need to depend on each other
  - Coupling rises

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

Variable Scope



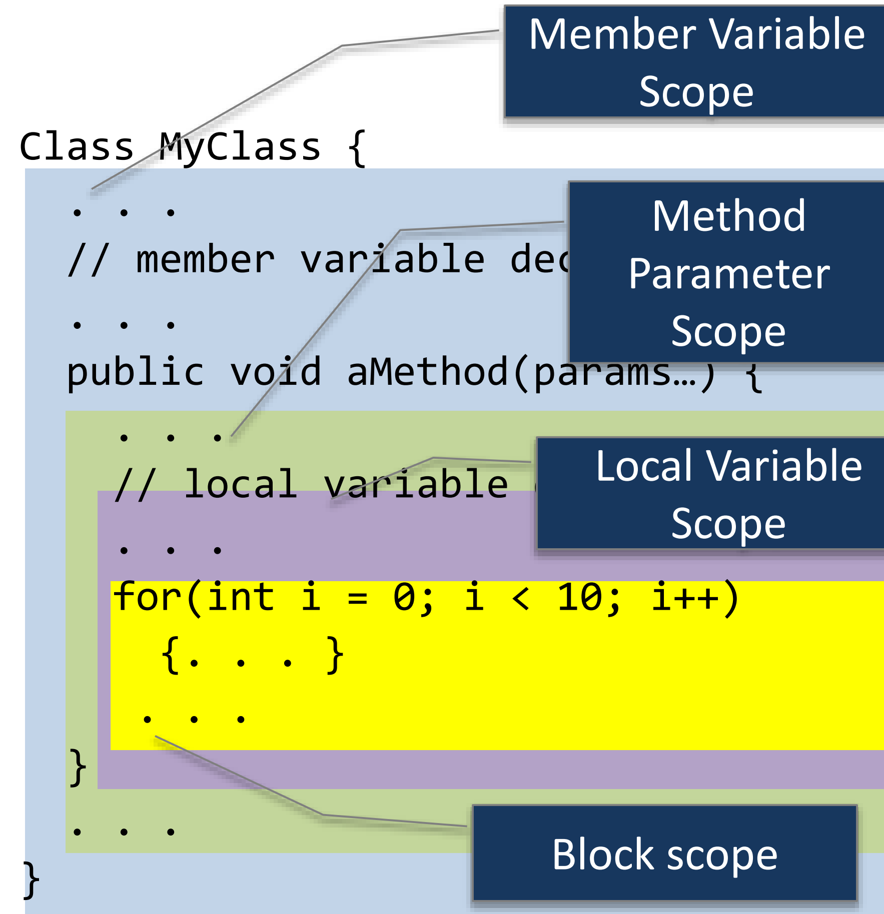
Scope **is the region of a program in which a variable can be accessed**

- ***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`



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.

Work Time

- Crazy Eights – see due date on schedule page
- Work with your partner on the Crazy Eights project
  - Get help as needed
  - Finding your partner…

Before you leave today, make sure that you and your partner have ***scheduled a session to complete the Crazy Eights project***

- Where will you meet?
  - ***Try the CSSE lab F-217/225***
- When will you meet?
  - ***Consider this evening***,

    7 to 9 p.m. ***Exchange contact info*** in case one of you needs to reschedule.
- **Do it *with your partner.*** If your partner bails out, **DON'T** do it alone until you communicate with your instructor.