CSSE 220 Day 12

Coupling and Cohesion
Scoping

Please download VideoStore from your SVN

The plan

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

Coupling and Cohesion

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

At a very high level:

- 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 which 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

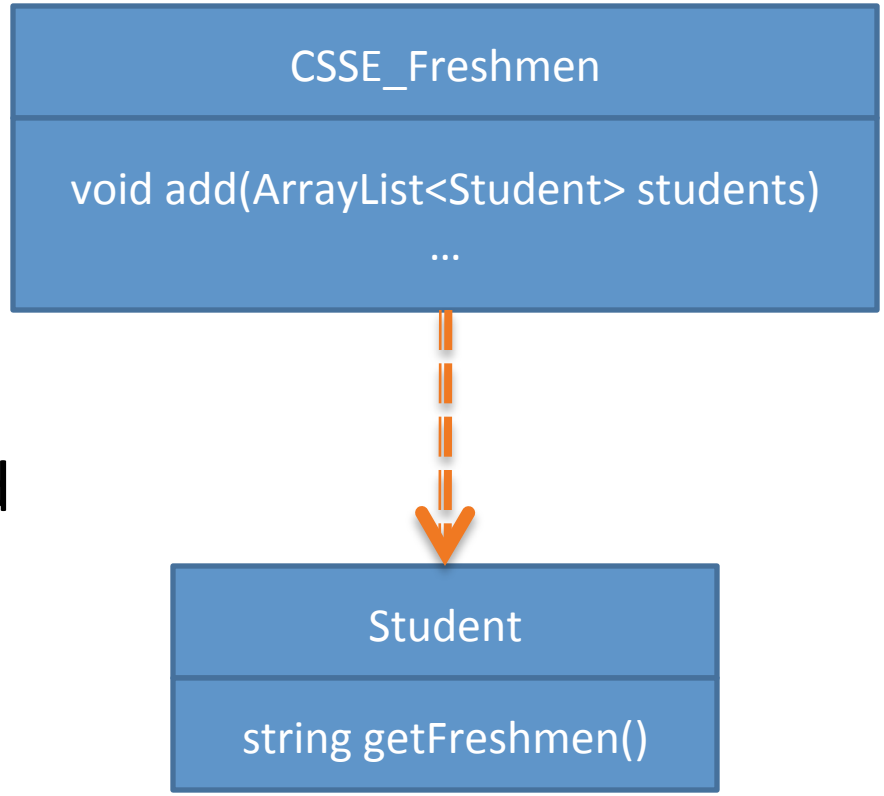
- A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a class is said to be cohesive.

  - Your textbook

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



Coupling

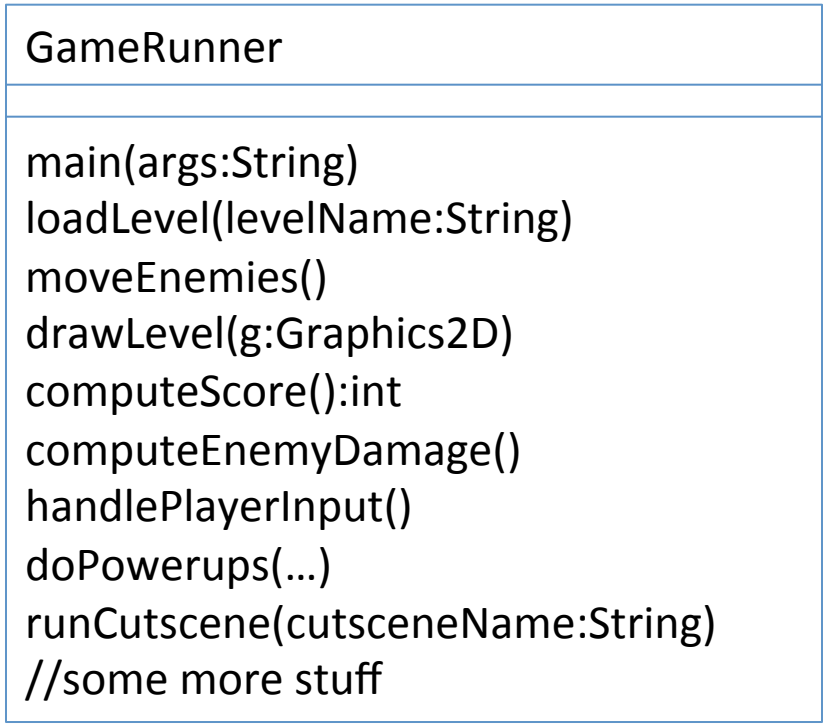
- Coupling is one object depends strongly on another

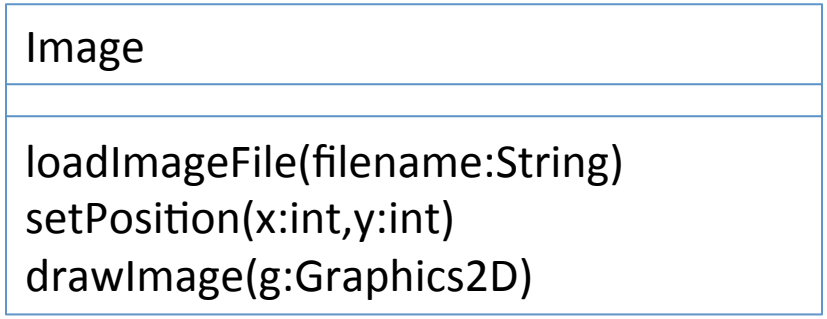
```
//do setup must be called first
this.otherObject.doSetup(var1, var2, var3);

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

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

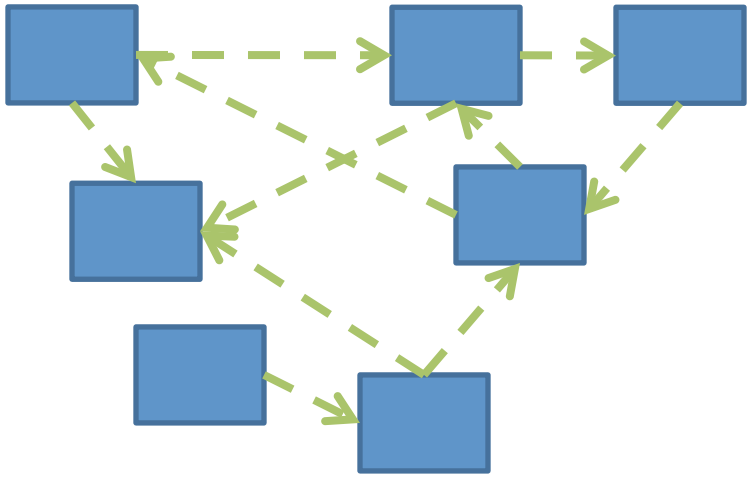
Note that in this design, GameRunner probably had many objects of the image class, but Image does not know the GameRunner class even exists. That's a sign of low coupling between Image and GameRunner.

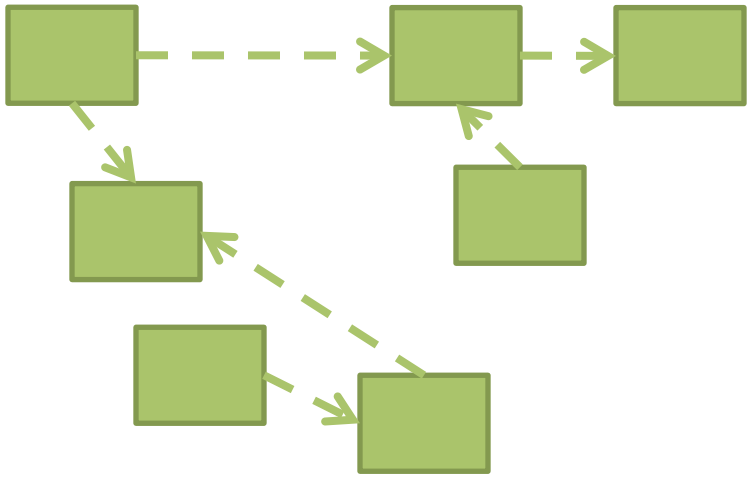




Coupling

- 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 nearby you. How many classes does you system need?

Note that

- Cohesion will tend to want us to make many smaller classes, each of which will do only one thing
- But if the classes are too small, they'll tend to need to depend on each other to do work, and the coupling will get bad

Hints #1 for Designing Objects

- Look for the nouns in your problem, consider making them objects
- Keep any one objects from getting too “fat” – containing too many methods or fields
- Avoid Plural Nouns
- Avoid Parallel Structures

Practice

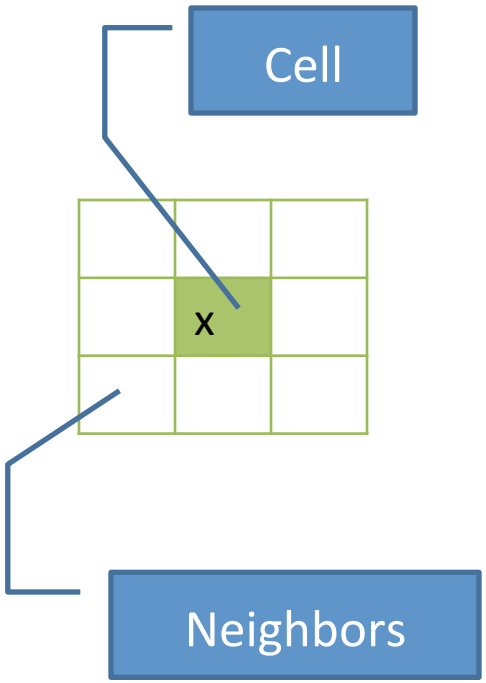
- Step 1 – Get into pairs
- Step 2 – Do the Video Store Quiz (you should talk together but each of you will submit a separate page)
- Step 3 – the mystery step, where we try and fix the problem

The Mystery Step

- The problem is that the customer object is not very cohesive – knows way to much about how things should be priced
- Add a getCost(int daysRented) method to Movie and make statement() call it
- Try to do something similar to rental points if you can

Game of Life

1. A new cell is born on an empty square if it has exactly 3 neighbor cells
2. A cell dies of overcrowding if it is surrounded by 4 or more neighbor cells
3. A cells dies of loneliness if it has just 0 or 1 neighbor cells



Developed by John Conway, 1970

Game of Life hints:

- Follow the TODO's. ***Test as frequently as practical.***
  - If a part is hard, break it down into sub-parts and test each sub-part as you go.
- There are some clever ways to avoid cluttering code that references cells with IF's to ensure that you are properly retrieving neighbors that wrap around the grid:
  - How to "Wrap" -- If the board is 10x10, attempting to reference: `board[10][3]` `--` convert to `board[0][3]`
    - (using the `%` operator on rows and columns) 10%10 = 0 ; 10 % 3 = 3.
    - (totalRows % x = row value)
    - totalColumns % y = columnVlu
  - Write a "getter" that gets the value of a cell and returns the correct value (0?) if the reference is off the edge of the board. Ditto for a "setter" if needed.

Animating Game of Life

- How: use `Timer` class to automatically "click" button

- Details: in `GameOfLifeMain`:
  - Use local variable for `UpdateButton` object
  - Add timer code to end of main to repeatedly click button at regular intervals:
    - ```
      Timer mrClicker =
                  new Timer(INTERVAL,
      updateButton);
      mrClicker.start();
      ```
- Learn more: Big Java, Ch. 9.9
      ```

Work Time

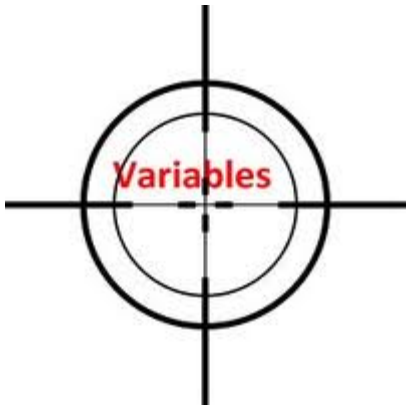
- Game of life due a week from Monday 11:59
- Work with your partner on the Game of Life project
  - Get help as needed
  - Finding your partner…

Before you leave today, make sure that you and your partner have ***scheduled a session to complete the Game of Life 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.

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);
}
```

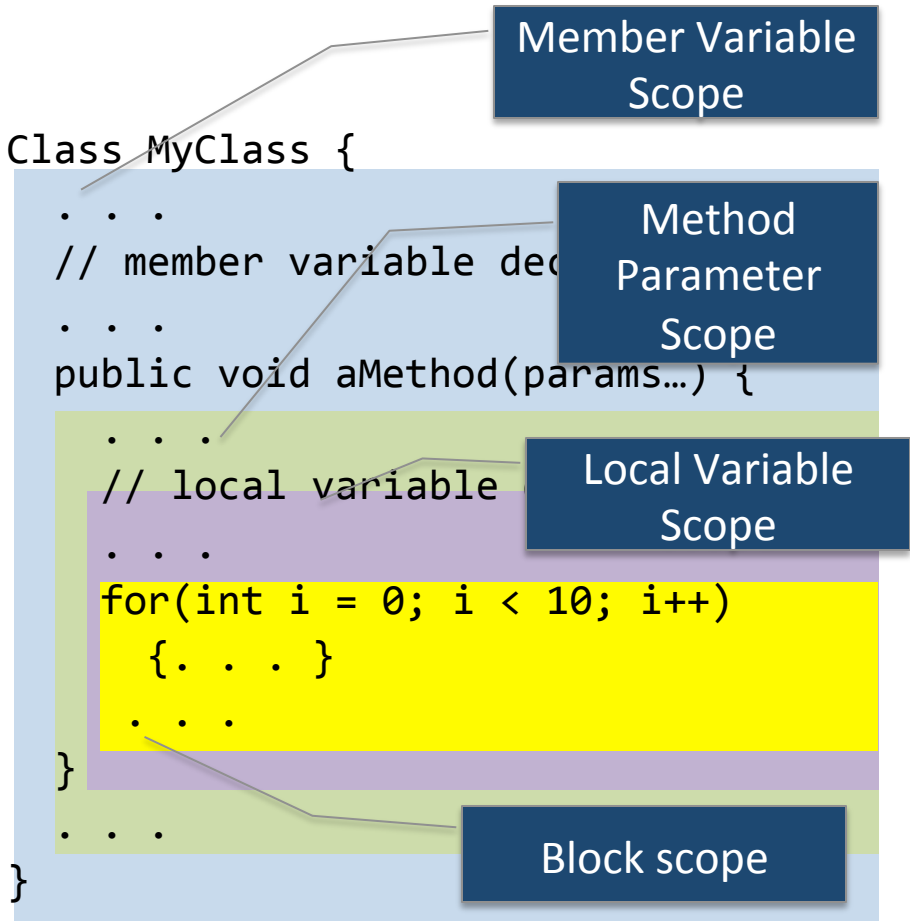
Why do you suppose scoping exists? What happens if two variables have the same name in the same code location?

- Please take 15 seconds and think about it
- Turn to neighbor and discuss it for a minute
- Then let's talk?

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.

What you have learned

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