CSSE 220 Day 13

Details on class implementation, Interfaces and Polymorphism

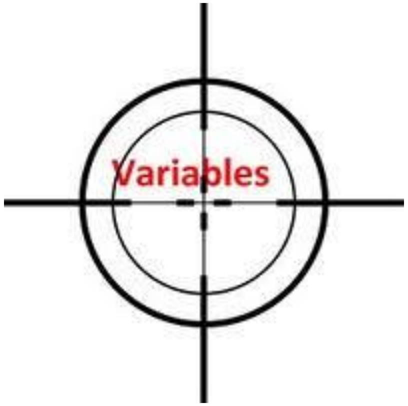
Check out *OnToInterfaces* from SVN

Questions?

Today

- Variable scope
- Interfaces and polymorphism

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);
}
```

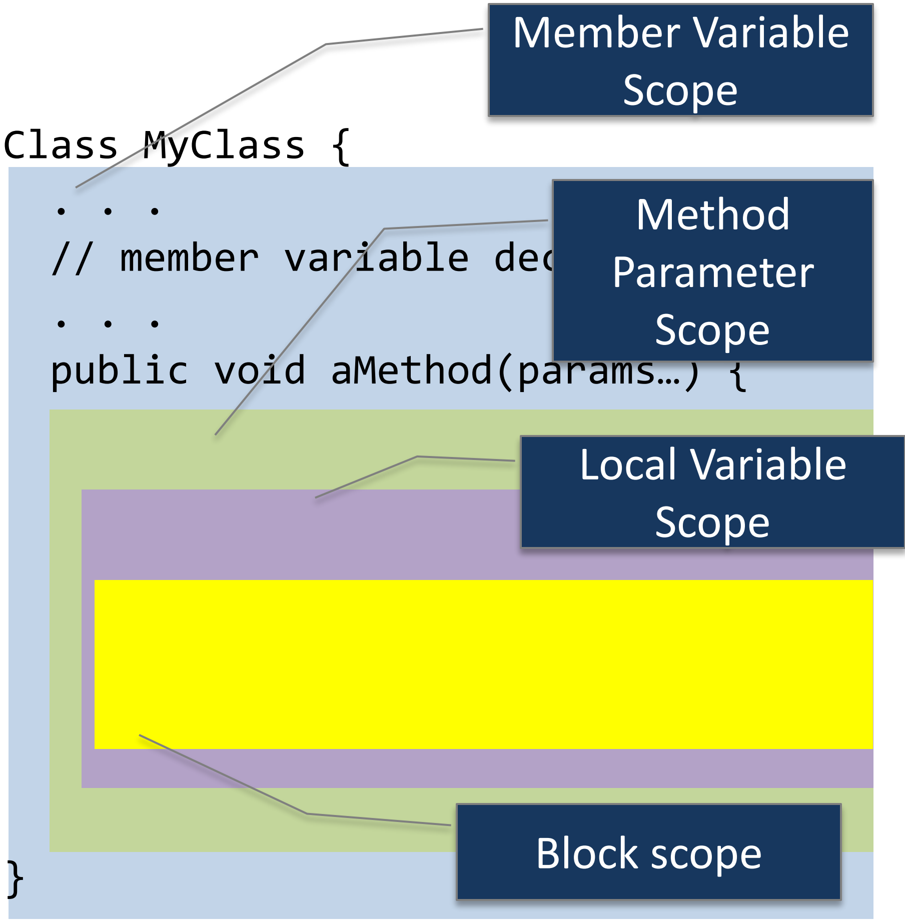
Why do you suppose scoping exists? What happens if two variables have the same name in the same code location?

- 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`



```
Class MyClass {
  . . .
  // member variable dec
  . . .
  public void aMethod(params…) {
}
```

Member Variable Scope

Method Parameter Scope

Local Variable Scope

Block scope

Q1-2

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.

Q3

Today

- Variable scope
- Interfaces and polymorphism

Interface Types

- Express common operations that multiple classes might have in common

- Make “client” code more reusable

- Provide method signatures and documentation

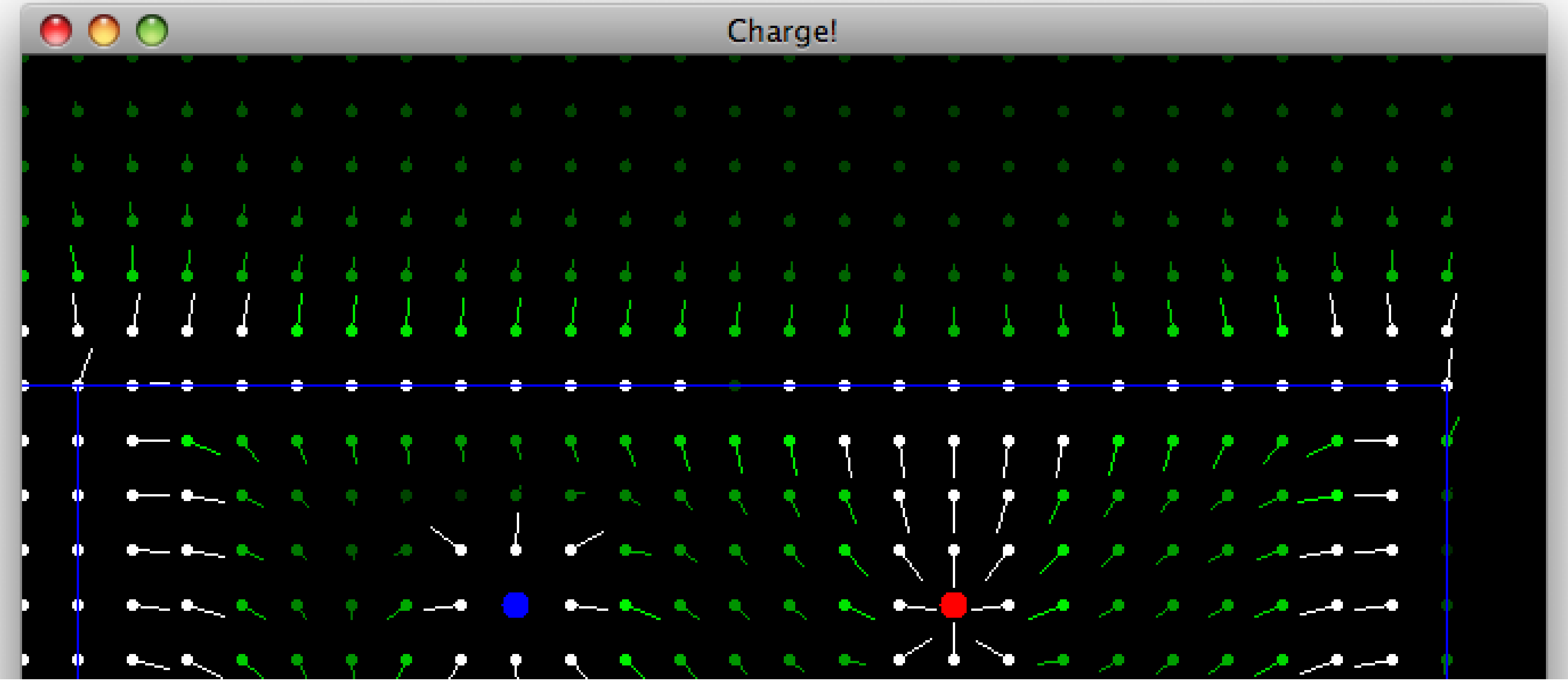
- Do **not** provide method implementations or fields

Interface Types: Key Idea

- Interface types are like **contracts**
  - A class can promise to **implement** an interface
    - That is, implement every method
  - Client code knows that the class will have those methods
    - Compiler verifies this
  - Any client code designed to use the interface type can automatically use the class!

Live Coding Activity

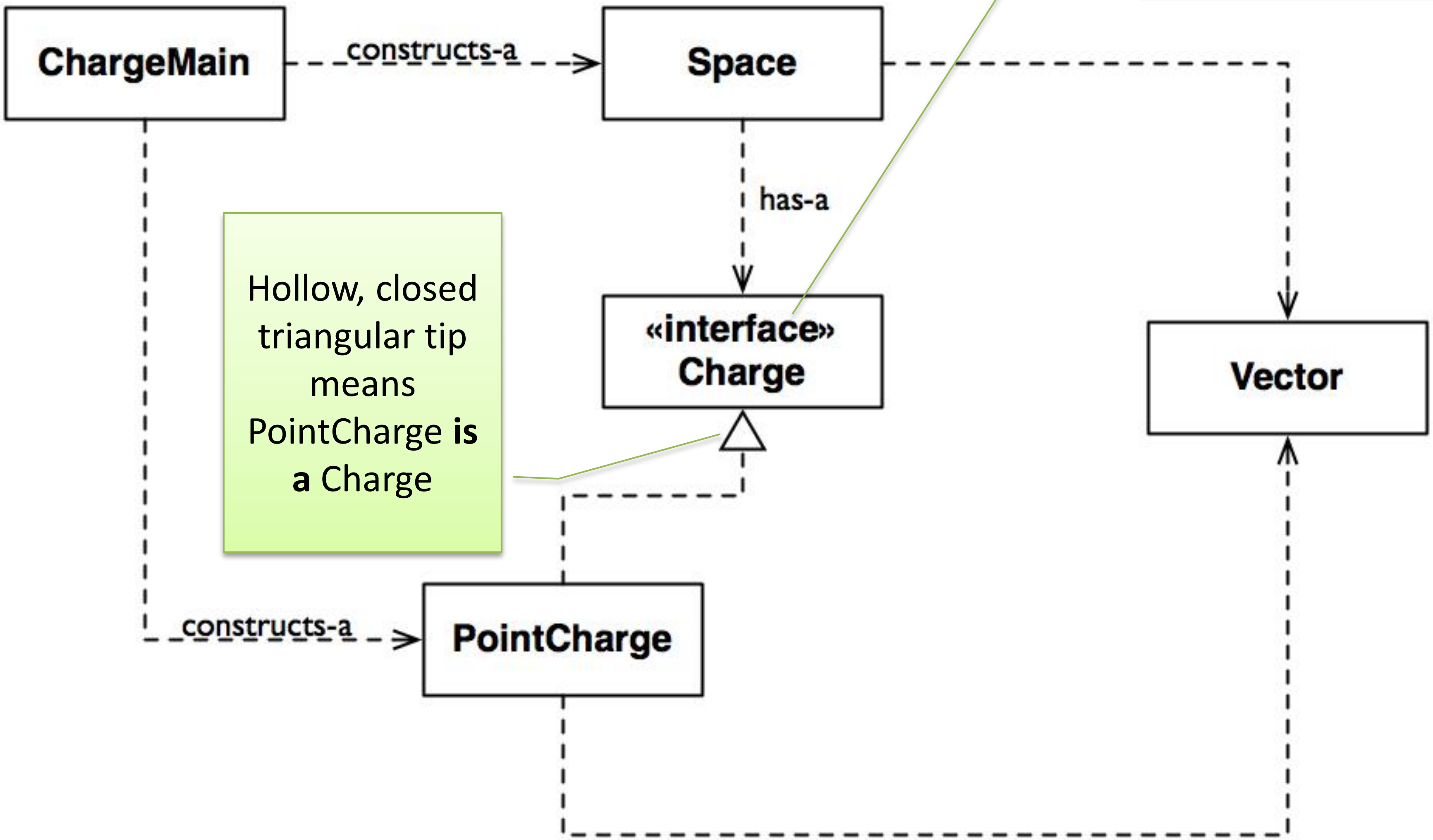
- Countries, Balances, and Measurable



Charges Demo

EXAMPLE

Charges UML



Distinguishes interfaces from classes

Hollow, closed triangular tip means PointCharge **is a** Charge

Q4

Notation: In Co

interface, not class

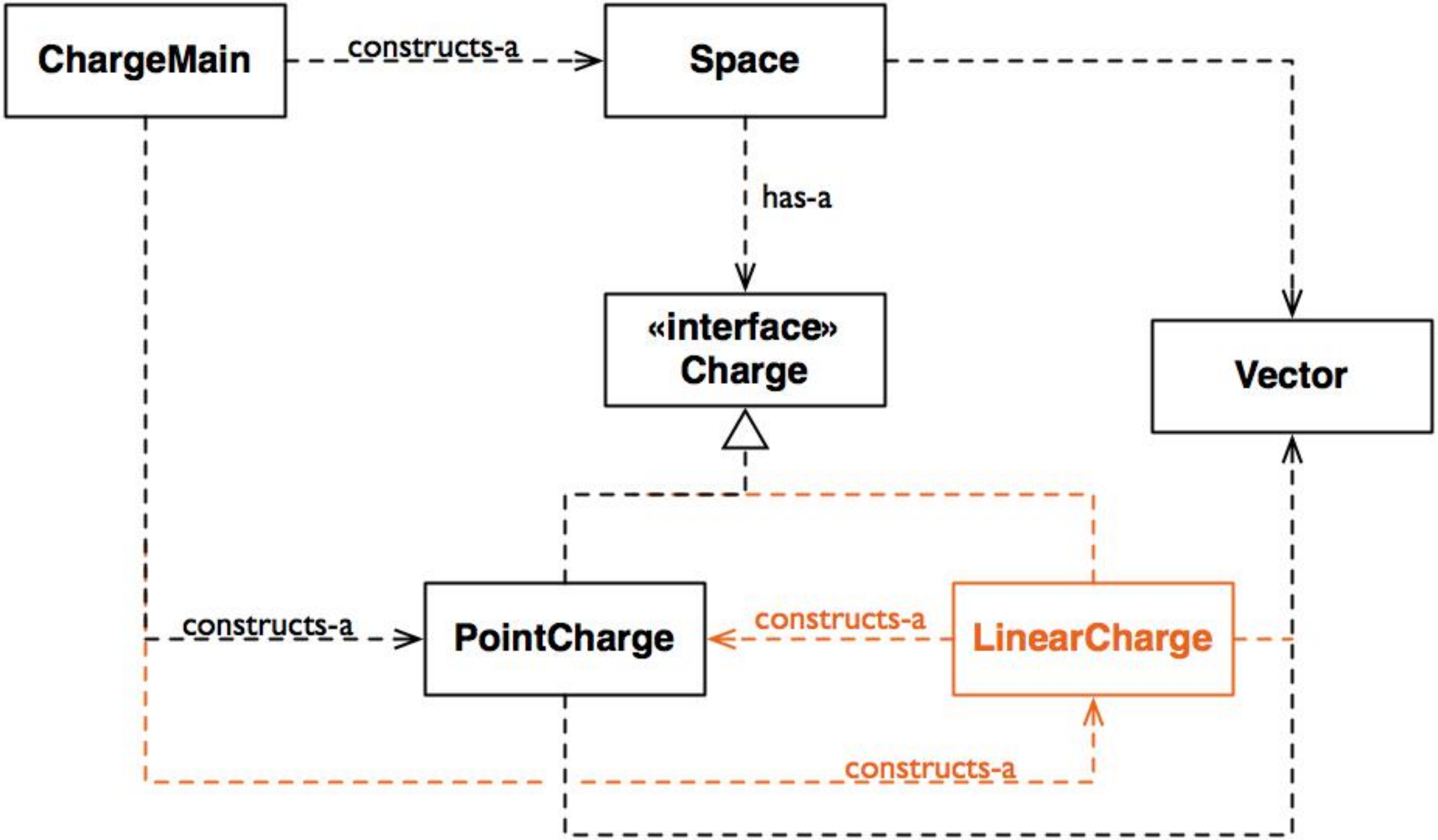
```
     public interface Charge {
          /**
           *  regular javadocs here
           */
          Vector forceAt(int x, int y);

          /**
           *  regular javadocs here
           */
          void drawOn(Graphics2D g);
     }

public class PointCharge implements Charge {

}
```

No “public”, automatically are so

No method body, just a semi-colon

PointCharge promises to implement all the methods declared in the **Charge** interface

Updated Charges UML



Interfaces reduce coupling!

Q6

How does all this help reuse?

- Can pass an **instance** of a class where an interface type is expected
  - But only *if the class* ***`implements`*** *the interface*
- We passed `LinearCharge`s to `Space`'s `addCharge(Charge c)` method without changing `Space`!

- **Use interface types** for field, method parameter, and return types whenever possible

Q6

Why is this OK?

-
  ```
  Charge c = new PointCharge(…);
  Vector v1 = c.forceAt(…);
  c = new LinearCharge(…);
  Vector v2 = c.forceAt(…);
  ```

- The type of the **actual object** determines the method used.

Polymorphism

- Origin:
  - Poly → many
  - Morphism → shape
- Classes implementing an interface give **many differently "shaped" objects for the interface type**

- Late Binding: choosing the right method based on the actual type of the implicit parameter **at run time**

Q8-Q9

WORK TIME