

# CSSE 220 Day 17

Details on class implementation,  
Interfaces and Polymorphism

Check out *OnToInterfaces* from SVN

Questions?

# Today

- ▶ Variable scope
  - ▶ Packages recap
  - ▶ Interfaces and polymorphism
- 

# Variable Scope

- ▶ *Scope*: 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 area() {  
    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 (Field or Method) Scope

- ▶ **Member scope**: anywhere in the class, including *before* its declaration
  - This lets methods call other methods later in the class.
- ▶ **public static** class members can be accessed from outside with “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.

# Last Bit of Static

- ▶ Static imports let us use unqualified names:
  - `import static java.lang.Math.PI;`
  - `import static java.lang.Math.cos;`
  - `import static java.lang.Math.sin;`
  
- ▶ See the `polygon.drawOn()` method in the `DesigningClasses` project

# Review: Packages

- ▶ Let us group related classes
- ▶ We've been using them:
  - `javax.swing`
  - `java.awt`
  - `java.lang`



# Avoiding Package Name Clashes

- ▶ Remember the problem with Timer?
  - Two Timer classes in different packages
  - Was OK, because packages had different names
- ▶ Package naming convention: reverse URLs
  - Examples:
    - `edu.roseHulman.csse.courseware.scheduling`
    - `com.xkcd.comicSearch`



Specifies the company or organization



Groups related classes as company sees fit

# Qualified Names and Imports

- ▶ Can use import to get classes from other packages:
  - `import java.awt.Rectangle;`
- ▶ Suppose we have our own Rectangle class and we want to use ours and Java's?
  - Can use “fully qualified names”:
    - `java.awt.Rectangle rect =  
new java.awt.Rectangle(10,20,30,40);`
  - U-G-L-Y, but sometimes needed.

# Interface Types

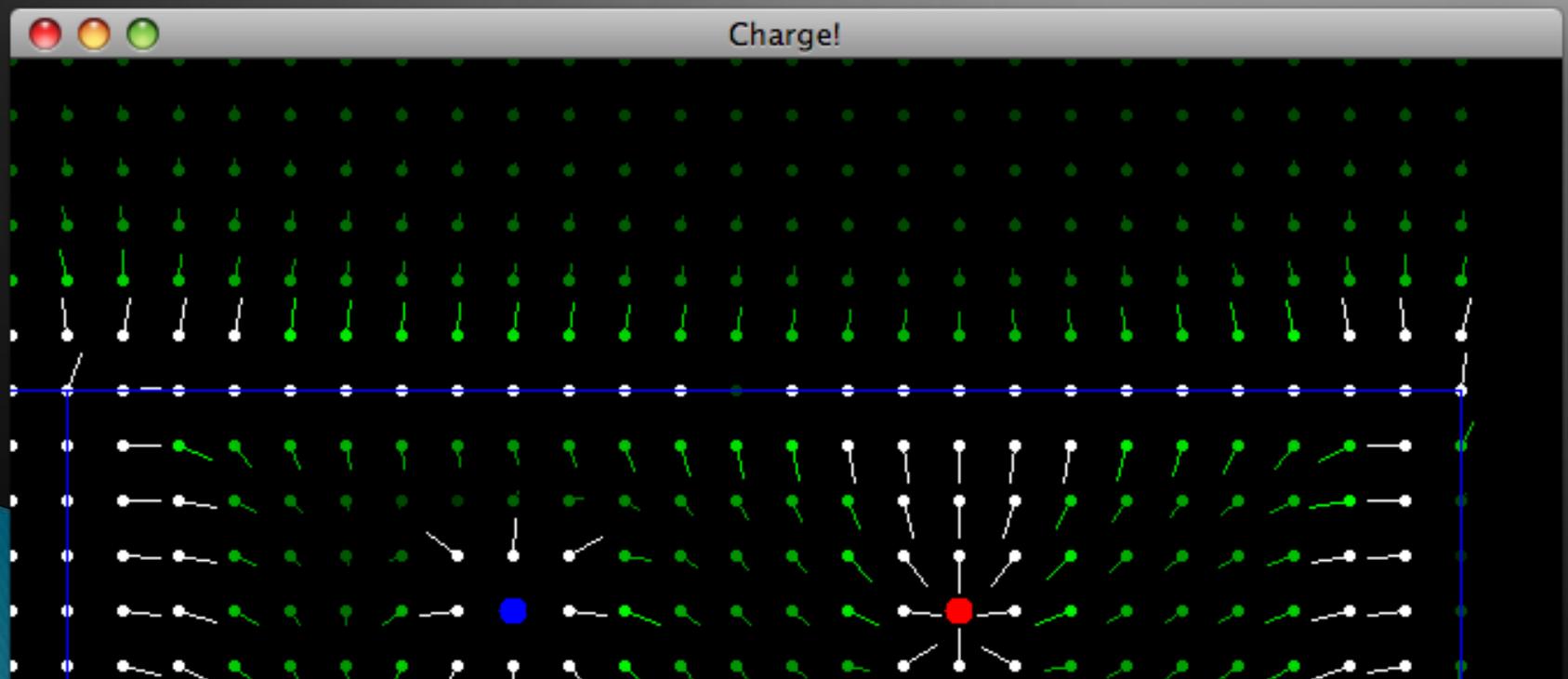
- ▶ Express common operations that multiple classes might have in common
- ▶ Make “client” code more reusable
- ▶ Provide method signatures and docs.
- ▶ Do **not** provide implementation or fields

# 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
  - Any client code designed to use the interface type can automatically use the class!

# Example

## »» Charges Demo



# Package Tracking

I don't even want this package. Why did I sign up for the stinging insect of the month club anyway?

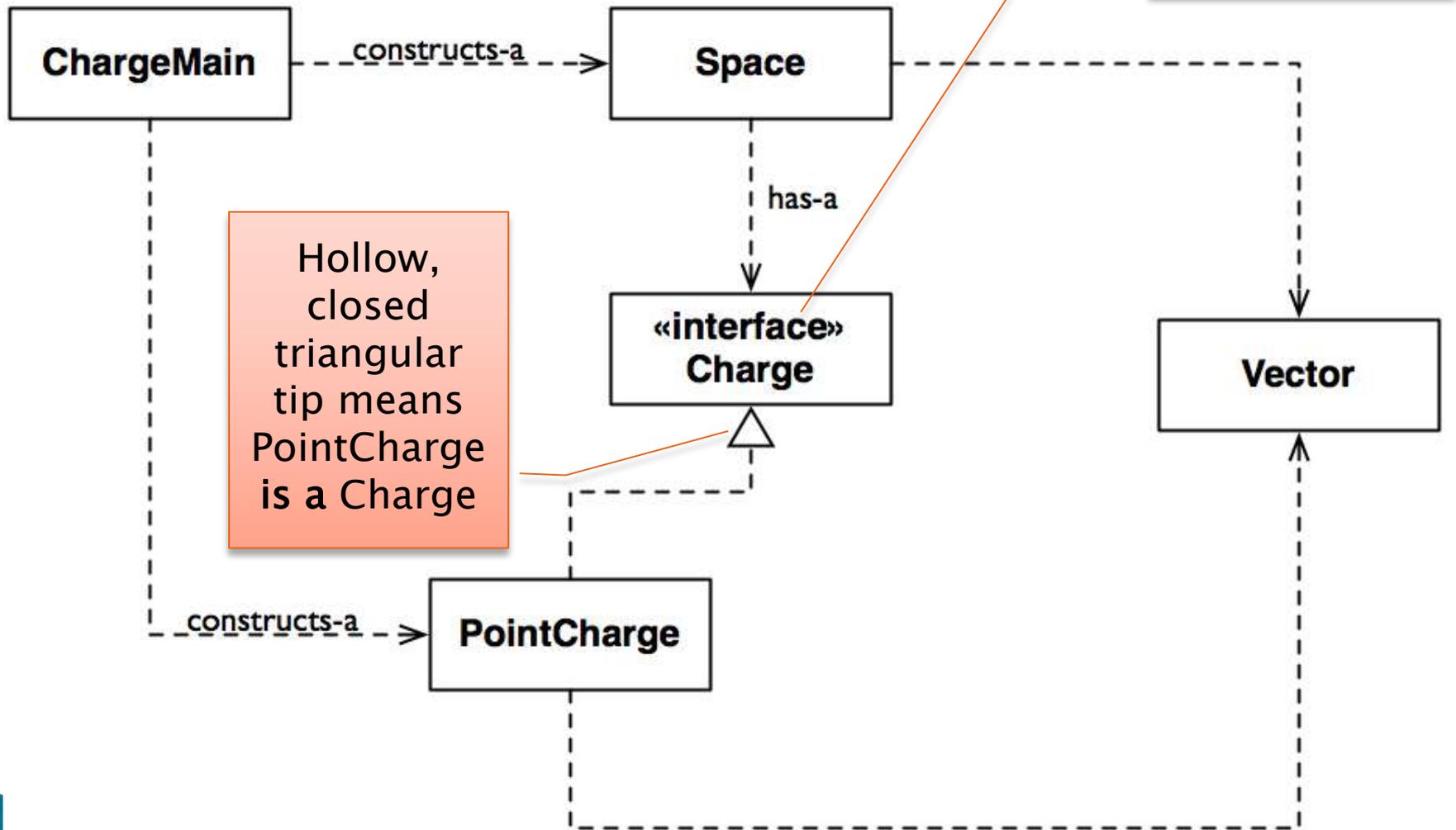
## ONLINE PACKAGE TRACKING:

PROs:  
CONVENIENT  
USEFUL

CONS:  
MAKES YOU  
CRAZY



# Charges UML



# Notation: In Code

interface, not class

```
public interface Charge {  
    /**  
     * regular javadocs here  
     */  
    Vector forceAt(int x, int y);  
    /**  
     * regular javadocs here  
     */  
    void drawOn(Graphics2D g);  
}
```

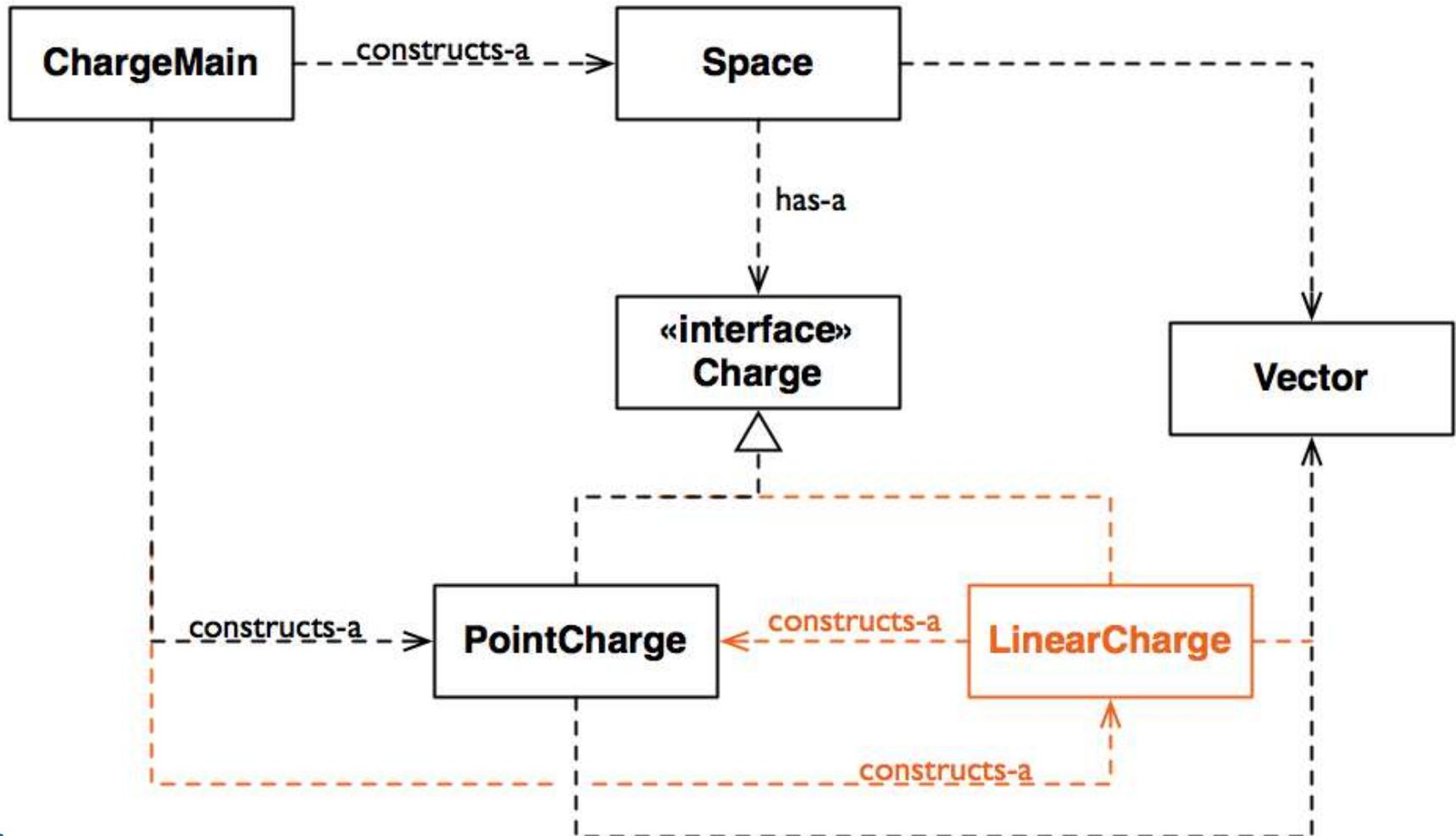
No "public",  
automatically  
are so

No method  
body, just a  
semi-colon

```
public class PointCharge implements Charge {  
}
```

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

# Updated Charges UML



Interfaces reduce coupling!

# 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 **LinearCharges** to **Space**'s **addCharge (Charge c)** method without changing **Space**!
- ▶ Use **interface types** for field, method parameter, and return types whenever possible

# 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**