CSSE 220 Day 14

Details on class implementation, Interfaces and Polymorphism

Check out *OnToInterfaces* from SVN

Questions?

Side Effects

- Side effect: any modification of data

- Method side effect: any modification of data *visible* outside the method
  - Mutator methods: side effect on implicit parameter
  - Can also have side effects on other parameters:

```
public void transfer(double amt, Account other)
{
    this.balance -= amt;
    other.balance += amt;
}
```

Avoid this if you can! Document it if you can't

Documenting Side Effects

```
/**
 * Transfers the given amount from this
 * account to the other account. Mutates
 * this account and other.
 *
 * @param amt
 *             amount to be transferred
 * @param other
 *             receiving account (mutated)
 *
 */
public void transfer(double amt, Account other) {
      this.balance -= amt;
      other.balance += amt;
}
```

Today: A Very Full Schedule

- Static fields and methods
- Variable scope
- Packages
- Interfaces and polymorphism

Call by Value

```
public static void main(String[] args) {
	double x= 1.0;
	double y = 2.5;
	swapOrNot(x,y);
	System.out.println("x is " + x);
}

private static void swapOrNot(double a, double b) {
	double temp = a;
	a = b;
	b = temp;
}
```

Draw a box-and-pointer diagram and predict the output.

Q1

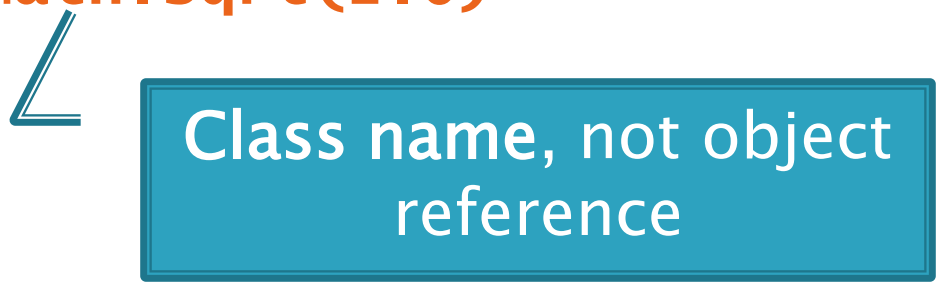
What is `static` Anyway?

- `static` members (fields and methods)…
  - are **not** part of objects
  - are **part of the class itself**

- Mnemonic: objects can be passed around, but static members stay put

Static Methods

- Cannot refer to `this`
  - They aren't in an object, so there is no `this`!
- Are called without an implicit parameter
  - `Math.sqrt(2.0)`



When to Declare Static Methods

- Helper methods that don't refer to this
  - Example: creating list of **`Coordinates`** for glider
- Utility methods
  - Example:
    - ```
      public class Geometry3D {
       public static double sphereVolume(double radius) {
              …
       }
      }
      ```
- **`main()`** method
  - Why static? What objects exist when program starts?

Q2
      ```

Static Fields

- We've seen static final fields

- Can also have static fields that aren't final
  - Should be private
  - Used for information shared between instances of a class

Two Ways to Initialize

- private static int nextAccountNumber = 100;

- or use “static initializer” blocks:

```
public class Hogwarts {
        private static ArrayList<String> FOUNDERS;

        static {
                FOUNDERS = new ArrayList<String>();
                FOUNDERS.add("Godric Gryfindor");
                // ...
        }

        // …
}
```

Exercise

- Polygon

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);
      }
      ```

Q4
      ```

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` class members can be accessed outside the class using "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.

Q6

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

Packages

- Let us group related classes
- We've been using them:
  - `javax.swing`
  - `java.awt`
  - `java.lang`
- Can (and should) group our own code into packages
  - Eclipse makes it easy…

Q7

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.

Package Tracking

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

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

Notation: In Code

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

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

public class PointCharge implements Charge {
}
```

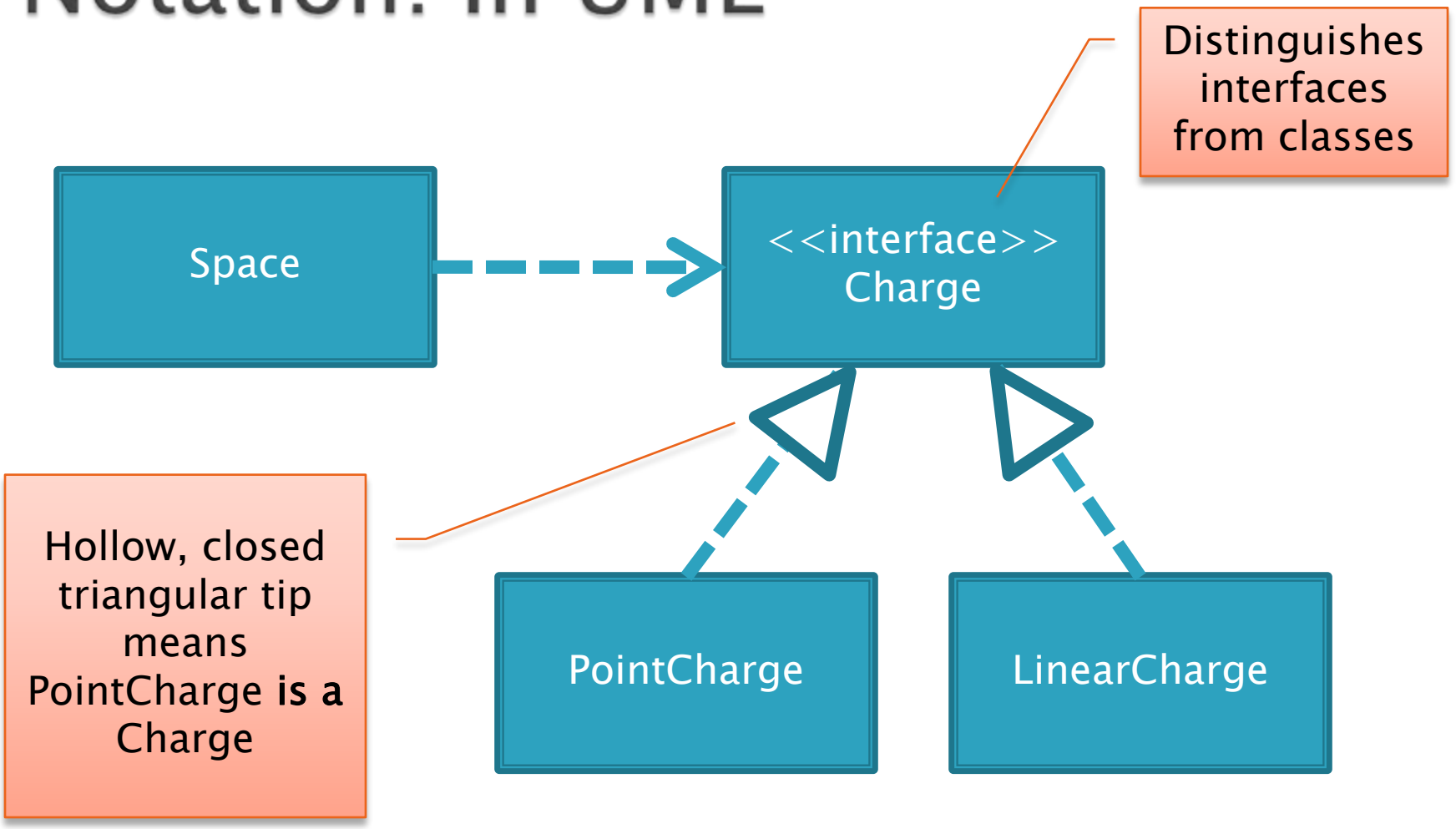
interface, not class

No "public", automatically are so

No method body, just a semi-colon

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

Notation: In UML



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 could pass `LinearCharge`s to `Space`'s `add(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**

Q13,14