CSSE 220 Day 15

Details on class implementation, Interfaces and Polymorphism

Check out *OnToInterfaces* from SVN

Questions?

Today: A Very Full Schedule

- Scope
  - Variables, fields and methods, class names
- Packages
- Interfaces and polymorphism

Scope – for parameters and local variables

- *Scope* : the region of a program in which a name 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);
      }
      ```

Q1

Scope – for fields and methods (*members* of a class)

- *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`
    Static
  - `list.size()`
    `p.x`
    Instance

Where *list* is an ArrayList and *p* is a Point

Q2

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

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;`

  Can then refer to just

  ```
  PI
  cos
  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…

Q4

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

Q5

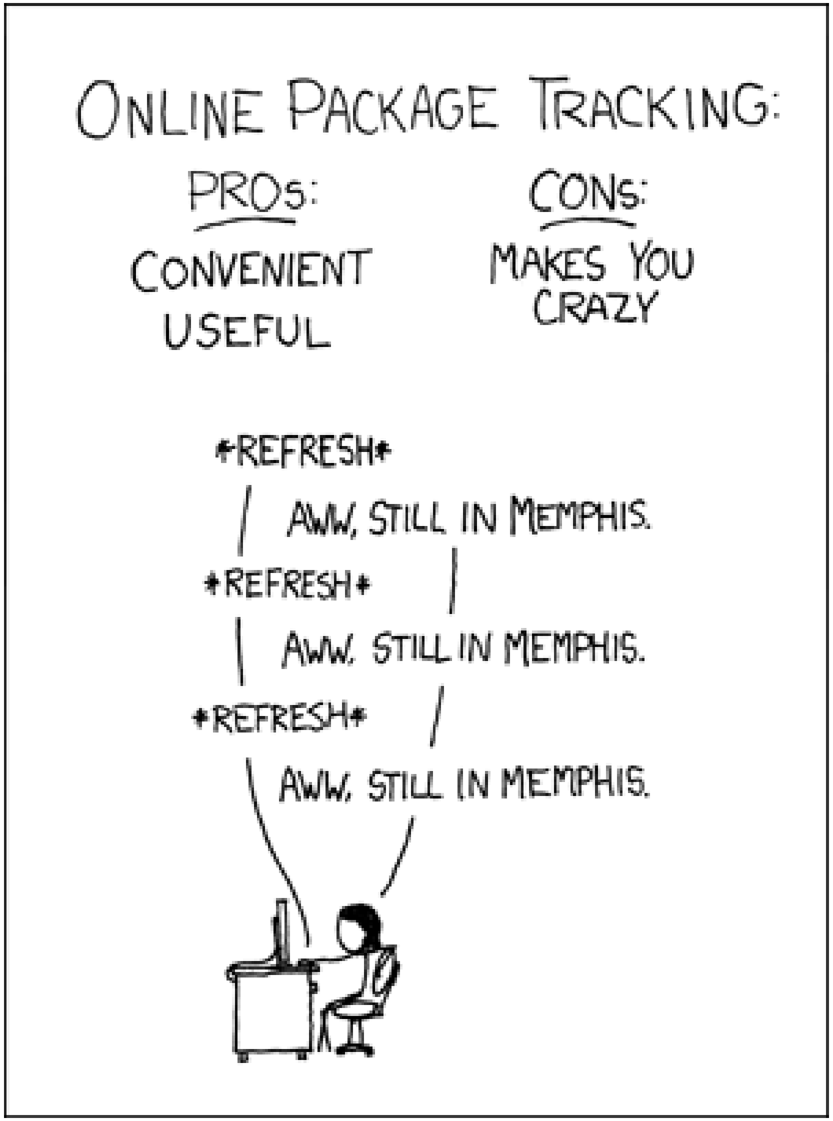
Qualified Names and Imports

- 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: Key Idea

- Interface types are like **contracts**
  - A class X can promise to **implement** an interface Y
    - That is, X will implement *every method specified* in the interface Y
  - Consider code C that has variables declared to be type Y
    - That is, it has interface type variables
    - Such code is called a Client of the interface Y
  - Code C can automatically call the methods of class X that are specified by interface Y!
    - Because C “knows” (from X implementing Y) that X will have the methods specified in Y

Example

- Suppose you are writing a sorting method. You could write:
  - `public void sort(int[] array) ...`
  - `public void sort(Double[] array) ...`
  - `public void sort(BigInteger[] array) ...`
  - `etc`
- Can you think of a better approach?
- Write a *single* sort method
  - `public void sort(Comparable<T> array) ...`
- where Comparable<T> specifies the comparison method `compareTo` to use

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
- Example:
  - Suppose you want to write a sort method.
  - If you just sort integers, why is your code not very reusable?

Q6

Notation: In Code

interface, not class

Type parameter – Comparable to type T objects

```
public interface Comparable<T> {
    /**
     * Compares this object with the specified
     * object for order. Returns a negative integer,
     * zero, or a positive integer as this object is
     * less than, equal to, or greater than the
     * specified object.
     */
     int compareTo(T object);
}
```

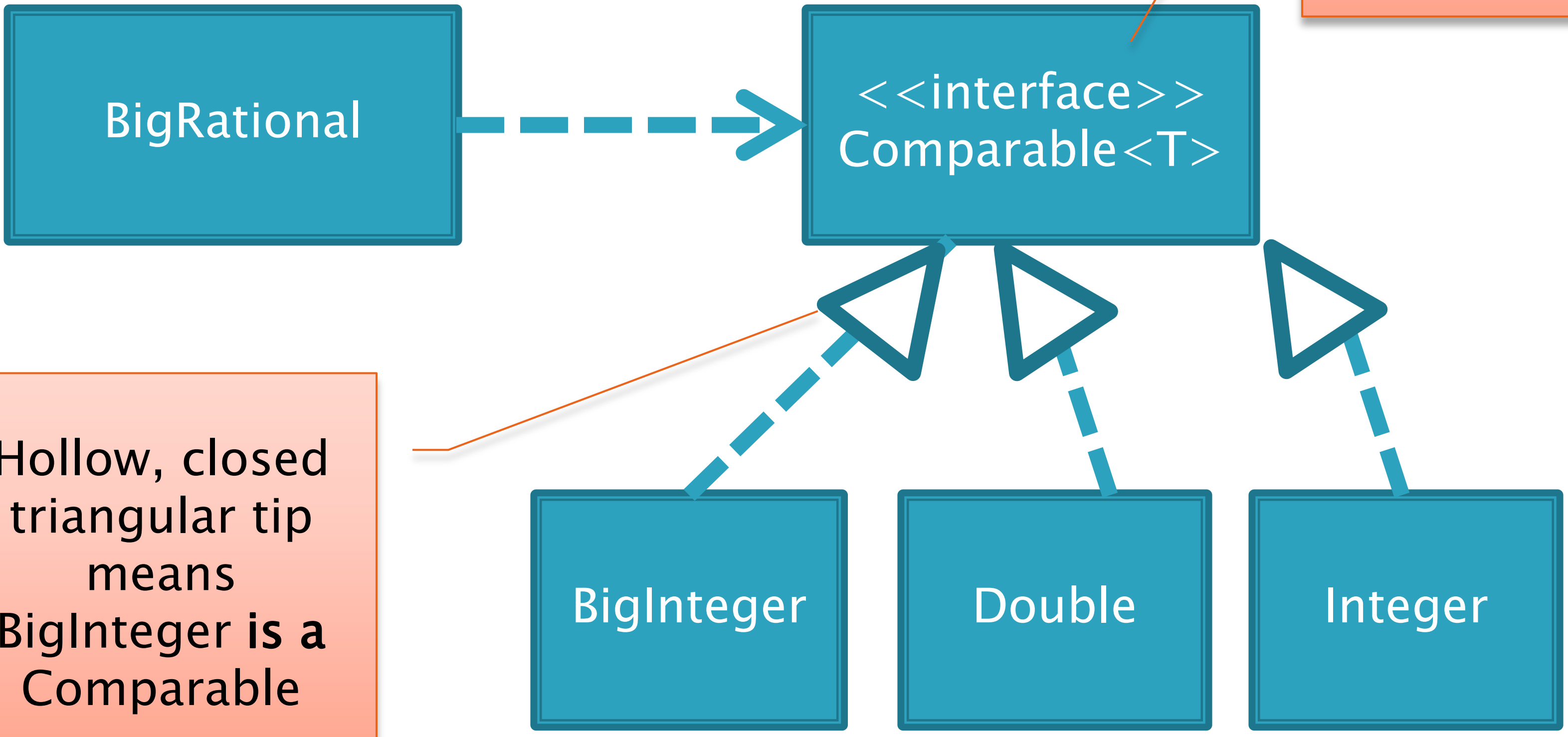
No "public", automatically are so

No method body, just a semi-colon

```
public class BigInteger implements Comparable<BigInteger> {
        …
}
```

BigInteger promises to implement all the methods declared in the **Comparable** interface

Notation: In UML



Distinguishes interfaces from classes

Hollow, closed triangular tip means BigInteger is a Comparable

Q7

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 **`Comparable`**s to **`BigRational`**'s **`compareTo(BigRational other)`** method without changing **`BigRational`**!

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

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**

Q9,10