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?

Reminder: 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

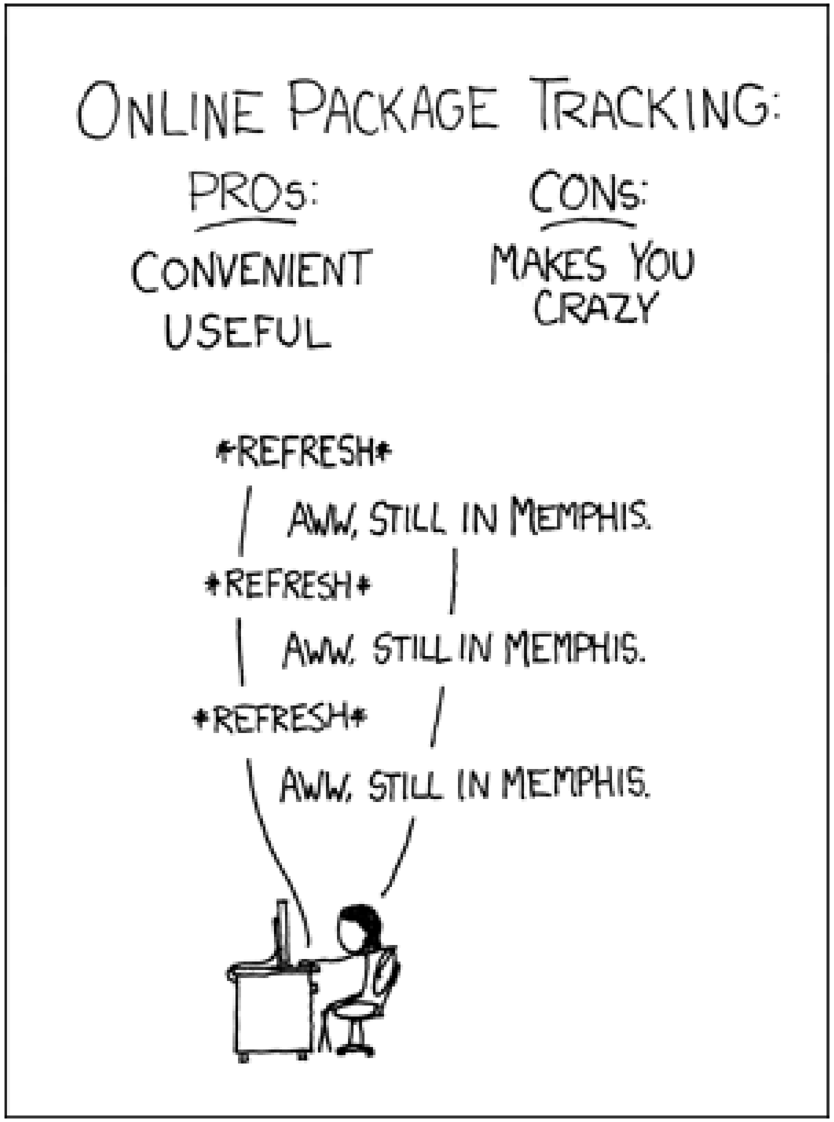
Qualified Names and Imports

- 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?

Interfaces for Algorithm Reuse

- Motivation: say I write a sort method for Students, which compares them by student ID. Relies on the fact that students can be compared with each other.

- What if I want to sort BankAccounts by balance instead?

Interfaces

- Specify a *contract* to implement every method in the interface
- Some code (called *client* of the interface) can use variables that implement the interface.
- Other code can implement the interface
- This clean separation allows the code that implements the interface to be changed without changing the client code at all!

- Why might I want to re-use the client code?

Q6

Notation: In Code

```
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);
}
```

interface, not class

Type parameter – Comparable to type T objects

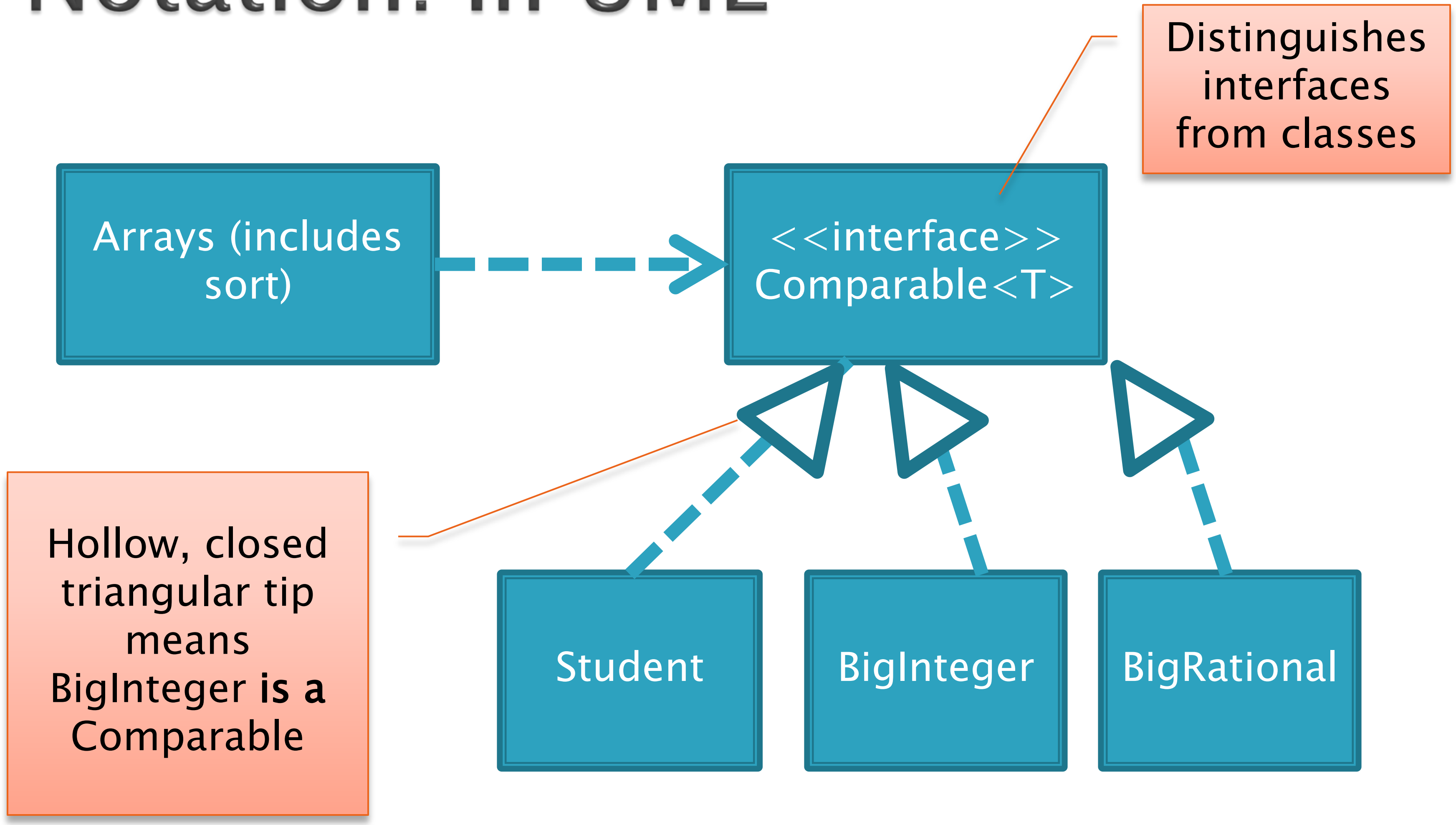
No "public", automatically are so

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
Arrays (includes sort)
<<interface>> Comparable<T>
Hollow, closed triangular tip means BigInteger is a Comparable
Student
BigInteger
BigRational

Q7

Why is this OK?

-
  ```
  Comparable c = new Student(…);
  if (c.compareTo(other) < 0) { … }
  c = new BigInteger(…);
  if (c.compareTo(other) < 0) { … }
  ```
- The type of the **actual object** determines the method used.

Q8a–c

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**

BigRational example

- Tonight's homework
- Our unit tests are a Client to Arithmetic objects and Comparable objects.
- You will write a BigRational class that implements each interface.
- Let's look at the starting code…