Inheritance and Polymorphism

CSSE 221

Fundamentals of Software Development Honors

Rose-Hulman Institute of Technology

ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Announcements

- Capsules:
  - Summary, quiz, and key each in a separate document
  - Quiz has place for students' names, questions are numbered
  - Quiz: max of 1 side
  - Key is marked as such
- Look for email about my BigRational unit tests
- Questions?

This week: BallWorlds assignment

- Last class:
  - Intro to UML as a communication tool
  - Writing methods you don't call
  - Using *this*
- Today:
  - **Inheritance**
  - **Polymorphism**
- Friday:
  - Introducing next week's assignment
  - Arrays and ArrayLists
  - (Using the debugger)

Inheritance

- Some slides inspired by Fall 2006-2007 CSSE221 students:
  - Michael Auchter
  - Michael Boland
  - Andrew Hettlinger

Inheritance

- Objects are unique
- But they often share similar behavior!

Software Engineer

Student

Professor

Drummer

Guitarist

Chemical Engineer

Physicist

Why not just copy-and-paste?

- Say I have an **Employee** class and want to create an **HourlyEmployee** class that adds info about wages. Why not copy-and-paste, then modify?

The Basics of Inheritance

- Inheritance allows you to reuse methods that you've already written to create more specialized versions of a class.
- Syntax:

```
public class HourlyEmployee extends Employee
```

Subclass

Superclass

HourlyEmployee IS-A Employee

Your turn

- Question: What is the relationship between a parrot and a bird?

Your turn

- What is the relationship between a parrot and a bird?
  - Every parrot is a bird, but not every bird is a parrot.
  - So if you had a Java class for each, which class would extend which?

Some Key Ideas in Inheritance

- Code reuse
- Overriding methods
- Protected visibility
- The “super” keyword

Code re-use

- The subclass inherits all the public and protected methods and fields of the superclass.
  - Constructors are not inherited
  - Constructors can be invoked by the subclass
- Subclass can add new methods and fields.

Overriding Methods

- DudThatMoves extends Dud
- DudThatMoves will define an **act()** method with the same signature that **overrides** Dud's method

What do you think happens if our child class doesn't override a method in the superclass?

It's exactly the same as in the superclass!

Visibility Modifiers

- **Public** – Accessible by any other class in any package.
- **Private** – Accessible only within the class; for fields.
- **Protected** – Accessible only by classes within the same package **and** any subclasses in other packages.
  - We won't use protected fields, but use private with protected accessors.
  - Private fields are *encapsulated*
- Default (No Modifier) – Accessible by classes in the same package but not by classes in other packages.
  - Use sparingly!

The “super” Keyword

- It’s like the word “this,” only “super”:
- Two uses:
  - To call a superclass' method, use **super.methodName(…)**
  - To call a superclass' constructor, use **super(some parameter)** from the child class’ constructor
- Don't use **super** for fields (they're private anyway).

The "super" Keyword

- Methods can call super.*methodName*(...)
  - To do the work of the parent class method, plus...
  - Additional work for the child class

```
public class Workaholic extends Worker {
    public void doWork() {
        super.doWork();
        drinkCoffee();
        super.doWork();
    }


}
```

The "super" Keyword

- Methods can call super.*methodName*(...)
  - To do the work of the parent class method, **plus**...
  - **Additional** work for the child class

```
public class Workaholic extends Worker {
    // If a Workaholic just worked
    // like a worker, it would inherit doWork
    // NEVER write code like this:
    public void doWork() {
        super.doWork();
    }

}
```

The “super” Keyword

- A common experience?

```
public class RoseStudent extends Worker {
    public void doWork() {
        while (!isCollapsed) {
            super.doWork();
            drinkCoffee();
        }
        super.doWork();
    }
}
```

Rules of using *super* in constructors

- A **super(...)** call ***must be the first*** line of the code of a class's constructor if it is to be used.

The this Keyword

1. **this.someField** and **this.someMethod()**: nice style
2. **this** alone is used to represent the whole object: **environment.addBall(this)**

The *this* Keyword

3. **this** calls another constructor

   this must be the first thing called in a constructor.

   Therefore, **super(…)** and **this(…)** cannot be used in the same constructor.

```
public class Foo {
  private String message;
  public Foo(){
    this("This is sad.");
  }
  public Foo(String s){
    this.message = s;
  }
}
```

Overriding vs. Overloading

- Recall: **overriding** a method is when a subclass has method with the same signature (name and parameter list) as its superclass
  - Mover's act() and Bouncer's act()

- **Overloading** a method is when two methods have the same name, but **different parameter lists**

  Arrays.sort(array) and Arrays.sort(array, new ReverseSort())

More notes

- **Every** object in Java extends java.lang.Object
  - Don't have to say it explicitly
  - This is why every class has a basic toString() and a basic clone() method.

- *Abstract classes* contain *abstract (*unimplemented) methods.
  - Abstract classes can't be instantiated, just extended

Final notes

- What does it mean to be declared **final**?
  - **Final fields** can't be assigned a new value
  - **Final methods** cannot be overridden
  - **Final classes** cannot be extended
- There is only single inheritance in Java

Next

- Finish the inheritance quiz
- Do the Inheritance Demo linked from the Schedule page
- Take a break

Polymorphism

- Polymorphism allows a reference to a superclass or interface to be used instead of a reference to its subclass

```
// Rectangle and Circle could implement or extend Shape
Shape rect = new Rectangle();
Shape circle = new Circle();

void printArea(Shape shape) {
        System.out.println(shape.getArea());
}
```

Polymorphism

```
double totalArea(ArrayList<Shape> shapes) {
        double totalArea = 0;
        for (Shape s : shapes) {
        totalArea += s.getArea();
        }
        return totalArea;
}
```

Example

- In the bird and parrot example, consider a bird method:

```
static void printCall(Bird bird) {
    System.out.println(bird.call);
}
```

```
Bird b = new Parrot();
printBirdCall(b);
Parrot p = new Parrot();
printBirdCall(p);
```

- Generic: printBirdCall expects a Bird, but any type of bird is OK.
- **Cannot** write Parrot p = new Bird(); -there's not enough info!
- However, without casting, b can only use bird methods; parrot-specific information can't be accessed!

Casting and `instanceof`

- If we know that b is a Parrot, we can cast it and use Parrot methods:

  ((Parrot)b).speak()

- At runtime, if b is just a Bird, the JVM will throw a ClassCastException.

- To test this, use **instanceof**:

```
if (b instanceof Parrot) { ((Parrot)b).speak()) }
```

Late Binding: The Power of Polymorphism

```
HourlyEmployee h = new HourlyEmployee("Wilma Worker", new
   Date("October", 16, 2005), 12.50, 170);

SalariedEmployee s = new SalariedEmployee("Mark Manager",
   new Date("June", 4, 2006), 40000);

Employee e = null;
if (getWeekDay().equals("Saturday")
   e = h;
else
   e = s;
System.out.println(e);
```

Is e's actual type (and thus which toString() to use) known at compile-time or run-time?

Wrap-up

- Finish the quiz and turn it in
- Finish the demo: this part is much shorter