Outline

- Course Syllabus
- Our First Lab
  - Post-Mortem and Motivation
- Inheritance
  - Mechanism
  - Protected Access
    - super(), this()
- Conclusion
A New Access Modifier: protected

- **Superclass:**
  ```java
  public class Animal {
    private String name;
    private String quote;

    //methods omitted
  }
  ```

- **Here, Bird will not compile (why?)**
  ```java
  public class Bird extends Animal {
    private int eggs;

    public void layEgg() {
      System.out.println(this.name + ": Squawk squawk *plop*");
      this.eggs++;
    }

    //other methods omitted
  }
  ```

  But `name` is private to `Animal`, and is inaccessible from other classes!
protected Access (Cont.)

- **One solution:** Make sure there’s a getter method for the field in the superclass, and call it from subclass.

```java
public class Animal {
    private String name;
    private String quote;

    public String getName() {
        return this.name;
    }
}
```

```java
public class Bird extends Animal {
    private int eggs;

    public void layEgg() {
        System.out.println(this.getName() + ": Squawk squawk *plop*");
        this.eggs++;
    }

    // other methods omitted
}
```

Can be called directly since `getName()` was inherited. *i.e.*, it also belongs in `Bird`. 
Another solution: Use the `protected` visibility modifier.

- Grants access to all subclasses (regardless of depth).

```java
public class Animal {
    protected String name;
    private String quote;

    //methods omitted
}
```

```java
public class Bird extends Animal {
    public void layEgg() {
        System.out.println(this.name + ": Squawk squawk *plop*" );
        this.eggs++;
    }

    //other methods omitted
}
```

(Are subclasses' fields accessible to superclasses? No inheritance is a 1-way street)
Protected Access in Class Diagrams

- denotes private access
+ denotes public access
# denotes protected access

Dotted edge denotes the "uses" or "has-a" relation
Solid edge (always points to superclass) denotes the “extends” or “is-a” relation
Outline

- Course Syllabus
- Our First Lab
  - Post-Mortem and Motivation
- Inheritance
  - Mechanism
  - Protected Access
    - super(), this()
- Conclusion
Step 4: Writing constructors

- Constructors initialize fields when objects are created
  - We’ve done it for Animal class already
- Let's re-write the constructors for subclasses:
  - How to initialize name and quote?
    - (They're still private in Animal)
Let's say Animal's constructors are defined as follows:

```java
public class Animal {
    private String name;
    private String quote;

    public Animal() {
        this.name = "Quincy"; // a default animal has no name
        this.quote = "Grunt"; // a default animal makes this noise
    }

    public Animal(String name, String quote) {
        this.name = name;
        this.quote = quote;
    }
}
```

What's a default cow's name and quote?

```java
public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        this.gallonsMilked = 0;
    }
}
```

Cow c = new Cow();
c.speak(); // what's the output?
Let's say Animal's constructors are defined as follows:

```java
public class Animal {
    private String name;
    private String quote;

    public Animal() {
        this.name = "Quincy";  // a default animal has no name
        this.quote = "Grunt";  // a default animal makes this noise
    }

    public Animal(String name, String quote) {
        this.name = name;
        this.quote = quote;
    }
}
```

What's a default cow's name and quote?

```java
public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        this.gallonsMilked = 0;
    }
}
```

A call to the constructor of the `superclass` is automatically inserted by Java!
Explicitly Calling the Superclass' Constructor

- To call the constructor of the immediate superclass, we use:
  - Syntax: `super(parameters);`
  - Caveat: Must be the first statement in subclass' constructor
  - Important! `super()` automatically inserted by Java if no explicit `super()` call is made

```java
public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        super();
        this.gallonsMilked = 0;
    }
}
```

```java
public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        super();
        this.gallonsMilked = 0;
    }
}
```

Same effect
But... Our Default Cows are named “Bessies”

- We don't actually want this to happen:

```java
Cow c = new Cow();
c.speak();
> Quincy: Grunt
```

- Animals are *too* general. Cows are *specialized* Animals, and a default Cow is named Bessie and doesn't “grunt”:

  - What we want:

```java
Cow c = new Cow();  // construct default cow
c.speak();
> Bessie: Moo
```

```java
Cow c2 = new Cow("Daisy", "Mooo000000000!");  // construct a customized cow
c2.speak();
> Daisy: Mooo00000000!
```
public class Animal {
    private String name;
    private String quote;

    public Animal() {
        this.name = "Quincy";  // a default animal has no name
        this.quote = "Grunt";  // a default animal makes this noise
    }

    public Animal(String name, String quote) {
        this.name = name;
        this.quote = quote;
    }
}

public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        super("Bessie", "Moo!");  // call the overloaded constructor in the superclass
        this.gallonsMilked = 0;
    }

    public Cow(String name, String quote) {
        super(name, quote);
        this.gallonsMilked = 0;
    }
}
public class Animal {
    private String name;
    private String quote;

    public Animal() {
        this.name = "Quincy";  // a default animal has no name
        this.quote = "Grunt";  // a default animal makes this noise
    }

    public Animal(String name, String quote) {
        this.name = name;
        this.quote = quote;
    }
}

public class Cow extends Animal {
    private double gallonsMilked;

    public Cow() {
        this("Bessie", "Moo");  // call the overloaded constructor in *this* class
    }

    public Cow(String name, String quote) {
        super(name, quote);
        this.gallonsMilked = 0;
    }
}
Add a new type of Animal to the current class diagram:

- **Goats**:
  - Default goat is named “Billy,” says "Maaaa"
  - Like to chew cans (keep a count)
  - Their fur can be shorn (how many pounds produced?)
  - Goats can be milked (how many gallons milked?)
Proposed class diagram: Good idea? Bad idea? Why?

No, goats and cows both produce milk.

"Milking" elements are duplicated now...
Better Design

Animal
- String name
- String quote
+ display()
+ getName()
+ speak()

Cow
- double gallonsMilked
+ milk()

Bird
- int eggs
+ layEgg()

Goat
- int cansChewed
- double furShorn
+ chew()
+ shear()
+ numCansChewed()
+ furProduced()

Does take care of code duplication problem, but functionally…

- Goats are not specialized Cows…
Add a new type of Animal to the current class diagram:

- **Goats:**
  - Default goat is named “Billy,” says "Maaaa"
  - Like to chew cans (keep a count)
  - Their fur can be shorn (how many pounds produced?)
  - Goats can be milked (how many gallons milked?)

![Goat image]

(Do we even need a Cow class?)
Outline

- Course Syllabus
- Our First Lab
  - Post-Mortem and Motivation
- Inheritance
  - Mechanism
  - Protected Access
  - super(), this()
- Conclusion
Conclusion

- Inheritance allows for easier code maintenance

- Know these concepts:
  - Why Inheritance?
    - Motivating drivers
    - When is it appropriate?
    - How do you refactor for it?
  - "is-a" vs. "has-a" relationship
  - Superclass vs. subclass
  - Protected access
  - super() vs. this()
Administrivia 9/3

- **Reading:**
  - Chap 1 of K&W, or/and
  - Chap 10 of the BlueJ book if you still have it

- **Homework 1 is posted (GET STARTED - people struggle with it):**
  - Due Fri, 9/13
  - Do this one alone; can be completed based on CSCI 161 knowledge

- **Lab 1 solution posted**
  - BetterFarm code is posted (We’ll write it this week)
Announcement from CWLT CS Tutor Lia Chin-Purcell

Reminders:

• Homework 1 Due Fri, 9/13
• Notes from last lecture posted
  - (I’ll stop reminding you from now on)

Last time…

• Quick rundown on UML notation
• The Lots-of-Specialized-Classes Problem (e.g., Cow, Bird, Gator, …)
• David’s 4-Step Process to Refactoring for Abstraction
  - We got through step 3 for Cow