Objectives

- Inheritance
- Polymorphism
  - Dispatch

Inheritance

- Build new classes based on existing classes
  - Allows code reuse
- Start with a class (**parent** or **super class**)
- Create another class that extends or *specializes* the class
  - Called **the child**, **subclass** or **derived class**
  - Use `extends` keyword to make a subclass

Examples?

Child class

- Inherits all of parent class's methods and fields
  - Note on `private` fields: all are *inherited*, just can't *access*
- Can also ***override*** methods
  - Use the same name and parameters, but implementation is different
- Adds methods or fields for *additional functionality*
- Use `super` object to call parent's method
  - Even if child class redefines parent class's method

Inheritance Rules

- Constructors are ***not*** inherited
  - For example: we will have to define
    `Rooster( String name, int height, double weight )`
    even though similar constructor in `Chicken`

`Rooster` class

- Could write class from scratch, but …
- A rooster is a chicken
  - But it adds something to (or *specializes*) what a chicken is/does
- Classic mark of inheritance: **is a** relationship
- Rooster is child class
- Chicken is parent class

Access Modifiers

- `public`
  - Any class can access
- `private`
  - No other class can access (including child classes)
    - Must use parent class's public accessor/mutator methods
- `protected`
  - Child classes can access
  - Members of package can access
  - Other classes cannot access

Access Modes

Default (if none specified)

Accessible to	Member Visibility			
	`public`	`protected`	package	`private`
Defining class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

protected

- Accessible to subclasses and members of package
- Can't keep encapsulation "pure"
  - Don't want others to access fields directly
  - May break code if you change your implementation
- Assumption?
  - Someone extending your class with protected access knows what they are doing

Access Modifiers

- If you're uncertain which to use (protected, package, or private), use the *most restrictive*
  - Changing to less restrictive later → easy
  - Changing to more restrictive → may break code that uses your classes

Rooster class

`extends` means that Rooster is a child of Chicken

```
public class Rooster extends Chicken {
   public Rooster( String name,
      int height, double weight) {
      // all instance fields inherited
      // from super class
      this.name = name;
      this.height = height;
      this.weight = weight;
      is_female = false;
   }

   // new functionality
   public void crow() {… }
   …
```

By default calls *default* `super` constructor with no parameters

(not one of the examples posted online)

Rooster class

```
public class Rooster extends Chicken {
   public Rooster( String name,
      int height, double weight) {
```

Call to super constructor must be **first** line in constructor

```
      super(name, height, weight, false);
   }

   // new functionality
   public void crow() { … }

   …
}
```

Constructor Chaining

- Constructor ***automatically*** calls constructor of parent class if not done explicitly
  - `super();`
- What if parent class does not have a constructor with no parameters?
  - Compilation error
  - Forces child classes to call a constructor with parameters

Overriding and New Methods

```
public class Rooster extends Chicken {
        …

        // overrides superclass; greater gains
        @Override
        public void feed() {
                weight += .5;
                height += 2;
        }

        // new functionality
        public void crow() {
                System.out.println("Cocka-Doodle-Doo!");
        }

}
```

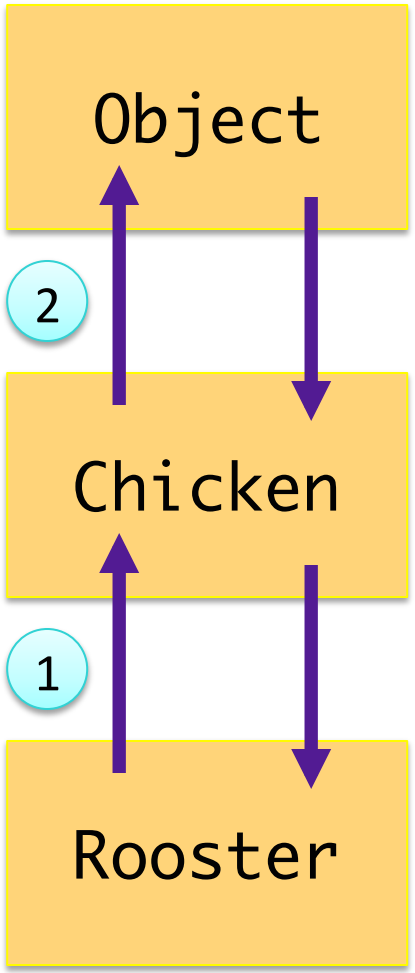
Same method signature as parent class

Specializes the class

Inheritance Tree

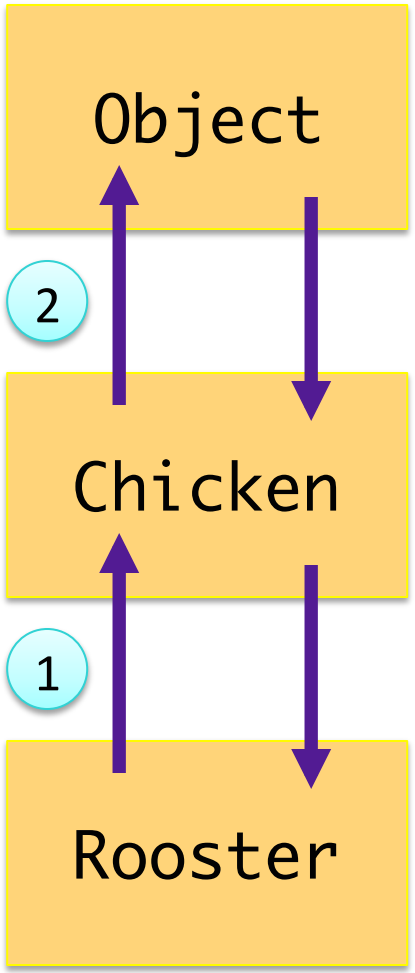
- java.lang.Object
  - Chicken
    - Rooster

- Call parent class's constructor first
  - Know you have fields of parent class before implementing constructor for your class



Inheritance Tree

- java.lang.Object
  - Chicken
    - Rooster



- No finalize() chaining
  - Should call super.finalize() inside of finalize method

Shadowing Parent Class Fields

- Child class has field with same name as parent class
  - You probably shouldn't be doing this!
  - But could happen
    - Example: more precision for a constant

```
field       // this class's field
this.field  // this class's field
super.field // super class's field
```

Multiple Inheritance

- In Python, it is possible for a class to inherit (or extend) more than one parent class
  - Child class has the fields from both parent classes
- This is NOT possible in Java.
  - A class may extend (or inherit from) **only one** class

POLYMORPHISM & DISPATCH

Polymorphism

- ***Polymorphism*** is the ability for an object to vary behavior based on its type
- You can use a child class object whenever the program expects an object of the parent class
- Object variables are ***polymorphic***
- A `Chicken` object variable can refer to an object of class `Chicken`, `Rooster`, `Hen`, or any class that *inherits from* `Chicken`

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

We can guess the actual types
But compiler can't

Compiler's Behavior

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

- We know `chickens[1]` is probably a `Rooster`, but to *compiler*, it's a `Chicken` so `chickens[1].crow();` will not compile

Compiler's Behavior

- When we refer to a `Rooster` object through a `Rooster` object variable, compiler sees it as a `Rooster` object
- If we refer to a `Rooster` object through a `Chicken` object variable, compiler sees it as a `Chicken` object.

→ Object variable determines how compiler sees object.

- We cannot assign a parent class object to a derived class object variable
  - Ex: `Rooster` is a `Chicken`, but a `Chicken` is not necessarily a `Rooster`

```
Rooster r = chicken;
```

Polymorphism

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

```
chickens[1].feed();
```

Compiles because `Chicken` has a `feed` method.

But, which `feed` method is called –
`Chicken's` or `Rooster's`?

Dynamic vs. Static Dispatch

- Dynamic dispatch is not necessarily a property of object-oriented programming in general
- Some OOP languages use **static dispatch**
  - Type of the object variable used to call the method determines which version gets run
- The primary difference is ***when*** **decision on which method to call is made**…
  - Static dispatch (C#) decides at compile time
  - Dynamic dispatch (Java, Python) decides at run time
- Dynamic dispatch is slow
  - In mid to late 90s, active research on how to decrease time

Inheritance Rules: Access Modifiers

Access modifiers in child classes

- Can make access to child class ***less*** restrictive but not more restrictive

- **Why?**
- What would happen if a method in the parent class is `public` but the child class's method is `private`?