Objectives

- Inheritance
- Polymorphism
  - Dispatch

Review

- What method should we implement to allow pretty printing of objects we define?
- What method should we implement for determining if two objects are equivalent?
- How does Java pass parameters?
- What does Java provide to prevent memory leaks?

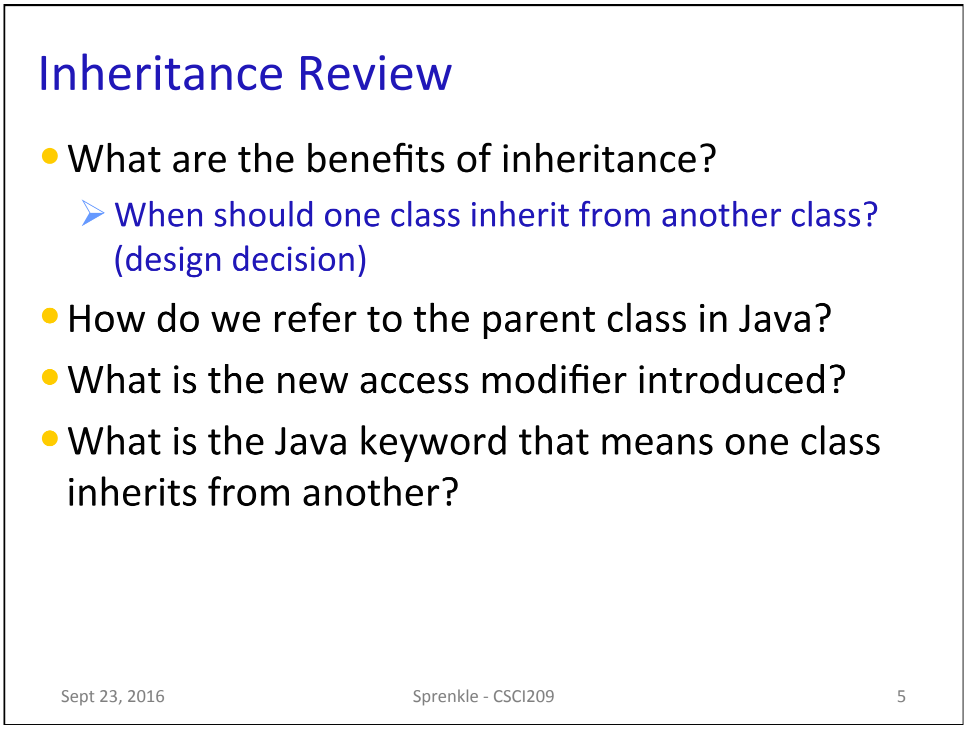
Assignment 4 Notes

- Document format for `toString` and how determines equivalence in `equals`

```
/**
 * Returns a string representation of the chicken.
 * Format:
 * Chicken name: <name>
 * weight: <weight>
 * height: <height>
 * female/male
 */
```

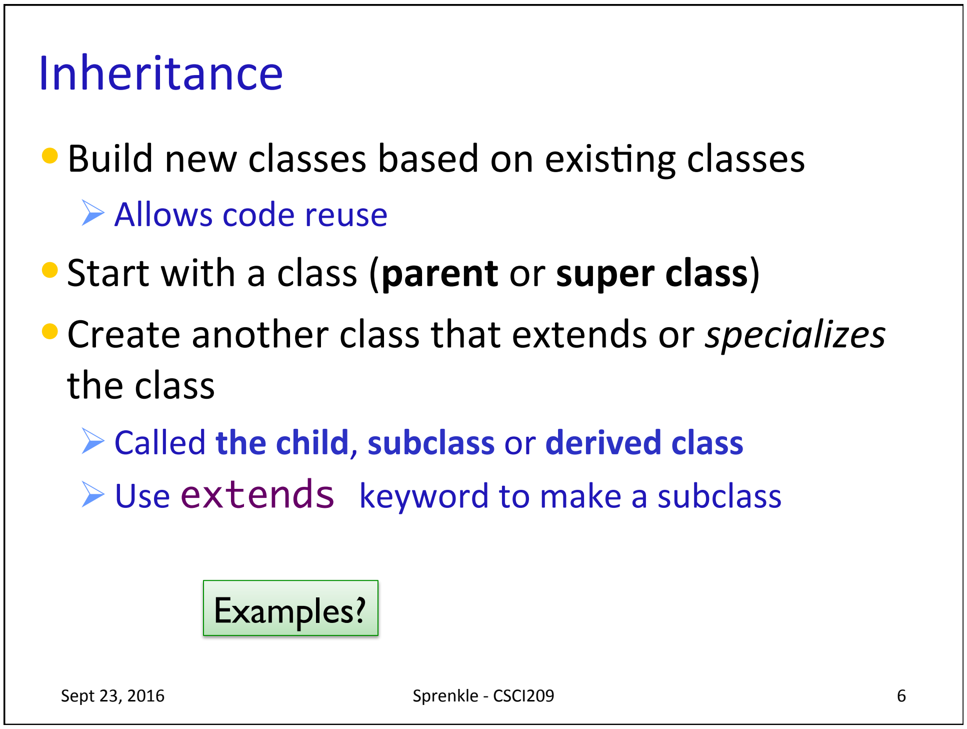
```
/**
 Determines if two Chickens are
equivalent, based on their name,
height, weight, and gender.
 */
```

INHERITANCE



Inheritance Review

- What are the benefits of inheritance?
  - When should one class inherit from another class? (design decision)
- How do we refer to the parent class in Java?
- What is the new access modifier introduced?
- What is the Java keyword that means one class inherits from another?



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

Modify Chicken Class

- What if we want instance variables to be accessible by child class
  - Can't be private*

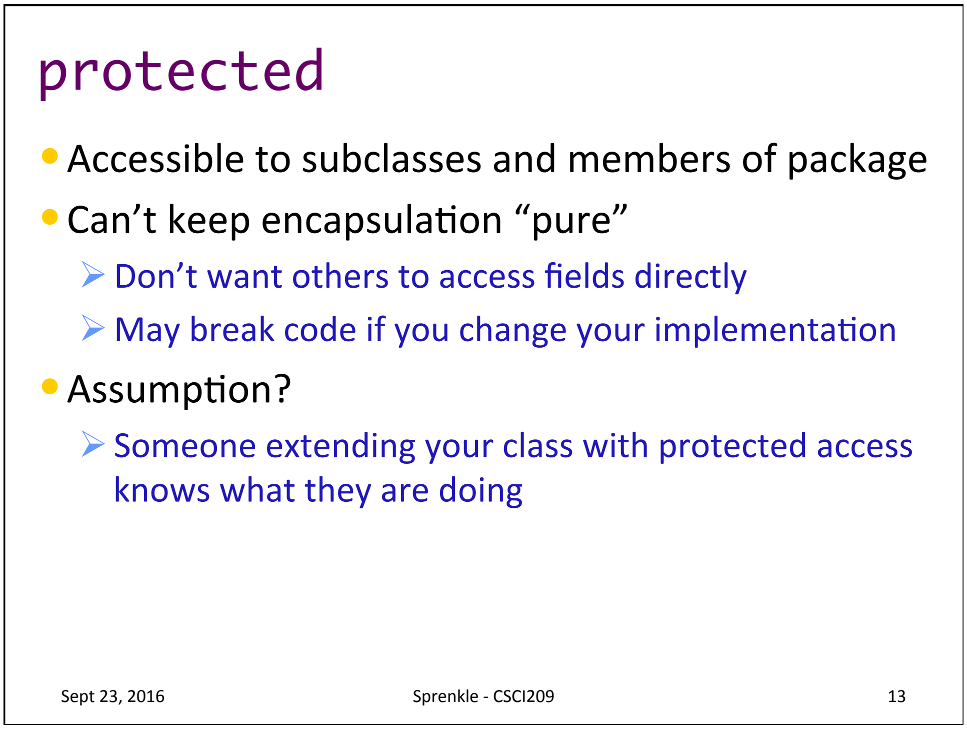
Access Modifiers

- `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

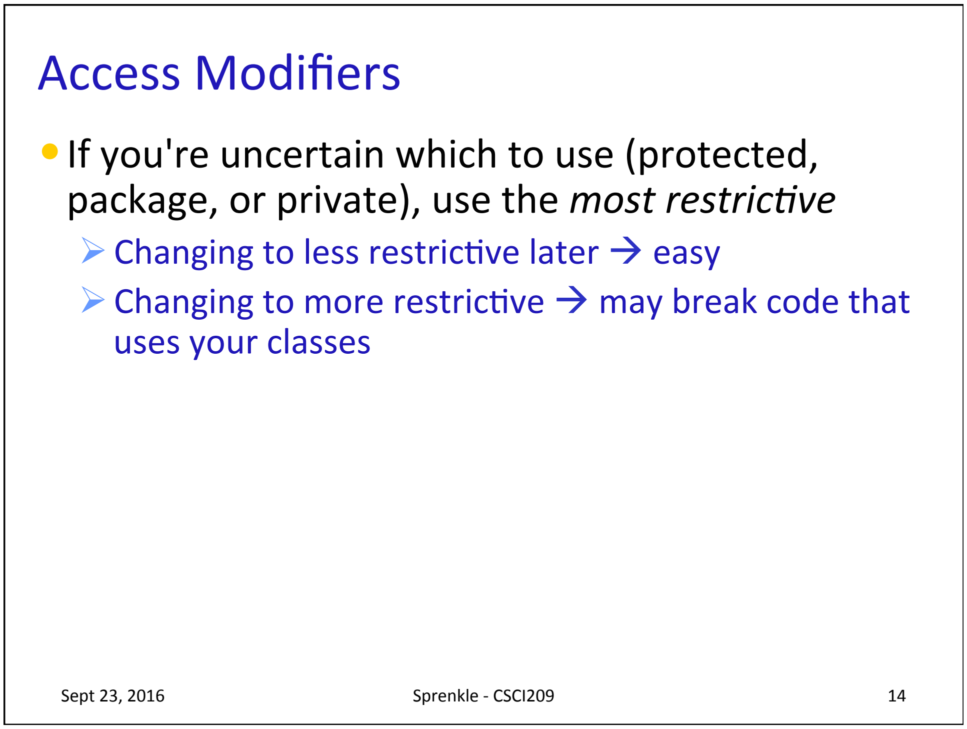
Default (if none specified)

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

Look at Modified Chicken Class

- Two examples:
  - one is extending the Chicken class, whose instance variables are private
  - one is extending the Chicken class, where the instance variables are protected.

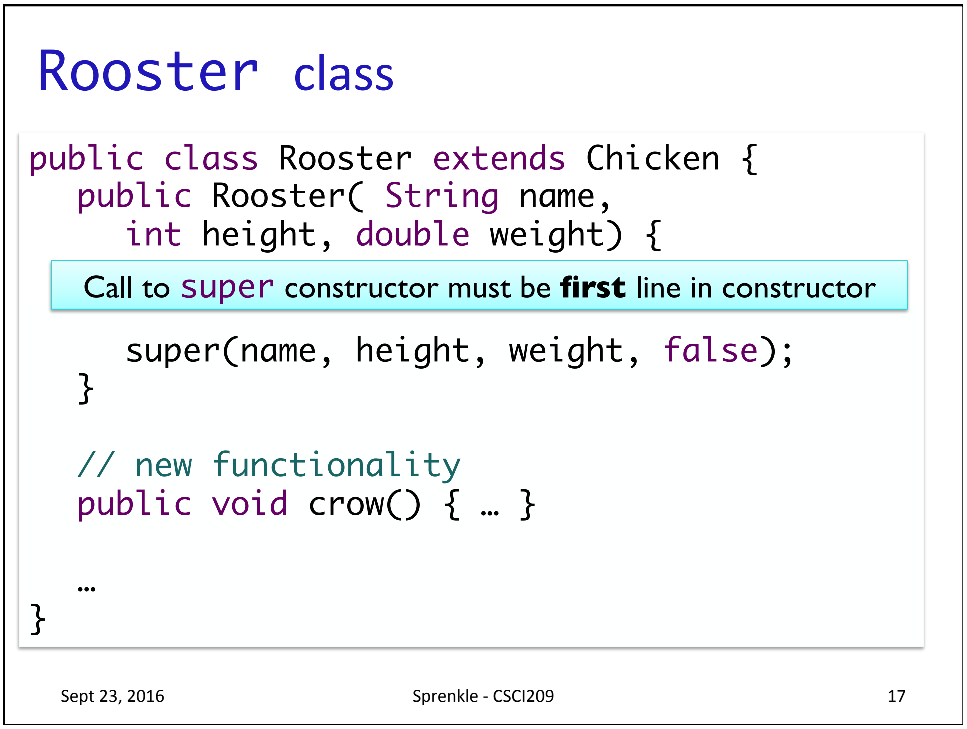
Rooster class

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

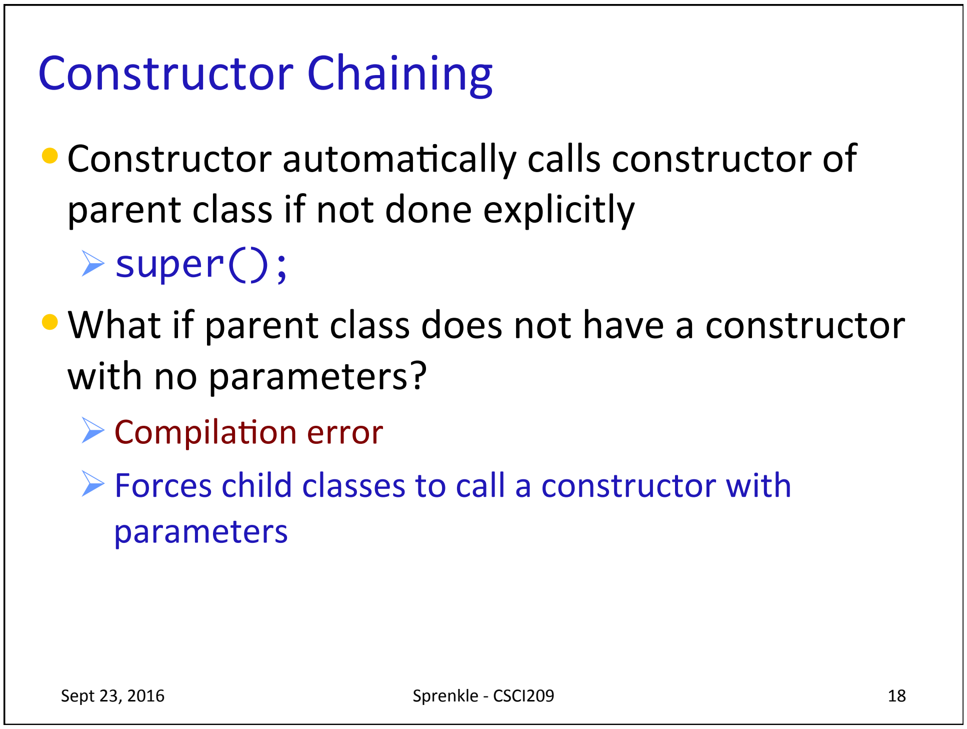
(not one of the examples posted online)



Rooster class

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

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

   …
}
```

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



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!");
      }

}
```

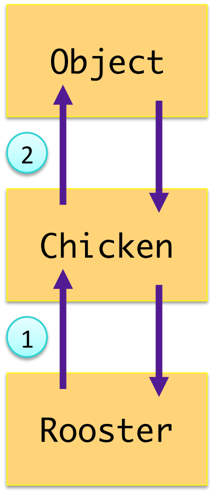
Same method signature as parent class

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



Object

2

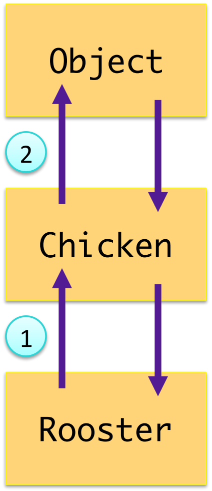
Chicken

1

Rooster

Inheritance Tree

- `java.lang.Object`
  - `Chicken`
    - `Rooster`



Object

2

Chicken

1

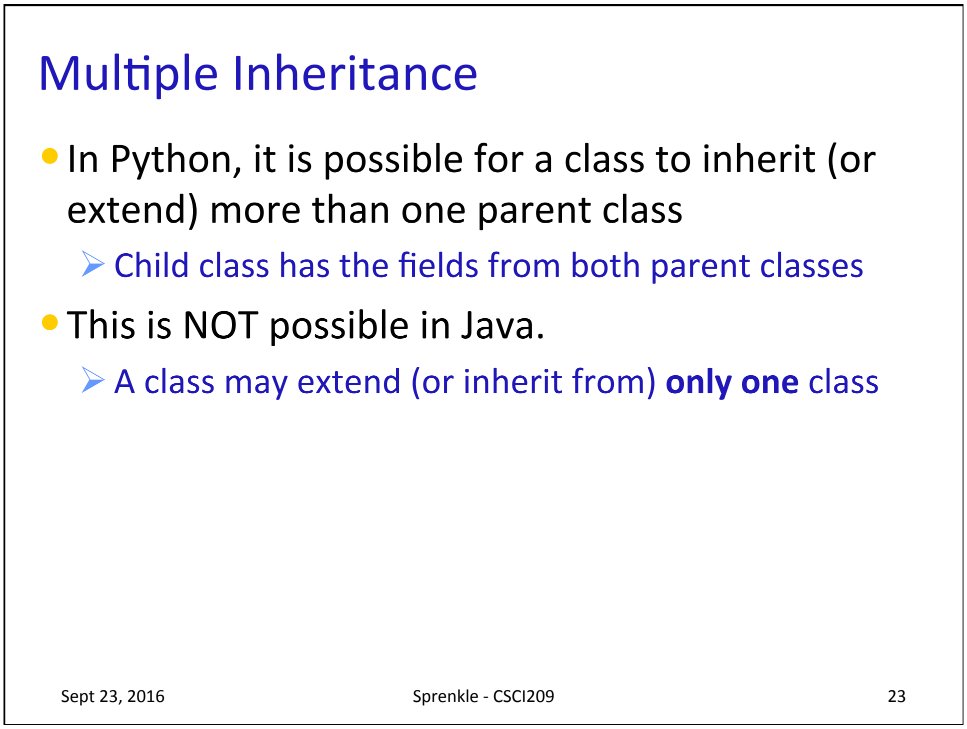
Rooster

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

Shadowing Parent Class Fields

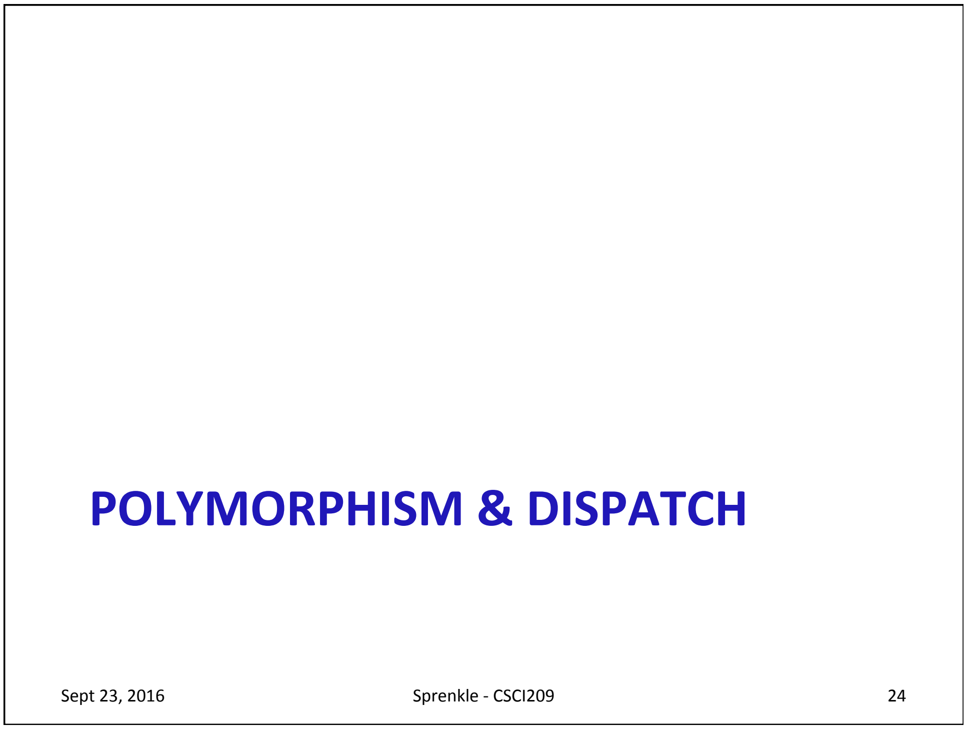
- Child class has field with same name as parent class
  - **You probably shouldn't be doing this!**
  - But could happen
    - Example: more precision for a constant

```
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`

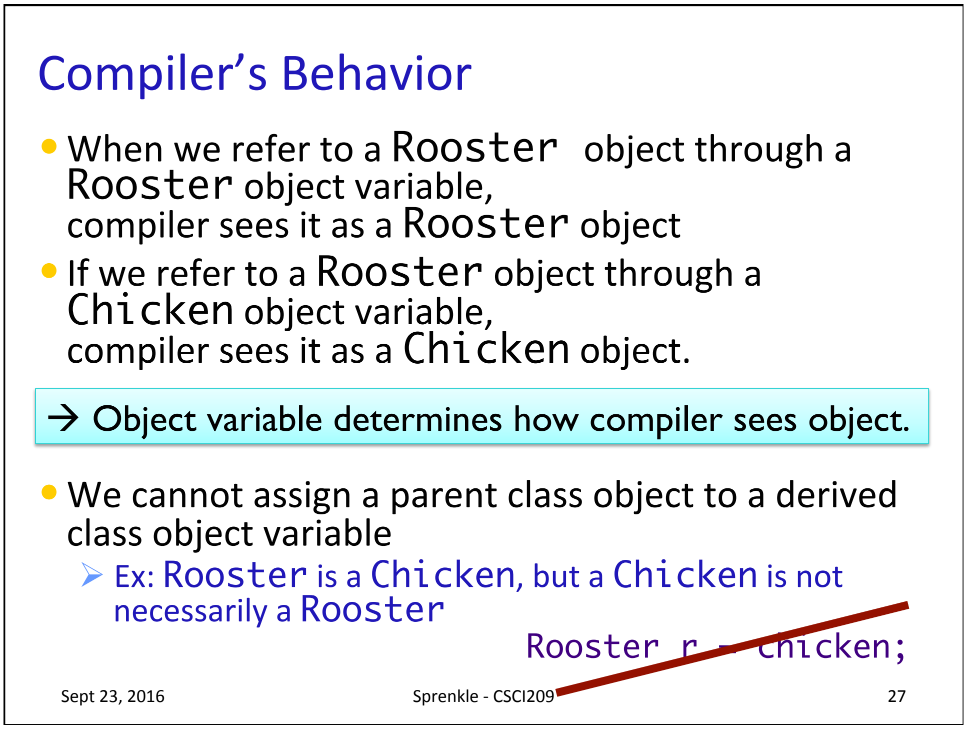
```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

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



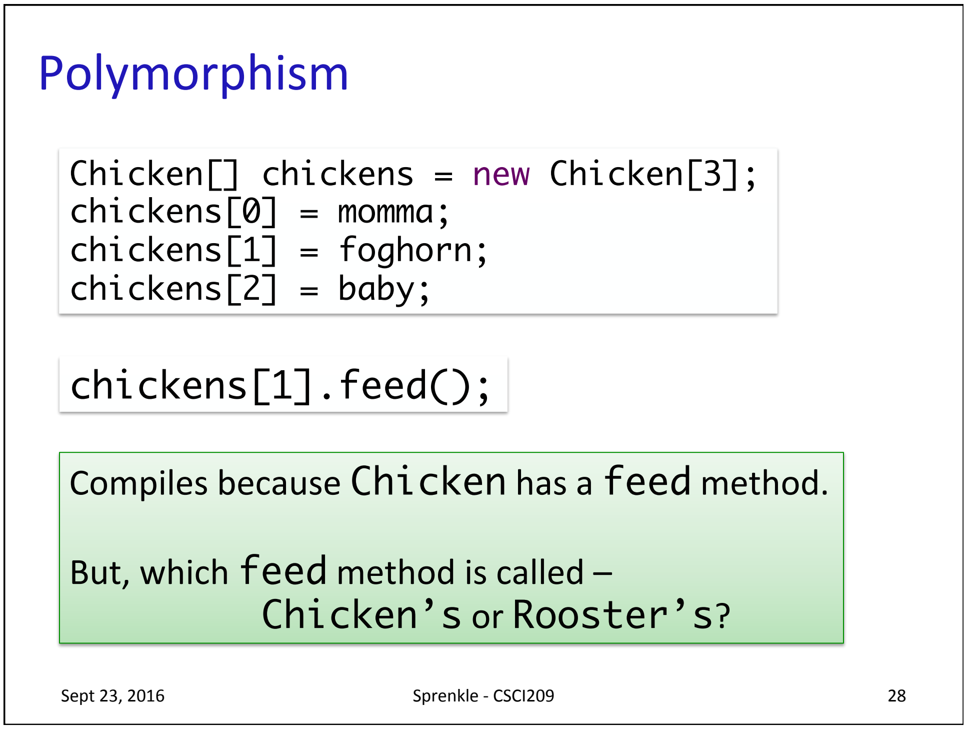
Compiler’s Behavior

- 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.

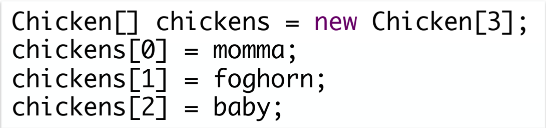
But, which `feed` method is called – `Chicken’s` or `Rooster’s`?

Polymorphism

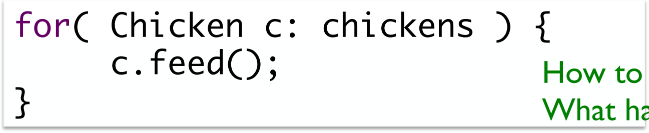
- Which method do we call when we call `chicken[1].feed()` Rooster's or Chicken's?
- In Java (and Python): Rooster's!
  - Object is a `Rooster`
  - JVM figures out its class at runtime and runs the appropriate method
- **Dynamic dispatch**
  - At runtime, the object's class is determined
  - Then, appropriate method for that class is dispatched

Feed the Chickens!

Recall:



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



```
for( Chicken c: chickens ) {
        c.feed();
}
```

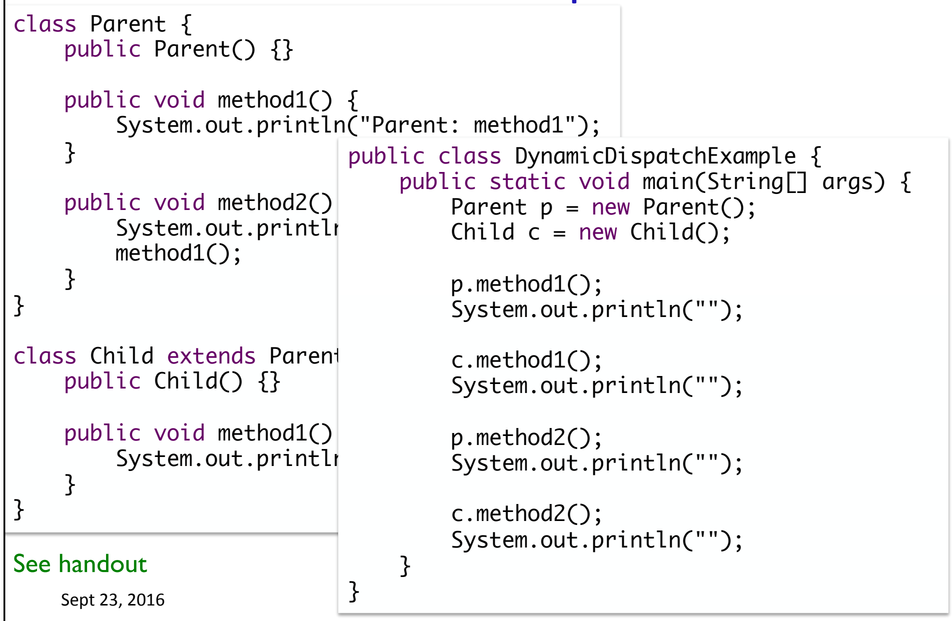
How to read this code?
What happens in execution?

- Dynamic dispatch calls the appropriate method in each case, corresponding to the actual class of each object
  - This is the power of polymorphism and dynamic dispatch!

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

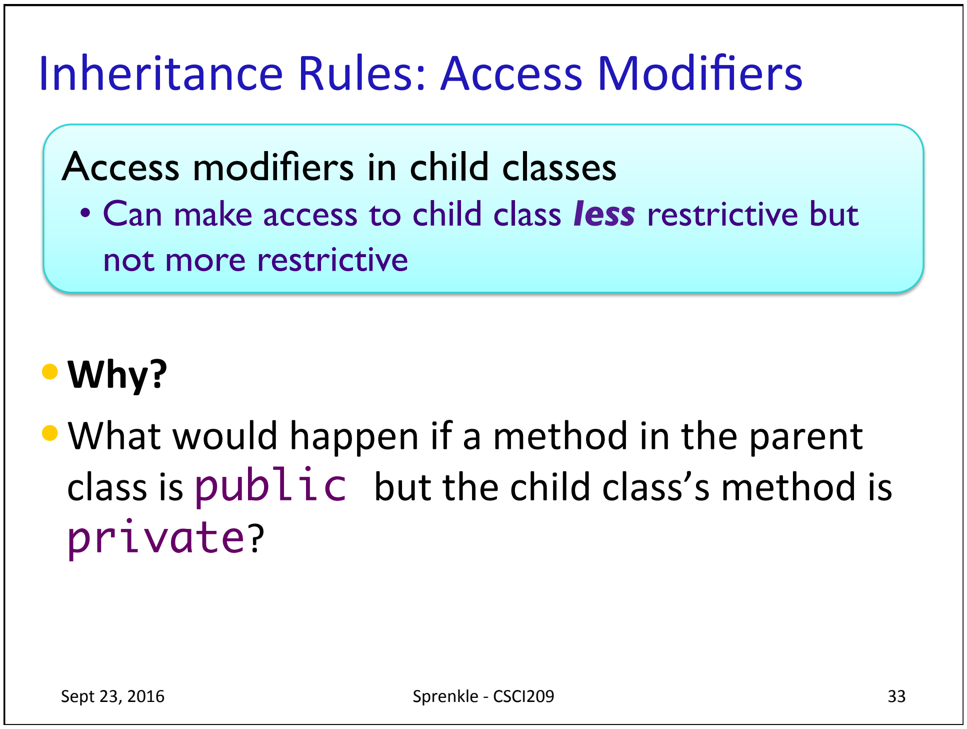
What Will This Code Output?



```
class Parent {
    public Parent() {}

    public void method1() {
        System.out.println("Parent: method1");
    }

    public void method2()
        System.out.printl
        method1();
    }
}

class Child extends Parent
    public Child() {}

    public void method1()
        System.out.printl
    }
}
```

```
public class DynamicDispatchExample {
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = new Child();

        p.method1();
        System.out.println("");

        c.method1();
        System.out.println("");

        p.method2();
        System.out.println("");

        c.method2();
        System.out.println("");
    }
}
```

See handout

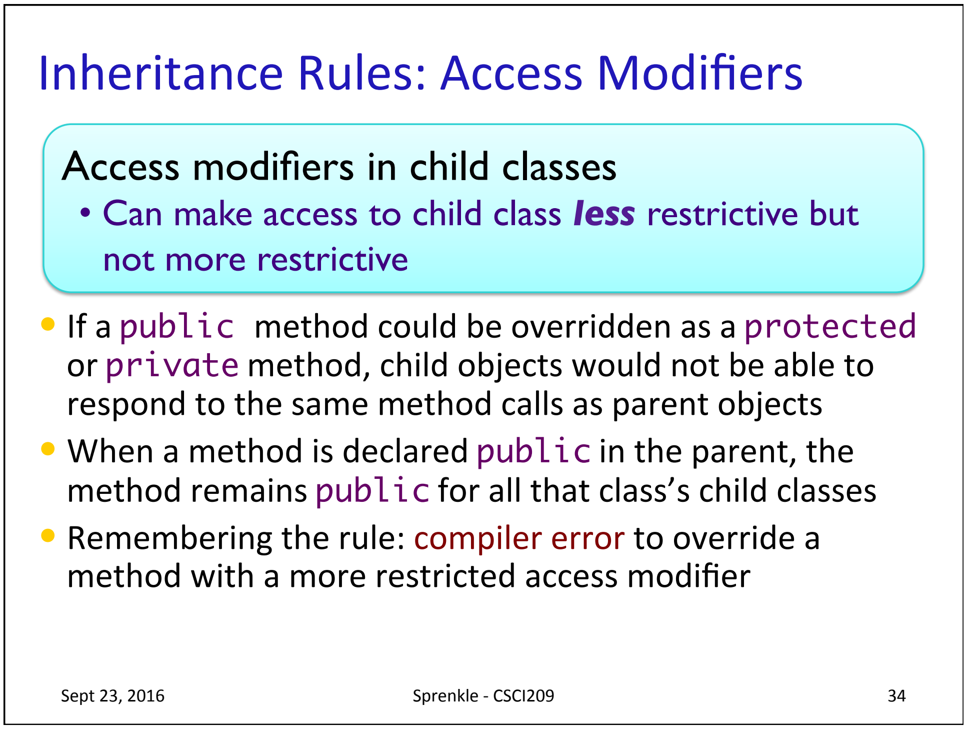


Inheritance Rules: Access Modifiers

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`?



Inheritance Rules: Access Modifiers

Access modifiers in child classes

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

- If a `public` method could be overridden as a `protected` or `private` method, child objects would not be able to respond to the same method calls as parent objects
- When a method is declared `public` in the parent, the method remains `public` for all that class's child classes
- Remembering the rule: compiler error to override a method with a more restricted access modifier

Assignment 5

- Start of a simple video game
  - `Game` class to run
  - `GamePiece` is parent class of other moving objects
- Some less-than-ideal design
  - Can’t fix until see other Java structures (Monday)
- Don’t need to understand all of the code, just some of it
- Create a `Goblin` class and a `Treasure` class
  - Move `Goblin` and Treasure
- Due Wednesday

Copy `/csdept/local/courses/cs209/handouts/assign5`

CS Alumni Visit

Friday, September 23

3:30 P.M. Parmly 405

Anurag Chandra ‘98

Matt Nelson ‘04

This Friday, September 23rd, Anurag Chandra, W&L '98, and Matt Nelson, W&L ’04, will be visiting the campus with the Alumni Science Advisory Board.

Mr. Chandra is a Director at PricewaterhouseCoopers in Decatur, Georgia.

Mr. Nelson is a Software Developer and Consultant with Applied Information Sciences in Washington, DC.

They will meet with interested students in Parmly 405 at 3:30 P.M. to cover such topics as employment opportunities, the interview process, questions, approaches, and challenges.