CISC370: Inheritance

Sara Sprenkle

Questions?

- Review
- Assignment 0 due
  - Submissions
- CPM Accounts

Quiz!

Inheritance

- Build new classes based on existing classes
  - Allows you to reuse code
- Start with a Class (**superclass**)
- Create another class that extends the class (**subclass** or **derived class**)
  - subclass inherits all of superclass's methods and fields (unless they're `private`)
  - can also *override* methods
    - use the same name, but the implementation is different

Inheritance

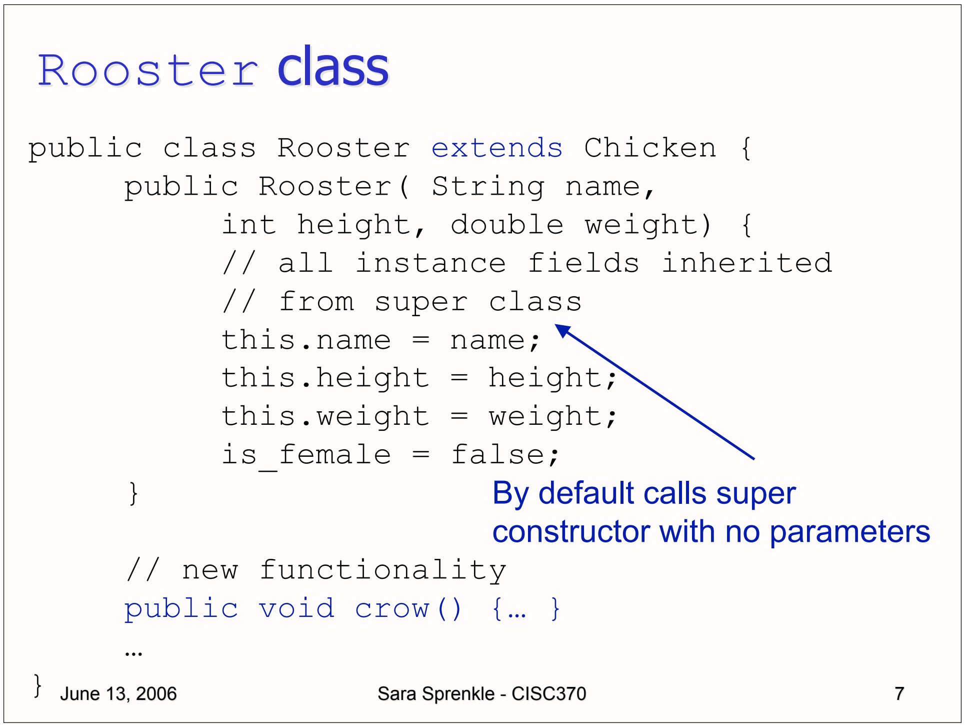
  - Subclass adds methods or fields for additional functionality
  - If the subclass redefines a superclass method, can still call the superclass method on the “super” object

- Use `extends` keyword to make a subclass

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

- The is a relationship
  - Classic mark of inheritance
- Rooster will be subclass
- Chicken will be superclass

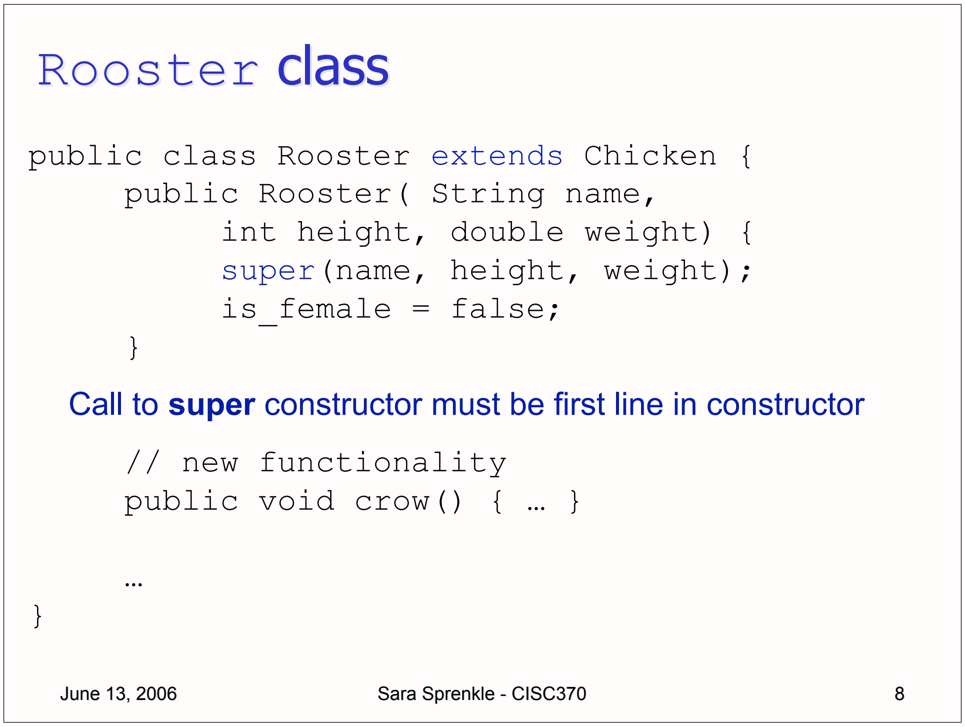


Rooster class

```
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 super constructor with no parameters



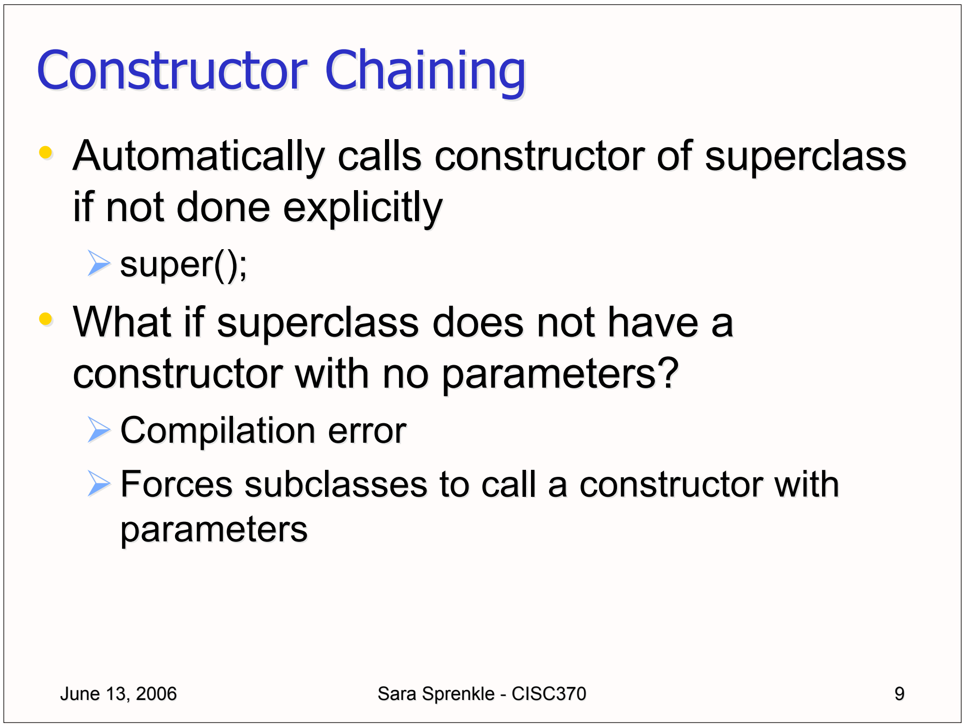
Rooster class

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

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

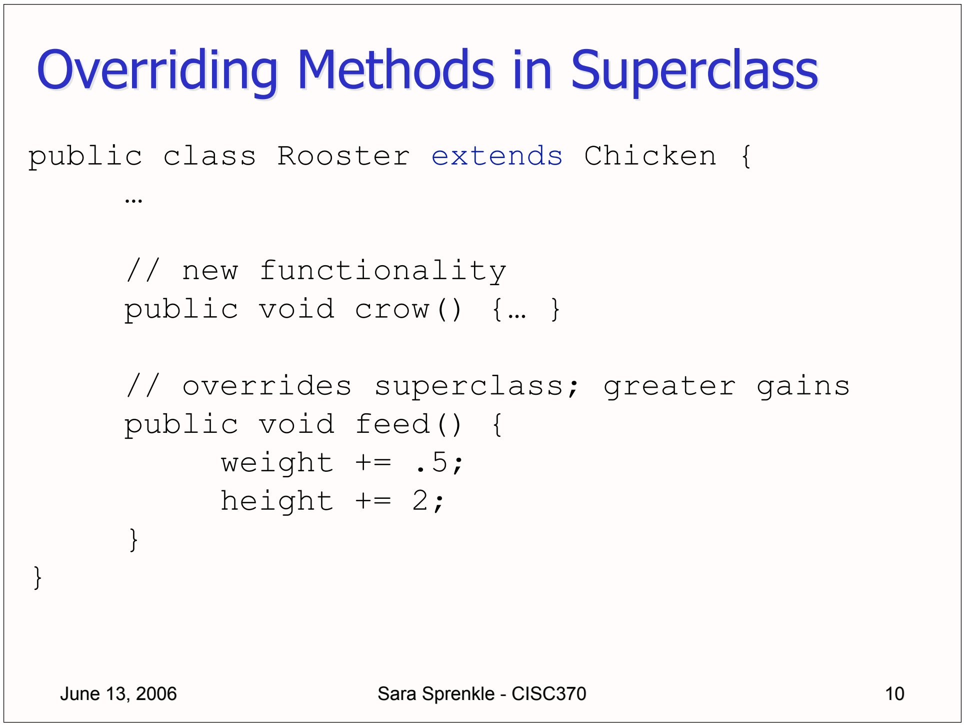
```
     // new functionality
     public void crow() { … }

     …
}
```



Constructor Chaining

- Automatically calls constructor of superclass if not done explicitly
  - super();
- What if superclass does not have a constructor with no parameters?
  - Compilation error
  - Forces subclasses to call a constructor with parameters

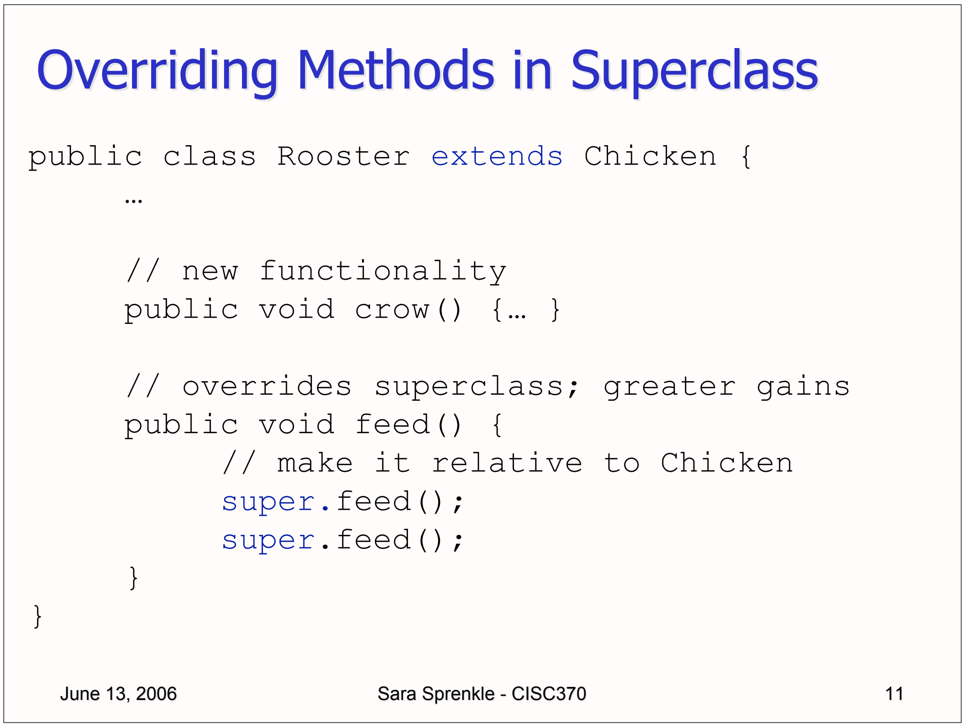


Overriding Methods in Superclass

```
public class Rooster extends Chicken {
     …

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

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

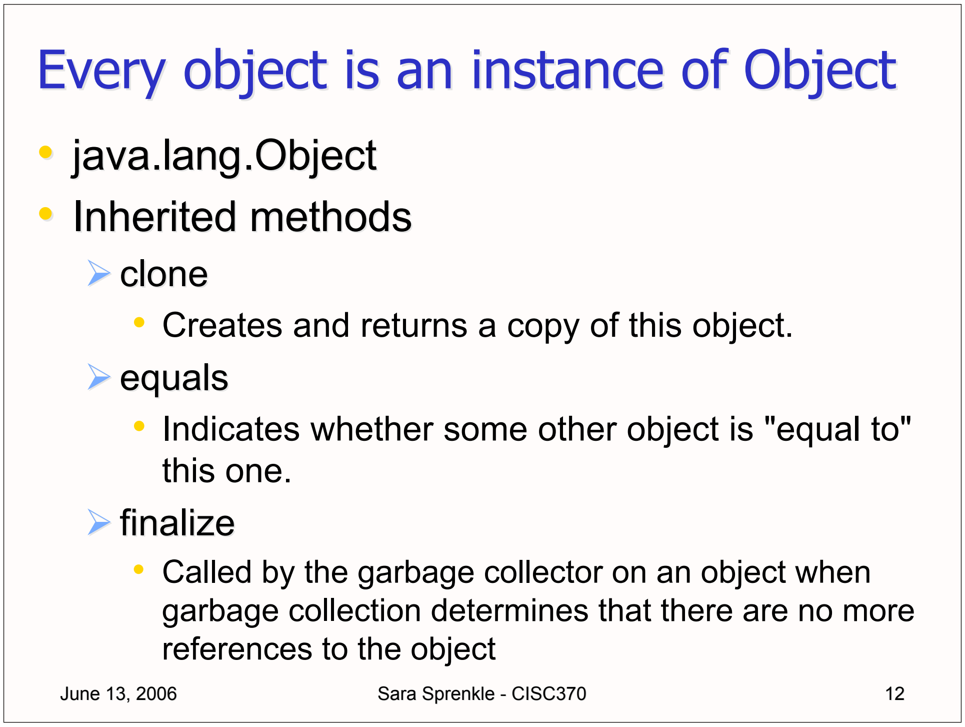


Overriding Methods in Superclass

```
public class Rooster extends Chicken {
     …

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

     // overrides superclass; greater gains
     public void feed() {
          // make it relative to Chicken
          super.feed();
          super.feed();
     }
}
```



Every object is an instance of Object

- java.lang.Object
- Inherited methods
  - clone
    - Creates and returns a copy of this object.
  - equals
    - Indicates whether some other object is "equal to" this one.
  - finalize
    - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

Aside on finalize()

- No deconstructors in Java
  - No explicit freeing of memory
- Garbage collector calls finalize()
  - Garbage collector is low-priority thread
    - Or runs when available memory gets tight
  - Before can clean up memory, object may have generated temp files or open network connections that should be deleted/closed first
- Benefits of garbage collection?

Aside on finalize()

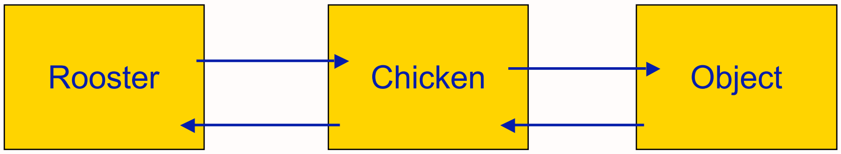
- Benefits of garbage collection
  - Fewer memory leaks
    - Less buggy code
    - But, memory leaks are still possible
  - Code is easier to write
- Cost: garbage collection may not be as efficient as explicit freeing of memory

Every object is an instance of Object

- java.lang.Object
- Inherited methods
  - getClass
    - Returns the runtime class of an object.
  - toString
    - Override to customize printout for use in System.out.println()
  - And others…

Inheritance Tree

- java.lang.Object
  - Chicken
    - Rooster

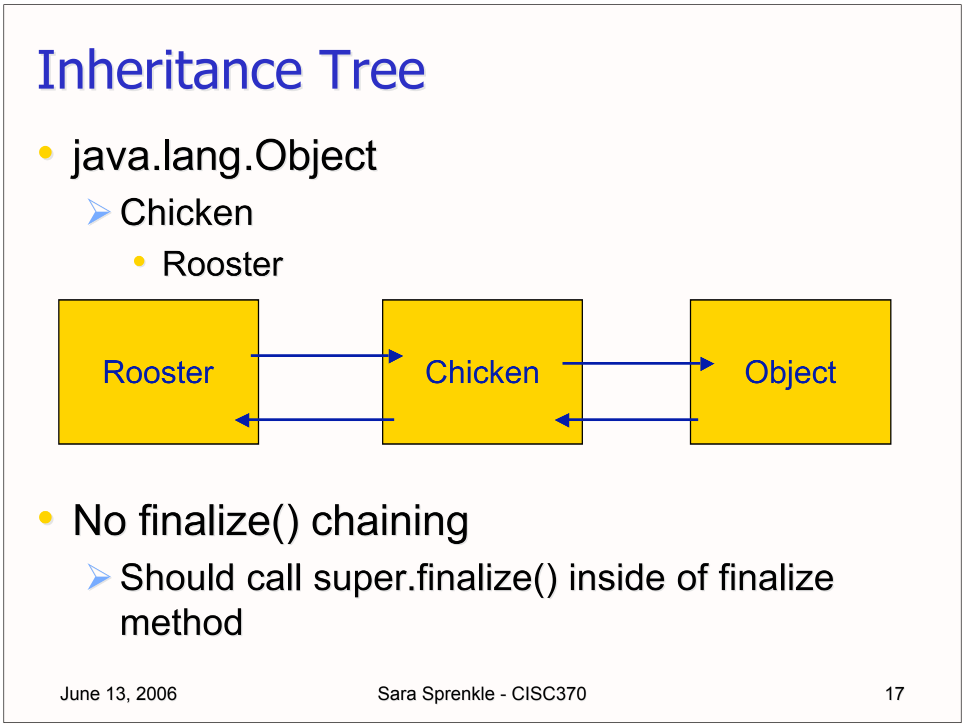


Rooster

Chicken

Object

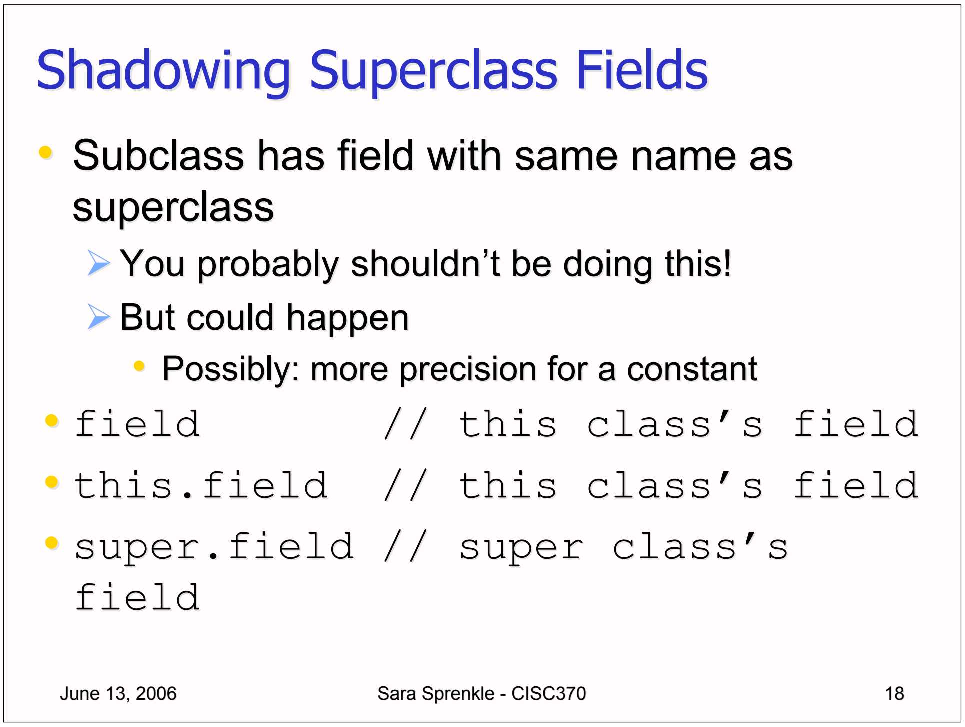
- Call constructor of superclass first
  - Know you have the fields of superclass before you implement constructor for your class



Inheritance Tree

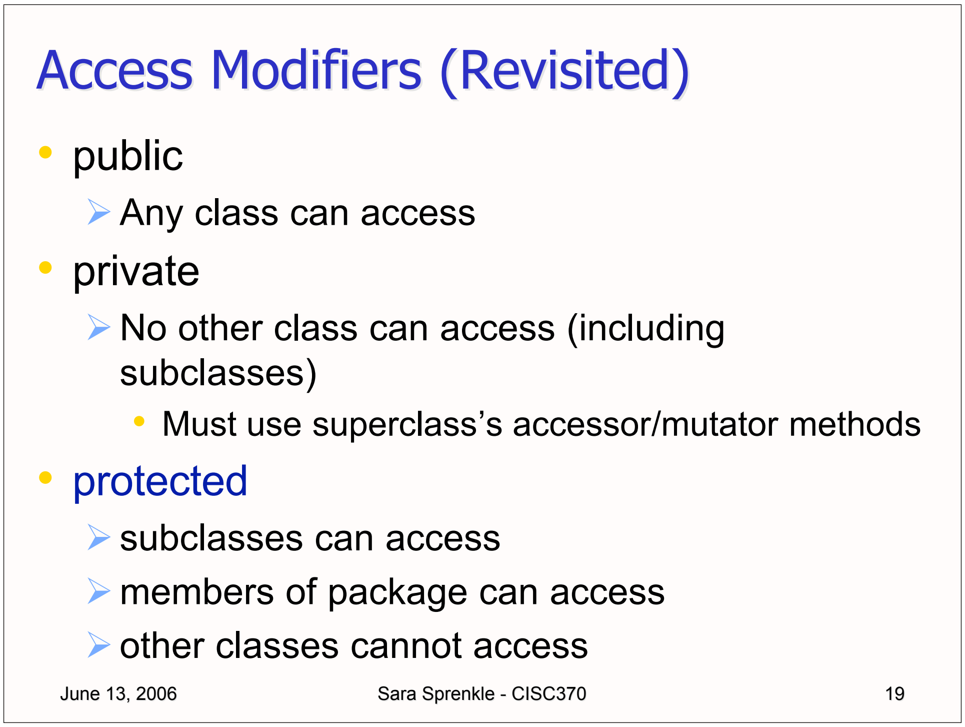
- java.lang.Object
  - Chicken
    - Rooster

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



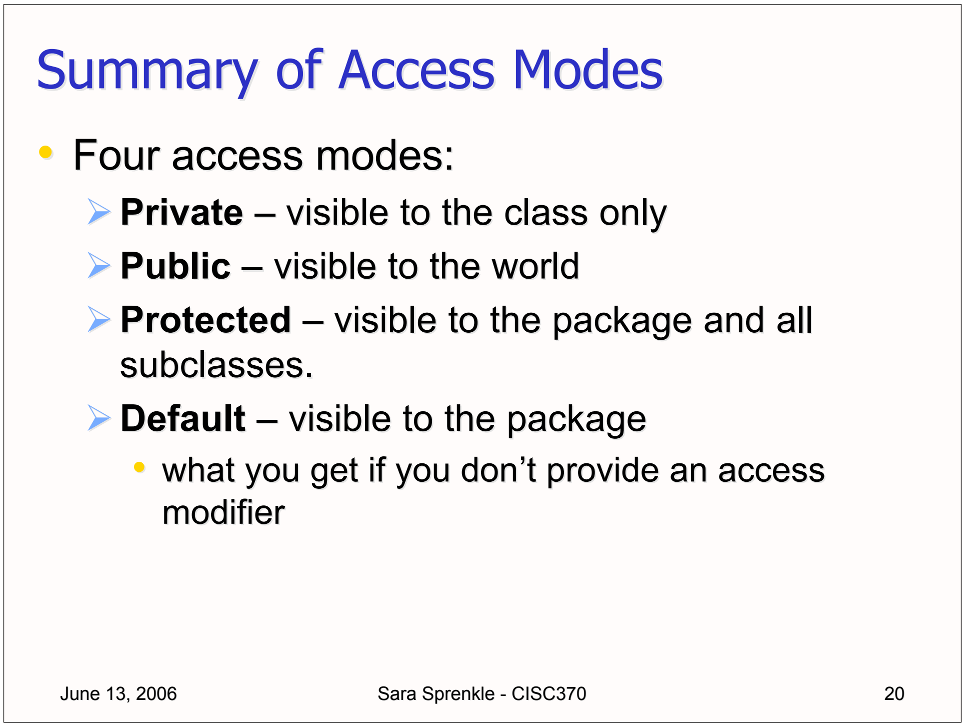
Shadowing Superclass Fields

- Subclass has field with same name as superclass
  - You probably shouldn't be doing this!
  - But could happen
    - Possibly: more precision for a constant
- `field        // this class's field`
- `this.field   // this class's field`
- `super.field  // super class's field`



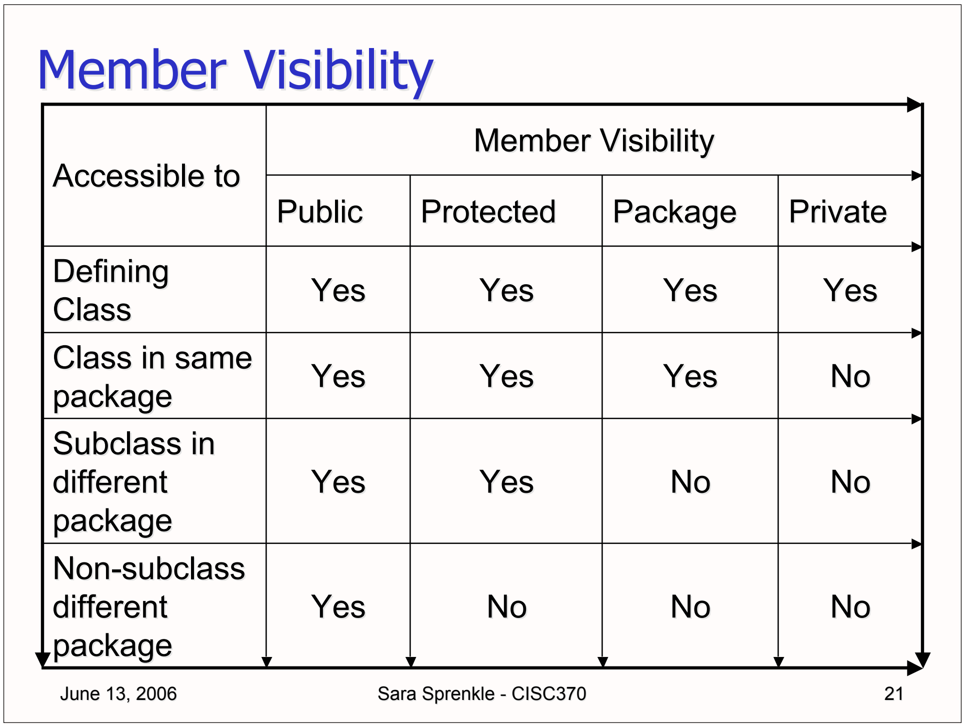
Access Modifiers (Revisited)

- public
  - Any class can access
- private
  - No other class can access (including subclasses)
    - Must use superclass's accessor/mutator methods
- protected
  - subclasses can access
  - members of package can access
  - other classes cannot access



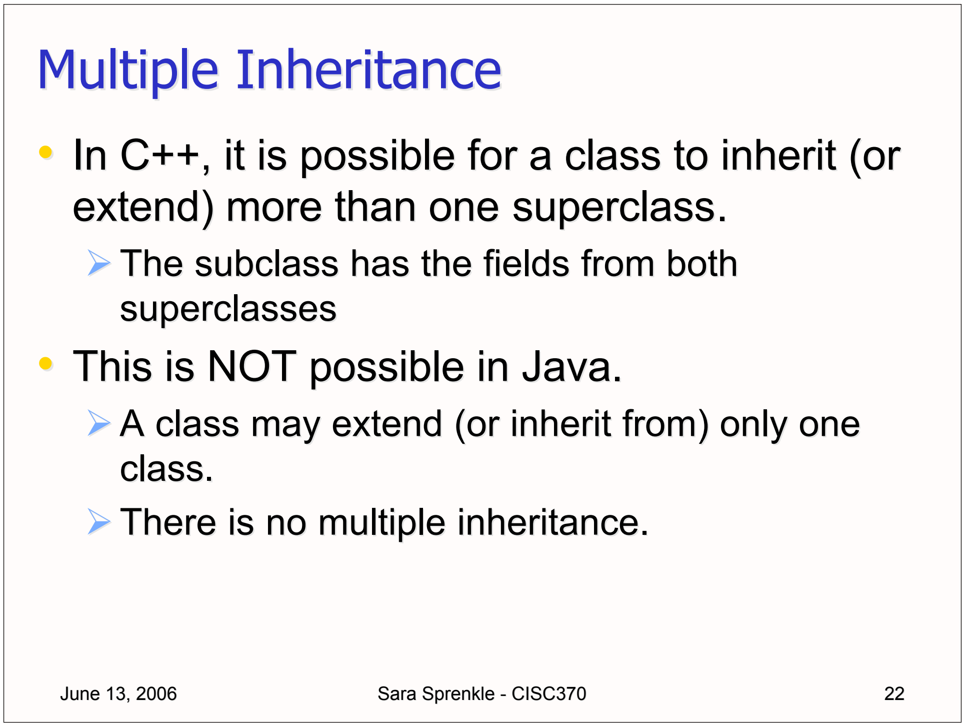
Summary of Access Modes

- Four access modes:
  - **Private** – visible to the class only
  - **Public** – visible to the world
  - **Protected** – visible to the package and all subclasses.
  - **Default** – visible to the package
    - what you get if you don't provide an access modifier



Member Visibility

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



Multiple Inheritance

- In C++, it is possible for a class to inherit (or extend) more than one superclass.
  - The subclass has the fields from both superclasses
- This is NOT possible in Java.
  - A class may extend (or inherit from) only one class.
  - There is no multiple inheritance.

Polymorphism

- You can use a derived class object whenever the program expects an object of the superclass
- object variables are *polymorphic*.
- A Chicken object variable can refer to an object of class Chicken, Hen, Rooster, or any class that inherits from Chicken

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

Polymorphism

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

- But, `chicken[1]` is still a Chicken object

  `chicken[1].crow();`

  will not work

Polymorphism

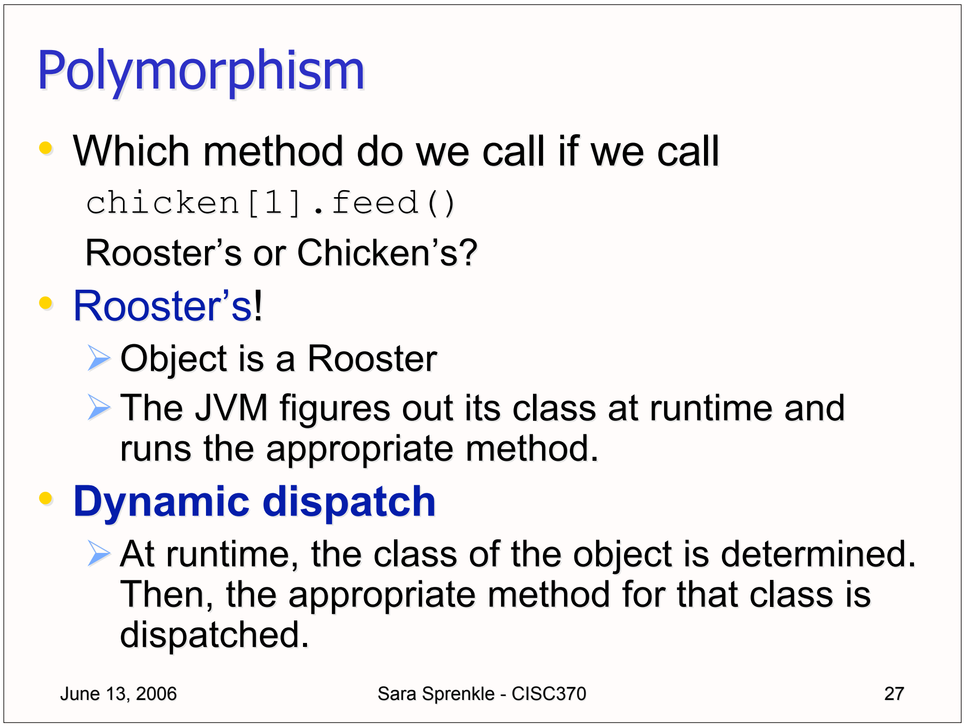
- When we refer to a Rooster object through a Rooster object variable, we see it as a Rooster object
- If we refer to a Rooster object through a Chicken object variable, we see it only as a Chicken object.
- We cannot assign a superclass object to a derived class object variable
  - A Rooster is a Chicken, but a Chicken is not necessarily a Rooster

Polymorphism

- Which method do we call if we call

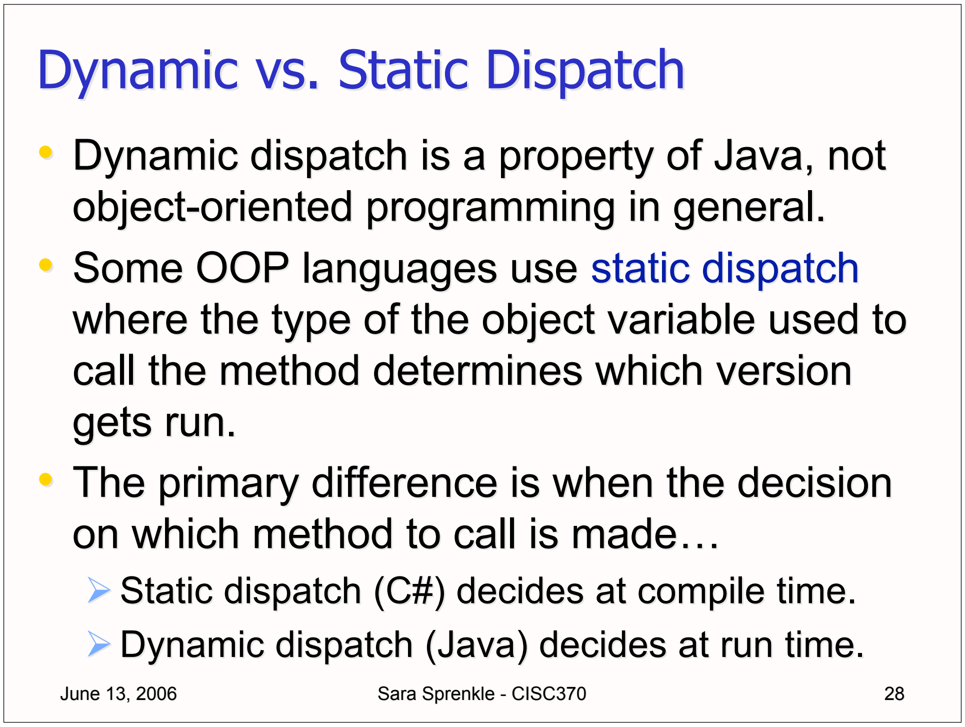
  ```
  chicken[1].feed()
  ```

  Rooster's or Chicken's?



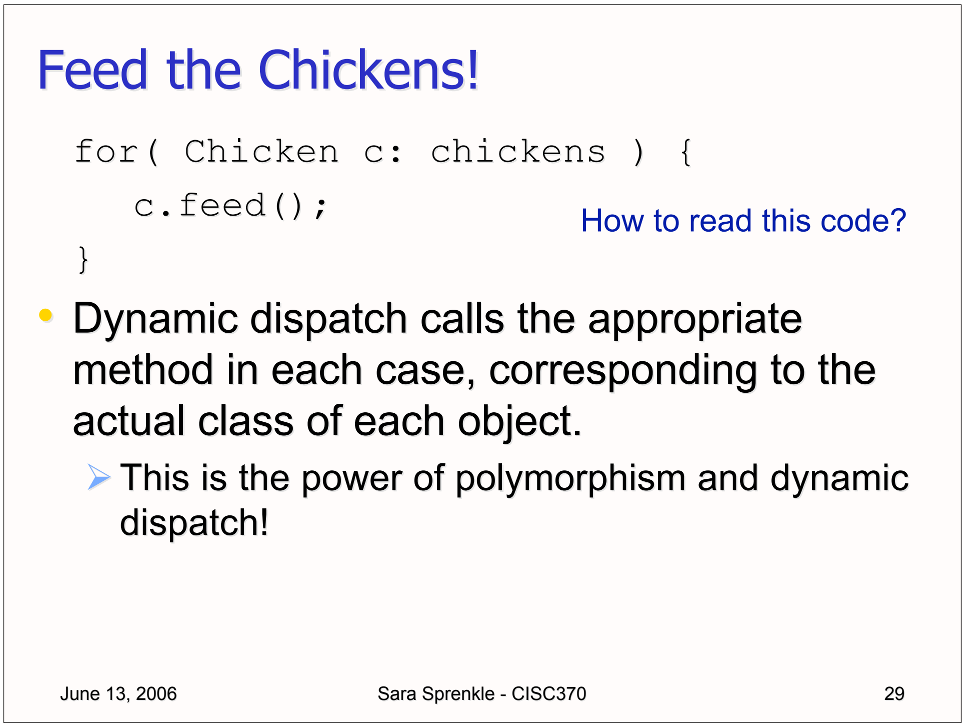
Polymorphism

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



Dynamic vs. Static Dispatch

- Dynamic dispatch is a property of Java, not object-oriented programming in general.
- Some OOP languages use static dispatch where the type of the object variable used to call the method determines which version gets run.
- The primary difference is when the decision on which method to call is made…
  - Static dispatch (C#) decides at compile time.
  - Dynamic dispatch (Java) decides at run time.

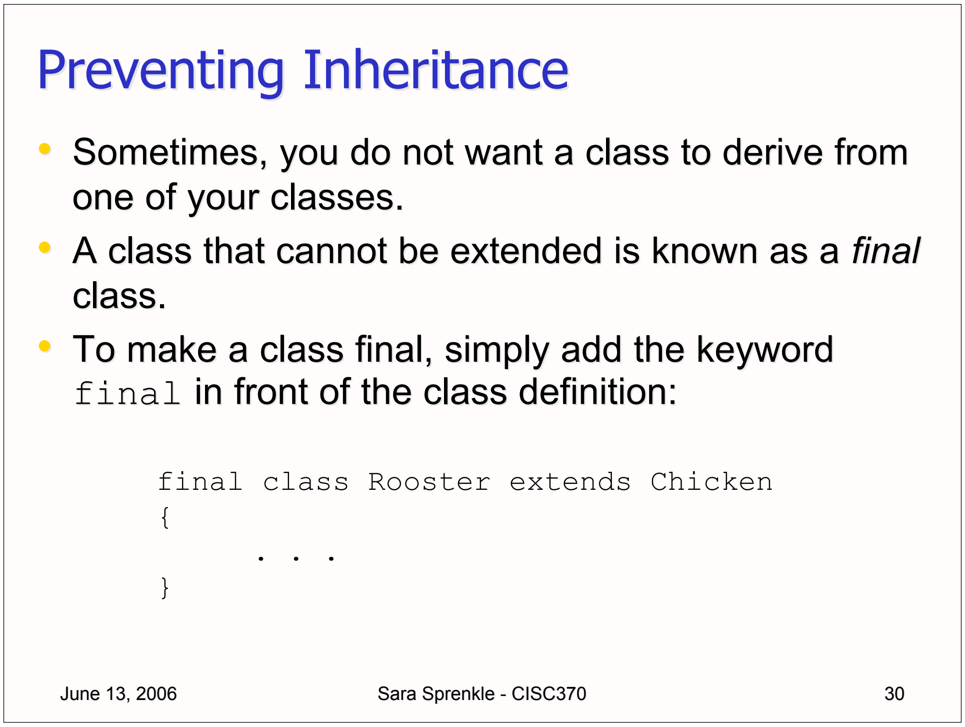


Feed the Chickens!

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

How to read this code?

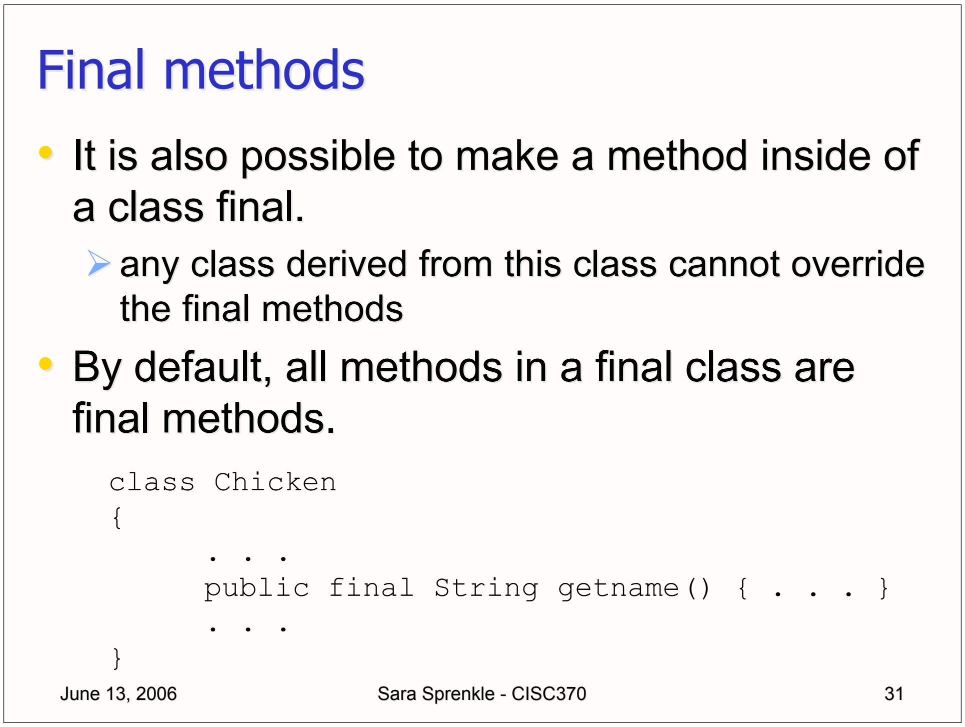
- Dynamic dispatch calls the appropriate method in each case, corresponding to the actual class of each object.
  - ➢ This is the power of polymorphism and dynamic dispatch!



Preventing Inheritance

- Sometimes, you do not want a class to derive from one of your classes.
- A class that cannot be extended is known as a *final* class.
- To make a class final, simply add the keyword `final` in front of the class definition:

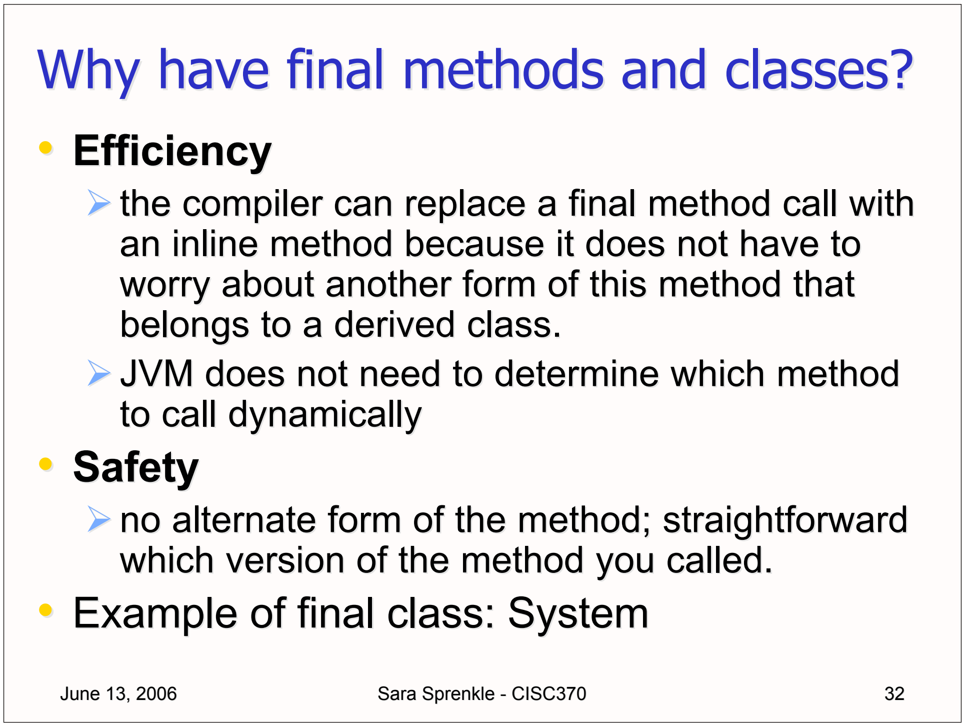
```
final class Rooster extends Chicken
{
        . . .
}
```



Final methods

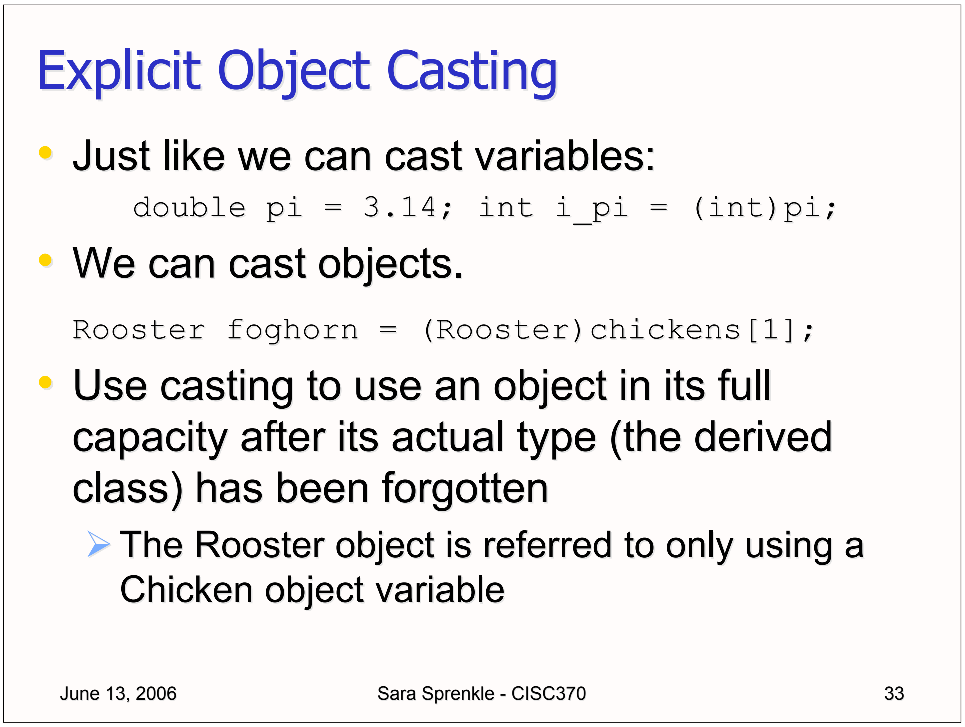
- It is also possible to make a method inside of a class final.
  - any class derived from this class cannot override the final methods
- By default, all methods in a final class are final methods.

```
class Chicken
{
        . . .
        public final String getname() { . . . }
        . . .
}
```



Why have final methods and classes?

- **Efficiency**
  - the compiler can replace a final method call with an inline method because it does not have to worry about another form of this method that belongs to a derived class.
  - JVM does not need to determine which method to call dynamically
- **Safety**
  - no alternate form of the method; straightforward which version of the method you called.
- Example of final class: System



Explicit Object Casting

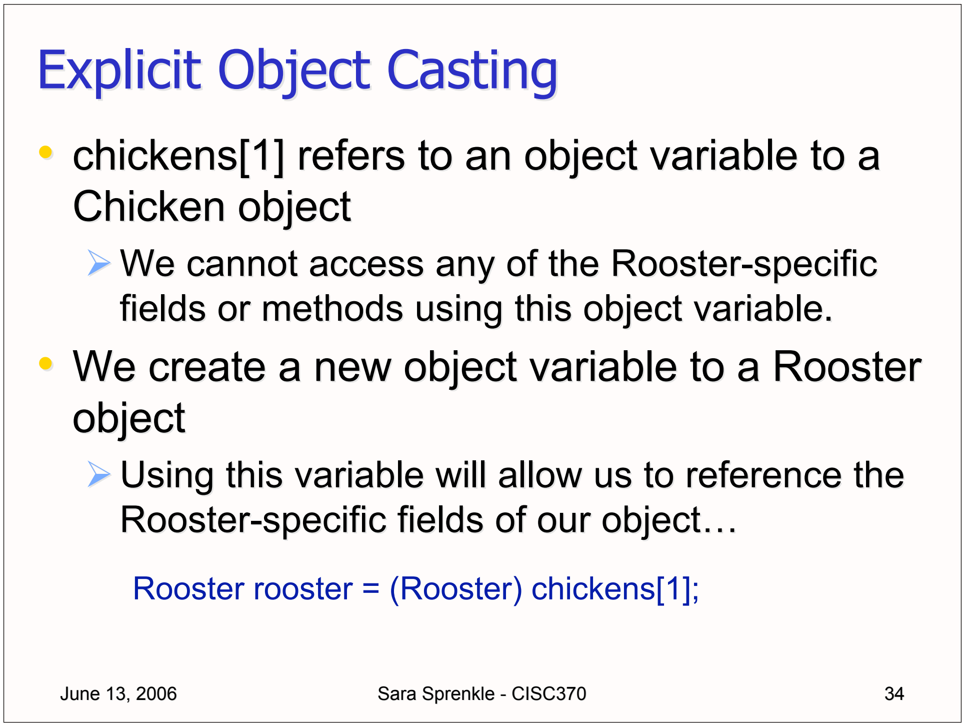
- Just like we can cast variables:

```
double pi = 3.14; int i_pi = (int)pi;
```

- We can cast objects.

```
Rooster foghorn = (Rooster)chickens[1];
```

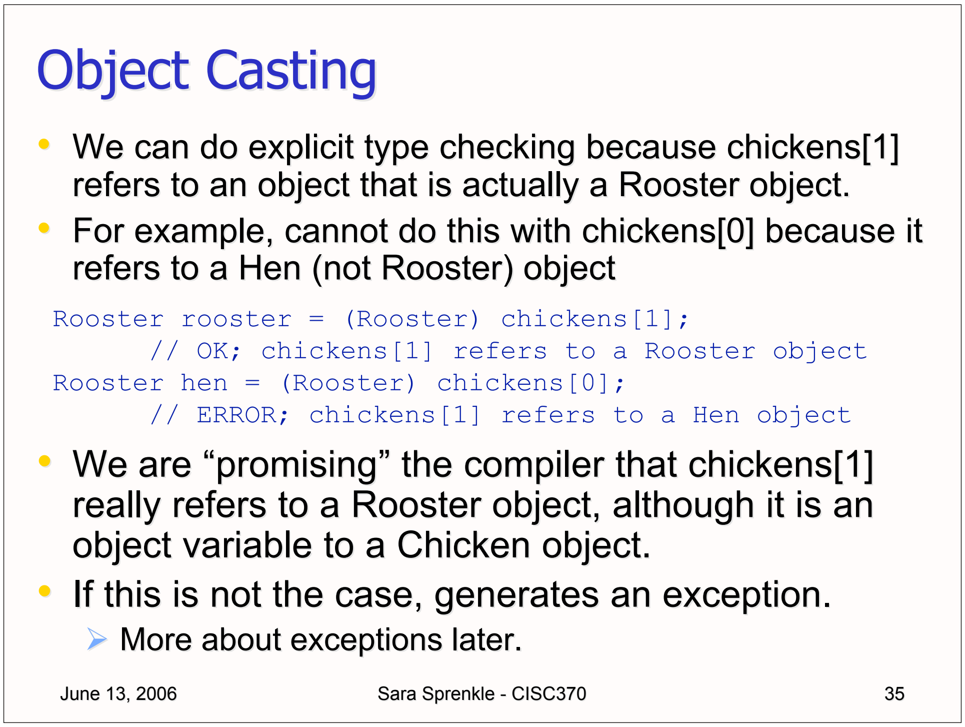
- Use casting to use an object in its full capacity after its actual type (the derived class) has been forgotten
  - The Rooster object is referred to only using a Chicken object variable



Explicit Object Casting

- chickens[1] refers to an object variable to a Chicken object
  - We cannot access any of the Rooster-specific fields or methods using this object variable.
- We create a new object variable to a Rooster object
  - Using this variable will allow us to reference the Rooster-specific fields of our object…

```
Rooster rooster = (Rooster) chickens[1];
```

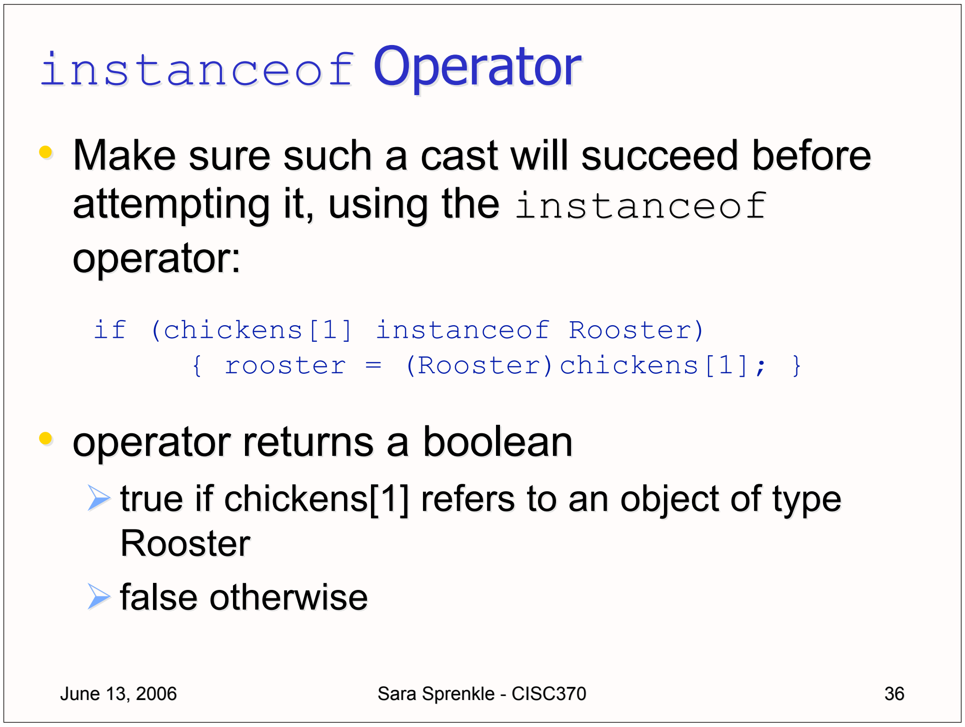


Object Casting

- We can do explicit type checking because chickens[1] refers to an object that is actually a Rooster object.
- For example, cannot do this with chickens[0] because it refers to a Hen (not Rooster) object

```
Rooster rooster = (Rooster) chickens[1];
      // OK; chickens[1] refers to a Rooster object
Rooster hen = (Rooster) chickens[0];
      // ERROR; chickens[1] refers to a Hen object
```

- We are “promising” the compiler that chickens[1] really refers to a Rooster object, although it is an object variable to a Chicken object.
- If this is not the case, generates an exception.
  - More about exceptions later.

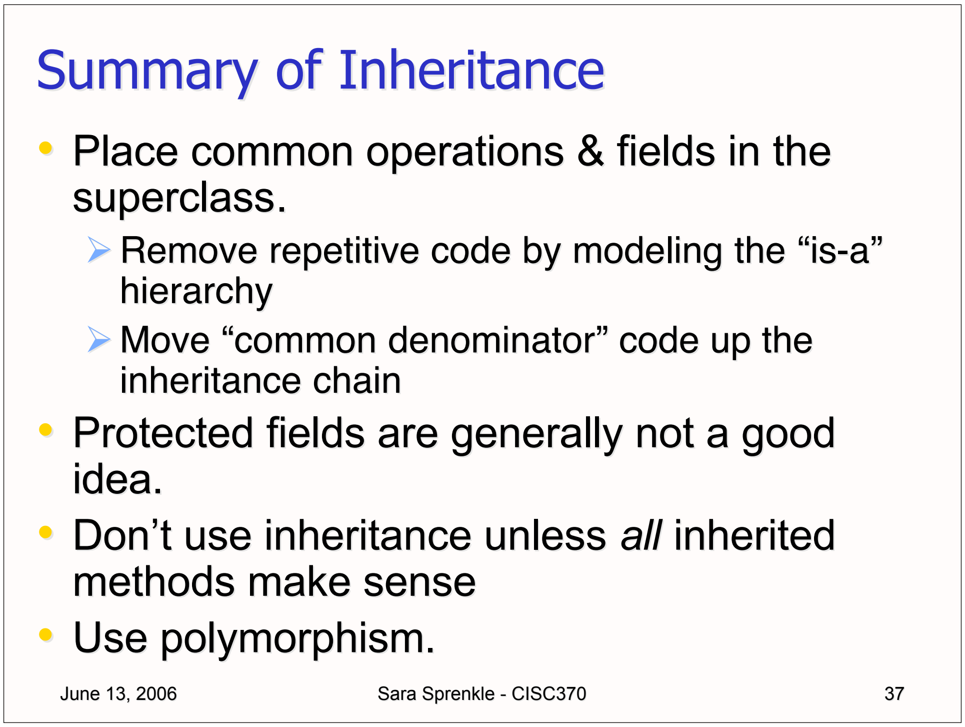


`instanceof` Operator

- Make sure such a cast will succeed before attempting it, using the `instanceof` operator:

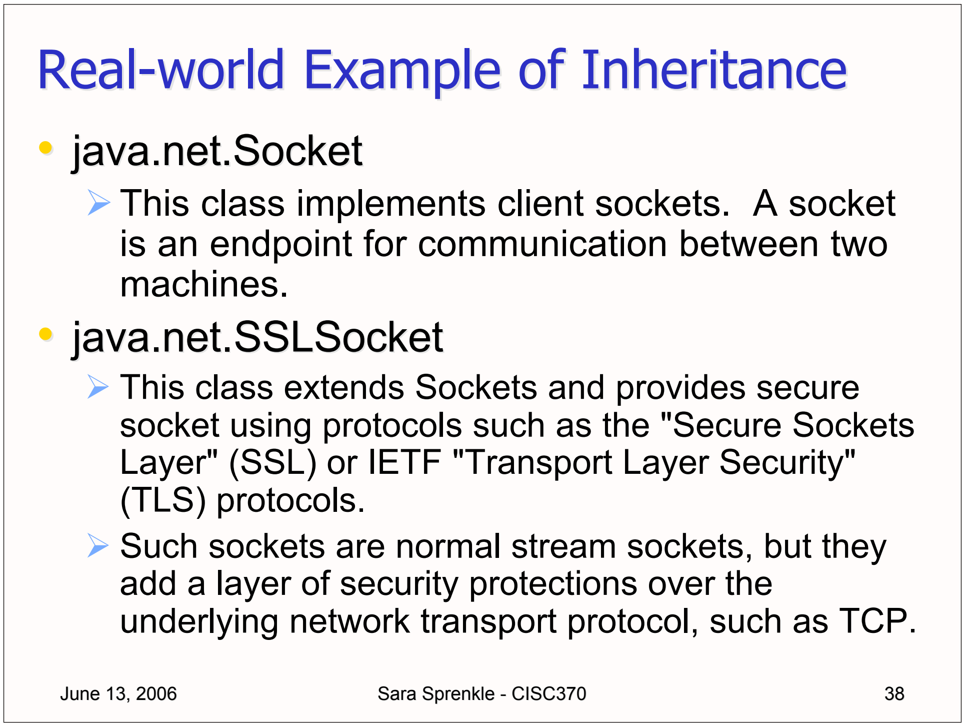
```
if (chickens[1] instanceof Rooster)
      { rooster = (Rooster)chickens[1]; }
```

- operator returns a boolean
  - true if chickens[1] refers to an object of type Rooster
  - false otherwise



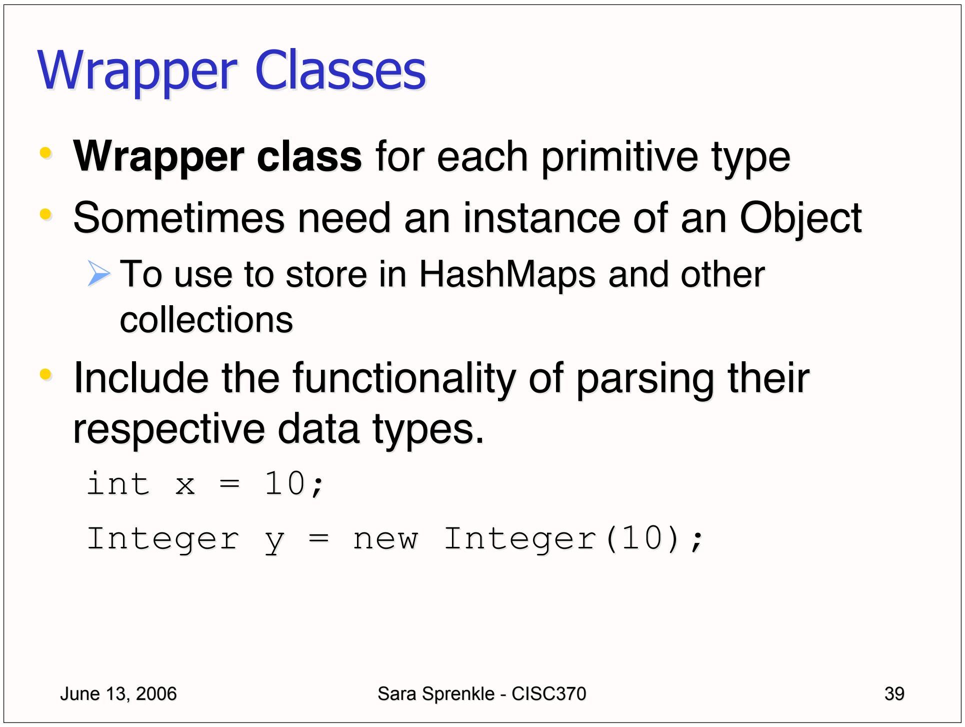
Summary of Inheritance

- Place common operations & fields in the superclass.
  - Remove repetitive code by modeling the “is-a” hierarchy
  - Move “common denominator” code up the inheritance chain
- Protected fields are generally not a good idea.
- Don’t use inheritance unless *all* inherited methods make sense
- Use polymorphism.



Real-world Example of Inheritance

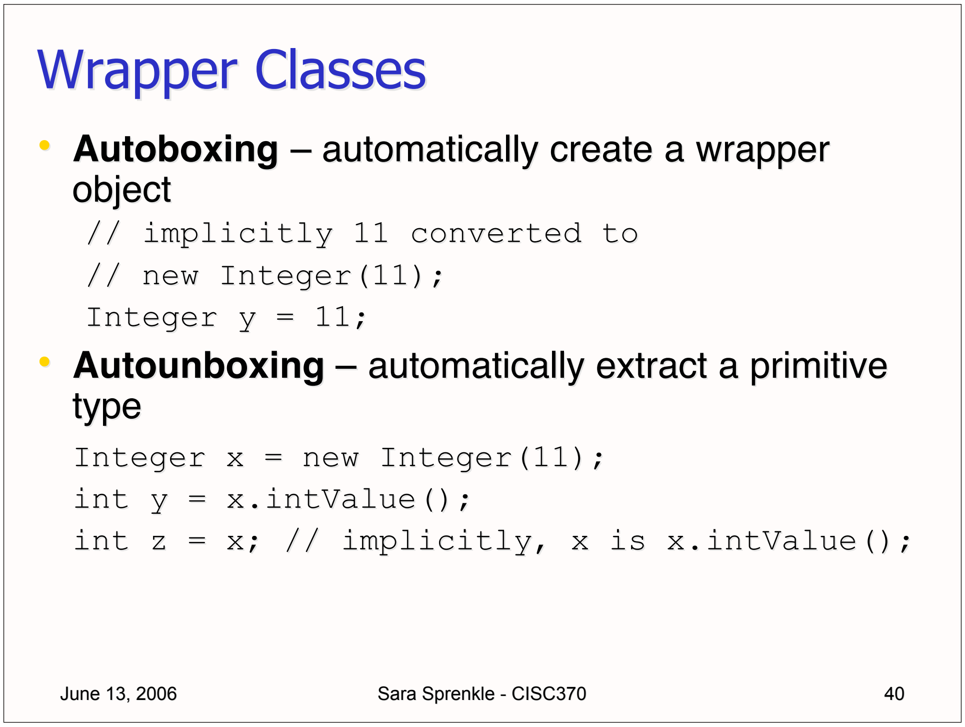
- java.net.Socket
  - This class implements client sockets. A socket is an endpoint for communication between two machines.
- java.net.SSLSocket
  - This class extends Sockets and provides secure socket using protocols such as the "Secure Sockets Layer" (SSL) or IETF "Transport Layer Security" (TLS) protocols.
  - Such sockets are normal stream sockets, but they add a layer of security protections over the underlying network transport protocol, such as TCP.



Wrapper Classes

- **Wrapper class** for each primitive type
- Sometimes need an instance of an Object
  - To use to store in HashMaps and other collections
- Include the functionality of parsing their respective data types.

```
int x = 10;
Integer y = new Integer(10);
```



Wrapper Classes

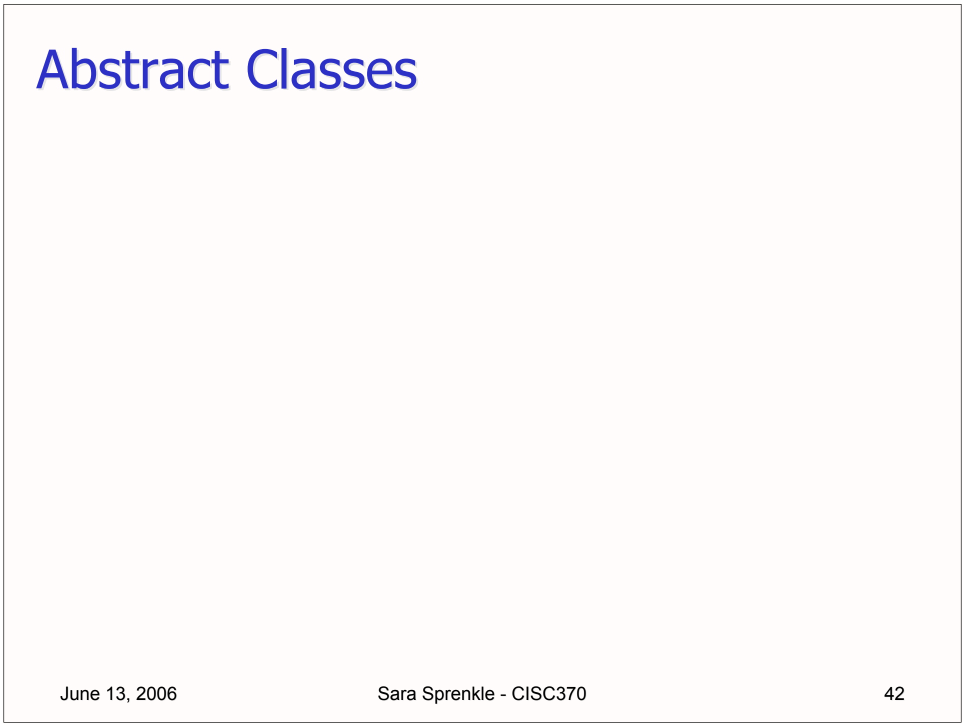
- **Autoboxing** – automatically create a wrapper object

```
// implicitly 11 converted to
// new Integer(11);
Integer y = 11;
```

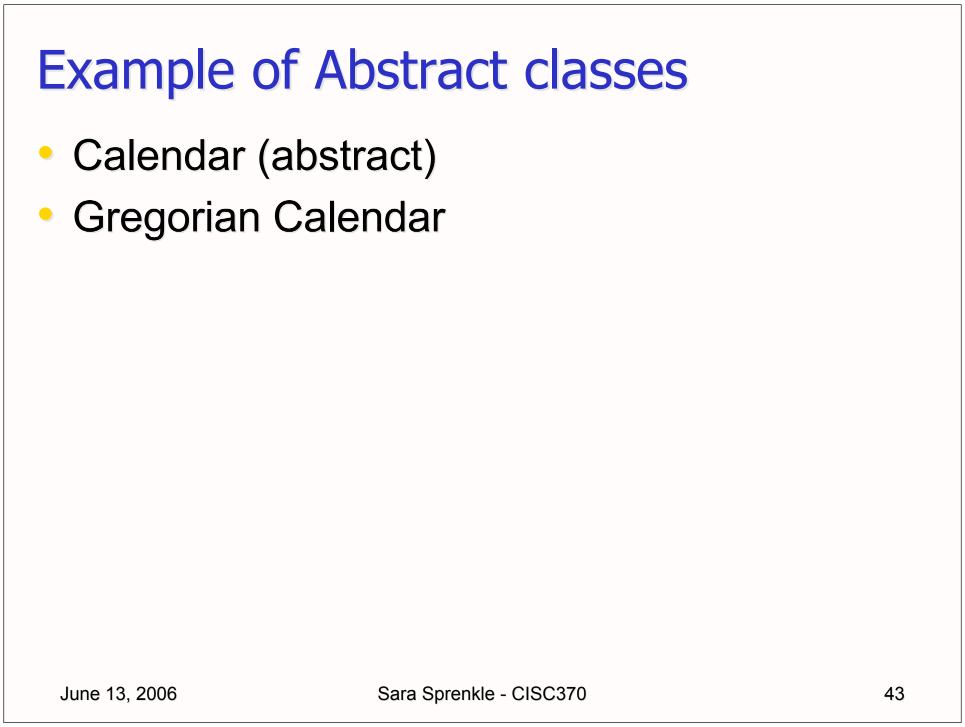
- **Autounboxing** – automatically extract a primitive type

```
Integer x = new Integer(11);
int y = x.intValue();
int z = x; // implicitly, x is x.intValue();
```

PACKAGES!

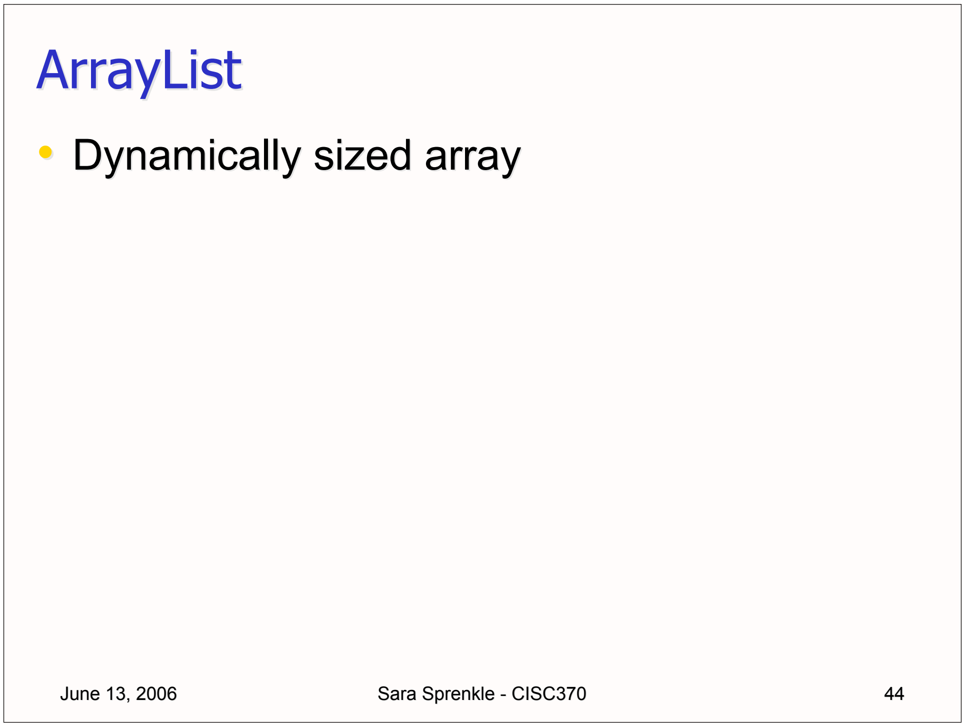


Abstract Classes



Example of Abstract classes

- Calendar (abstract)
- Gregorian Calendar



ArrayList

- Dynamically sized array