COP 3330, Spring 2013

Inheritance II

Instructor : Arup Ghosh
02-27-13

School of Electrical Engineering and Computer Science
University of Central Florida

Today

- Recap and some remarks
- The `Object` class
- The `protected` modifier
- The `final` modifier
  - As applied to inheritance, not for making constants
- Abstract classes
  - A hybrid of normal classes and interfaces

Recap

- Inheritance allows us to *extend* an existing class (superclass) to make a new class (subclass).

- The subclass *inherits* members from the superclass.

- There is an is-a relationship:
  - ***Subclass_Object*** *is-a* ***Superclass_Object***
  - Doesn’t work the other way!

- At the core of every object of the derived class, there lives an object of the parent class.

- This superobject **must be initialized first**, when creating an object of the subclass.

Extending a class

```
public class SportingDog extends Dog {
     ...
}
```

- Here SportingDog is derived from Dog. We say that:
  - SportingDog is a subclass/derived class of Dog
  - Dog is a superclass/base class of SportingDog
  - SportingDog extends Dog
  - SportingDog inherits from Dog
  - ...and so on

- This also creates the relationship: Every SportingDog is-a Cat
  - Polymorphism by subtyping.

Inheritance: Methods

- The *public* members of the base class are inherited by its descendants.
  - Its public face, as it were.
  - The subclasses can access these as if they were its own members.
  - Clients of the subclass can access them as if they were public members.

- The inherited *methods* can be overridden by simply providing an alternative method body in the subclass.

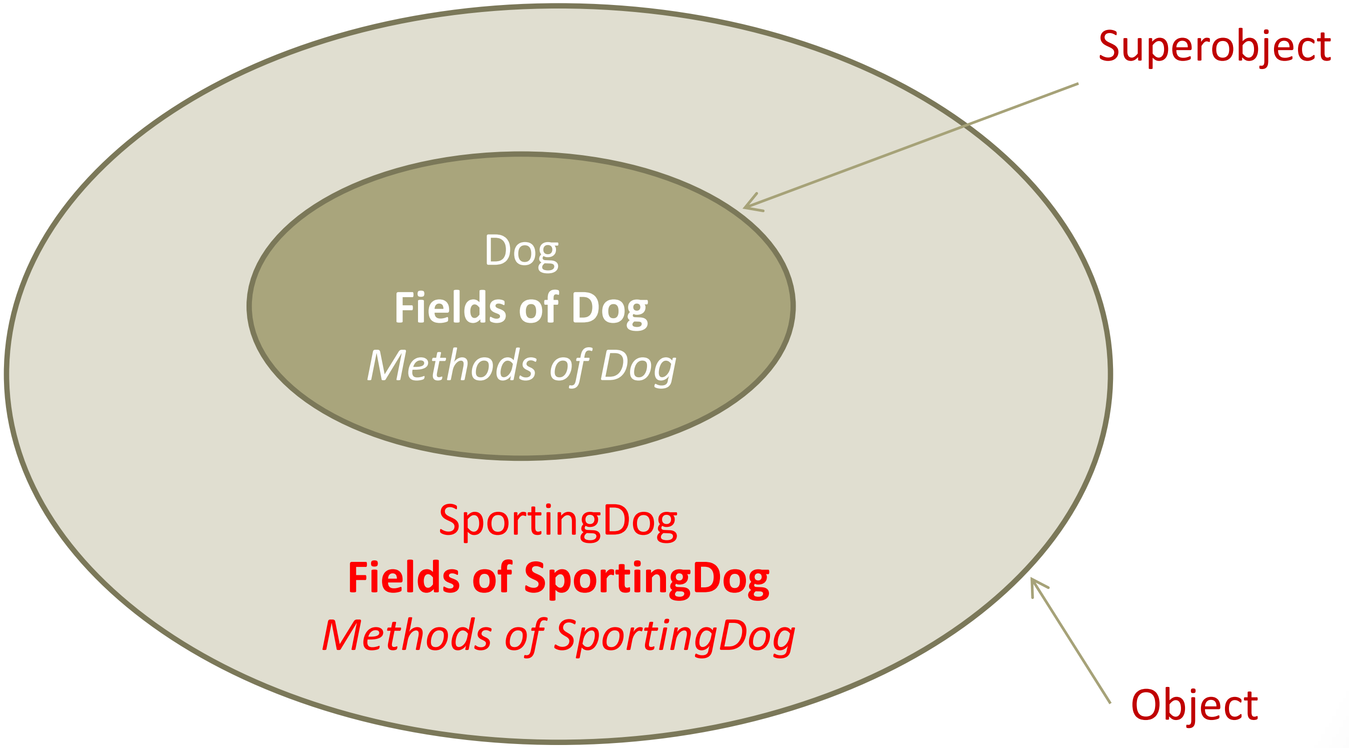
- Note that this overriding *hides* the original method.
  - It can still be accessed within the derived class using `super`.

Inheritance: Fields

- Public fields are inherited, just like methods.
- Naturally overriding doesn't make any sense in this context, since fields are just data, not functionality.
- However, if the derived class has a field with the same name as a public field of the parent, that inherited field is hidden.
  - Can still be accessed with super, of course.
- Happens even if the types of the variables are different.
- In general, avoid this. It is considered bad practice.

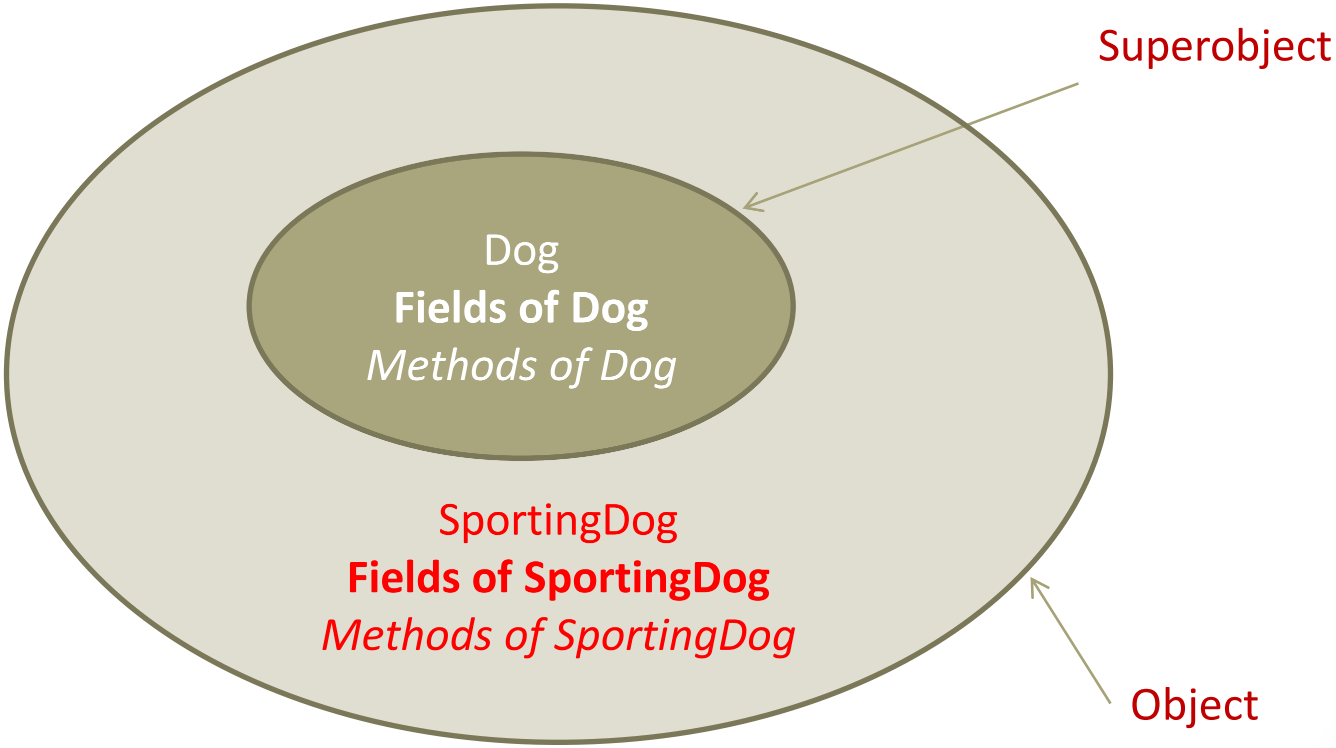
Structure

The superobject is the thing being accessed by the super keyword.



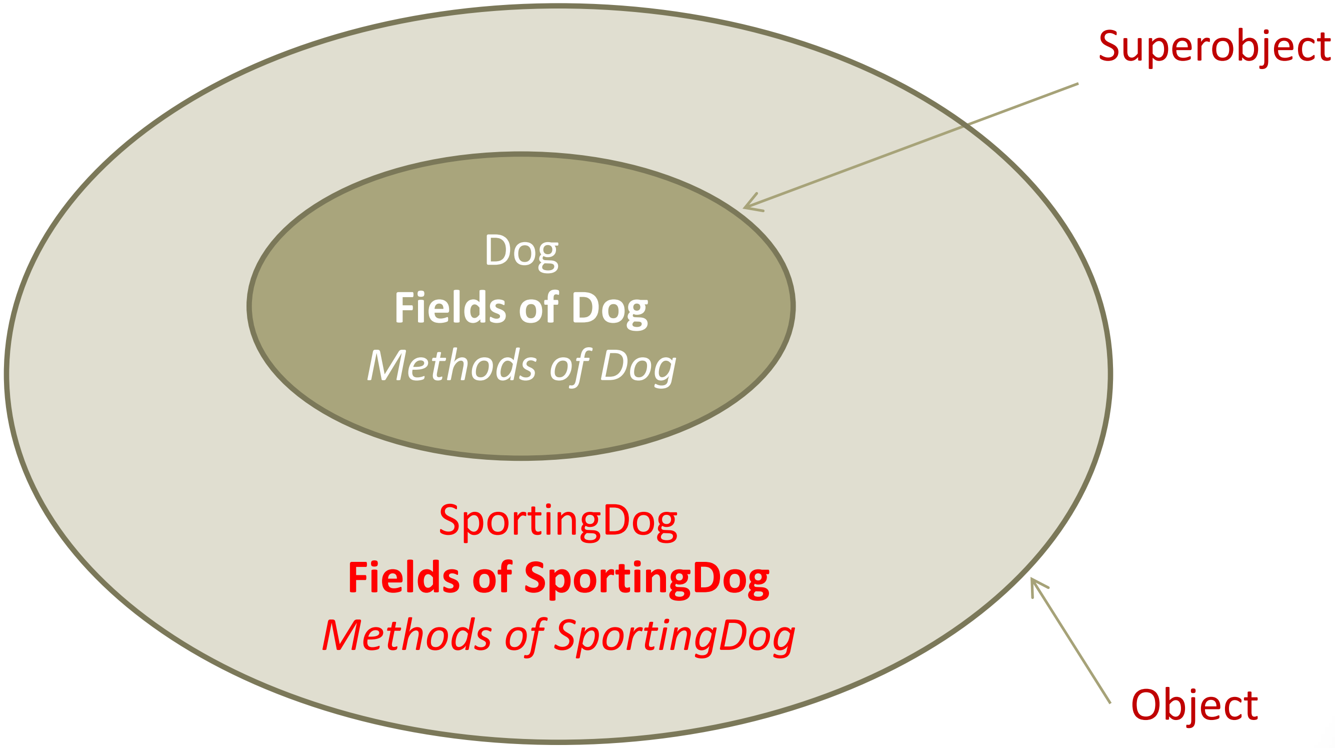
Structure

The superobject has to be initialized first, hence the requirement for a superconstructor call in any constructor.



Structure

The public members of the superobject ‘shine through’ to the outside, along with the public members of the object.



From the outside

- To a client of the subclass, there is no visible difference between the members that are inherited, and the ones specific to the subclass.

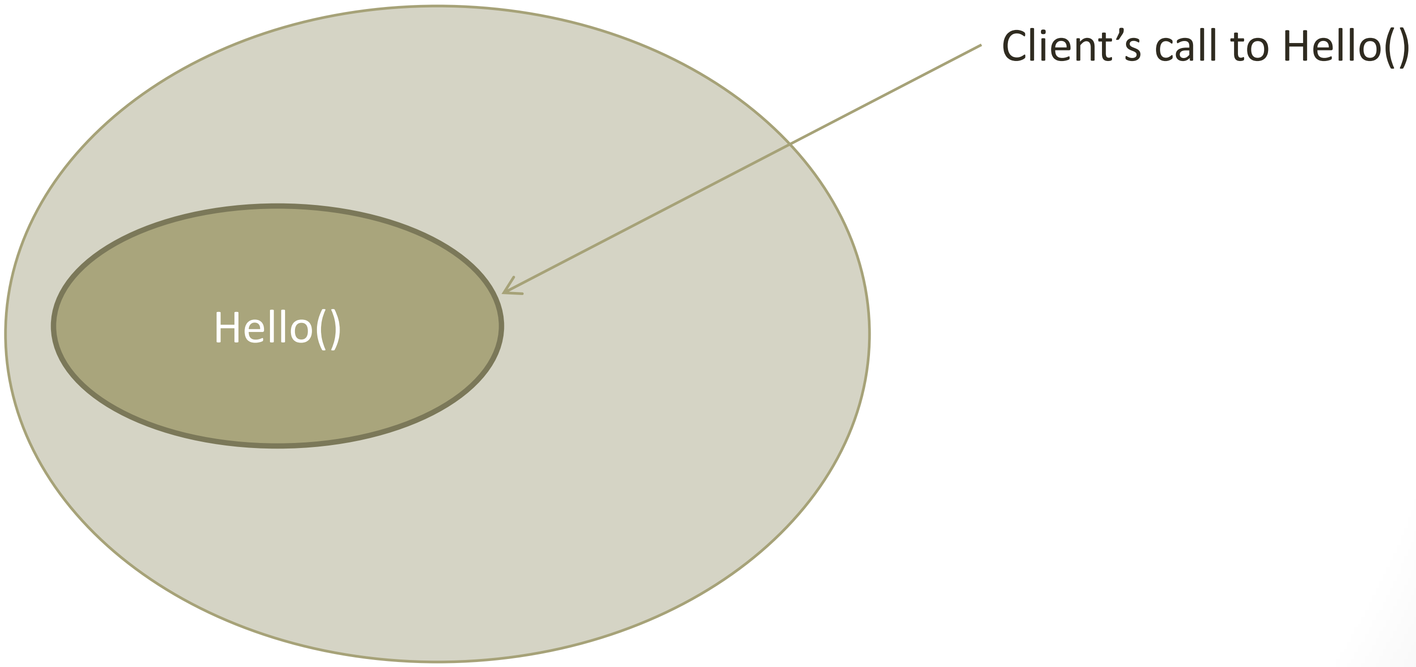
- They all appear to be members of the subclass.

- But whenever you reference an inherited member, the JVM really goes into the superobject to find it.

- Unless it’s an overridden method (or a hidden field), in which case it uses the object’s overridden version of it instead.

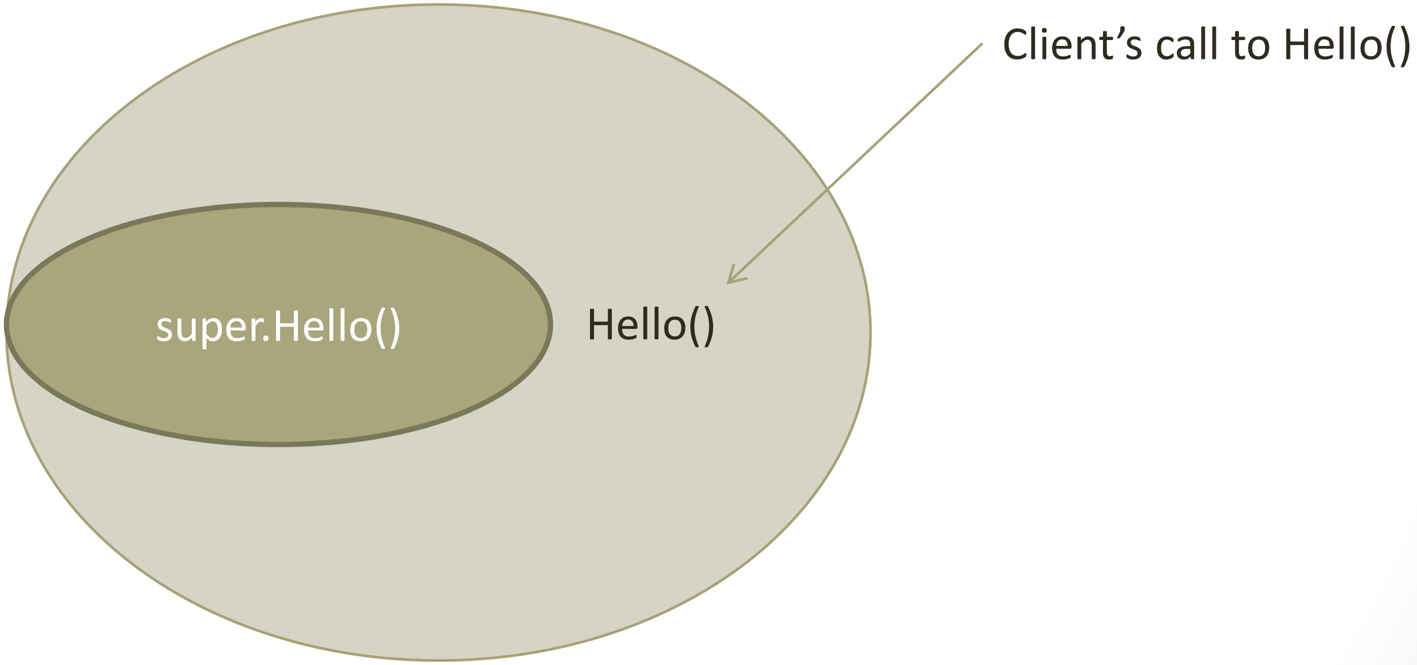
Method calls

- If not overridden, the call transparently goes to the superobject.



Method calls

- If overridden, the call goes to the overridden version.
- Superclass version can ***only*** be accessed internally.



@Override

- You may have seen Eclipse automatically place this text just before an overridden method.

- This is an annotation – a request to the compiler for some extra favors.

- Annotations never affect your code – they exist solely to add extra support for some things.
    - Compiler checking for overriding
    - Suppressing warnings
    - Indicating deprecated elements

@Override

- Suppose you wanted to override the method `Hello()` in SportingDog.

- If you accidentally typed `Hllo()`, the compiler will not know that this was a typo.
  - It'll just think you wanted a method called `Hllo()`.

- However, if you use @Override, it will search the parent class and complain that it has no method by that name.

- This makes writing code a bit safer.

- As a bonus, people reading your code can tell at a glance that a method was overridden.

Annotations

- There are several other annotations, and you can even define your own.

- See http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html if you’re interested.

The Object class

- For generality, Java makes sure that *every* class has a parent.
- Even when they don't explicitly extend anything.
- This universal base class is named `Object`, and itself doesn't have any superclass.
  - To add to the confusion, you can make an object of type Object…
- Anything that doesn't extend a class explicitly is invisibly extending Object.
- Object has a default constructor, which is invisibly called to satisfy the superconstructor call requirement.

Object's methods

- Object has a few methods inherited by every class.
- The methods toString() and equals() are among them.
- The default toString() output that looks like "`Dog@5d0385c1`" is the result of this toString().
- These methods can be overridden, of course. Every time you add a toString() method to a class, you're actually overriding the version inherited from Object.
- http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

Polymorphism

- Because inheritance creates an is-a relationship, *every* class has an is-a relationship to Object.

- No matter what class you write, it can always be stored in an Object reference.
  - Object x = “I’m a String!”;
  - Object y = new Dog();
  - Object z = new Scanner(System.in);

- Remember that is-a is a *transitive* relation, so if:
  - X is-a Y, and Y is-a Z
  - Then X is-a Z

Protected

- I said before that only the public members are inherited.

- Technically, the private members are also inherited – but they are in the superobject, and invisible (they're private after all).

- Sometimes this is inconvenient.
  - There may be members that we want to inherit.
  - But they shouldn't be public.

- The answer is to make them `protected`.

Protected

- Protected members are passed down to the derived class, and are visible to it.

- But they are not visible to any client of the derived class!

- So in effect, they are:
  - public when seen from inside the derived class
  - private when seen from outside of it.

When to use them?

- Private fields that you want derived classes to have free access to, without a getter/setter method.
- Private methods that would provide useful functionality to the derived class, but shouldn’t be exposed to clients.

Final, redux

- We've previously seen the final modifier used on variables, to indicate they are constants.
- It is possible to mark a *method* as final.
- This means that it cannot be overridden in any subclasses!
- Similarly, a *class* can be marked as final too.
- This means that it cannot be extended to make any subclasses.
  - For example, String is final.