COP 3330, Spring 2013

Introduction to Inheritance

Instructor : Arup Ghosh
02-25-13

School of Electrical Engineering and Computer Science
University of Central Florida

Today

- Inheritance
  - Motivation, Intuition, difference from interfaces

Looking like a duck…

- Interfaces provide a basis for polymorphism.
- In simple terms, “If it looks like a duck, it is a duck.”

- If a class X implements interface Y, it *looks like* Y.
  - Looks like = has the same public methods.
  - In fact, methods in an interface are *always* public (you can even leave out the public modifier).
- If X looks like Y, then X is a Y.
  - This is unimaginatively named an is-a relationship.
  - Just means that X also has the type Y.

Acting like a duck…

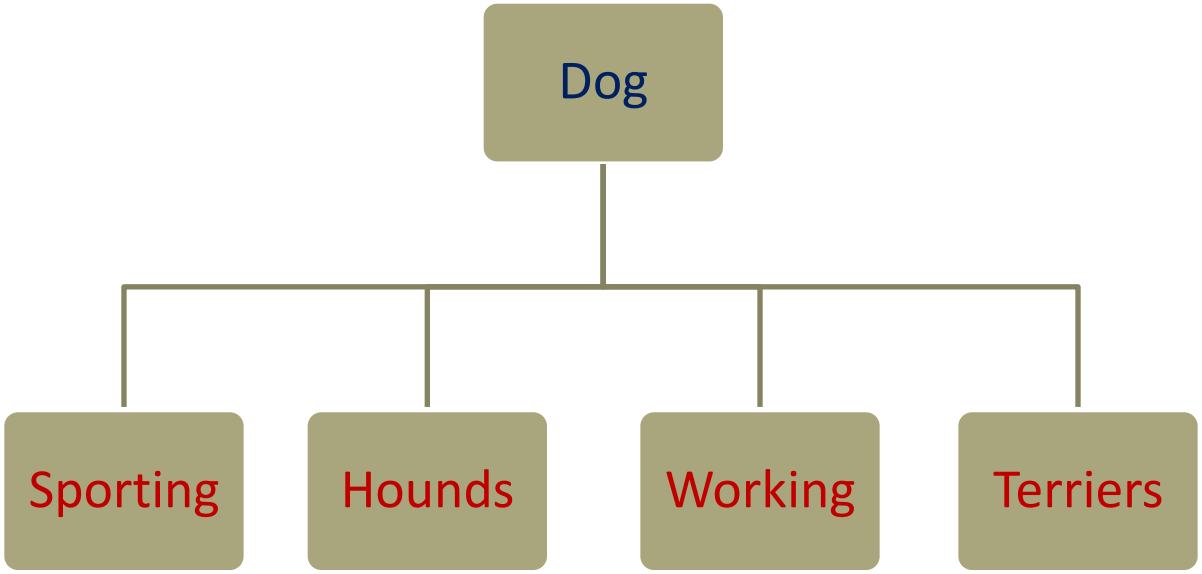
- There is an alternative way to create is-a relationships.
- Build a new type by *extending* an old type.
- The new type *inherits* data and functionality from the old type.
  - So it *acts* like a duck (the type it inherited from)
- It can also *override* some of the old functionality, and add new data and functionality of its own.
  - So it can be somewhat different, while still being a duck.

Dogs

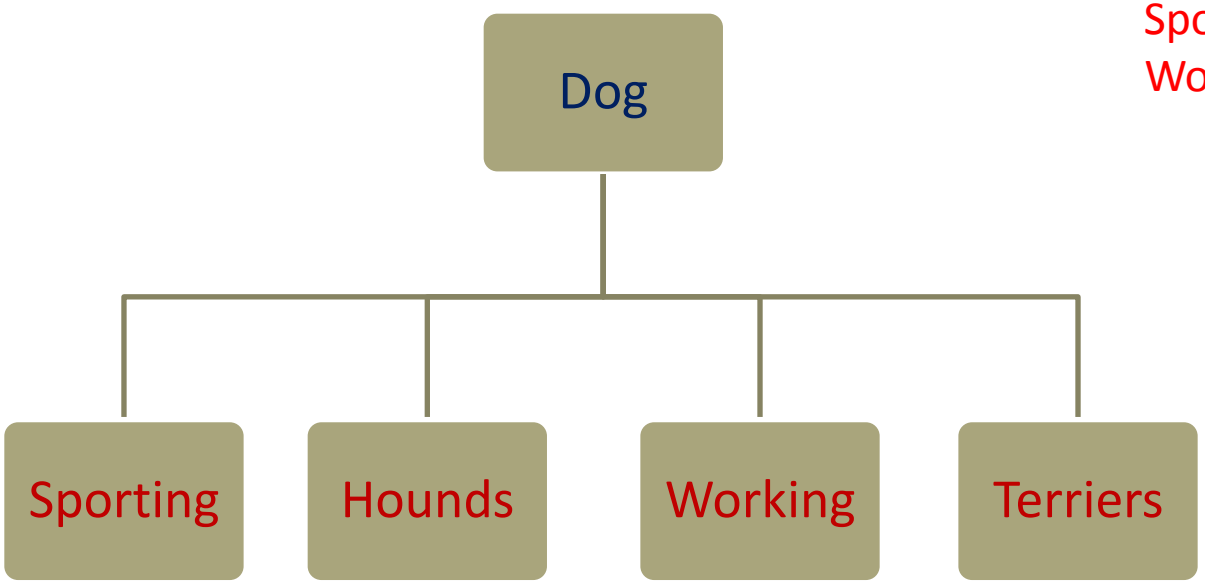
- Let’s say we have a Dog class
- Its fields and methods together define what a Dog is and does.

- This can be a foundation for building more specialized Dogs.

Dogs



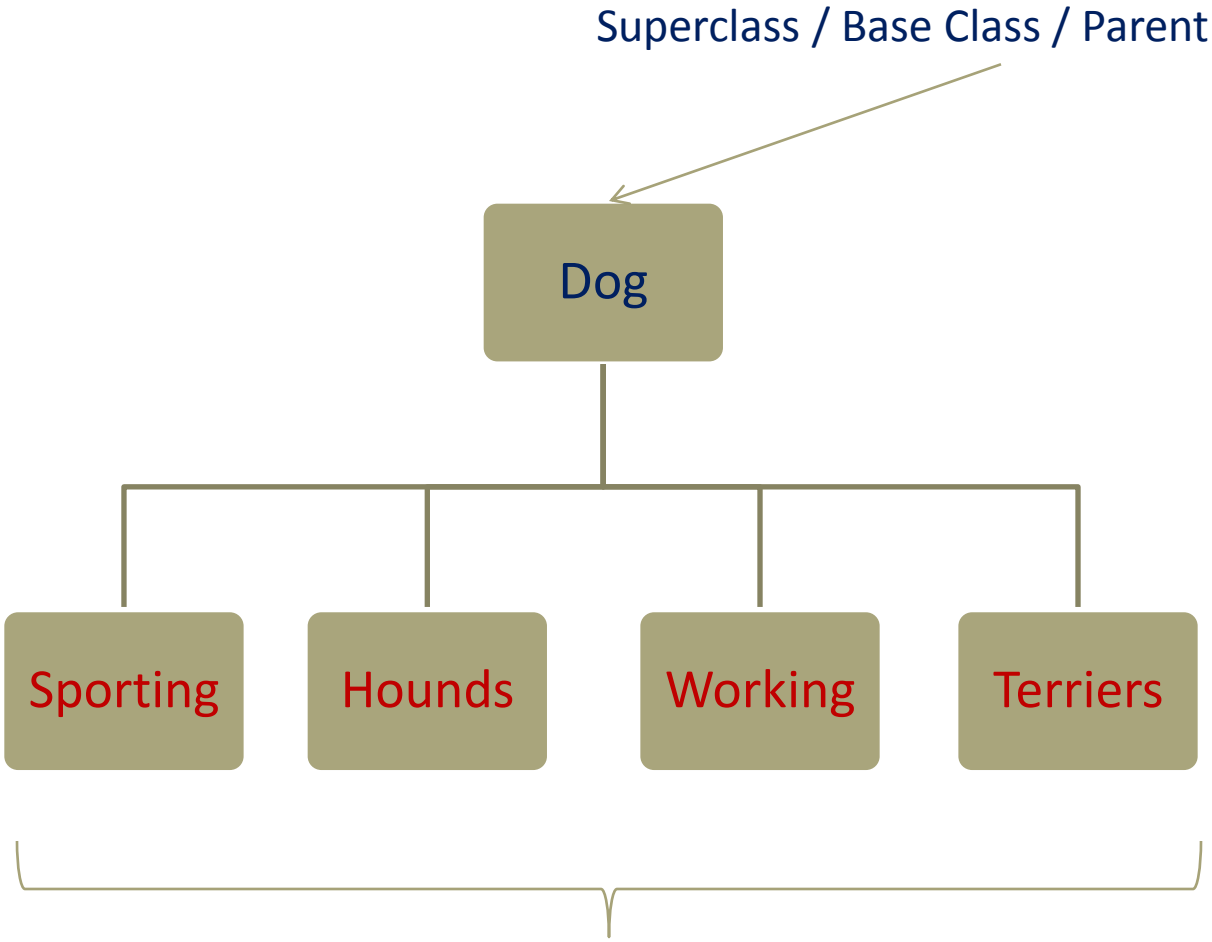
Dogs



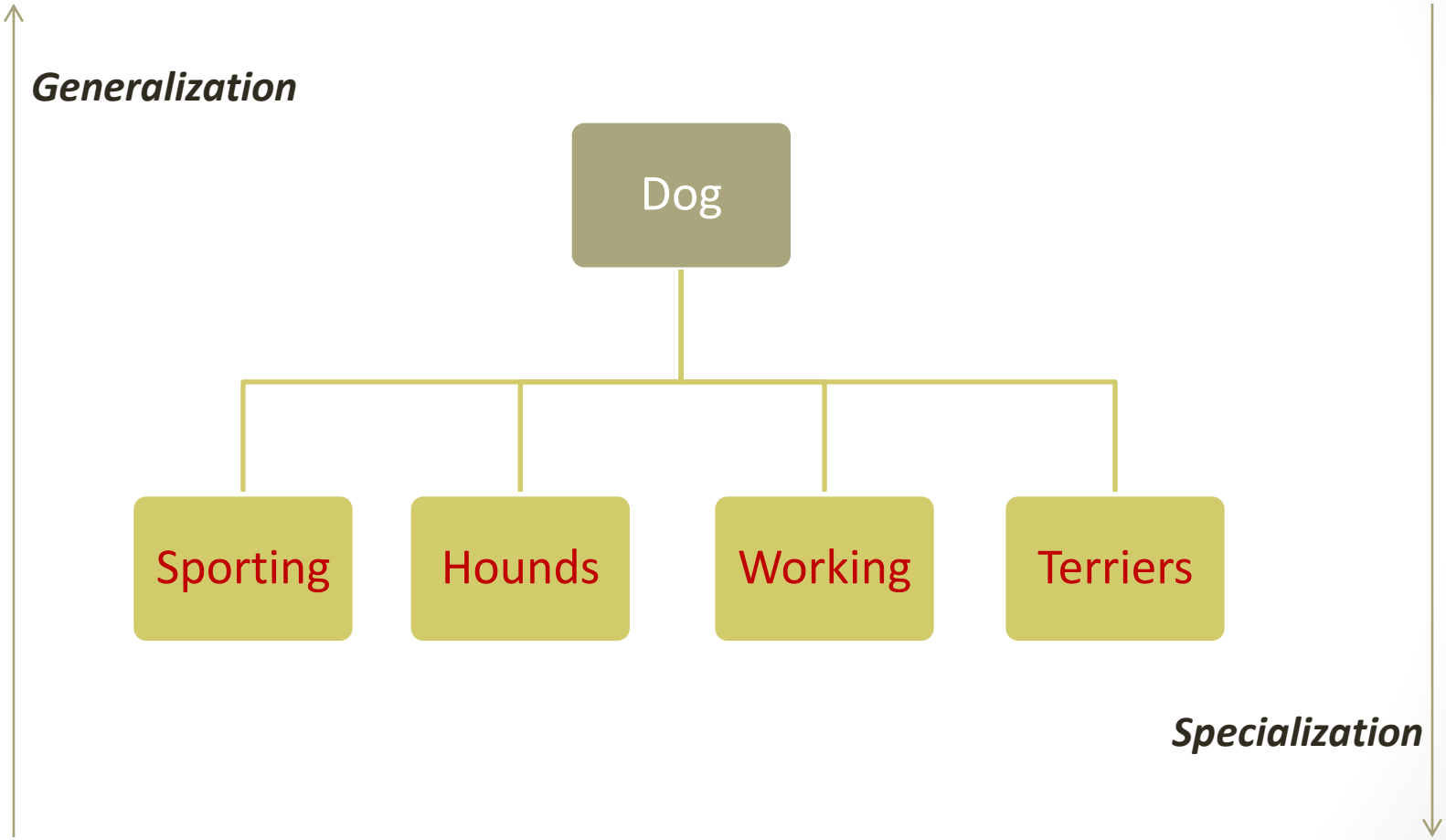
Sporting Dog is-a Dog
Working Dog is-a Dog
…, etc

These are all different *kinds* of Dogs.

Terminology



An inheritance hierarchy



Inheritance

- Subclasses inherit all the public members of the parent class.
  - Actual situation is more complicated, you will see gradually.
- For example, the field *name* and method makeNoise*()* are *passed down* to all the subclasses of Dog.
- "If it looks like a Dog and acts like a Dog…"
- If `racer` is a `SportingDog` object, then this is valid:
  - `racer.makeNoise();`

The *extends* keyword

- To inherit from a class, we use the *extends* keyword.
  - I'll extend Dog to make SportingDog now.
- All the inherited stuff comes along for free – no need to rewrite it.
- We can add new fields and methods!
- For example, SportingDog can have some field
  - So we'll make a field for that.
  - And maybe some methods that work with it.

Making SportingDog

- Before we deal with constructors in subclasses, some deeper understanding is needed.
- Only public members inherited? Not strictly true.
- The private stuff is passed down, but *inaccessible* to the derived class.
  - Same way it's inaccessible to any other class, really – it's private.
- But these private members are part of the derived class, so some special stuff happens.

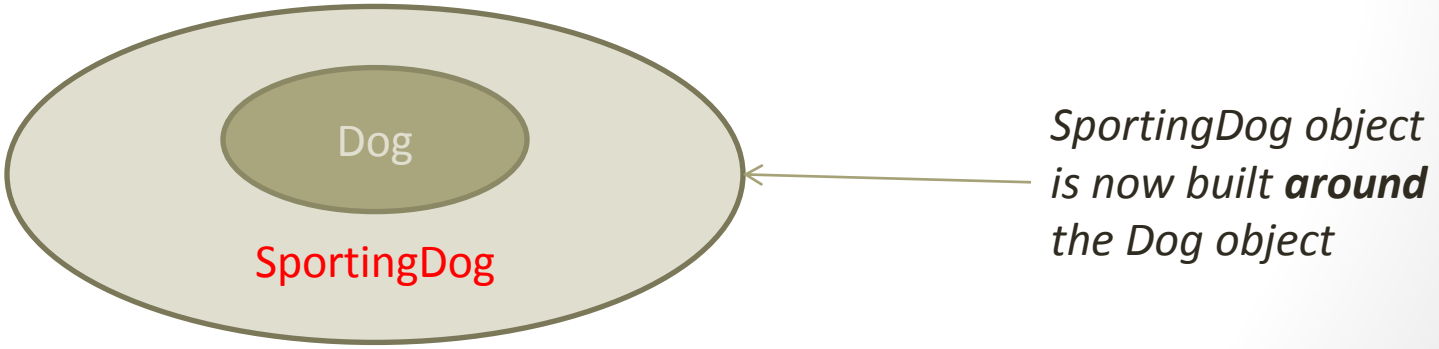
Anatomy of a SportingDog

- To make a SportingDog object, first a *Dog* object is created.
  - This brings along all the members of the Dog object.
  - Think of it as a foundation for the SportingDog.

Anatomy of a SportingDog

- To make a SportingDog object, first a *Dog* object is created.
  - This brings along all the members of the Dog object.
  - Think of it as a foundation for the SportingDog.

- Next, the extra SportingDog-specific members are added.
  - `Sport, displaySport(),` etc.



Anatomy of a SportingDog

- This inner core is sometimes called the *superobject*, and can be referenced using the keyword `super`.
  - It's just like the keyword `this`.
- So to write a SportingDog constructor…
- First, call the Dog constructor, using the super keyword.
  - Initializes all the private stuff that SportingDog can't even see!
- Now initialize the SportingDog specific stuff.
- Always call the superconstructor first!

Default constructors

- There is one exception to this structure:
  - If the parent class has a *default constructor* (i.e., no parameters) then you **don't** need to explicitly call the superconstructor.

- The default superconstructor will get called *implicitly* in such cases.

- Either way, a superconstructor gets called. No escaping that.

Summary

- 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.