How my Dog learned Polymorphism

What follows is an unedited transcript of a tutoring session between myself (CowGirl) and the ranch hound known as Clover. Clover's words are in blue.

So, Clover, let's review from our last lesson...

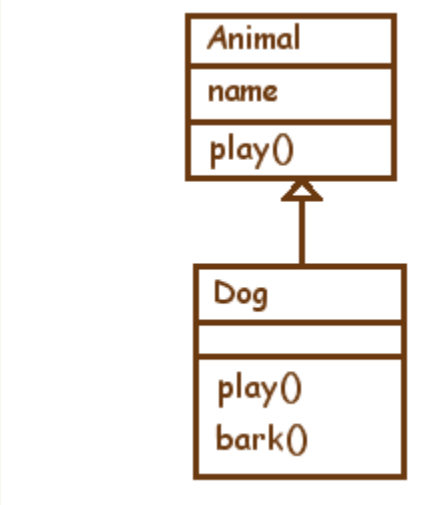
What do you get when you do this:

```
Dog d = new Dog();
```

You get a new Dog object right? An instance of class Dog, referenced by a variable named "d". [tail wags]

Good Girl! Here's a treat...

So let's go beyond that. First, I'm going to show you a class inheritance tree for the example we're going to work on:



This means:

Dog extends Animal
Animal is a superclass of Dog
Dog is a subclass of Animal
Dog inherits from Animal
Dog overrides the play() method
Dog adds one new method, bark()

Ruff!

Yes, very good.
So what happens now if I say:

```
Dog d = new Dog();
d.play();
```

[cocks head sideways]

Which play() method is called -- the one in Dog, or the one inherited from Animal?

Dog overrides the play() method, so you get the play() method in Dog.

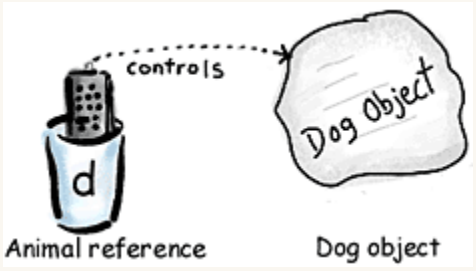
Of course. That's what overriding means. The subclass object has "extended" the superclass by adding more specific functionality to the play() method.

OK, now what if I say...

```
Animal d = new Dog();
```

Is that code even legal? Will it compile?

Yes! Because Dog "is a" Animal. (or so you keep reminding me). Dog is a subclass of Animal, so I know that it's legal to have a reference type which is a superclass of the actual object type. Here, let me draw it [puts marker in mouth]



Wow! I had no idea you could draw. And yes, that's correct. The reference type and the object type don't always match, but the rule is:

If the reference type is a class, the object it refers to MUST be either that same type or a subclass of that type.

If the reference type is an INTERFACE, the object it refers to MUST be an object from a class which implements the interface (either directly or through inheritance).

I understand how it works, but doesn't it "trick" a programmer into thinking that he (or she) has an Animal object when really a Dog object is hiding out there on the heap?

But what's the danger in that? A Dog can do ANYTHING an Animal can do (and more). So there's no harm. A programmer can safely call methods on the Animal reference, as though the object were really an Animal. The object is guaranteed to be able to respond to those method calls.

But let's get to the Big Question:

```
Animal d = new Dog();
d.play();
```

Which play() is really called? The one in Animal, or the one in Dog?

Here's a hint: Java uses "late-binding" for instance methods.

Oh, well, that explains it...

Late-binding means Java doesn't "bind" (think: choose) a method call to an actual method at compile time. The choice of method happens later... at run-time.

Compile time = early, run-time = late. I get it.

Yes, exactly.

So Java waits to see what the ACTUAL object is... rather than using the reference type! Cool. [wags tail again]

You're really a smart dog, Clover. Despite what those sheep say.
And you're right.
Java does the right thing at run-time.
So if a Dog object is at the other end of an Animal reference, you still get the Dog version of the method. Always. It's not like this in C++, where a method has to marked "virtual" if you want to get this "polymorphic" behavior where the method is looked-up at run-time, LATE.

Based on the actual object's class, not the reference type.

But I still have another question... WHY would you want to do this? If you wanted the Dog to play (is now OK?) why not just use a Dog reference? Why use the Animal reference?

Referring to an object in many different ways (reference type is different from actual object class) is the point of polymorphism, which means "many forms". I can treat a Dog object like a Dog, or I can treat it like an Animal, or I can even treat it like a Pet, when I have the Dog class implement the Pet interface.

...and the reason for this would be...

Many reasons. Here's a good one...
You have a program that uses many different animal subclasses, and tells all the animals to play().

You could put all of the different animal subclass objects into an Animal [] array rather than keeping them in separate arrays for each subclass type.

Then you can loop through the Animal [] array and tell each element to play().

And the RIGHT play() method will be called for the REAL object at each slot in the array... am I right?

Yes! So the Compiler is...

.What, no treat? [doing that cute, sad, dog look]

Hang on and let me finish. So the Compiler is happy, since it knows that Animal has a play() method, it knows that each animal in the array can be safely treated like an Animal.

And if any element in the array happens to be a subclass of Animal, and it overrides the play() method, then a different play() method will be called for that element.

Late. The choice of the method is made "late".

At run-time.

So why not just use a generic collection like Vector? Can't you just put any object in there -- they don't even have to be from the same inheritance tree.

You've missed the point. A Vector would be less efficient, for one thing, and you'd have to cast the objects back to something more specific before you could call methods (other than, say, toString()).

What a pain. And that's not the main problem.

What if another programmer comes along after the guy who wrote the Animal program is gone? What if you don't even have his source code, but you need to add a new animal subclass like Cow? If all the different Animal subclasses had to be figured out, and cast, etc. then you could not add new Cow animals into the program without changing the original code.

But if the Animal program is simply expecting an array of Animals, then all I have to do is make my new Cow class a subclass of Animal, then it can be safely included in an Animal [] array and passed to a method in the Animal program.

So that's what you meant about extensibility with object-oriented programming...

Good girl! Yes -- that's part of what makes OOP so cool. OK, here's your treat. Now go write some code for me and I promise to keep the sheep away from your Duke chew toy.

[In Clover's next lesson, we'll be tackling AWT event-handling]