Prototypal Inheritance

If you're coming from a traditional object-oriented language like Java or C++ you're used to what Doug Crockford refers to as "classical inheritance." That is—the language uses classes to organize and reuse bits of functionality; objects are specific instances of classes.

JavaScript doesn't have (or need) classes. You can just create new objects from thin air, no classes required. But what if you want to make several objects that all have the same capabilities without writing the same code over and over for each object? How do you share functionality?

Cheating

The most straight-forward way to solve the problem is just to avoid it altogether. (We've already used this method and it's worked beautifully.) Imagine you want to make a thousand Cat objects without understanding anything about classes or inheritance. You can just make a thousand raw objects with the same guts.

```
var i, cats = []
for( i = 0; i < 1000; i ++ ){
    cats.push({
        id: i,
        meow: function(){ return 'MEOW!' }
    })
}
```

That `for` loop will give you one thousand objects that all meow and have unique ID numbers. It's as if you had a Cat class and made a thousand instances of it.

```
cats[ 29 ].meow()
"MEOW!"
cats[ 29 ].id
29
```

But this is definitely cheating. Works well for simple tasks but it's not very robust. It uses more memory than it should because you've made a thousand *copies* of something instead of just *referencing* it (though in this case you won't notice any performance issues) and if you want to augment or modify the behavior of all these cats you'd have to patch all one thousand of them instead of just mending the thing they inherit from. It's not ideal. It's like … herding cats.

Pretend classical

JavaScript doesn't have classes but it can emulate them using functions—adding properties with the `this` keyword.

```
function Cat(){
    this.cuteness = 1000
    this.meow     = function(){ return 'MEOW!' }
    var nothing   = 'this is really nothing'
}
```

And it really is just a function.

```
typeof Cat
"function"
```

Now we can make a new Cat and it will inherit the properties and functions from our pseudo Cat class.

```
felix = new Cat()
► Cat
typeof felix
"object"
felix.cuteness
1000
felix.meow()
"MEOW!"
felix.nothing
undefined
```

Notice how we must use the `this` keyword in order for a property or method to be inheritable? And of course we can add new custom properties to Felix, like number of whiskers for example.

```
felix.whiskers = 57
```

Adding these custom properties to one object will not alter the class itself or other instances of it.

Playing with prototypes

But let's suppose we did want to alter every instance of Cat all at once—after execution has already begun? Does that even make sense in a classical language like C++ or Java? It can when you think in prototypes. Let's go back to our cat example.

```
function Cat(){
    this.cuteness = 1000
    this.meow = function(){ return 'MEOW!' }
}
```

Let's create two kitties this time.

```
felix = new Cat()
► Cat
fritz = new Cat()
► Cat
```

So we have a pretend class and we've already created two objects that inherit from it. Now let's add a new property to both of them at once.

```
Cat.prototype.cuddly = true
```

Now both of our cats are cuddly!

```
felix.cuddly
true
fritz.cuddly
true
```

Ok, thats great. But what just happened? Let's have a little in class discussion about what prototypes *might be*.

Object, create!

You know what? There's an easier way: `Object.create()`. This method's been kicking around years, but you used to have to code the mechanics of it yourself. (If you're interested in how that works see Doug Crockford's notes on prototypal inheritance.) Fortunately for you this is now baked right into JavaScript. Let's have a look.

```
var Cat = {
    cuteness: 1000,
    meow: function(){ return 'MEOW!' }
}
```

Notice how above we're creating a raw object instead of a function. And now we can easily create child objects that refer to our master Cat object.

```
felix = Object.create( Cat )
► Object
fritz = Object.create( Cat )
► Object
```

We can create custom properties and methods on `felix` or `fritz`—that's not news. What's news is how simple it is update all objects that inherit from `Cat` at once.

```
Cat.cuddly = true
```

There's no need to even mention prototypes because by using `Object.create()` we've already established that there is a one-way relationship between the master object, Cat, and the objects that inherit from it.

```
felix.cuddly
true
fritz.cuddly
true
```

Let's take a moment now to inspect `felix` and `fritz`. What's this `felix.__proto__` stuff all about? What about passing arguments to the pretend constructor? And if you're really looking for a brain workout, what about `Object A` that inherits from `Object B` that inherits from `Object A`?

Further reading

Here are some quick helpful references. It's not homework reading, but having a skim over them and then being able to refer back to them later will be a big help to you.

1. Prototypal Inheritance in JavaScript, Douglas Crockford.
2. Using "Object.create" instead of "new", Stack Overflow.
3. The *this* keyword, Quirksmode.
4. What is the difference between call and apply?, Stack Overflow.
5. Prototype-based programming, Wikipedia.