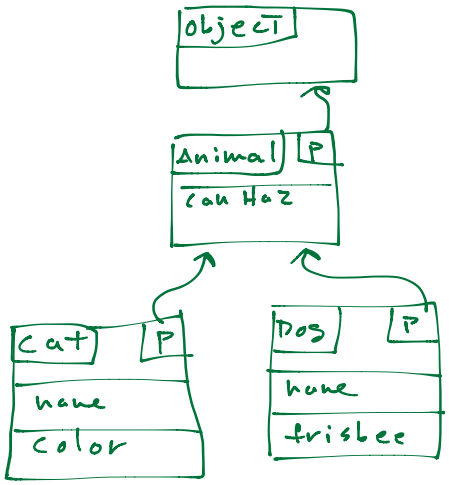
```
function Cat(name, color) {
    this.name = name
    this.color = color
}
function Dog(name, frisbee) {
    this.name = name
    this.frisbee
}
var Animal = {
    canHaz : function(haz) {
        return this.name +
            " can haz " + haz
    }
};
Cat.prototype = Animal
Dog.prototype = Animal
```




```
var c = Cat("spot", "red")
```

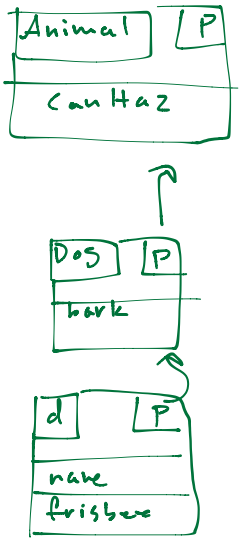
missing new!

```
c === undefined
name === "spot"
color === "red"
```

new vars!

```
var Animal = {
    canHaz : function(haz) {
        return this.name +
            "can haz " + haz
    }
}
var Cat = clone(Animal)
Cat.meow = function() {
    return "meow!"
}
var Dog = clone(Animal)
Dog.bark = function() {
    return "arf!"
}
```

```
var d = clone(Dog)
d.name = "fido"
d.frisbee = false
d.bark()
d.canHaz
```




```
class Animal
    canHaz : (haz) ->
        @name + " can haz " + haz

class Cat extends Animal
    construct : (name, color) ->
        @name = name
        @color = color

class Dog extends Animal
    construct : (@name, @color) ->
```

```
d = new Dog("fido", true)
d.canHaz("advice")
```

```
function clone(base) {
    function F() {}
    F.prototype = base
    return new F()
}
```

```
var Animal = {
      canHaz: function(haz) {
           return this.name + " can haz " + haz
      }
}
var Cat = Animal.extend({
      construct: function(name, color) {
           this.name = name
           this.color = color
      }
     meow: function() {
          return "Meow!"
      }
})
var Dog = Animal.extend({
      construct: function(name, fris) {
           this.name = name
           this.fris = fris
      }
      bark: function() {
          return "Arf"
      }
}
```

```
var d = Dog.create("fido", false)
d.bark()
d.canHaz("advice")
```

```
Object.prototype.create = function() {
     var obj = clone(this)
     if(typeof obj.construct === 'function') {
          obj.construct.apply(obj, arguments)
     }
     return obj
}

Object.prototype.extend = function(properties) {
    var result = clone(this)
    for(var prop in properties) {
       if(properties.hasOwnProperty(prop)) {
           result[prop] = properties[prop]
       }
    }
    return result
}

Object.prototype.hasPrototype = function(proto) {
    function F() {}
    F.prototype = proto
    return this instanceof F
}
```

Monads

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
    wrap v = Just v
    bind (Just v) f = f v
    bind Nothing f = Nothing

wrap :: a -> m a
bind :: m a -> (a -> m b) -> m b
```

```
setName :: () -> Str or null
setPhone :: () -> Str or null
setAddr = () -> Str or null
validatePhone :: Str -> Str or null

val setNameM = function () {
    val result = setName()
    if(result == null) {
        return Nothing.create()
    else
        return Just.create(result)
}
```

```
function signUP() {
    var name = setName()
    var phone = setPhone()
    var addr = setAddr()
    phone = validatePhone(phone)
    return name + addr + phone
}
```

```
var Maybe = {}
Var Nothing = Maybe.extend({})
var Just = Maybe.extend({
    construct: function (val) { this.val = val }
})

Maybe.bind = function(f) {
    if(this.hasPrototype(Just)) {
        return f(this.val)
    else
        return Nothing.create()
}
```

```
function signUP() {
    var name = setName()
    if (name != null) {
        var addr = setAddr()
        if(addr != null) {
            var phone = setPhone()
            if (phone != null) {
                phone = validatePhone(phone)
                if(phone != null) {
                    return name + addr + phone
                }
            }
        }
    }
}
```

```
function signUP() {
    var name = setNameM()
    if(!name.hasPrototype(Nothing)) {
        ...
}
```

```
function signUP() {
return setNameM().bind((name) =>
    setAddrM().bind((addr) =>
    setPhoneM().bind((phone) =>
    validatePhoneM(phone).bind((phone) =>
        Just.create(name + addr + phone) ))))
}
```