Pseudo-classical pattern

Ilya Kantor

Tweet

In pseudo-classical pattern, the object is created by a constructor function and it's methods are put into the prototype.

Pseudo-classical pattern is used is frameworks, for example in Google Closure Library. Native JavaScript objects also follow this pattern.

Pseudo-class declaration

The term *"pseudo-class"* is chosen, because there are actually no classes in JavaScript, like those in C, Java, PHP etc. But the pattern is somewhat close to them.

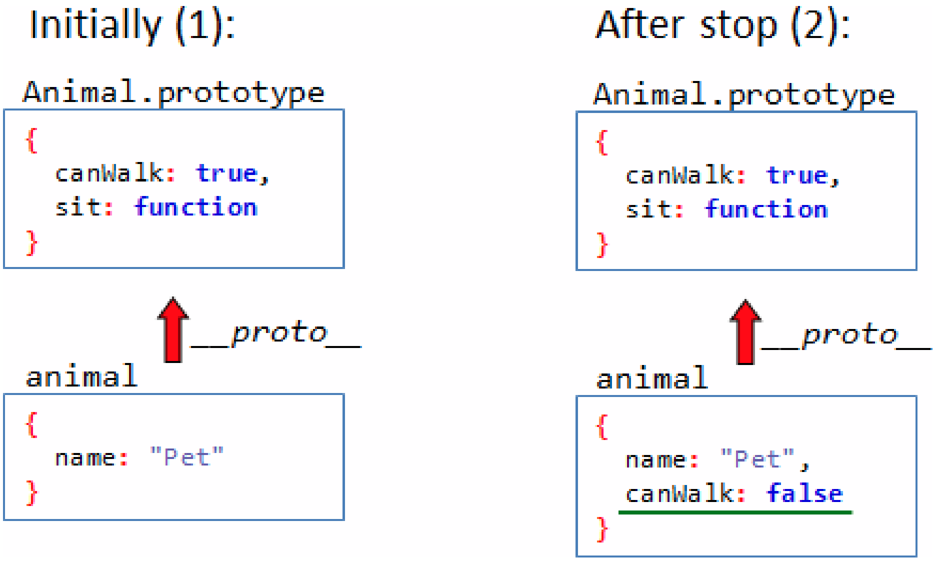
> The article assumes you are familiar with how the prototypal inheritance works.
> That is described in the article Prototypal inheritance .

A *pseudo-class* consists of the constructor function and methods.
For example, here's the `Animal` pseudo-class with single method `sit` and two properties.

F

```
function Animal(name) {
  this.name = name
}

Animal.prototype = {
  canWalk: true,
  sit: function() {
    this.canWalk = false
    alert(this.name + ' sits down.')
  }
}

var animal = new Animal('Pet') // (1)

alert(animal.canWalk) // true

animal.sit()              // (2)

alert(animal.canWalk) // false
```

1. When `new Animal(name)` is called, the new object recieves __proto__ reference to`Animal.prototype`, see that on the left part of the picture.
2. Method `animal.sit` changes `animal.canWalk` in the instance, so now this animal object can't walk. But other animals still can.



The scheme for a pseudo-class:

- Methods and default properties are in prototype.
- Methods in `prototype` use `this`, which is the *current object* because the value of `this` only depend on the calling context, so `animal.sit()` would set `this` to `animal`.

There are dangers in the scheme. See the task below.

You are a team lead on a hamster farm. A fellow programmer got a task to create Hamster constructor and prototype.

Hamsters should have a `food` storage and the `found` method which adds to it.

He brings you the solution (below). The code looks fine, but when you create two hamsters, then feed one of them - somehow, both hamsters become full.

What's up? How to fix it?

F

```
function Hamster() {   }
Hamster.prototype = {
  food: [],
  found: function(something) {
    this.food.push(something)
  }
}

// Create two speedy and lazy hamsters, then feed the first one
speedy = new Hamster()
lazy = new Hamster()

```

```
speedy.found("apple")
speedy.found("orange")

alert(speedy.food.length) // 2
alert(lazy.food.length) // 2 (!??)
```

Solution

Let’s get into details what happens in `speedy.found("apple")`:

1. The interpreter searches `found` in `speedy`. But `speedy` is an empty object, so it fails.
2. The interpreter goes to `speedy.__proto__ (==Hamster.prototype)` and luckily gets`found` and runs it.
3. At the pre-execution stage, `this` is set to `speedy` object, because of dot-syntax:`speedy.found`.
4. `this.food` is not found in `speedy`, but is found in `speedy.__proto__`.
5. The “apple” is appended to `speedy.__proto__.food`.

Hamsters share the same belly! Or, in terms of JavaScript, the `food` is modified in`__proto__`, which is shared between all hamster objects.

Note that if there were a simple assignment in `found()`, like `this.food = something`, then step 4-5 would not lookup `food` anywhere, but assign `something` to `this.food` directly.

Fixing the issue

To fix it, we need to ensure that every hamster has it’s own belly. This can be done by assigning it in the constructor:

F

```
function Hamster() {
  this.food = []
}
Hamster.prototype = {
  found: function(something) {
    this.food.push(something)
  }
}

speedy = new Hamster()
lazy = new Hamster()

speedy.found("apple")
speedy.found("orange")

alert(speedy.food.length) // 2
alert(lazy.food.length) // 0(!)
```

Inheritance

Let’s create a new class and inherit it from `Animal`.

Here you are.. A Rabbit!

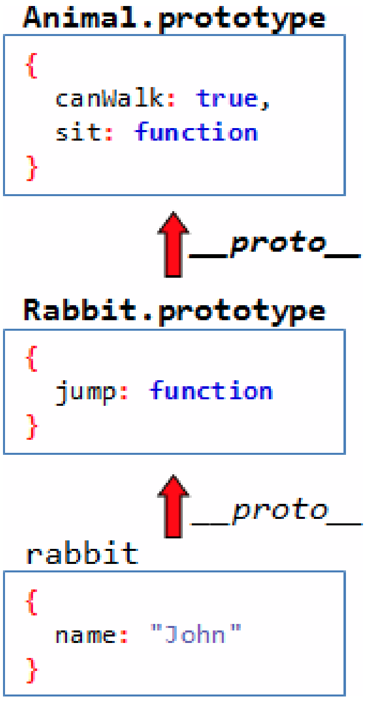
```
function Rabbit(name) {
  this.name = name
}
```

```

Rabbit.prototype.jump = function() {
  this.canWalk = true
  alert(this.name + ' jumps!')
}

var rabbit = new Rabbit('John')
```

As you see, the same structure as `Animal`. Methods in prototype.

To inherit from `Animal`, we need `Rabbit.prototype.__proto__ == Animal.prototype`. This is a very natural requirement, because if a method is not find in `Rabbit.prototype`, it should be searched in the parental method store, which is `Animal.prototype`.

That's how it should look like:



To implement the chain, we need to create initial `Rabbit.prototype` as an empty object inheriting from`Animal.prototype` and *then* add methods.

```
function Rabbit(name) {
  this.name = name
}

Rabbit.prototype = inherit(Animal.prototype)

Rabbit.prototype.jump = function() { ... }
```

`inherit`

In the code above, `inherit` is a function which creates an empty object with given`__proto__`.

```
function inherit(proto) {
  function F() {}
  F.prototype = proto
  return new F
}
```

See Prototypal inheritance for details.

And finally, the full code of two objects:

F

```
// Animal
function Animal(name) {
  this.name = name
}

// Animal methods
Animal.prototype = {
  canWalk: true,
  sit: function() {
    this.canWalk = false
    alert(this.name + ' sits down.')
  }
}

// Rabbit
function Rabbit(name) {
  this.name = name
}

// inherit
Rabbit.prototype = inherit(Animal.prototype)

// Rabbit methods
Rabbit.prototype.jump = function() {
  this.canWalk = true
  alert(this.name + ' jumps!')
}

// Usage
var rabbit = new Rabbit('Sniffer')

rabbit.sit()   // Sniffer sits.
rabbit.jump()  // Sniffer jumps!
```

Don't create new Animal to inherit it

There is a well-known, but *wrong* way of inhereting, when instead of `Rabbit.prototype = inherit(Animal.prototype)` people use:

```
// inherit from Animal
Rabbit.prototype = new Animal()
```

As a result, we get a `new Animal` object in `prototype`. Inheritance works here, because`new Animal` naturally inherits `Animal.prototype`.

… But who said that `new Animal()` can be called like without the `name`? The constructor may strictly require arguments and die without them.

Actually, the problem is more conceptual than that. **We don’t want to create an `Animal`. We just want to inherit from it.**

That’s why `Rabbit.prototype = inherit(Animal.prototype)` is preferred. The neat inheritance without side-effects.

Calling superclass constructor

The “superclass” constructor is not called automatically. We can call it manually by applying the `Animal`function to current object:

```
function Rabbit(name) {
  Animal.apply(this, arguments)
}
```

That executes `Animal` constructor in context of the current object, so it sets the `name` in the instance.

Overriding a method (polymorphism)

To override a parent method, replace it in the prototype of the child:

```
Rabbit.prototype.sit = function() {
  alert(this.name + ' sits in a rabbity way.')
}
```

A call to `rabbit.sit()` searches `sit` on the chain `rabbit -> Rabbit.prototype -> Animal.prototype`and finds it in `Rabbit.prototype` without ascending to `Animal.prototype`.

Of course, we can even more specific than that. A method can be overridden directly in the object:

```
rabbit.sit = function() {
  alert('A special sit of this very rabbit ' + this.name)
}
```

Calling a parent method after overriding

When a method is overwritten, we may still want to call the old one. It is possible if we directly ask parent prototype for it.

F

```
Rabbit.prototype.sit = function() {
  alert('calling superclass sit:')
  Animal.prototype.sit.apply(this, arguments)
}
```

All parent methods are called with `apply/call` to pass current object as `this`. A simple call`Animal.prototype.sit()` would use `Animal.prototype` as `this`.

Sugar: removing direct reference to parent

In the examples above, we call parent class directly. Either it’s constructor: `Animal.apply...`, or methods:`Animal.prototype.sit.apply...`.

Normally, we shouldn’t do that. Refactoring may change parent name or introduce intermediate class in the hierarchy.

Usually programming languages allow to call parent methods using a special key word, like *parent*`.method()` or *super()*.

JavaScript doesn’t have such feature, but we could emulate it.

The following function `extend` forms inheritance and also assigns `parent` and `constructor` to call parent without a direct reference:

```
function extend(Child, Parent) {
  Child.prototype = inherit(Parent.prototype)
  Child.prototype.constructor = Child
  Child.parent = Parent.prototype
}
```

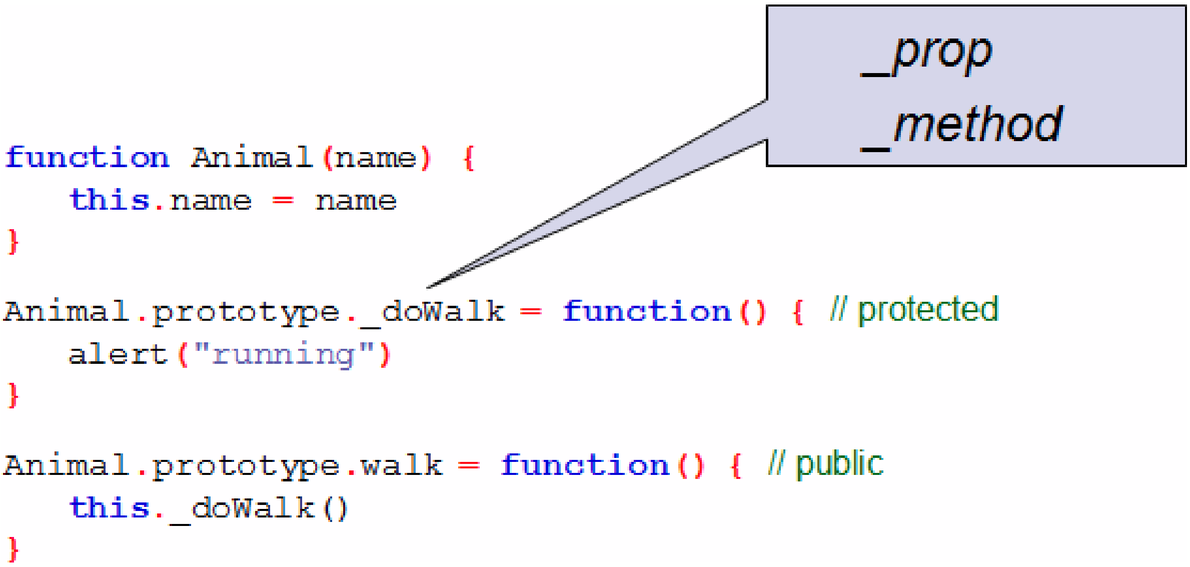
Usage:

```
function Rabbit(name) {
  Rabbit.parent.constructor.apply(this, arguments) // super constructor
}

extend(Rabbit, Animal)

Rabbit.prototype.run = function() {
    Rabbit.parent.run.apply(this, arguments) // parent method
    alert("fast")
}
```

As the result, we can now rename `Animal`, or create an intermediate class `GrassEatingAnimal` and the changes will only touch `Animal` and `extend(...)`.

Private/protected methods (encapsulation)

Protected methods and properties are supported by naming convention. So, that a method, starting with underscore '_' should not be called from outside (technically it is callable).



```
function Animal(name) {
    this.name = name
}

Animal.prototype._doWalk = function() { // protected
    alert("running")
}

Animal.prototype.walk = function() { // public
    this._doWalk()
}
```

Private methods are usually not supported.

Static methods and properties

A static property/method are assigned directly to constructor:

F

```
function Animal() {
  Animal.count++
}
Animal.count = 0

new Animal()
new Animal()

alert(Animal.count) // 2
```

Summary

And finally, the whole suppa-mega-oop framework.

```
function extend(Child, Parent) {
  Child.prototype = inherit(Parent.prototype)
  Child.prototype.constructor = Child
  Child.parent = Parent.prototype
}
function inherit(proto) {
  function F() {}
  F.prototype = proto
  return new F
}
```

Usage:

F

```
// --------- the base object ------------
function Animal(name) {
  this.name = name
}

// methods
Animal.prototype.run = function() {
  alert(this + " is running!")
}

Animal.prototype.toString = function() {
  return this.name
}


// --------- the child object -----------
function Rabbit(name) {
  Rabbit.parent.constructor.apply(this, arguments)
}

// inherit
extend(Rabbit, Animal)

// override
Rabbit.prototype.run = function() {
  Rabbit.parent.run.apply(this)
  alert(this + " bounces high into the sky!")
}

var rabbit = new Rabbit('Jumper')
rabbit.run()
```

Frameworks may add a bit more sugar, like function `mixin` which copies many properties from one object to another:

```
mixin(Animal.prototype, { run: ..., toString: ...})
```

But in fact you don’t need much to use this OOP pattern. Just two tiny functions will do.