All-in-one constructor pattern

Ilya Kantor

Tweet

All methods and properties of the object can be added in the constructor. This method doesn't use`prototype` at all.

Declaration

The object is declared solely by it's constructor. As an example, let's build the `Animal` with property `name`and method `run`:

F
```
function Animal(name) {
  this.name = name
  this.run = function() {
    alert("running "+this.name)
  }
}

var animal = new Animal('Foxie')
animal.run()
```

As you see, properties and methods are assigned to `this`. As the result, we have a full object.

Inheritance

To create a `Rabbit`, inheriting from `Animal`:

1. First apply `Animal` constructor to `this`. We've got an `Animal`
2. Modify `this`, add more methods to get a `Rabbit`.

For example:

F
```
function Rabbit(name) {

  Animal.apply(this, arguments) // inherit

  this.bounce = function() {
    alert("bouncing "+this.name)
  }
}

rabbit = new Rabbit("Rab")

rabbit.bounce() // own method

rabbit.run()    // inherited method
```

The superclass constructor is called automatically when inhereting. There is no non-ugly way to first inherit, then do something, and then call the superclass constructor.

We can call constructor with custom parameters too:

F

```
function Rabbit(name) {
  Animal.call(this, "Mr. " + name.toUpperCase())
  // ..
}

rabbit = new Rabbit("Rab")

rabbit.run()
```

At the end of `new` call we always have a single object with both own and parent methods in it. The`prototype` is not used at all.

Overriding (polymorphism)

Overriding a parent method is as easy as overwriting it in `this`. Of course we may want to copy the old method and call it in the process.

F

```
function Rabbit(name) {

  Animal.apply(this, arguments)

  var parentRun = this.run  // keep parent method

  this.run = function() {
    alert("bouncing "+this.name)
    parentRun.apply(this)  // call parent method
  }
}

rabbit = new Rabbit("Rab")

rabbit.run()    // inherited method
```

Here we use `apply` to provide right `this`.

Private/protected methods (encapsulation)

Private methods and properties are supported really good in this pattern.

A local function or variable are private. All constructor arguments are private automatically.

That's because all functions created in the scope of `Rabbit` can reference each other through closure, but the outside code can only access those assigned to `this`.

In the example below, `name` and `created` are private properties, used by private method `sayHi`:

F

```
function Rabbit(name) {

  Animal.call(this, "Mr. " + name.toUpperCase())

  var created = new Date()  // private

  function sayHi() {  // private
    alert("I'm talking rabbit " + name)
  }

  this.report = function() {
    sayHi.apply(this)
    alert("Created at " + created)
  }
}

rabbit = new Rabbit("Rab")

rabbit.report()
```

It is quite inconvenient to use call methods through `apply`, like `sayHi.apply(this)`. So, the functions are usually bound to the object. Read more about that in Early and Late Binding

Local variables/functions become private, not protected. A child can't access them:

```
function Animal() {
  var prop = 1
}

function Rabbit() {
  Animal.call(this) // inherit
  /* can't access prop from here */
}
```

Protected properties are implemented same way as in pseudo-classical approach. That is, by naming convention: "`_prop`".

F

```
function Animal() {
  this._prop = 'test'  // protected
}

function Rabbit() {
  Animal.call(this) // inherit
  alert(this._prop) // access
}

new Rabbit()
```

Summary

- The object is fully described by it's constructor.
- Inheritance is done by calling the parent constructor in the context of current object.
- All local variables/functions become private, all assigned to `this` become public. Local functions are usually bound to the object.
- Protected properties can be prepended with underscore `'_'`, but their protection can't be forced on language level.
- Overriding is done as replacing the property in `this`. The old property may be copied and reused.

Comparison with pseudo-classical pattern

- `rabbit instanceof Animal` doesn't work here. That's because `Rabbit` does not inherit from `Animal` in prototype-sense.

- Slower creation, more memory for methods, because every object carries all methods in it, without prototype as shared storage. But on frontend programming we *shouldn't*create many objects. So that's not a big problem.
- Inheritance is joined with parent constructor call. That's architectural inflexibility, because one can't call parent method before parent constructor. Not so fearful though.
- Private methods and properties. That's safe and fast. Especially because JavaScript compressors shorten them.
- There are no "occasionaly static" properties in prototype.

Actually, we have minor problems and advantages. Choose the method depending on how they refer to your application.

‹ Pseudo-classical pattern