Factory constructor pattern

Ilya Kantor

Tweet

This pattern is special, because it doesn’t use `"new"`.
The object is created by a simple function call, similar to *Python-style*:

```
var animal = Animal("fox")
var rabbit = Rabbit("rab")
```

~~new~~

Declaration

The constructor is defined as a function which returns a new object:

```
function Animal(name) {

  return {
    run: function() {
      alert(name + " is running!")
    }
  }

}
```

Usage:

F
```
var animal = Animal("fox")
animal.run()
```

Inheritance

`Rabbit` is made by creating an `Animal`, and then mutating it:

F
```
function Rabbit(name) {

  var rabbit = Animal(name) // make animal

  rabbit.bounce = function() { // mutate
    this.run()
    alert(name + " bounces to the skies! :)")
  }

  return rabbit // return the result
```

```
}

var rabbit = Rabbit("rab")
rabbit.bounce()
```

Private/protected methods (encapsulation)

Local variables and functions become private:

```
function Bird(name) {

  var speed = 100                      // private prop
  function openWings() { /* ... */ } // private method

  return {
    fly: function() {
      openWings()
      this.move()
    },
    move: function() { /*...*/ }
  }
}
```

The code above looks simple, but still there is a gotcha.

A public method can be called as `this.move()` from another public method, but *not* from a private method.

A private method like `openWings` can't reference `this`. There's no reference to the new object in a local function.

One way to solve that is to bind the new object to a local variable prior to returning:

```
function Bird(name) {

  function doFly() {
    openWings()
    self.move()
  } // private method


  var self ={
    fly: function() { doFly() },
    move: function() { /*...*/ }
  }
  return self
}
```

Summary

- The *factory constructor* uses a function which creates an object on it's own without `new`.
- Inheritance is done by creating a parent object first, and then modifying it.
- Local methods and functions are private. The object must be stored in closure prior to returning if we want to access it's public methods from local ones.

Comparison with All-in-one constructor

The two methods: “All-in-one constructor” and the “Factory constructor”, described here, are actually the same.

Compare the two code pieces below. How similar they are.

```
function Animal(name) {
    //…
}

function Rabbit(name) {
    var rabbit = Animal(name)

    var parentRun = rabbit.run

    rabbit.jump = function() {
      alert(name + " jumped!")
    }

    rabbit.run = function() {
      parentRun.call(this)
      alert("fast")
    }

    return rabbit
}

rabbit = Rabbit("rab")
```

```
function Animal(name) {
    // …
}

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

  var parentRun = this.run

  this.jump = function() {
    alert(name + " jumped!")
  }

  this.run = function() {
    parentRun.call(this)
    alert("fast")
  }
}

rabbit = new Rabbit("rab")
```

The result of both codes is same: they create a `rabbit` object with all methods assigned to it.

Initially, the object is created by literal on the left, and by `new` (as `this`) on the right.

Inheritance is performed similarly. The factory method uses `rabbit = Animal()` to get the parent object as`rabbit`. The all-in-one constructor uses `Animal.apply(this, arguments)` to get parent as `this`.

The only minor difference is syntax. Choose the one you’d prefer.

‹ All-in-one constructor pattern