Programación por prototipos

Junio 2017, repaso

Repaso de objetos en Javascript

Objeto literal:

```
var literalObj = {
      name : 'myObjectName',
      lastname : 'myObjectLastName',
      sayName: function(){ return this.name +' '
+ this.lastname}
};
=============================================
var myObj = {};
myObj.name='myObjectName';
myObj.lastname='myObjectLastName';
myObj.sayName: function(){ return this.name +' '
+ this.lastname};
```

Con función constructora:

```
function Person(name, lastname) {
        this.name = name,
        this.lastname = 'myObjectLastName',
        this.sayName= function(){ return this.name +' ' +
this.lastname
}
var person = new Person( 'juan' , 'perez' );
person.sayName(); // "juan perez"
person.constructor // function Person(name) {...}
```

Encapsulamiento

¿Cómo hacemos para controlar el acceso?

```
delete myObj.name // true
myObj.name = ‘jaja’
myObj.sayName() // jaja myObjectLastName
```

Sería deseable que name no pueda ser eliminado ni modificado

Encapsulamiento

¿Cómo hacemos para controlar el acceso?

```
delete myObj.name // true
myObj.name = ‘jaja’
myObj.sayName() // jaja myObjectLastName

Sería deseable que name no pueda ser eliminado ni modificado

function Person(name) {
      var name = name;
      this.sayName = function(){ return name;}
};
```

Funciones

- Son objetos
- Tienen siempre un **prototype**
- Usadas como constructor, crean nuevos objetos que comparten el prototype.

Funciones

```
function PersonWithPrototype(name, lastname) {
     var name = name;
     var lastname = lastname;
     this.sayName= function(){ return name + ' ' + lastname; }
};
var p3 = new PersonWithPrototype("pepe", "argento");
var p4 = new PersonWithPrototype("pepe", "argento2");
p3.sayName(); //"pepe argento"
p4.sayName(); // "pepe argento2"
p3.family // undefined
PersonWithPrototype.prototype.family = "test" // "test"
p3.family // "test"
p4.family // "test"
PersonWithPrototype.prototype.family = "prod" // "prod"
p3.family // "prod"
p4.family // "prod"
```

Patrones de reutilización de código

Herencia clásica

- Los objetos son creados bajo una función constructora ‘hija’ y tienen las propiedades que provienen de otra función constructora ‘padre’.
  → Child() debería crear objetos con las propiedades de Parent().

Patrón N° 1

```
function inherit ( C, P) {
        C.prototype = new P();
}
```

- heredan tanto las propiedades como las propiedades del prototipo y métodos.
- **Pero**, por lo general solamente queremos heredar las propiedades del prototipo (ver más patrones en la bibliografía)

Herencia por prototipos

- Los objetos heredan de otros **objetos**.
- Se reutiliza código a partir de un **objeto existente**.

Herencia por prototipos

```
function Person() {
        this.name: ”Adam”
};
Person.prototype.getName = function() {
        return this.name;
}
var child = object(person.prototype)
typeof kid.getName // “function”
typeof kid.name //”undefined”
```

```
function object(o) {
        function F(){}
        F.prototype = o
        return new F();
}
```

Buenas noticias!..

En ECMAScript 5 esta herencia viene incluida en el lenguaje bajo la función:
Object.create() ← “function object(o)...

var child = Object.create(Person.prototype)

¡y acepta parámetros adicionales!
var child = Object.create(parent, { age: { value: 2}});
child.hasOwnProperty(“age”) // true

Bibliografía

- JavaScript Patterns Build Better Applications with Coding and Design Patterns By Stoyan Stefanov - Capítulo 6