[Resumen] Javascript Orientado a Objetos.

Si bien la mayoría de los desarrolladores utilizan a Javascript solamente a través de algunas funciones y no más que algunas llamadas durante ciertos eventos, lo cierto es que Javascript es mucho más que eso. Vamos a ser sinceros, no es un lenguaje orientado a objetos como los que estamos acostumbrados utilizar, con el concepto universalmente aceptado de "Clases" e "Instancias". En Javascript las cosas son un tantito diferentes.

Javascript fue originalmente concebido como un lenguaje de programación funcional, y no un lenguaje típicamente orientado a Objetos, porque vamos, modas son modas y en ese entonces estaban muy de moda los lenguajes al estilo C (C, C++, Java, el que prefieran). Esto causó dos modificaciones en el lenguaje. La aparición del concepto de Objeto - junto a la herencia prototípica-, y el operador "new".

Así, Javascript es muy confuso al comienzo para muchos desarrolladores, sobre todo aquellos que no han programado mucho, por pensar que están en un mundo, y en realidad se encuentran en otro. Pero comprendiendo solo un par de conceptos clave, se podrá entender mucho mejor este lenguaje.

Solo tengan en cuenta que este es un artículo inicial, que solo tocará varios de estos temas muy por encima y será necesario más investigación por parte del estudiante de este lenguaje, para comprender más profundamente estos conceptos.
Como nota aclaratoria, vamos a estar utilizando Javascript puro para los ejemplos, sin ningún agregado extra. O sea que si solo sabes utilizar jQuery (o alguna librería como ésta) y nunca has programado solamente en Javascript, esta es una buena oportunidad para que veas un poco más a fondo este lenguaje.

El Objeto Literal

Un Objeto, en Javascript, difiere respecto a otros lenguajes, a que no necesita ser la instancia de una Clase. Simplemente es una estructura de propiedades/valor. Nada más. Nada menos. Muy similar a un Array, solo que éste último solo acepta índices numéricos, cuando el Objeto en Javascript acepta "cadenas de texto" (strings) como claves. Y como valor, podemos asignar otras cadenas, números, constantes, funciones (aquí es donde comenzamos con el concepto de "método") y hasta otros objetos. Un objeto se puede declarar de varias maneras en Javascript. Vamos a ver la más simple de todas:

```
var circulo   = {};
circulo.radio = 9;
circulo.color = 'red';
```

Pero también podemos definir todos los valores de un Objeto de una manera más directa:

```
var circulo = {
    radio : 9,
    color : 'red'
}
```

Métodos

Cómo habíamos dicho, un Objeto puede contener como valor, una función o una referencia a una función. Como sabemos, en Javascript una función es un elemento de "primer nivel", o sea, pueden ser guardadas dentro de variables. Veamos como entra esto en juego cuando hablamos de Objetos guardando funciones.

```
function calcularArea(radio) {
    return radio * radio * Math.PI;
}
```

```
var circulo = {
    radio : 9,
    area : calcularArea(this.radio)
}
```

De esta manera, el objeto circulo posee un atributo llamado area que contiene el resultado de llamar a la función calcularArea. Esto es... guarda el RESULTADO de la función, pero no la función en si. Para que el objeto guarde la función, podemos hacer lo siguiente:

```
var circulo = {
    radio: 9,
    area : calcularArea,
}
```

Así, para conocer el área de nuestro Objeto Círculo, simplemente llamamos al método que acabamos de crear.

```
var area = circulo.area(circulo.radio);
```

¿No les suena que hay algo raro acá? Si definimos el método area como la referencia a una función, entonces, nuestro método no es más que una función global! No nos sirve tenerla de esta manera. Por eso, vamos a definir los métodos como realmente corresponde. Y aquí es donde entra el concepto de this.

```
var circulo = {
    radio : 9,
    area : function() {
        return this.radio * this.radio * Math.PI;
    }
}
```

¿Qué función tiene el operador this aquí?

Imaginen a this como un parámetro implícito que se le pasa a cada función, indicando en su valor, al contexto en el que se encuentra la función. En este

caso del ejemplo, this obtiene como valor una referencia el objeto circulo. Por eso cuando nos estamos refiriendo athis.radio Javascript sabe que no nos referimos a una variable radio global, sino a la que está definida dentro de this.

Cuando definimos una función global, como el calcularArea del ejemplo anterior, la variable thisapunta al contexto global, que es lo mismo que window. ¿Cómo pueden chequear esto rápidamente? Abran la consola de alguno de sus navegadores y escriban:

```
function test() {
    console.log(window == this);
}
test();
```

Un Objeto. Varias Instancias.

Cuando en Javascript creamos un objeto como hemos visto hasta ahora, no estamos hablando de una clase. Estamos hablando de un Objeto, listo, preparado para funcionar. Lo que en otros lenguajes llamaríamos *una instancia*. Pero esto no es del todo útil si queremos tener *varias* instancias de un mismo objeto al mismo tiempo. Así que aquí es donde entra en juego el operador new.

El operador new se aplica, por más que parezca extraño, sobre una función y no sobre un objeto. ¿Por qué?, se preguntarán. Básicamente, porque Javascript utilizará esta función en particular como **constructor** del Objeto que queremos instanciar, y nos devolverá un Objeto inicializado. Vamos a ver un ejemplo:

```
function Circulo() {}
var redondo = new Circulo();
```

Lo que estamos haciendo aquí es crear una instancia de un

tipo *nuevo* de Objeto llamado Circulo, y asignándoselo a la variable redondo. Vamos un paso más allá:

```
function Circulo() {
    this.radio = 2;

    this.area = function() {
        return this.radio * this.radio * Math.PI;
    }
}

var redondo = new Circulo();
redondo.area() // devuelve 12.566370614359172
```

Aqui vemos nuevamente el uso de this para indicar que estamos con un contexto nuevo. En este caso this no representa al objeto window ya que estamos utilizando new. Ahora, this representa el contexto del Objeto Circulo que estaremos devolviendo y asignando a la variable. Así, podemos contar con distintas instancias aunque utilicen todas el mismo código de inicialización. Vamos con un ejemplo más completo donde paramos el parámetro radio:

```
function Circulo(radio) {
    this.radio = radio;

    this.area = function() {
        return this.radio * this.radio * Math.PI;
    }
}

c1 = new Circulo(3);
c2 = new Circulo(4);

c1.area() // devuelve 28.274333882308138
c2.area() // devuelve 50.26548245743669
```

Herencia. Prototype.

Una de las diferencias claves de Javascript con otros lenguajes de programación, es la manera en la que implementa la Herencia de

Objetos. En Javascript se utiliza la Herencia Prototípica, y esto implica ciertas cosas:

* Un Objeto por defecto, siempre hereda sus métodos de otro Objeto, que se encuentra accesible a través del atributo prototype.
* Uno puede cambiar, en tiempo de definición, de qué tipo de Objeto se hereda.
* La Herencia prototípica es una cadena, si se hereda de un Objeto, a su vez éste heredará de otro objeto. Al final de la cadena, siempre se heredará del Objeto Object.

Volviendo con los ejemplos anteriores, vamos a extender nuestro Objeto circulo directamente modificando el Objeto prototype que hereda.

```
function Circulo(radio) {
    this.radio = radio;

    this.area = function() {
        return this.radio * this.radio * Math.PI;
    }
}

Circulo.prototype.diametro = function() {
    return this.radio * 2;
}

Circulo.prototype.circunferencia = function() {
    return Math.PI * this.diametro();
}

var circulo = new Circulo(3);
circulo.area() // devuelve 28.274333882308138
circulo.diametro() // devuelve 6
circulo.circunferencia() // devuelve 18.84955592153876
```

De esta manera, pudimos modificar agregando dos nuevos métodos, diametro y circunferencia, tan solo modificando el objeto prototype. Pero si queremos que un objeto herede completamente los métodos de OTRO objeto, entonces la manera de llevar a cabo la herencia es **reemplazando** el objeto prototype por completo. Veamos esto en un ejemplo totalmetne distinto:

```
function humano() {
    this.ojos = 2;
    this.brazos = 2;
    this.alimentacion = "omnivoro";
}

function freddito() {
    this.alimentacion = "paleo";
    this.conocimiento = Infinity;
}

freddito.prototype = new humano();
var Freddier = new freddito();
console.log(Freddier.brazos); // devuelve 2
console.log(Freddier.alimentacion); // devuelve "paleo"
```

Composición de Objetos.

Para completar este repaso de cómo trabajar con Javascript orientado a objetos, un tema que no es menor, la composición de objetos.Ya sabemos como crear clases, dotándolas de atributos y métodos, como extender un objeto de otro, y como llegar a tener nuestra funcionalidad encapsulada. Pero aún nos queda ver cómo podemos hacer que los objetos trabajen juntos entre si. La manera en la que se crean estructuras de objetos y se trabaja con ellos se llama *Composición* y hay ciertos tipos pre-definidos.

Composición.

En este tipo de composición, el Objeto principal o *"padre"* es quién define o contiene los objetos internos o *"hijos"*. Al eliminarse este objeto principal, los objetos internos no tienen sentido de existencia por si mismos, y son eliminados también. Un ejemplo de esto sería al decir *El Objeto A* ***tiene*** *un Objeto de clase B*.

```
function Tentaculo() {}

function Pulpo() {
```

```
    this.extremidades = [];

    for(var i = 1; i <= 8; i++) {
        this.extremidades.push(new Tentaculo());
    }
}

var pulpopaul = new Pulpo();
console.log(pulpopaul.extremidades);
```

Asociación.

En la Asociación, se termina definiendo o componiendo un objeto utilizando otros objetos, pero con una relación distinta a la Composición. Aquí, al elminiarse uno de los Objetos, los demás aún mantienen un sentido y siguen su curso. Un ejemplo de esto sería decir *El Objeto A **usa** un Objeto de tipo B*.

```
function Profesor() {
    this.name = "Pablo Rigazzi";
}

function Curso (profesor) {
    this.profesor = profesor;

    this.saludar = function() {
        console.log("Hola alumnos, soy el profesor " +
this.profesor.name);
    }
}

var yo = new Profesor();
var TeAmoJavascript = new Curso(yo);
TeAmoJavascript.saludar();
```

Polimorfismo.

Si bien ya vimos ejemplos de esto en otros ejemplos de este mismo texto, ahora vamos a ponerle oficialmente un nombre. El Polimorfismo en un lenguaje orientado a objetos, nos ofrece la oportunidad de que distintos Objetos, puedan responder a métodos o atributos del mismo nombre, y por eso, puedan ser reemplazados por otros Objetos. En el caso del Polimorfismo, la manera más habitual de referirse es *El Objeto A **es un** subtipo o clase de Objeto B*.

En el siguiente ejemplo, vamos a crear dos Objetos, con una interfaz similar (los métodos que contienen) y veremos como pueden ser utilizados indistintamente, y Javascript reconoce perfectamente cuál tiene que utilizar en cada caso.

```
function expresarse( ser_vivo ) {
    ser_vivo.hablar();
}

function Gato()
{
    this.hablar = function() {
        var parasiempre = '';
        for(var i=0; i<1000; i++) {
            parasiempre += 'nyan';
        }
        console.log(parasiempre);
    }
}

function Humano()
{
    this.hablar = function() {
        console.log('Hola, soy un humano!');
    }
}

function Freddier()
{
    this.hablar = function() {
        console.log('Christian, contexto!');
    }
}

Freddier.prototype = new Humano();
```

Teniendo esta definición de clases, y gracias al Polimorfismo, el siguiente código es válido y se ejecuta correctamente:

```
var humano   = new Humano();
var freddito = new Freddier();
var nyancat  = new Gato();

expresarse(humano);
'Hola, soy un humano!'
expresarse(freddito);
```

```
"Christian, contexto!"
expresarse(nyancat);
Se imaginaran...
```