JavaScript Orientado a Objetos

Primero empezemos con una diferencia fundamental (sobre todos para los que ya programaron en otros lenguajes orientados a objetos); la diferencia entre herencia de objetos *clásica* vs *basada en prototipos*.

> **Herencia**: Un objeto puede heredar propiedades y métodos de otro objeto (o clases), esto se conoce como herencia o inheritance, es un concepto fundamental de la programación orientada a objetos.

No vamos a hablar mucho de la herencia clásica, ya que JavaScript no soporta este tipo de herencia. JavaScript tiene una forma de herencia mucho más simple, donde no hay tanto protocolo ni restricciones. Esto hace que sea más flexible y fácil de entender.

Prototypal Inheritance

Como sabemos, los objetos de javascript tienen propiedades y métodos, y sabemos cómo acceder a ellos. Lo que no sabíamos es que además de las propiedades y métodos que nosotros le agregamos **todos los objetos tienen una referencia a otro objeto llamado proto**. Veamos para que le sirve:

En el ejemplo de arriba tenemos al objeto `Objeto`, que contiene dos propiedades `propiedad1` y `propiedad2`. Por lo tanto si quisiera acceder a cualquiera de esas propiedades podría usar la `dot notation`: `Objeto.propiedad1`. Ahora, como se ve en la imagen `Objeto` tiene una referencia a otro objeto llamado `proto`, y a su vez este objeto tiene una propiedad llamada `propiedad3`. Ahora lo interesante, es que si nosotros queremos acceder a la `propiedad3` del objeto `Objeto`, lo vamos a poder hacer! Cuando escribimos `Objeto.propiedad3` lo que ocurre es que el intérprete busca en el objeto por esa propiedad, y si no la encuentra antes de lanzar un error, busca en el objeto `proto` (que lo tienen *todos* los objetos) a ver si no encuentra esa propiedad, si la encuentra la devuelve.

De hecho, el objeto al que hace referencia `proto` también podría tener una referencia a otro `proto`. Digamos el objeto al que hace referencia el segundo `proto` contiene la propiedad `propiedad4`. Si nosotros intentaramos acceder a `propiedad4` desde `Objeto` usando `Objeto.propiedad4`, el interprete primero buscaría en `Objeto`, como no está esa propiedad ahí entonces va a buscar en el objeto al que hace referencia `proto`, como tampoco está ahí se fija si ese objeto tiene una referencia en `proto`, como la tiene, va a buscar la propiedad en ese objeto al que hace referencia. En nuestro ejemplo, finalmente encuentra la `propiedad4` en este último y por lo tanto la accede. Esto es lo que se conoce como **Prototype Chain**.

> Las propiedades *parecen* que están en el Objeto que intentamos acceder, pero en realidad están en *otro* objeto y son accedidas a traves del **prototype chain**.

Lo más importante de esto, es que si tuvieramos un segundo objeto: `Objeto 2`, cuya propiedad `proto` hace referencia al *mismo objeto* al que hacia referencia el `proto` de `Objeto`, entonces ambos objetos **compartirian** un subset de propiedades. En este caso, si quisieramos acceder a `Objeto2.propiedad3` (que no existe en el objeto 2), la encontraríamos siguiendo el prototype chain, y accederiamos a la misma propiedad que si hicieramos `Objeto1.propiedad3`.

 Herencia

Veamos algunos ejemplos:

```
var persona = {
 nombre: 'Default',
 apellido: 'Default',
 getNombre: function() {
  return this.nombre + ' ' + this.apellido;
 }
}

var juan = {
 nombre: 'Juan',
 apellido: 'Maquiavelo'
}

// no hacer esto en producción es sólo para demostración
// hay otras formas de asiganar prototipos.

juan.__proto__ = persona;

// Ahora podemos usar los métodos de `persona`

juan.getNombre();
```

> En el ejemplo, ambos objetos (persona y juan) tienen la propiedad `firstname`. En estos casos, el interprete va a devolver la primera que encuentra, y no va a seguir la cadena de prototipos.

Todo es un objeto en JavaScript (o una primitiva)

Cualquier cosa en JavaScript que NO sea una primitiva, es un objeto! O sea: funciones, arreglos, objetos, todos tienen un prototipo. Sin embargo hay un objeto especial, que no tiene un prototipo. Este objeto se llama `base object`. Este objeto se encuentra siempre al final del prototype chain, y termina ahí porque `base object` no tiene un prototype.

De hecho, `base object` tiene definido una serie de propiedades y métodos. Y como todos los demás objetos lo tiene en su cadena de prototipos, entonces, estos métodos y propiedades son accesibles por todos los objetos de JavaScript. Por ejemplo, el método `toString` está definido en el `base object`.

En el caso de cualquier función, su prototipo por defecto es un objeto llamado `Empty`, que es a su vez una función. Cualquier función que creemos va a tener este proto y por ende van a tener acceso a todas las propiedades y métodos de `Empty`. Por ejemplo, la funciones `apply`, `bind` y `call` están definidas en este Objeto.

Con los arreglos pasa algo similar, todos los arreglos tiene como proto a un `arreglo base`. En este último se encuntran definido todos los métodos que usamos en los arreglos, como `push`, `shift`, `length`, etc...

Veamos un ejemplo:

```
var a = {}; // un objeto vacio
var b = function(){ };
var c = [];

a.__proto__ // veamos que tiene en su proto
// tiene Object {} que es el `base object`
a.__proto__.toString // contiene este metodo

// funcion b
b.__proto__ // function Empty()
b.__proto__.bind // funcion bind!

//arreglo
c.__proto__ // [] arreglo vacio base
c.__proto__.push // funcion push!

// y el proto del proto?

a.__proto__.__proto__ // Object {}
b.__proto__.__proto__ // Object {}
c.__proto__.__proto__
```

Esta es la razón por la que tenemos acceso a todos esas propiedades y métodos que vienen `por defecto` y que usamos a menudo.

Reflexion and Extend

La *Reflexion* es la capacidad que tienen los objetos de mirarse a si mismos, listando y cambiando sus propias propiedades y métodos. Todos los objetos tiene un método llamado `hasOwnProperty` que recibe un string, y devuele un booleano en base a si el objeto en cuestión tiene o no la propiedad con el nombre del string pasado. Lo importante es que este método, se fija **solo** en las propiedades del objeto y **no** sigue el `prototype chain`.

Por ejemplo:

```
var persona = {
 nombre: 'Default',
 apellido: 'Default',
 getNombre : function() {
  return this.nombre + ' ' + this.apellido;
 }
};

var santi = {
 nombre: 'Santi',
 apellido: 'Scanlan'
};

santi.__proto__ = persona;
```

```
for (var key in santi){
 console.log( key + ":" + santi[key] );
} // imprime todo! inclusive getNombre, que esta en su proto.

for(var key in santi){
 if(santi.hasOwnProperty(key)){
  console.log( key + ":" + santi[key] );
 }// imprime solo las propiedades del objeto Santi.
}
```

Este concepto nos permite realizar algo simililar al prototipado, pero con ciertas diferencias importantes. Esta forma no es nativa en JavaScript, pero es tan útil que muchos frameworks y librerías la implementan. Por ejemplo, underscore.js, tiene una implementación. Esta nueva forma es conocida como **extend**, y vamos a explicarla con un ejemplo usando la librería antes mencionada:

En este ejemplo, tenemos varios objetos que tiene algunas propiedades y/o métodos que nos gustaría reutilizar.

```
var guille = {
 direccion : 'Armenia 636 6to F',
 getNombreFormal : function(){
  return this.apellido + ', ' + this.nombre
 }
}
var toni = {
 getPrimerNombre : function() {
  return this.nombre;
 }
}

_.extend(santi, guille, toni);
```

Lo que hace `extend`, es combinar todas los métodos y propiedades de `guille` y `toni` dentro del objeto `santi`. Si ahora vemos el objeto `santi`, vemos que conserva sus propiedades y métodos, pero ahora tiene las de `guille` y `toni`. Y, a diferencia de haberlas obtenido a traves del `prototype chain`, *estas propiedades y métodos* **son** de `santi`.

> Pueden bajar la librereia underscore y buscar entre sus funciones a la función *extend* y ver cómo está implementada. Se pueden imaginar cómo la hicieron antes de ver el código?

Construyendo Objetos

Ahora que conocemos el prototipado, la cadena cadena de prototipos, y conocemos los Objetos base, podemos empezar a hablar de las mejores formas de *construir objetos*.

Ya conocemos algunas, como *objects literals*:

```
var objeto = {
 propiedad1: valor1,
 propiedad2: valor2,
 metodo1   : function1() {

 }
}
```

Pero hay varias formas de construirlos, sobre todo cuando se trata de setear el **proto** del objeto creado.

Functions Constructors y new

Hay algunas features del lenguaje cuya inclusión no fueron por razones técnicas. Es más, existen algunas que tienen que ver con el **marketing** del lenguaje. (de hecho, el nombre `javascript` fue elegido para atraer a desarrolladores `java`, que estaba muy de moda en aquel momento). Lo que termina ocurriendo es que aparecen ciertas formas de escribir o hacer cosas que vienen de otros lenguajes, ya que los desarrolladores estaban acostumbrados a hacerlas de ese modo. Finalmente lo que pasa es que escribimos algo de una forma que sí tiene sentido para otro lenguajes, pero no tanto para JavaScript. El caso del keyword `news` es este. Esta forma de crear objetos, tiene que ver con la forma de instanciar clases de lenguajes como `java`. Pero en JavaScript no tendría sentido, porque no existen las clases. En JavaScript son todos objetos, no hay clases. De todas formas, esa forma de crear objeto fue heredada de esta manera, por marketing.

Veamos un ejemplo de la sintaxis del keyword `this` y cómo funciona examinando el siguiente código:

```
function Persona() {
 this.firstname = 'Juan';
 this.lastname = 'Perez';
}

var juan = new Persona();
console.log(juan)
```

Qué ven de raro en la función `Persona()`? por empezar no retorna nada, y además no está claro a qué hace referencia el keyword `this`. Pero si nos fijamos, cuando la usamos con `new`, vemos que nos creó un objeto con las propiedades definidas en esa función. Para empezar a entender, primero tenemos que saber el `new` es en realidad un operador en JavaScript. Lo que hace es operador es, primero, crear un objeto vacío. Luego invoca la función que le pasamos como argumento, con la particularidad que bindea el nuevo objeto vacio que había creado, de tal forma que en ese nuevo contexto de ejecucieon el keyword `this` haga referencia a este objeto nuevo. Por último retorna ese objeto que había creado (y que fuera modificado por la función ejecutada).

```
var a = {};
Persona().call(a);

return a; // algo asi
```

Por lo tanto, esto nos permite crear un objeto al invocar una función, justamente esta función (en el ejemplo `Persona()`) cumple el rol de `function constructor`. Seguro estarán pensando que la función `Persona` me sirve para crear objetos que sean iguales, veamos como mejorar eso. Cómo los `functions constructors` **son funciones**, entonces podemos hacer lo siguiente:

```
function Persona(nombre, apellido){
 this.nombre = nombre || 'Juan';
 this.apellido = apellido || 'Perez';
}
var toni = new Persona('Toni', 'Tralice');
var santi = new Persona('Santi', 'Scanlan');
var guille = new Persona('Guille', 'Aszyn');
```

No pierdan de vista, que estamos invocando una función, asi que podemos utilizar adentro **todo** los que sabemos de funciones.

Lo que todavía no vimos, es cómo setear el **proto** cuando creamos objetos usando `function constructors`. Antes que nada, veamos qué objeto tienen seteado como **proto** los objetos creados con algún `function constructor`. Sorpresa! Ya tienen seteados un **proto** y cómo vemos, hace referencia a un objeto que tiene el mismo nombre que la función que los construye. Para entender esto, tenemos que recordar qué las funciones son objetos especiales, que tenian algunas propiedades extras cómo `Code` y `Name`. También tenemos que saber que *todas* las funciones tienen la propiedad `prototype`, y que se setea siempre cómo un objeto vacío, **cuando invocamos la función con el operador** `new` la propiedad `prototype` de la función va a ser usada como el __proto__ de **todos** los objetos que hayan sido creadas con ella.

> Mucho cuidado con confundirse la propiedad prototype con el __proto__ de un objeto. No es lo mismo, prototype es simplemente el nombre que , unfortunadamente, eligieron para esa propiedad.

Por lo tanto, todo lo que pongamos dentro de la propiedad `prototype` de la función constructora, va a ser *heredado* por los objetos creados usando está función con `new`. Por ejemplo:

```
function Persona(nombre, apellido){
 this.nombre = nombre || 'Juan';
 this.apellido = apellido || 'Perez';
}
Persona.prototype.getNombre = function () {
 return this.nombre + ' ' + this.apellido;
}

var toni = new Persona('Toni', 'Tralice');
var santi = new Persona('Santi', 'Scanlan');
var guille = new Persona('Guille', 'Aszyn');

toni.getNombre() // funciona!
santi.getNombre() // tambien!
guille.getNombre() // :D
```

> Agregar funciones en el prototype del constructor y no dentro del mismo, es una buena practica. **Ya que tener metodos replicados en cada objeto ocupa mucho espacio**. En cambio, si los tenemos en el proto, todos comparten el mismo.

Object.create y Pure Pototypal Inheritance

Dijimos que los *function constructors* fueron creados pensados en imitar el comportamiento de otros lenguajes de programación, por lo tanto son un poco extraños. Existen muchos programadores que directamente dicen que es mejor aceptar el hecho que JavaScript tiene un modelo de herencia basado en prototipos (a diferencia de otros modelos de herencia de otros lenguajes) y que para crear objetos usemos métodos consecuentes a este hecho. Para hacerlo, javaScript nos provee de la función 'Object.create'. Veamos como funciona:

```
var person = {
 nombre: 'Defecto',
 apellido: 'Defecto'
}; // un objeto cualquiera

var Santi = Object.create(person); // le paso el objeto que creamos
console.log(Santi) // es un objeto
console.log(Santi.__proto__) // es el objeto person que creamos al principio!!
```

Object.create recibe un objeto como parametro y crea un nuevo objeto cuyo prototipo es el primero que le pasamos. Por lo tanto, vamos a poder aceder a todas las propiedades y métodos del objeto *base* en los nuevos objetos creados. Ahora, si quisieramos cambiarle algunas propiedades, lo único que deberíamos hacer es escribirle nuevas propiedades con le mismo nombre al objeto en cuestión. En nuestro ejemplo, si quisieramos que le nuevo objeto tuviese un nombre y apellido que no fuera el por defecto (heredado) deberíamos hacer lo siguiente:

```
Santi.nombre = 'Santiago';
Santi.apellido = 'Scanlan';
```

Esto es posible gracias a cómo funciona el *Prototype Chain*, que va a buscar esa propiedad primero en el objeto en sí, si la encuentra no tiene la necesidad de buscarla en el prototipo.

ES6 y Clases (class)

En la última versión del estandar JS va a tener la posibilidad de declarar clases, que es el concepto que utilizan otros lenguajes para implementar herencia. De todos modos, en JS las clases no van a se exactamente iguales que en otros lenguajes, ya que en JS existe el modelo de prototipado. Veamos cómo se ven las clases en JavaScript:

```
class Persona {

 constructor (nombre, apellido){
```

```
    this.nombre = nombre,
    this.apellido = apellido
   }

   saludar() {
    console.log('Hola!' + this.nombre);
   }
  }

  var toni = new Persona('Toni', 'Tralice');
  toni.saludar();
```

Cuando definimos una clase usamos el nuevo statement **class**. Dentro de el primero definimos su constructor, que es el método que va a recibir todos los parámetros y va *crear* el objeto nuevo basado en esta clase. Podemos decir que es similar a los *functions constructors* que habiamos visto. Luego podemos agregar métodos que van a estar disponibles para todos los objetos creados con esa *clase*.

> Los que ya programaron en otros lenguajes orientados a Objetos tienen que tener en cuenta que la implementación de Clases no es igual en JS. De hecho, en este ejemplo la clase *Persona* ES un **objeto**, este objeto es utilizado como prototipo para los nuevos objetos creados con ella.

Para agregar un prototipo a la clase creada se utiliza el keyword **extends**, que también está inspirado en otros lenguajes:

```
class Empleado extends Persona {
 constructor (nombre, apellido, empleo, sueldo){
  super(nombre, apellido);
  this.empleo = empleo;
  this.sueldo = sueldo;
 }
}

var toni = new Empleado('Toni', 'Tralice', 'Profesor', 100);
toni.saludar();
```

El método **super** nos permite usar el constructor de la clase de la cual estamos *heredando*, en este caso de la clase *Persona*.

> Estas nuevas formas son **sólo distantas formas de *escribir* lo mismo**, por atrás JS crea los objetos de la misma forma que usando otros métodos. Esto es conocido como *syntactic sugar*: Una forma distinta de escribir algo, pero que no cambia cómo el intérprete trabaja.

Modelando Objetos

Programación Orientada a Objetos

Un **paradigma de programación** representa un enfoque particular o filosofía para diseñar soluciones. Los paradigmas difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema, en otras palabras, el cómputo. La

OOP (Object Oriented Programming) es un paradigma de programación. donde los datos están encapsulados en **objetos** que tienen *propiedades* y *métodos*, y todas las operaciones que hacemos la hacemos sobre estos objetos.

Veamos algunos conceptos claves de OOP:

- **Clase**: Una clase define las características de un objeto. Es el template donde se definen las *propiedades* y *métodos* que van a tener las instancias de esa clase. Una *clase* es algo abstracto, representa algo pero no *es* ese algo, hasta que se instancie. Por ejemplo: la *clase* Empleados, representa a los empleados, pero no es ninguno en particular.
- **Objeto** : *Instancia* de una clase. Es cuando usamos la abstraccion de la clase para *Crear* (instanciar) un objeto. Siguiendo el ejemplo de *Empleados*, podriamos instanciar un objeto *Jorge*(instancia) que *es un* Empleado (clase).
- **Atributos**: Características que tiene la clase, por ejemplo, en Empleados, puede ser: *nombre*, *apellido*, *salario*, etc...
- **Método** : Es el comportamiento de la clase, en general son funciones. Por ejemplo, en empleados un método puede ser: *cobrarSueldo()*, o *trabajar()*.
- **Herencia**: Una clase (subclase) puede *heredar* de otra clase (superclase). Esto quiere decir que extiende a la superclase, es decir que tiene todas las propiedades y métodos de ella y que además tiene otros atributos o métodos particulares. Por ejemplo, la clase *Manager* puede heredar de *Empleado* y tener un método nuevo: *darOrdenes()*.
- **Abstracción**: Es el principio básico bajo el cuál diseñamos las clases, básicamente *abstraemos* un conjunto de objetos con sus atributos y métodos. Esto nos da la posibilidad de pensar un problema desde un nivel más alto.
- **Encapsulamiento**: Es la capacidad de poder agrupar propiedades y métodos en un entorno con límites bien definidos. De hecho las *clases* son son otra cosa que *abstracciones _encapsuladas* bajo un nombre en particular.
- **Polimorfismo**: Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando.
- **Modularidad**: Se denomina "modularidad" a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos.
- **Principio de ocultación**: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una "interfaz" a otros objetos que especifica cómo pueden interactuar con los objetos de la clase.

Como la OOP es un *paradigma*, cada lenguaje lo interpreta y lo implementa de distintas maneras. De hecho, Javascript no usa el concepto de *herencia*, si que utiliza el concepto de *prototipado*, que de alguna forma lo emula. Es decir, que hay lenguajes donde podemos programar orientado a objetos pero que no es la única forma en la que podemos programar. Vamos a hacer una comparación con **Java**, que es un lenguaje muy preparado para programar OOP:

Class-based (Java)	**Prototype-based (JavaScript)**
Clases y Objetos son entidades distintas.	Todos los objetos pueden heredad de otro objeto.

Class-based (Java)	**Prototype-based (JavaScript)**
Se define una clase con una definición de clase, y se instacia un objeto de esa clase usando un constructor	Se define y se crean objetos usando una función constructor.
Crea un nuevo objeto con el operador *new*	Lo mismo.
Construye una jerarquía de clases, donde cada clase hereda de otras clases existentes.	Construye una jerarquía de objetos, donde cada objeto tiene asociado un prototipo con un constructor.
Los objetos heredan propiedades y métodos según la cadena de clases.	Los objetos heredan propiedades y métodos siguiente la cadena de prototipos.
La definición de una clase especifica *TODAS* las propiedades y métodos de ella, no se pueden cambiar dinámicamente durante el *run time*.	Las funciones constructoras especifican una serie de propiedades y métodos *INICIALES*. Se pueden agregar o sacar propiedades y méetodos dinámicamente durante el *run time*.