**Lección 6: Javascript V (Clases y prototype)**

En esta lección cubriremos:

* Clases
* prototype

**Clases**

Muchas veces cuando creamos un objeto, estamos creando una plantilla. En lugar de copiar esa plantilla una y otra vez, Javascript nos da acceso a lo que llamamos un constructor o class. Las clases comparten gran parte de la misma funcionalidad que los objetos normales, pero también se expande mucho en esa funcionalidad. Las clases son útiles para crear muchos objetos que comparten algunas de las mismas propiedades y métodos (como los usuarios en un sitio web).

**Class e instanciación pseudo-clásica**

Si tienes experiencia en un lenguaje orientado a objetos (como Java o C#), probablemente estés familiarizado con el concepto de clases. Si bien Javascript no proporciona un “verdadero” sistema de clases, hay algo muy familiar. En aras de la discusión, llamaremos a nuestros objetos de clase ‘clases’. Se instancia de manera pseudo clásica, usando la palabra clave new, y puede tomar argumentos.

En este ejemplo crearemos una clase Gato. La convención para las clases consiste en dar un nombre en mayúscula al nombre de todo lo que se pueda instanciar con la palabra clave new. Cuando usamos la palabra clave new, Javascript hace un gran trabajo detrás de escena para nosotros y crea y devuelve un objeto automáticamente.

**function** Gato(nombre) {

*// El nuevo operador crea un objeto, "this"*

**this**.nombre **=** nombre;

**this**.maullar **=** **function**() {

**return** 'Mi nombre es ' **+** **this**.nombre **+** ' ... Meow!';

}

*// Devuelve el objeto "this"*

}

**const** sam **=** **new** Gato('Sam');

**const** kitty **=** **new** Gato('Kitty');

console.log(sam.maullar()); *// 'Mi nombre es Sam ... Meow!'*

console.log(kitty.maullar()); *// 'Mi nombre es Kitty ... Meow!'*

**this en las clases**

La palabra clave this puede comenzar a volverse muy confusa cuando comenzamos a usarla en clases. En el último ejemplo lo usamos en el método de los maullidos. Una buena regla general si no está seguro de a qué se refiere this, es observar dónde se llama el método y el objeto a la izquierda del ‘punto’. Ese es el objeto al que se refiere this.

**prototype**

La creación de funciones es costosa (refiriéndonos a la capacidad de memoria de una computadora) y cada vez que creamos un nuevo objeto de clase con métodos, estamos recreando esos métodos en la memoria. Puede imaginar que si estamos creando miles de objetos de clase a partir de una clase con docenas de métodos, la memoria se acumulará rápidamente (20.000 - 40.000 métodos). Las clases tienen una forma única de establecer un método una vez y dar acceso a cada objeto de esa clase a esos métodos. Esto se llama el prototype. Cada clase tiene una propiedad *prototype*, que luego podemos establecer en métodos:

**function** Usuario(nombre, github) {

**this**.nombre **=** nombre;

**this**.github **=** github;

}

Usuario.prototype.introduccion **=** **function**(){

**return** 'Mi nombre es ' **+** **this**.nombre **+** ', mi usuario de Github es ' **+** **this**.github **+** '.';

}

**let** juan **=** **new** Usuario('Juan', 'juan.perez');

**let** antonio **=** **new** Usuario('Antonio', 'atralice');

console.log(juan.introduccion()); *// Mi nombre es Juan, mi usuario de Github es juan.perez.*

console.log(riley.introduccion()); *// Mi nombre es Antonio, mi usuario de Github es atralice.*

Los métodos de prototype tienen acceso a la palabra clave this y, al igual que antes, siempre apuntará al objeto (a la izquierda del punto) que lo está llamando.

Hasta ahora siempre que teníamos que crear un objeto nuevo declarábamos un object literal, pero vamos a ver que hay otros métodos que nos da el prototype de Object para cumplir esa tarea

**Object.create**

El método create de los objetos nos permite crear un nuevo objeto a partir de un prototype especificado.

*// creo un objecto con un objeto vacio como proto*

**>** **var** obj **=** Object.create({})

**>** obj

**<** Object {}

*// creo que un objeto a partir de un proto de Objeto*

**>** **var** obj **=** Object.create(Object.prototype)

*// que es lo mismo que crear un objeto vacio literal*

**>** **var** obj **=** {}

**Object.assign**

El método assign de los objetos te permite agregar propiedades a un objeto pasado por parámetro

**>** **var** obj **=** {}

*// No hace falta guardar el resultado porque los objetos se pasan por `referencia`*

**>** Object.assign(obj, {nombre:'Emi', apellido:'Chequer'})

**>** obj.nombre

**<** 'Emi'

**Herencia Clásica**

En el paradigma de *Programación Orientada a Objetos* un tema muy importante es la *Herencia y Polimorfismo* y de las clases (los vamos a llamar constructores por ahora).

Cuando hacemos referencia a **Herencia** nos referimos a la capacidad de un constructor de *heredar* propiedades y métodos de otro constructor, así como un Gato es Mamífero antes que Gato, y hereda sus ‘propiedades’ (nace, se reproduce y muere).

Cuando hablamos de **Polimorfismo** nos referimos a la capacidad de que objetos distintos puedan responder a un llamado igual de acuerdo a su propia naturaleza.

**Herencia en JavaScript**

En JS a diferencia de la herencia clásica nos manejamos con prototipos, que van a tomar los métodos pasados por sus ‘padres’ mediante la Prototype Chain.

Cuando generamos un arreglo nuevo podemos acceder a métodos como map o slice gracias a que los heredamos del Objeto Array que esta vinculado en la propiedad \_\_proto\_\_ y es el siguiente en el Prototype Chain.

Nosotros también podemos generar nuestros propios constructores que de los cuales heredar. Creemos un constructor de el cual pueda haber variantes.

**>** **function** Persona(nombre,apellido,ciudad) {

**this**.nombre **=** nombre;

**this**.apellido **=** apellido;

**this**.ciudad **=** ciudad;

}

**>** Persona.prototype.saludar **=** **function**() {

console.log('Soy '**+this**.nombre**+**' de '**+this**.ciudad);

}

**>** **var** Emi **=** **new** Persona('Emi', 'Chequer', 'Buenos Aires');

**>** Emi.saludar()

**<** 'Soy Emi de Buenos Aires'

Ahora todo Alumno de Henry antes de Alumno es una Persona, asique podríamos decir que un Alumno hereda las propiedades de ser Persona.

**>** **function** Alumno(nombre,apellido,ciudad,curso) {

*// podría copiar las mismas propiedades de Persona acá adentro*

**this**.nombre **=** nombre;

**this**.apellido **=** apellido;

**this**.ciudad **=** ciudad;

**this**.curso **=** curso

}

**Constructores Anidados**

Pero en este caso estaríamos repitiendo código, y si en un futuro quisiera cambiar una propiedad tendría que hacerlo en ambos constructores. Descartemos esta opción.

*// lo que nosotros queremos es poder reutilizar las propiedades de persona,*

**>** **function** Alumno(nombre, apellido, ciudad, curso) {

*// usemos nuestro constructor Persona dentro del de Alumno*

Persona.call(**this**, nombre, apellido, ciudad);

*// vamos a necesitar el call porque queremos que las propiedades de persona, queden en bajo el objeto que va a devolver Alumno, y no uno nuevo del constructor Persona.*

*// luego le paso los valores que quiero que reciba el constructor de Alumno*

*// finalmente le agrego los puntos propios de Alumno*

**this**.curso **=** curso;

**this**.empresa **=** 'Soy Henry';

}

**>** **var** toni **=** **new** Alumno('Toni', 'Tralice', 'Tucuman', 'Web Full Stack')

*// Ahora si tenemos nuestra instancia creada a partir de ambos constructores*

**>** toni.curso

**<** Web Full Stack

**>** toni.apellido

**<** Tralice

**>** toni.saludar()

**<** Uncaught TypeError: toni.saludar is not a 'function'

*// que paso?*

Como podemos ver los métodos de *Personas* no fueron pasados a nuestros *Alumnos*. Veamos un poco el porqué.

El constructor del \_\_proto\_\_ esta ligado a Alumno y luego al Object Object de JS. Pero el método saludar esta en el objeto prototype de Personas… , y esta perfecto, así es como debería funcionar, las instancias acceden al \_\_proto\_\_ que fue vinculado por el constructor para ver que métodos tienen. Nuestro problema es que al llamar a Persona con call en vez de con el método new no se esta haciendo ese vinculo en el que Persona.prototype se mete en nuestro Prototype Chain, y entonces las instancias de Alumno no tienen acceso a los métodos de Persona

Vamos a solucionar ese problema agregando al prototipo los métodos de Persona, para esto vamos a usar el método Object.create.

*// usamos `Object.create` porque este guardaba el argumento pasado como `\_\_proto\_\_` del objeto a retornar*

**>** Alumno.prototype **=** Object.create(Persona.prototype)

*// si recuerdan el objeto prototype siempre tenia una propiedad constructor que hacia referencia a la función en si, con la asignación que hicimos arriba lo pisamos, por lo que deberíamos volver a agregarlo.*

**>** Alumno.prototype.**constructor** **=** Alumno

**>** **var** Franco **=** **new** Alumno('Franco','Etcheverri','Montevideo','Bootcamp')

**>** Franco.saludar()

**<** 'Soy Franco de Montevideo'

**Abre la carpeta “homework” y completa la tarea descrita en el archivo README**

[Homework](https://github.com/atralice/Curso.Prep.Henry/tree/master/06-JS-V/homework)

**Recursos adicionales**

* [MDN: Classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)
* [MDN: Prototype](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype)