Hecho con 💛 por alumnos de Henry

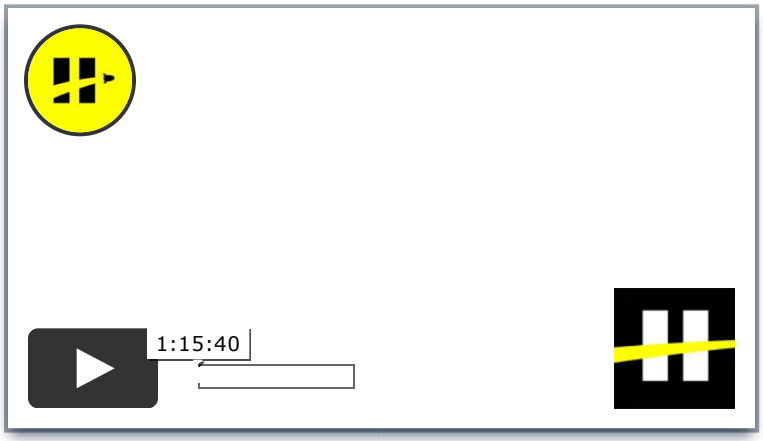
Contenido de la clase

Tiempo de lectura
14 min

Homework

JavaScript V

Clases y 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-

Dejanos tu feedback! 👍

Contenido de la clase

Dejanos tu feedback! 👍

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 +
    }
    // Devuelve el objeto "this"
}

const sam = new Gato('Sam');
const kitty = new Gato('Kitty');
console.log(sam.maullar()); // 'Mi nombre es Sa
console.log(kitty.maullar()); // 'Mi nombre es
```

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`.

Contenido de la clase

Dejanos tu feedback! 👍

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 + ', m
}

let juan = new Usuario('Juan', 'juan.perez');
let antonio = new Usuario('Antonio', 'atralice'

console.log(juan.introduccion()); // Mi nombre
console.log(antonio.introduccion()); // Mi nomb
```

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

Contenido de la clase

Dejanos tu feedback! 👍

```
// creo un objecto con un objeto vacio como pro
> var obj = Object.create({})

> obj
< Object {}

// creo un objeto a partir de un proto de Objet
> var obj = Object.create(Object.prototype)
// que es lo mismo que crear un objeto vacio li
> 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 lo
> Object.assign(obj, {nombre:'Emi', apellido:'C

> 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

Contenido de la clase

Dejanos tu feedback! 👍

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.
  }

> var Emi = new Persona('Emi', 'Chequer', 'Buen

> 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.

Contenido de la clase


```
    // podría copiar las mismas propiedades de
    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
> function Alumno(nombre, apellido, ciudad, cur
    // usemos nuestro constructor Persona dentr
    Persona.call(this, nombre, apellido, ciudad
    // vamos a necesitar el call porque queremo
    // luego le paso los valores que quiero que

    // finalmente le agrego los puntos propios
    this.curso = curso;
    this.empresa = 'Soy Henry';
  }

> var toni = new Alumno('Toni', 'Tralice', 'Tuc

// Ahora si tenemos nuestra instancia creada a
> toni.curso
< Web Full Stack

> toni.apellido
< Tralice

> toni.saludar()
< Uncaught TypeError: toni.saludar is not a 'fu
// que paso?
```

Dejanos tu feedback! 👍

Contenido de la clase

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
> Alumno.prototype = Object.create(Persona.prot

// si recuerdan el objeto prototype siempre ten
> Alumno.prototype.constructor = Alumno

> var Franco = new Alumno('Franco','Etcheverri'

> Franco.saludar()
< 'Soy Franco de Montevideo'
```

Recursos adicionales

- **MDN: Classes**
- **MDN: Prototype**

Dejanos tu feedback! 👍

Contenido de la clase

Homework

Completa la tarea descrita en el archivo **README**

Si tienes dudas sobre este tema, puedes consultarlas en el canal *06_js-v* de Slack

Dejanos tu feedback! 👍