Programación Orientada a Objetos

No, este no es un documento de 18 páginas hablando mal de la POO, ¡aquí aprenderemos como aplicar la Programación orientada a Objetos en Javascript!

¿Se imaginan mi cara cuando descubrí que la POO existe en Javascript y es vital para software a gran escala?

La programación orientada a objetos en Javascript es diferente a la de otros lenguajes, principalmente porque no existe. Es algo así como café descafeinado. Tiene el mismo sabor, pero no es café real.

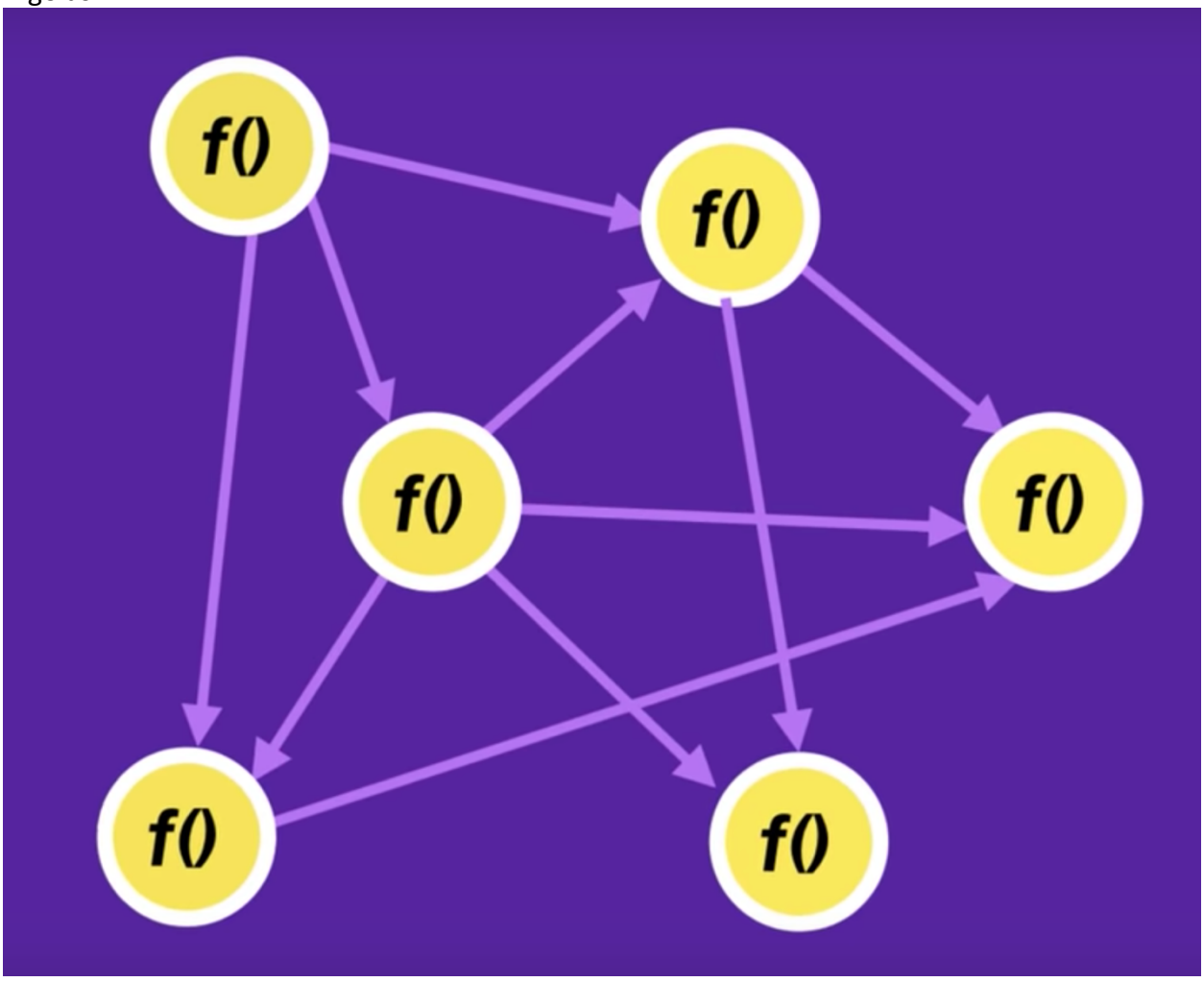
Esta intro la basaré en las clases de POO de Programming with Mosh. Lo demás es de mi querido Brad Traversy <3

Los cuatro pilares de la programación orientada a Objetos

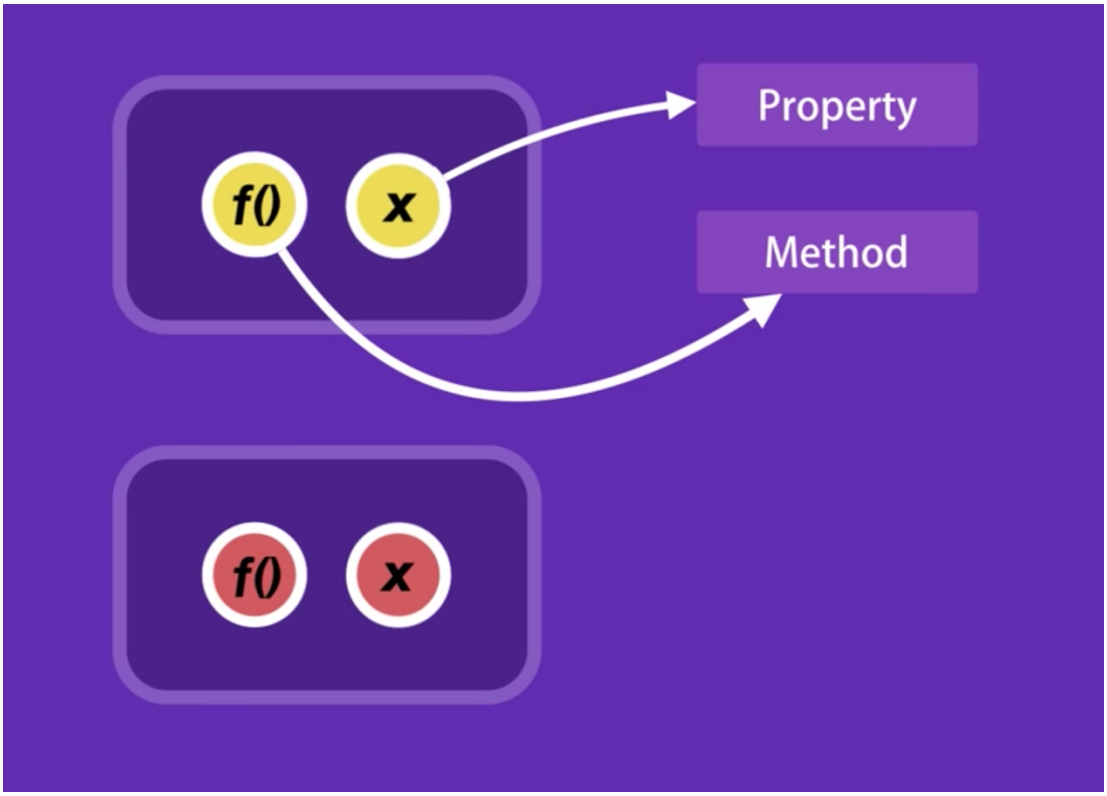
Encapsulación, Abstracción, herencia y polimorfismo.

Antes de la programación orientada a objetos teníamos a la programación procesal.
Dividía el programa en una combinación de funciones. Es muy simple y lineal, pero mientras más crece se hace un solo espagueti y luego de un tiempo el código se hace muy difícil de leer.

Algo así:

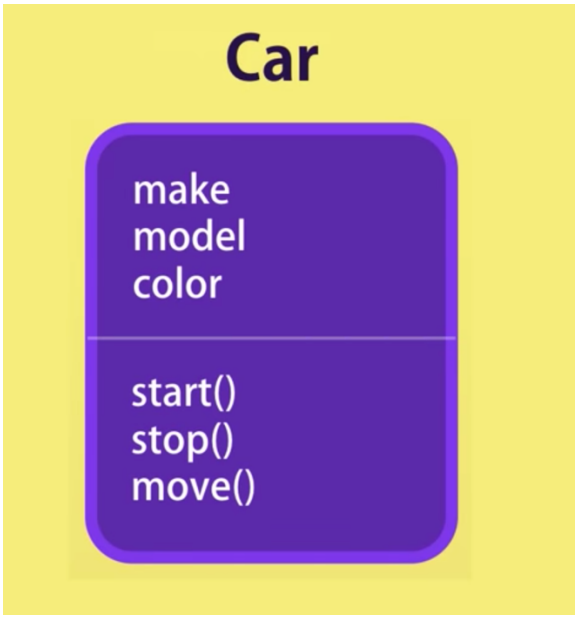


En la programación orientada a objetos, combinamos un conjunto de funciones y variables en una sola unidad, le llamamos a esa unidad un objeto (nada que ver con los objetos de Javascript)



Aquí, las funciones son métodos y los parámetros o variables, son propiedades.

Aquí hay un ejemplo:



Donde make, model y color son propiedades y las funciones start, stop y move son métodos.
PERO NO TENEMOS CARROS EN LA PROGRAMACION, COMO CARAJO SE APLICA IN RIAL LAIF
(Lo digo porque siempre dan ejemplos así en las clases de POO y solo me confundo más.

```
Console was cleared                                          VM100:1
undefined
> let numeros = [1,2,3,4,5]
undefined
> console.log(numeros)
                                                             VM130:1
▼ (5) [1, 2, 3, 4, 5] i
    0: 1
    1: 2
    2: 3
    3: 4
    4: 5
    length: 5
  ▼ __proto__: Array(0)
    ▶ concat: ƒ concat()
    ▶ constructor: ƒ Array()
    ▶ copyWithin: ƒ copyWithin()
    ▶ entries: ƒ entries()
    ▶ every: ƒ every()
    ▶ fill: ƒ fill()
    ▶ filter: ƒ filter()
    ▶ find: ƒ find()
    ▶ findIndex: ƒ findIndex()
    ▶ forEach: ƒ forEach()
    ▶ includes: ƒ includes()
    ▶ indexOf: ƒ indexOf()
    ▶ join: ƒ join()
    ▶ keys: ƒ keys()
    ▶ lastIndexOf: ƒ lastIndexOf()
      length: 0
    ▶ map: ƒ map()
    ▶ pop: ƒ pop()
    ▶ push: ƒ push()
    ▶ reduce: ƒ reduce()
    ▶ reduceRight: ƒ reduceRight()
    ▶ reverse: ƒ reverse()
    ▶ shift: ƒ shift()
    ▶ slice: ƒ slice()
    ▶ some: ƒ some()
    ▶ sort: ƒ sort()
    ▶ splice: ƒ splice()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ unshift: ƒ unshift()
    ▶ values: ƒ values()
```

Aquí simplemente se creó un array. No importa lo que hay dentro, es un array y ya.
Si console.loggeamos el array, nos dará los valores dentro de él, pero también tendremos los prototipos que tenemos, como concat, full o join. Un Objeto también tine sus propios prototipos y así. Esto es programación orientada a objetos.

Así que, en la programación orientada a objetos agrupamos determinado tipo de variables y funciones que operan en ellas. A esto se le llama ENCAPSULACIÓN.

(¿Cansado de tecnicismos? Pues que mal, aun faltan 3 y solo llevamos 1)

Ejemplo de encapsulación:

```
let salarioBase = 10000;
let tiempoExtra = 10;
let radio = 20;

function obtenerSalario(salarioBase, tiempoExtra, radio) {
    return salarioBase + (tiempoExtra * radio);
}
```

Esta función recibe tres parámetros y los procesa. Tenemos las variables en un lado y las funciones en otro :s al crecer la app esto se hará más absurdo, no queremos eso verdad?

```
let empleado = {
  salarioBase: 10000,
  tiempoExtra: 10,
  radio: 20,
  obtenerSalario: function(){
    return this.salarioBase + (this.tiempoExtra * this.radio);
  }
};
empleado.obtenerSalario();
```

Quizás hay algunas dudas al respecto, luego hablaremos de cosas como el “this”.
Pero por qué esto es mejor? Porque encapsuló todo lo que queremos hacer en una sola variable, aislado. En “empleado” no recibe parámetros, mientras que obtenerSalario sí.

“Las mejores funciones son aquellas sin parámetros”

Y así y así y así… la verdad me da pereza siquiera comprender esas cosas, yo ni las sabía ni las quiero saber :^)
Mejor usemos POO aplicada a la vida real en vez de ver tanto ternísismos, ¿no? Digo, en lo personal no lo entiendo así que creo que saldría mejor usarla nada más.

```
let s1 = "hola";
console.log(typeof(s1)) //retorna "string"

const s2 = new String("hola");
console.log(typeof(s2)) // retorna "object"
```

los objetos no son variables realmente, son eso, objetos.
De hecho uno de los objetos más grandes que hay está en su navegador. Al abrir la consola, pueden escribir “window” y abrirá las características del objeto.
Una función de este objeto window es “alert”. Alert despliega ese mensajito castroso que todo el mundo odia, el popup. Podemos invocarlo como window.alert(“hola”)… pero lo que pasa es que el documento lo estamos escribiendo dentro de Window :00000000000
Entonces podemos solamente escribir “alert(‘hola;)” y sería exactamente lo mismo. Es la hermosa “herencia”, pero aquí es más casual. Si no entendiste está bien, no espero que lo entiendas, esto se empieza a poner complicado.

Ahora, veamos una definició de que es el “this”:

```
const libro1 = {
  titulo: "Libro Uno",
  autor: "John Doe",
  anio: 2018,
  descripcion: function()  {
    return `${this.titulo} fue escrito por ${this.autor}, en
${this.anio}`;
  }
}
console.log(libro1.descripcion());
```

Okay, este es un ejemplo similar al anterior pero con una sintaxis diferente.
Qué es el this? “this” se refiere al padre de la función en la que nos encontramos.
Es decir, nos encontramos en “descripción”. Más específicamente, “libro1,descripción()”. Si escribimos ${titulo} dará como resultado “undefined” porque en ningún lado de la función la

has declarado, pero si en el objeto. Así que para acceder al objeto, simplemente accedes a él con un “this” y ya, nada más.

```
console.log(Object.keys(libro1));
```

Este es para obtener las keys del objeto:

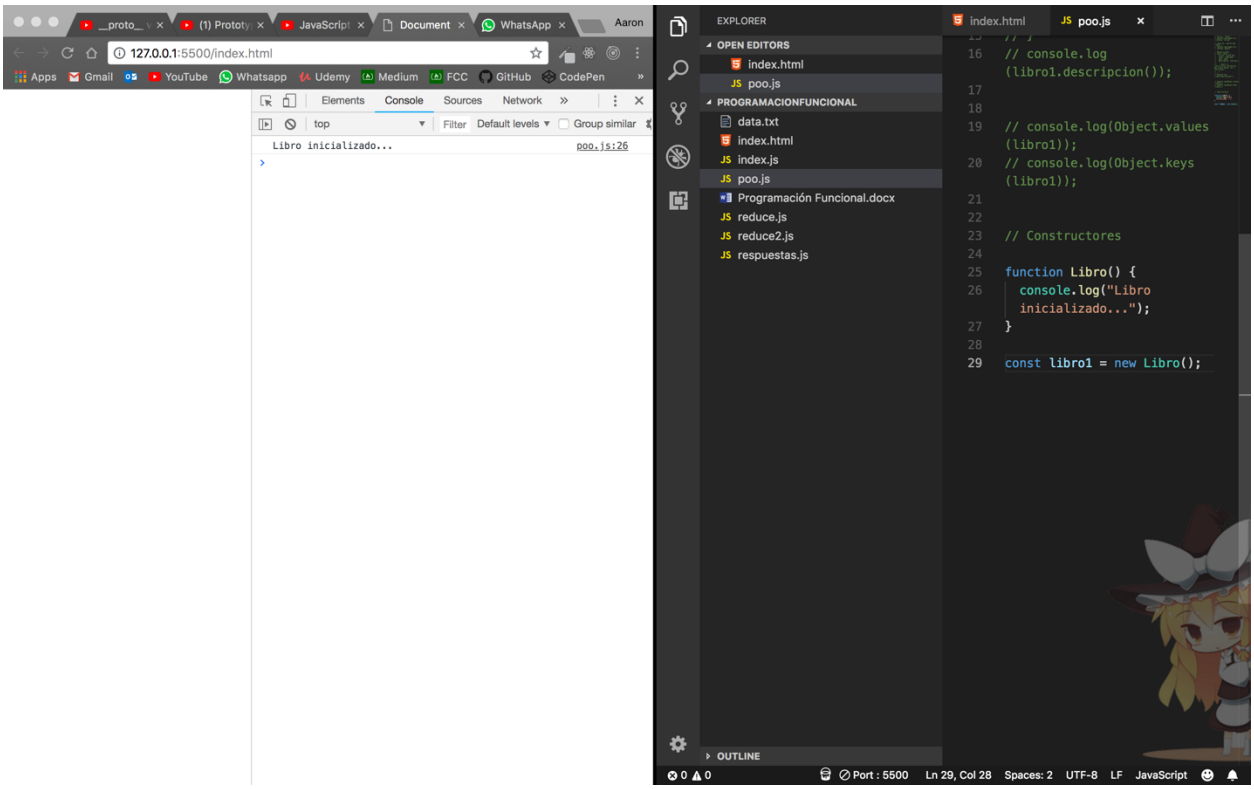

```
console.log(Object.values(libro1));
```

Y este para ver los valores de un objeto:

Ahora, supongamos que queremos tener varios libros… pero para cada libro, además de tener que agregarle los valores, hay que agregarle la función de “descripción” para cada uno. Eso no es optimo :s entonces que hacemos? UN CONSTRUCTOR
WOOOOOO

```
// Constructores

function Libro() {
  console.log("Libro inicializado...");
}

const libro1 = new Libro();
```

HAY
MUCHO
ESPACIO
EN
BLANCO
ASI
QUE
ESCRIBO
ASI
PARA
LLENAR
EL
VACIO
QUE ELLA ME DEJO :(



Si hace Zoom se mira.

Ahora, podríamos tener varios libros de esta manera:

```
function Libro() {
  console.log("Libro inicializado...");
}

const libro1 = new Libro();
const libro2 = new Libro();
const libro3 = new Libro();
```

:D

Pero el código es inútil. Hagamoslo interesante.

```

const libro1 = new Libro('Libro 1', 'John Doe', '2018');
const libro2 = new Libro('Libro 2', 'Camilla Orantes', '1914');
const libro3 = new Libro('Mi lucha', 'Adolfo', '1939');

console.log(libro3.titulo);
```

Ya sirve de algo :B

```
function Libro(titulo, autor, anio) {
  this.titulo = titulo;
  this.autor = autor,
  this.anio = anio;
  this.descripcion = () => {
    return `${this.titulo} fue escrito por ${this.autor} en ${this.anio}`;
  }
}

const libro1 = new Libro('Libro 1', 'John Doe', '2018');
const libro2 = new Libro('Libro 2', 'Camilla Orantes', '1914');
const libro3 = new Libro('Mi lucha', 'Adolfo', '1939');

console.log(libro3.descripcion());
```

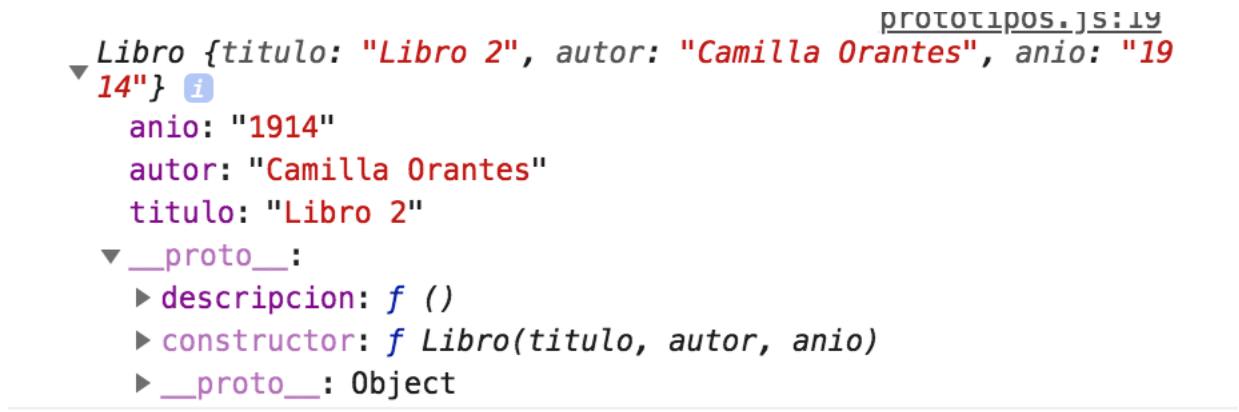
Listo, ya está, sirve de algo.
Ahora todos los libros pueden devolver por una función de heredada de los objetos tipo “Libro” una string con los datos de ese libro.

Dato curioso: En la primera forma mostrada no acepta sintaxis de ES6. Pero en esta forma si. La POO está gonita a que no… CUANDO ES UTIL Y NO TIENES QUE USARLA CADA PUTA VEZ AUNQUE NO QUIERAS.

```
// Constructor
function Libro(titulo, autor, anio) {
  this.titulo = titulo;
  this.autor = autor,
  this.anio = anio;
}

// prototipo de descripcion
Libro.prototype.descripcion = function() {
  return `${this.titulo} fue escrito por ${this.autor} en ${this.anio}`;
}

const libro1 = new Libro('Libro 1', 'John Doe', '2018');
const libro2 = new Libro('Libro 2', 'Camilla Orantes', '1914');
const libro3 = new Libro('Mi lucha', 'Adolfo', '1939');

console.log(libro2.descripcion());
```

¿Cual es la diferencia de un prototipo a una propiedad?
Pues console.logueemoslo…

```
console.log(libro2);
```



WAAAAAAAAAAO AHORA DESCRIPCION NO ES PARTE DEL LIBRO SINO QUE ES PARTE DE SUS PROTOTIPOS OOOOOOOOHHH MAAAAAI GAAAAAAAAAAAAAAAAAAAAD

Aunque no lo crean, varias personas hacen su vida trabajando de Javascript y no saben sobre esto. Es realmente interesante, especialmente porque la gente cree que los que crean los prototipos como “map” son genios que hicieron no sé que cambio con el diablo para poder meter algo nuevo al lenguaje. Map es tan aceptado que los novatos lo considera casi una parte nativa de Javascript, pero no lo es, es solo un prototipo. De hecho, si te esfuerzas te darás cuenta que es tan simple como un for de 0 al this.array.length. ASI DE SIMPLE :000
Pero escribir map se siente más delicioso que hacer un jodido for, ¿a que si?

PONGAMOSLE SABOR A LA COSA, USEMOS OBJETOS DENTRO DE OBJETOS HUEHUEHUE

```
Libro.prototype.obtenerEdad = function() {
  return new Date().getFullYear() - this.anio;
}
```

New Date() retorna una cadena de cosas raras que interpreta como momento actual. Desde el año hasta el milisegundo. getFullYear() solo lo convierte a 2018 y ya ☺
2018 – fechaDelLibro

```
console.log(libro2.obtenerEdad());
console.log(libro1.obtenerEdad());
```

```
104                                                    prototipos.js:24
0                                                      prototipos.js:25
```

CUANDO LES VAN A ENSEÑAR COSAS TAN UTILES EN LA UNIVERSIDAD PAPAAAAAAAAA

¿QUIEREN MAS? ECHEMOSLE MAS AZUCAR SINTETICA AL CAFÉ DESCAFEINADO JAJAJAJA

```
// Yo tambien puedo hacer mis funciones como map
Libro.prototype.cambiarAnio = function(nuevoAnio) {
  this.anio = nuevoAnio;
}


console.log("el año actual es: ",libro1.anio);
libro1.cambiarAnio(2000);
console.log("el nuevo año es ",libro1.anio);
```

AHUEVOS

```
el año actual es:  2018                                prototipos.js:32
el nuevo año es  2000                                  prototipos.js:34
```

¿Han notado que el primer año está en negrito y el segundo en azul? Es porque el primero es String y el segundo es int.

Si quieren operarlo cambienlo de

```
const libro1 = new Libro('Libro 1', 'John Doe', "2018");
```

a

```
const libro1 = new Libro('Libro 1', 'John Doe', 2018);
```

o pueden aceptar cualquiera de los dos y editarlo en el constructor:

```
function Libro(titulo, autor, anio) {
  this.titulo = titulo;
  this.autor = autor,
  this.anio = parseInt(anio);}
```

¿Que bonito que es no? Creando prototipos en los constructores de objetos…

Hasta ahora todo bien… sería una lastima que alguien decidiera utilizar la programación orientada a objetos en todo su esplendor para que escalar la aplicación

```
// Constructor

function Revista(titulo, autor, anio, mes) {
  Libro.call(this, titulo, autor, anio);

  this.mes = mes;
}

// Instanciando el Objeto
const res1 = new Revista("revista 1", "John Doe", "2018", "Febrero");

//Invocandolo
console.log(res1);
```

creamos un constructor “Revista” que, como desea crear los objetos “titulo, autor, anio” y eso mismo hace el constructor Libro, lo incluimos para ahorrar código :D
la sintaxis es:
otroObjeto.call(this, parametro1, parametro2, parametroN);

el parámetro “this” siempre debe de ser asignado para que el resultado lo retorne.

También agregamos el this.mes = mes; porque eso no lo puede hacer el libro, así que lo hacemos manualmente.

Y obtenemos nuestra revista:

```
prototipos.js:48
Revista {titulo: "revista 1", autor: "John Doe", anio: 2018, mes: "Febrero"}
  anio: 2018
  autor: "John Doe"
  mes: "Febrero"
  titulo: "revista 1"
  __proto__: Object
```

Y COMO LA PROGRAMACION ORIENTADA A OBJETOS SE BASA EN HERENCIA PODEMOS OBTENER LOS MISMOS PROTOTIPOS HAHA

```
res1.obtenerEdad();
```

daría como resultado:

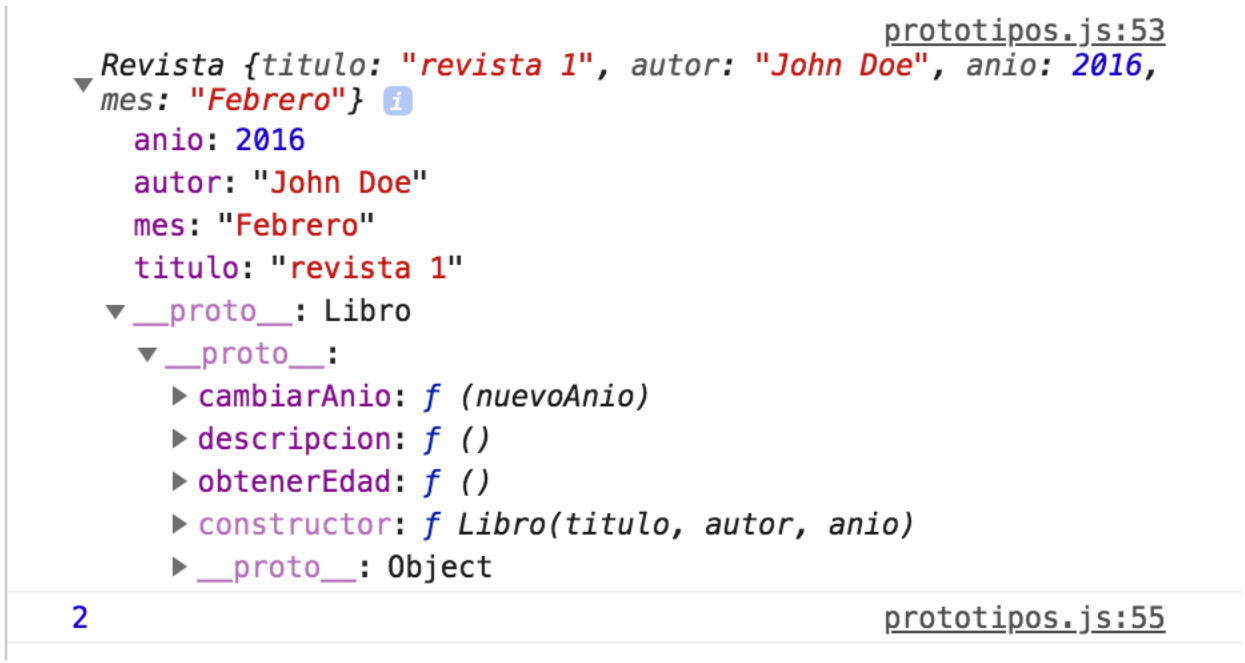
```
Uncaught TypeError: res1.obtenerEdad is not a function
    at prototipos.js:52
```

Es café descafeinado, no es perfecto, ¿OK?

```
// Constructor
function Revista(titulo, autor, anio, mes) {
  Libro.call(this, titulo, autor, anio);

  this.mes = mes;
}

// Heredando el Prototype
Revista.prototype = Object.create(Libro.prototype);

// Instanciando el Objeto
const res1 = new Revista("revista 1", "John Doe", "2016", "Febrero");


//Invocandolo
console.log(res1);

console.log(res1.obtenerEdad());
```

Lo único que se agrega es el el Object.create(), es tan simples como eso.

Hoy si da:B



Object create

Esta es una aproximación más orientada a lo común de los objetos en Javascript:

```
// Objeto de Protos
const ProtosDeLibros = { // Antes se llamaba "obtenerDescripcion"
  getResumen : function() {
    return `${this.titulo} fue escrito por ${this.autor} en ${this.anio}`;
  },
  getEdad: function() { // Antes se llamaba "obtenerEdad"
    return new Date().getFullYear() - this.anio;
  }
}

// Crear Objeto
const libro1 = Object.create(ProtosDeLibros);
libro1.titulo = "Libro 1";
libro1.autor = "John Doe";
libro1.anio = "2013";

console.log(libro1);
```

también podemos declarar el objeto con los parámetros instantáneamente:

```
// Crear Objeto
const libro1 = Object.create(ProtosDeLibros, {
  titulo: {value: "Libro 1"},
  autor: {value: " John Doe"},
  anio: {value: "2013"}
});

// libro1.titulo = "Libro 1";
// libro1.autor = "John Doe";
// libro1.anio = "2013";

console.log(libro1);
```

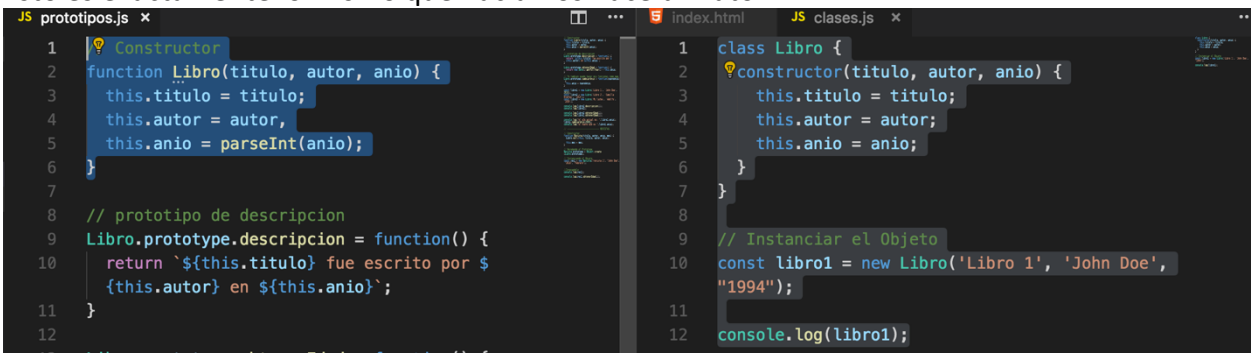
Uff, eso fue largo… verdad?
De hecho eso fue algo así como un for. Ahora vamos a ver la parte fácil, eso mismo pero con sintaxis de ES6, osea como si fuera un map.

QUE VENGAN LAS CLASES

Si ya has programado en un lenguaje de POO como Java, esto es izi moni:

```
class Libro {
  constructor(titulo, autor, anio) {
    this.titulo = titulo;
    this.autor = autor;
    this.anio = anio;
  }
}

// Instanciar el Objeto
const libro1 = new Libro('Libro 1', 'John Doe', "1994");

console.log(libro1);
```

Esto es exactamente lo mismo que hacíamos hace un rato



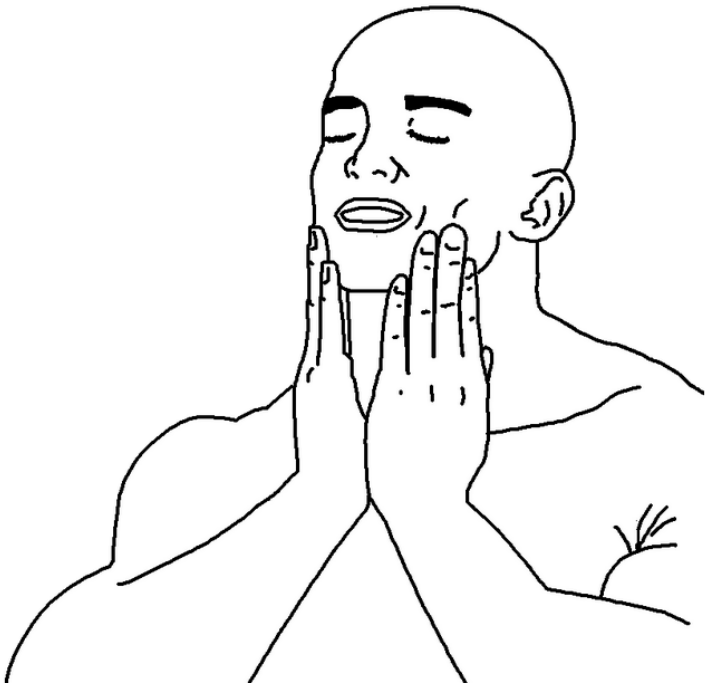
Pero en ES6 :^)

Pero wait… THERE'S MORE!

Si queremos agregar un método a la clase:

```
class Libro {
  constructor(titulo, autor, anio) {
    this.titulo = titulo;
    this.autor = autor;
    this.anio = anio;
  }
  getResumen() {
    return `${this.titulo} fue escrito por ${this.autor} en ${this.anio}`;
  }
  getEdad() {
    return new Date().getFullYear() - this.anio;
  }
  setAnio(nuevoAnio) {
    this.anio = nuevoAnio;
  }
}

// Instanciar el Objeto
const libro1 = new Libro('Libro 1', 'John Doe', "1994");

console.log(libro1);
```

PERO MIRATE ESOOOO PAPAAAAAAA sin prototipos, ni nada de eso, solo declaramos la función dentro de la clase y ya está. **DELICIOSO**



Y hasta ahí con las clases… ahora vamos a las subclases

```
class Libro {
  constructor(titulo, autor, anio) {
    this.titulo = titulo;
    this.autor = autor;
    this.anio = anio;
  }
  getResumen() {
    return `${this.titulo} fue escrito por ${this.autor} en ${this.anio}`;
  }
}

class Revista extends Libro {
  constructor(titulo, autor, anio, mes) {
    super(titulo, autor, anio);
    this.mes = mes;
  }
}
// Instanciar el Objeto
const rev = new Revista('Revista 1', "John Doe", "2018", "Abril");

console.log(rev);
```

Listo, es un Objeto llamado Revista que extiende Libro.
Puedes hacer así un objeto dentro de otro y otro y otro y otro hasta que odies a Java :^)

Este Script hace lo mismo que le script de hace un rato para las revistas en ES5.