PROGRAMACIÓN AVANZADA EN JAVASCRIPT

Módulo 3

OTEC PLATAFORMA 5 CHILE

MÓDULO 3

CLASE 7

Introducción

Herencia

Prototype Chain

Proto

Funciones constructoras

New

Introducción

En estas clases, tanto la de hoy como la siguiente, trabajaremos partes muy conceptuales de los Objetos.

Sabemos que son abstractos y no son fáciles de comprender, es por eso que iremos concepto a concepto.

Cabe destacar que al momento de realizar los ejercicios, no es necesario que tengan todos estos conocimientos, pero si son imprescindibles para que pueda realizar códigos escalables y profesionales.

Herencia

La herencia es la capacidad de ciertos elementos de adquirir propiedades o métodos de otros elementos que podrían englobarlos. Así como una persona hereda características de sus padres, un modelo informático puede heredar - es decir, copiar estructuras- de otro modelo que se defina como más genérico.

Cuando trabajamos en JavaScript, los conceptos de herencia son distintos a los lenguajes típicos de Programación Orientada a Objetos. Esto se debe a que utilice Prototype Inheritance como método de herencia.

Prototype Chain

En lo que a herencia se refiere, JavaScript sólo tiene una estructura: objetos. Cada objeto tiene un enlace interno a otro objeto: su prototype. Ese objeto prototype tiene su propio prototype, y así sucesivamente hasta que se alcanza un objeto cuyo prototype es null. Por definición, null no tiene prototype, y actúa como el final de esta cadena de prototipos (Prototype Chain).

Si queremos ver estos prototypes podemos encontrarlos en la propiedad __proto__. Pruebe el siguiente código en la consola:

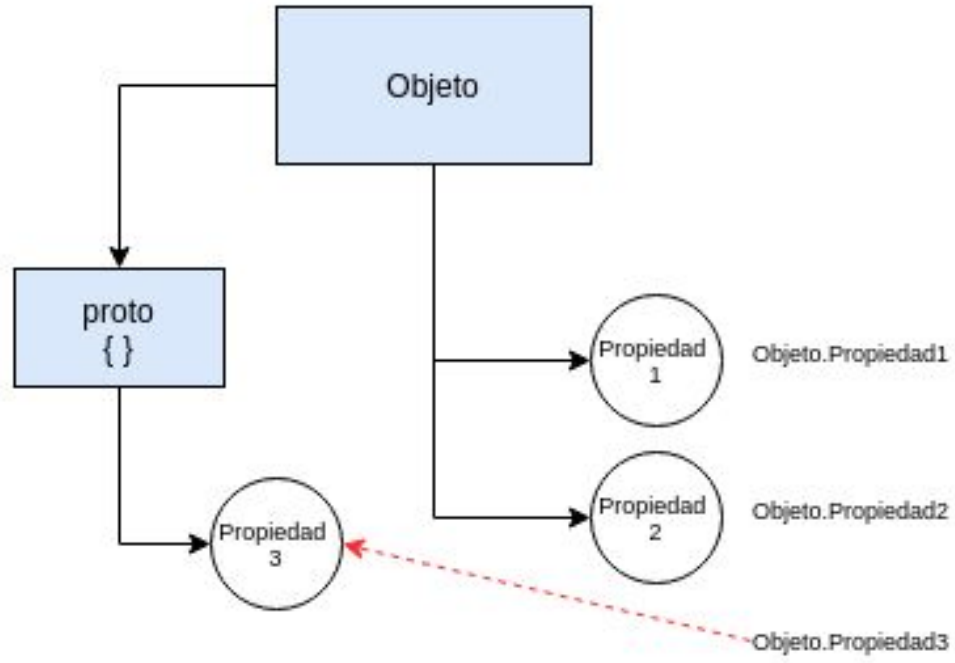
```
var a = [1,2,3]
a.__proto__
```

Puede ver que en la respuesta están todas los métodos de los Arreglos que conoce.

La Propiedad proto Y La Prototype Chain

En el ejemplo tenemos al 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 que, a su vez, tiene una propiedad llamada propiedad3. Lo interesante es que si nosotros queremos acceder a la propiedad3 del objeto Objeto lo vamos a poder hacer.



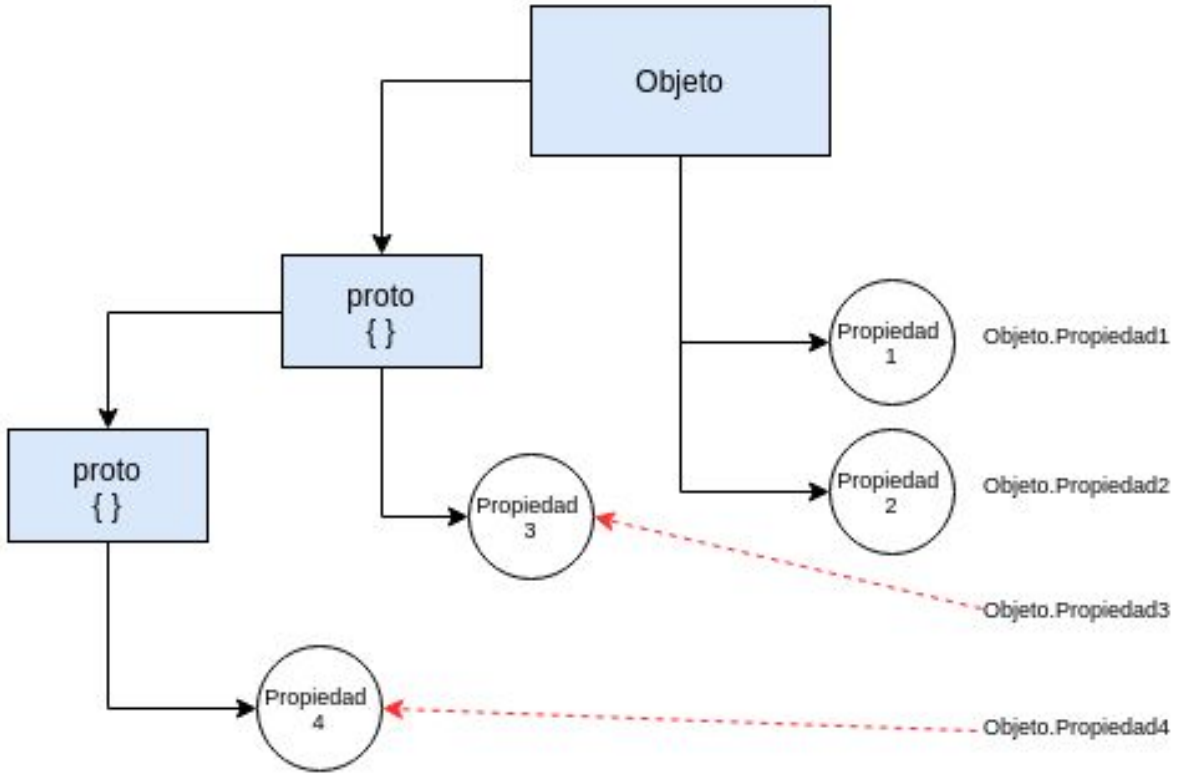
Cuando escribimos Objeto.propiedad3 lo que ocurre es que el intérprete busca en el objeto por esa propiedad y si no la encuentra, antes de lanzar un error, busca en el objeto proto (que lo tienen todos los objetos).

La Propiedad proto Y La Prototype Chain

De hecho, el objeto al que hace referencia proto también podría tener una referencia a otro proto. Si nosotros intentamos acceder a propiedad4 desde Objeto usando Objeto.propiedad4, el intérprete primero buscaría en Objeto. Como no está en esa propiedad, va a buscar en el objeto al que hace referencia proto. Como tampoco se encuentra ahí, se fija si ese objeto tiene una referencia en proto. En este caso, como la tiene, va a buscar la propiedad en ese objeto al que hace referencia.

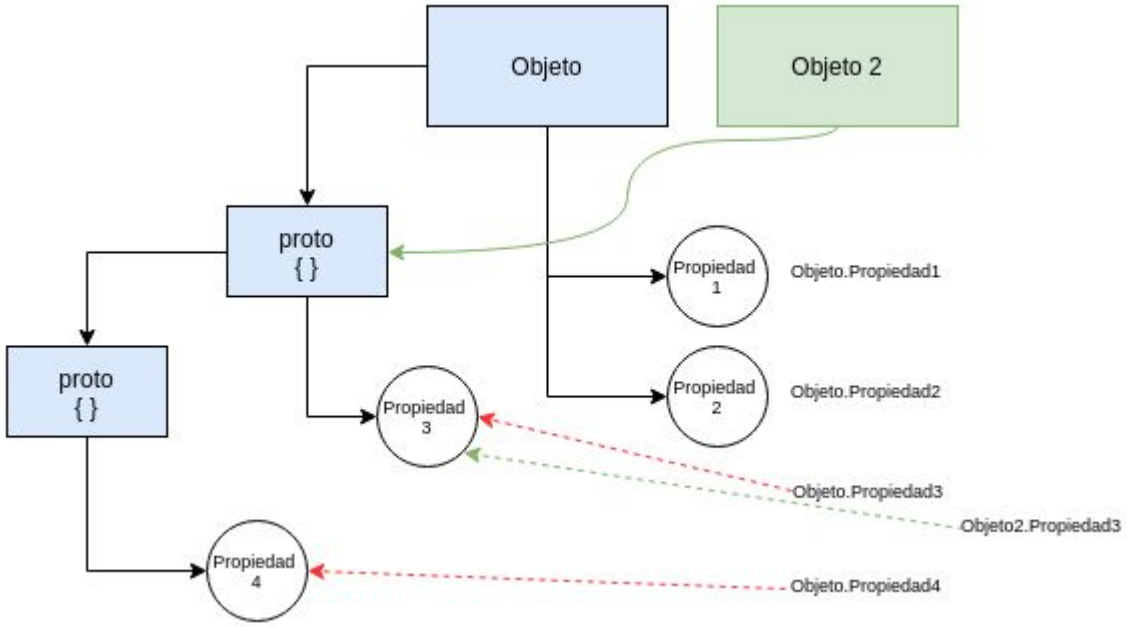
En el ejemplo, finalmente encuentra la propiedad4 en este último y por lo tanto la accede.

Esto es gracias al Prototype Chain.



La Propiedad proto Y La Prototype Chain

En este caso, si quisiéramos acceder a Objeto2.propiedad3 (que no existe en el objeto 2), la encontraríamos siguiendo el prototype chain, y accederemos a la misma propiedad que si hiciéramos Objeto1.propiedad3.




```
//Ejemplo
var persona = {
 getNombre: function () {
   return this.nombre + " " +
this.apellido;
 },
};

var facu = {
 nombre: "Flor",
 apellido: "Páez",
};

// esta no es una buena forma de asignar
prototypes
// es solo de demostración
facu.__proto__ = persona;

// Ahora podemos usar los métodos de
`persona`
facu.getNombre();
("Flor Páez");
```

Todo elemento es un Objeto

Todo tipo de dato complejo en JavaScript es un Objeto. Sin embargo, hay un objeto especial que no tiene un prototipo. Este objeto se llama base object. 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: por lo tanto, estos métodos y propiedades son accesibles por todos los objetos de JavaScript. Por ejemplo, el método toString está definido en el base object.

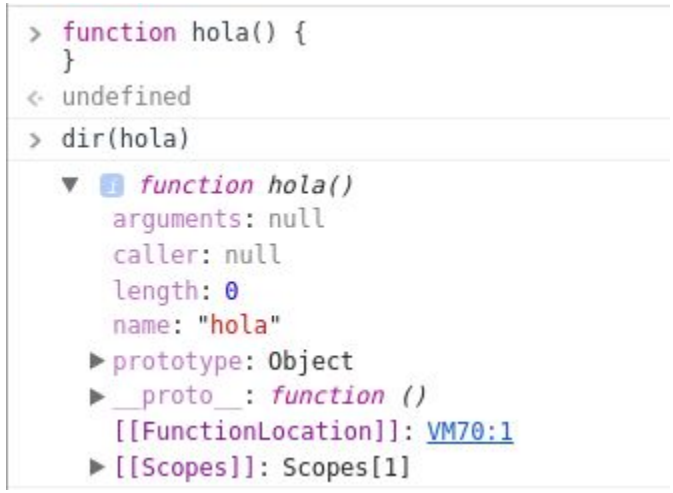
En el caso de cualquier función, su prototipo por defecto es un objeto llamado Empty, que es a su vez una función. Cualquier función que creemos va a tener este proto y, por ende, van a tener acceso a todas las propiedades y métodos de Empty. Por ejemplo, la funciones apply, bind y call están definidas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

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


Cuando me dicen que ignore la complejidad, pero ignorar está bien difícil

Funciones constructoras

Vimos cómo funciona la propiedad proto y que contiene un vínculo que los relaciona al siguiente elemento del prototype chain. Si analizamos una función podemos ver que hay una variable prototype además de __proto__:



Esta variable prototype no es otra manera de llamar a proto. Es un objeto totalmente distinto que aparece solo en las funciones: tiene un método inicial llamado constructor y se utiliza solo cuando una función está siendo usada como constructor.

Un *constructor* es una función que tiene como finalidad crear un nuevo objeto.

Los nombres de las funciones constructoras se ponen con la primer letra en mayúscula por convención. Así, podés reconocer cuándo una función es un constructor.

Pero, hay algo raro, ¿no? Por empezar no retorna nada. Además, no está claro a qué hace referencia el keyword this. Si se fija, cuando se usa con *new*, crea un objeto con las propiedades definidas en esa función.

```
function Persona(name, surname)
{
 this.nombre = name;
 this.apellido = surname;
}

var flor = new Persona("Flor",
"Páez");

flor; // Persona {nombre:'Flor',
apellido:'Páez'}
```

New

Primero, tiene que saber que new es en realidad un operador de JavaScript. Este es su funcionamiento:

1. **Crea un objeto vacío.**
2. **Invoca la función que le pasamos como argumento, con la particularidad de que lo hace con el contexto de ejecución ligado a dicho objeto (call). De esta forma, en esa función, el keyword this hace referencia a este objeto nuevo.**
3. **Una vez que la función pasada por parámetro ya corrió, retorna ese objeto que había creado (y que debería haber sido modificado por la función ejecutada) como el objeto creado.**

```
// una versión simplificada de las funcionalidades de new se vería así
function new (constructor){
  var a = {};
  constructor.call(a);
  return a;
}
```

New

Así, podemos crear muchas instancias distintas a partir de esta función constructora:

```
var javi = new Persona('Javiera', 'Alfaro');
var doris = new Persona('Doris', 'Hidalgo');
var flor = new Persona('Flor', 'Páez');
```

Recuerde: **estamos invocando una función, así que podemos utilizar dentro todo lo que sabemos de funciones.**

Si quiere saber si un objeto es una instancia de cierto constructor, puede usar el método instanceof:

```
javi instanceof Persona
true
```

MÓDULO 3

GRACIAS

Aquí finaliza la clase n°7 del módulo 3

OTEC PLATAFORMA 5 CHILE