

Curso Básico de Programación Orientada a Objetos con JavaScript

Los **prototipos** son un mecanismo mediante el cual los objetos en JavaScript heredan características entre sí.

JavaScript es a menudo descrito como un lenguaje basado en **prototipos** - para proporcionar mecanismos de herencia, los objetos pueden tener un objeto prototipo, el cual actúa como un objeto plantilla que hereda métodos y propiedades.

Un objeto prototipo del objeto puede tener a su vez otro objeto prototipo, el cual hereda métodos y propiedades, y así sucesivamente. Esto es conocido con frecuencia como la **cadena de prototipos**, y explica por qué objetos diferentes pueden tener disponibles propiedades y métodos definidos en otros objetos.

Los métodos y propiedades son definidos en la propiedad **prototype**, que reside en la función constructora del objeto, no en la instancia misma del objeto.

En JavaScript, se establece un enlace entre la instancia del objeto y su prototipo (su propiedad **__proto__**), y las propiedades y metodos son encontrados recorriendo la cadena de prototipos.

Nota: Es importante entender que, tanto el prototipo de la instancia de un objeto como el prototipo que contiene el constructor hacen referencia al mismo objeto.

Se puede acceder al prototipo de la instancia de un objeto mediante

Object.getPrototypeOf(obj) o a través de la propiedad **__proto__**.

Ejemplo: Definimos nuestro constructor Persona()

```
function Persona(nombre, apellido, edad, genero, intereses) {  
  
    // definiendo de propiedades y métodos  
    this.first = first;  
    this.last = last;  
    //...  
}
```

```
var person1 = new Persona('Bob', 'Smith', 32, 'hombre', ['music', 'skiing']);
```

Si escribe "person1." en su consola JavaScript, debería ver que el navegador intenta completarlo automáticamente con los nombres de miembro disponibles en este objeto.

En esta lista, podrá ver los miembros definidos en el objeto prototipo de person1, que es la Persona() (Persona() es el constructor). **Sin embargo**, también verá algunos otros miembros - watch, valueOf, etc - que están definidos en el objeto prototipo de Persona() 's, que es un Objeto (**Object**). Esto demuestra que el **prototipo cadena** funciona.

Entonces, ¿dónde se definen las propiedades y métodos heredados?

La respuesta es que los heredados son los que están definidos en la propiedad **prototype** (podría llamarse subespacio de nombres), es decir, los que empiezan con **Object.prototype**, y no los que empiezan sólo con **Object**. El valor de la propiedad del prototipo es un objeto, que es básicamente un repositorio(bucket) para almacenar propiedades y métodos que queremos que sean heredados por los objetos más abajo en la cadena del prototipo.

Si escribe en la consola: **Object.prototype** verá un gran número de métodos definidos en la propiedad Prototype de Object, que están disponibles en los objetos que heredan de Object, como se ha mostrado anteriormente.

Hay otros ejemplos de herencia de cadena de prototipos en todo JavaScript; los métodos y propiedades definidas en el prototipo de los objetos globales String, Date, Number y Array, por ejemplo.

Así que cuando se crea una cadena, como ésta:

```
var myString = 'Esto es mi String.';
```

myString inmediatamente tiene una serie de métodos útiles disponibles en él, como **split()**, **indexOf()**, **replace()**, etc.

Object.create() crea una nueva instancia de objeto.

```
var person2 = Object.create(person1);
```

Lo que hace **create()** es crear un nuevo objeto a partir de un objeto prototipo específico.

Aquí, **person2** se crea utilizando la **person1** como objeto prototipo. Puede comprobarlo introduciendo lo siguiente en la consola: **person2.__proto__**. Esto devolverá el objeto Persona.

La propiedad constructor

Cada función de constructor tiene una propiedad **prototype** cuyo valor es un objeto que contiene una propiedad **constructor**. Esta propiedad **constructor** apunta a la función constructor original.

Las propiedades definidas en la propiedad **Persona.prototype** (o en general en la propiedad **prototype** de una función de constructor, que es un objeto, como se mencionó en la sección anterior) se hacen disponibles a todas las instancias de objetos creadas utilizando el constructor **Persona()**. Por lo tanto, la propiedad del **constructor** también está disponible tanto para los objetos **person1** como para los objetos **person2** y así sucesivamente.

La propiedad **constructor** tiene otros usos. Por ejemplo, si se tiene una instancia y se quiere devolver el nombre del que el constructor de la instancia, se puede usar lo siguiente:

instanceName.constructor.name

VER IMAGEN ABAJO

```
> [1,2].constructor.name
< 'Array'
> cadena.constructor.name
< 'String'
> persona2.constructor.name
< 'Persona'
```

Modificando prototipos

Vamos a echar un vistazo a un ejemplo para modificar la propiedad prototype de una función constructor (los métodos añadidos a la propiedad prototipo están disponibles en todas las instancias de los objetos creados a partir del constructor).

El siguiente código, el cuál añade un nuevo método a la propiedad prototype del constructor:

```
Person.prototype.farewell = function() {
    alert(this.name.first + ' has left the building. Bye for now!');
};
```

```
person1.farewell();
```

Con esto, deberías obtener un mensaje de alerta mostrando el nombre de la persona como se define dentro del constructor. Esto es realmente útil, pero lo que es más útil es que toda la cadena de herencia se ha actualizado dinámicamente; **automáticamente hace que este nuevo método esté disponible en todas las instancias del objeto creadas desde el constructor.**

Un patrón bastante común para la mayoría de definiciones de objetos es declarar las propiedades dentro del constructor, y los métodos en el prototipo. Esto hace el código más fácil de leer, ya que el constructor sólo contiene las definiciones de propiedades, y los métodos están en bloques separados. Por ejemplo:

```
// Constructor with property definitions

function Test(a, b, c, d) {
    // property definitions
}

// First method definition

Test.prototype.x = function() { ... };

// Second method definition

Test.prototype.y = function() { ... };
```

¿Qué es abstracción?

La abstracción consiste en abstraer los datos de un objeto para crear su molde, prototipo o clase. Para posteriormente crear instancias de dichos prototipos.

¿Qué es encapsulamiento?

Es la forma de proteger, encapsular, guardar, limitar, esconder el acceso de ciertos atributos y métodos de nuestros objetos.

Encapsular consiste en:

- Esconder métodos y atributos.
- No permitir la alteración de métodos y atributos.

Getters y Setters en JavaScript

Los getter y setters en las instancias no se manejan como metodos, sino como propiedades. En este caso: **cursoProgBasica.name** - obtiene el valor de propiedad a traves del getter.

cursoProgBasica.name = "Otro valor"; - actualiza la propiedad a traves del setter.

```
1  class Course {
2    constructor({name, classes = []}) {
3      this._name = name;
4      this.classes = classes;
5    }
6
7    get name() {
8      return this._name;
9    }
10
11   set name(nuevoNombre) {
12     if(nuevoNombre.toLowerCase().includes("malo")) {
13       console.warn("Nombre de curso inválido");
14     } else {
15       this._name = nuevoNombre;
16     }
17   }
18 }
```

```
> cursoProgBasica
< ▶ Course {_name: 'Curso gratis de programación básica', classes: Array(3)}
> cursoProgBasica.name;
< 'Curso gratis de programación básica'
> cursoProgBasica.name = "Curso gratis de programación básica de Platzi";
< 'Curso gratis de programación básica de Platzi'
> cursoProgBasica.name;
< 'Curso gratis de programación básica de Platzi'
```

¿Qué es la herencia?

La herencia permite que se puedan definir nuevas clases basadas de unas ya existentes a fin de reutilizar el código, generando así una jerarquía de clases dentro de una aplicación. Si una clase deriva de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.

En lo que a herencia se refiere, **JavaScript** sólo tiene una estructura: objetos. Cada objeto tiene una propiedad privada (referida como su `[[Prototype]]`) que mantiene un enlace a otro objeto llamado su prototipo. Ese objeto prototipo tiene su propio prototipo, y así sucesivamente hasta que se alcanza un objeto cuyo prototipo es `null`. Por definición, `null` no tiene prototipo, y actúa como el enlace final de esta cadena de prototipos.

Los objetos en JavaScript poseen un enlace a un objeto prototipo. Cuando intentamos acceder a una propiedad de un objeto, la propiedad no sólo se busca en el propio objeto sino también en el prototipo del objeto, en el prototipo del prototipo, y así sucesivamente hasta que se encuentre una propiedad que coincida con el nombre o se alcance el final de la cadena de prototipos.

NOTA: Cuando una función heredada se ejecuta, el valor de **this** apunta al objeto que hereda, no al prototipo en el que la función es una propiedad. Es decir: No importa dónde se encuentre el método: en un objeto o su prototipo. En una llamada al método, **this** es siempre el objeto antes del punto.


```
var o = {
  a: 2,
  m: function() {
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// Cuando en este caso se llama a o.m, 'this' se refiere a o

var p = Object.create(o);
// p es un objeto que hereda de o

p.a = 12; // crea una propiedad 'a' en p
console.log(p.m()); // 13
// cuando se llama a p.m, 'this' se refiere a p.
// De esta manera, cuando p hereda la función m de o,
// 'this.a' significa p.a, la propiedad 'a' de p
```

```
class Student {  
  constructor(name) {  
    this.name = name;  
  }  
}  
  
class FreeStudent extends Student {}
```



Herencia a la antigua con prototipos:

```
> const rabbit = {jumps: true };  
< undefined  
> const animal = {eats: true};  
< undefined  
> rabbit.__proto__ = animal;  
< ▶ {eats: true}  
> rabbit  
< ▼ {jumps: true} ⓘ  
  jumps: true  
  ▼ [[Prototype]]: Object  
    eats: true  
    ▶ [[Prototype]]: Object  
> rabbit.eats;  
< true
```

Otra forma de herencia con prototipos:

IMAGEN ABAJO (SIGUIENTE PAGINA)

```

function Padre(a, b) {
  this.a = a;
  this.b = b;
}
Padre.prototype.sumar = function () {return this.a + this.b;};

const obj1 = new Padre(1,2);
console.group("obj1");
console.log(obj1.sumar());
console.groupEnd();

function Hijo(y,z) {
  this.y = y;
  this.z = z;
}

//Se aplica la herencia
Hijo.prototype = new Padre(1, 2);

const obj2 = new Hijo("a", "b");
console.group("Obj2");
console.log(obj2.z);
console.log(obj2.a);
console.log(obj2.sumar());
console.groupEnd();

```

El método **Object.getPrototypeOf()** devuelve el prototipo (es decir, el valor de la propiedad interna `[[Prototype]]`) del objeto especificado. Ejemplo: **Object.getPrototypeOf(obj)**

```

var proto = {};
var obj= Object.create(proto);
Object.getPrototypeOf(obj) === proto; // true

```

Prototype	proto
Prototypes is a simple way to share behavior and data between multiple objects access using .prototype	proto is also a way to share behavior and data between multiple objects access using __proto__
All the object constructors (function) have prototype properties.	All the objects have proto property.
The prototype gives access to the prototype of function using function. Syntax: (function.prototype)	proto gives access to the prototype of the function using the object. Syntax: (object.__proto__)
It is mostly used to resolve issues of memory wastage when creating an object in constructor mode then each object has separate behavior.	It is used in the lookup chain to resolve methods, constructors, etc.
It is the property of the class.	It is the property of the instance of that class.
The prototype property is set to function when it is declared. All the functions have a prototype property.	proto property that is set to an object when it is created using a new keyword. All objects behavior newly created have proto properties.
It is introduced in EcmaScript 6.	It is introduced in ECMAScript 5.
It is also called it as .prototype	It is also called dunder proto.
It is mostly used in javaScript.	It is rarely used in JavaScript.

¿Qué es polimorfismo?

En POO, es la capacidad para hacer que al invocar al mismo método desde distintos objetos, cada uno de esos objetos pueda responder a esa invocación de forma distinta.

A grandes rasgos, el polimorfismo permite que nombres dos acciones del mismo modo dentro de tu código, pero que cada una de ellas acepte diferentes parámetros y comportamientos. También puede haber polimorfismo con los atributos de la superclase.

Tipos de polimorfismo: **Sobrecarga, Paramétrico, Inclusión.**

En **JavaScript** al menos por ahora, sólo podemos aplicar polimorfismo por Inclusión.

En este caso, sobreescribimos un método de una superclase en una subclase, para cambiar el comportamiento de éste. Esto se puede reflejar/ver a través de la cadena de prototipos en la consola del navegador.

IMAGEN ABAJO


```
> freddy
< TeacherStudent {name: 'Freddy Vega', email: 'freddy@platzi.com', username: 'freddyfeliz', socialMedia: {...}, approved
  Courses: Array(0), ...} ⓘ
  ▶ approvedCourses: []
    email: "freddy@platzi.com"
  ▶ learningPaths: []
    name: "Freddy Vega"
  ▶ socialMedia: {twitter: undefined, instagram: 'freddier', facebook: undefined}
    username: "freddyfeliz"
  ▼ [[Prototype]]: Student
    ▶ approveCourse: f approveCourse(newCourse)
    ▶ constructor: class TeacherStudent Subclase (herencia)
    ▶ publicarComentario: f publicarComentario(commentContent) Polimorfismo, se sobrescribe el método.
  ▼ [[Prototype]]: Object
    ▶ constructor: class Student Superclase
    ▶ publicarComentario: f publicarComentario(commentContent)
    ▶ [[Prototype]]: Object
```