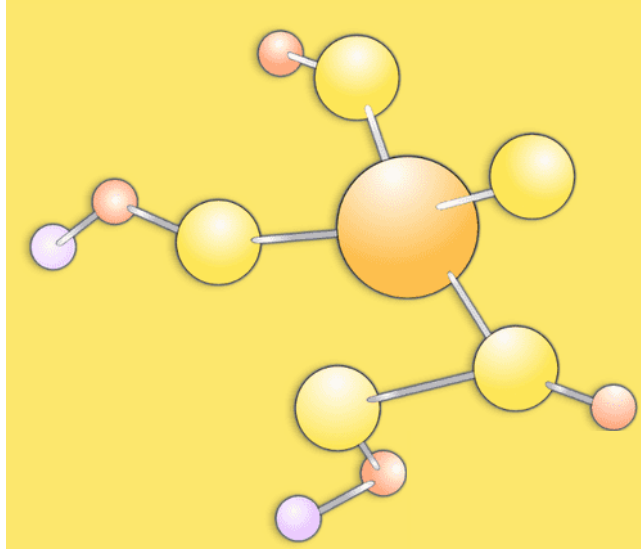


Desarrollo Web en Entorno Cliente

Tema 23





PROTOTYPES

[objetos en js]

Hace ya muchos años, antes de ES6, no existía la funcionalidad de las clases y para poder crear objetos en JS se utilizaban objetos literales.

Así, si se quiere crear una estructura para representar los productos que se vende en un sitio web, utilizando objetos literales sería de la siguiente forma:

No obstante, al no utilizar ninguna plantilla, cada objeto no tiene ninguna obligación de tener las mismas propiedades, *producto2* podría, por ejemplo, no tener una propiedad llamada *precio*.

```
const producto1 ={  
  nombre: "Patatas Fritas",  
  precio: 10 };
```

```
const producto2 ={  
  nombre: "Patatas Bravas",  
  precio: 12 };
```

Esta problemática se ve más acentuada si se quiere implementar algún método en los objetos.

[objetos en js]

Continuando con el ejemplo, si se implementa un método que devuelva el nombre junto con su precio, sería de la siguiente forma:

```
const producto1 = {  
  nombre: "Patatas Fritas",  
  precio: 10,  
  mostrar(){  
    return `${this.nombre} - ${this.precio} euros`; }  
};
```

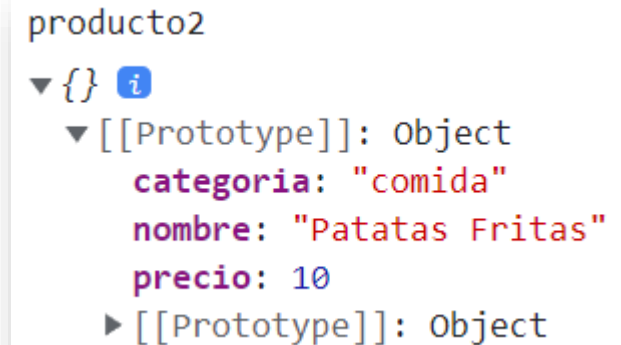
El problema es que hay que repetir la creación del método *mostrar()* en todos los objetos literales del mismo tipo, y si existen muchos objetos literales esto puede ser muy tedioso.

[objetos en js]

Para solucionar estos problemas se ideó una forma sencilla que funciona gracias al “prototipo” (*Prototype*) que fue introducido en ES6.

La solución consiste en almacenar en cada objeto “algo” (el *prototype*) que sirva de plantilla para crear objetos a partir de él y que, además, permita acceder desde el objeto copiado al objeto que sirvió de plantilla, para así poder recorrerlo, accediendo a los ancestros del objeto y, de esta manera, a sus propiedades y métodos. Salvando las distancias, podría considerarse ese *prototype* como “el ADN del objeto”.

```
const producto1 = {  
  nombre: "Patatas Fritas",  
  precio: 10 };  
  
const producto2 = Object.create(producto1);  
producto1.categoria = "comida";  
console.log(producto2);
```



A screenshot of a JavaScript console showing the prototype chain for 'producto2'. The console displays 'producto2' followed by a collapsed object icon. When expanded, it shows '[[Prototype]]: Object' with the following properties: 'categoria: "comida"', 'nombre: "Patatas Fritas"', and 'precio: 10'. Below this, another '[[Prototype]]: Object' is shown, indicating the chain continues to the next object in the hierarchy.

prototype

En JS la mayoría de las cosas son objetos, como, por ejemplo, las funciones, los *string*, los *arrays*... sin embargo, a pesar de que JS es un lenguaje orientado a objetos, está basado en prototipos y no en clases.

Los prototipos son un mecanismo que permite crear objetos a partir de un diseño previo.

Los objetos contienen, de manera implícita, una propiedad prototipo (*prototype*), un objeto literal que actúa como una plantilla (de ahí el nombre de prototipo) y que permite que otro objeto copie esta plantilla y así pueda heredar métodos y propiedades.

Un objeto prototipo puede tener a su vez otro objeto prototipo, el cual hereda métodos y propiedades, y así sucesivamente. Esto se conoce como *cadena de prototipos*, y explica por qué objetos diferentes pueden tener propiedades y métodos definidos en otros objetos.

Los métodos y propiedades de un objeto se definen en la propiedad *prototype*, que se encuentra en el constructor del objeto y no en la instancia de dicho objeto.

prototype

En este ejemplo se define una función constructor, así:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}
```

Se puede observar que *Producto* se escribe en *Upper Camel Case*, se utiliza como convención para funciones que permiten crear instancias, objetos, y de igual manera para definir clases.

La palabra reservada *new* indica que se quiere crear una nueva instancia del objeto devuelva la función.

Y creamos una instancia, un objeto, como este:

```
const producto1= new Producto("Patatas Fritas", 10);
```

prototype

Si se visualiza *producto1* en la consola del navegador se puede ver la cadena de prototipos:



Esto significa que desde *producto1* se pueden acceder a las propiedades y métodos del objeto *Object*.

```
producto1.valueOf()  
► Producto {nombre: 'Patatas Fritas', precio: 10, mostrar: f}
```

Por ejemplo, el método *valueOf()* del objeto *Object* que simplemente devuelve el valor del objeto sobre el que se llama.

```
producto1  
▼ Producto {nombre: 'Patatas Fritas', precio: 10, mostrar: f} ⓘ  
  ► mostrar: f ()  
  nombre: "Patatas Fritas"  
  precio: 10  
  ▼ [[Prototype]]: Object  
    ► constructor: f Producto(nombre, precio)  
    ▼ [[Prototype]]: Object  
      ► constructor: f Object()  
      ► hasOwnProperty: f hasOwnProperty()  
      ► isPrototypeOf: f isPrototypeOf()  
      ► propertyIsEnumerable: f propertyIsEnumerable()  
      ► toLocaleString: f toLocaleString()  
      ► toString: f toString()  
      ► valueOf: f valueOf()  
      ► __defineGetter__: f __defineGetter__()  
      ► __defineSetter__: f __defineSetter__()  
      ► __lookupGetter__: f __lookupGetter__()  
      ► __lookupSetter__: f __lookupSetter__()  
      ► __proto__: (...)  
      ► get __proto__: f __proto__()  
      ► set __proto__: f __proto__()
```


{prototype}



```
producto1.valueOf()  
▶ Producto {nombre: 'Patatas Fritas', precio: 10, mostrar: f}
```

En este ejemplo:

- ❑ El navegador comprueba si el objeto *producto1* tiene un método *valueOf()*.
- ❑ Si no lo tiene el navegador comprueba si el constructor del objeto prototipo del objeto *producto1*, esto es, el constructor del prototipo (*Producto()*) tiene un método *valueOf()*.
- ❑ Si tampoco lo tiene, entonces el navegador comprueba si el constructor del objeto prototipo del objeto prototipo *Producto()* (*Object()*) tiene un método *valueOf()*, y, como lo tiene, lo invoca.

[prototype]

Como se ha podido apreciar en el ejemplo anterior, los métodos y propiedades no se copian de un objeto a otro en la cadena del prototipo, sino que son accedidos recorriendo la cadena de prototipos.

No existe una forma **oficial** de acceder directamente al objeto prototipo de un objeto, los "enlaces" entre los elementos de la cadena de prototipos están definidos en una propiedad interna, denominada `[[Prototype]]`, a la que no puede accederse directamente.

No obstante, desde ECMAScript 2015 se puede acceder indirectamente al objeto prototipo de un objeto mediante `Object.getPrototypeOf(obj)`.

Además, en la actualidad, la mayoría de los navegadores modernos ofrecen y permiten acceder a una propiedad llamada `__proto__` que contiene el constructor del objeto prototipo.

Cuando se crea una instancia de un objeto, se crea una propiedad llamada `__proto__` que deriva de la propiedad `prototype` del constructor.

prototype

La propiedad `__proto__` es un enlace a la propiedad `prototype` del constructor, y así, recorriendo este enlace se puede acceder a las diferentes propiedades y métodos heredados que ha ido heredando el objeto.

La propiedad `__proto__` es accesible desde los objetos, mientras que la propiedad `prototype` solo es accesible desde la función constructor.

Tanto el prototipo de la instancia de un objeto, que se accede mediante `Object.getPrototypeOf(obj)` o a través de la propiedad `__proto__`, como el prototipo que contiene el constructor, que se encuentra en la propiedad `prototype` del constructor, hacen referencia al mismo objeto.

```
> function Producto(nombre, precio){
  this.nombre=nombre;
  this.precio=precio;
  this.mostrar=function(){
    return `${this.nombre} - ${this.precio} euros`;
  }
}
const producto1= new Producto("Patatas Fritas", 10);
< undefined

> producto1
< ▼ Producto {nombre: 'Patatas Fritas', precio: 10, mostrar: f} ⓘ
  ► mostrar: f ()
  nombre: "Patatas Fritas"
  precio: 10
  ▼ [[Prototype]]: Object
    ► constructor: f Producto(nombre, precio)
    ► [[Prototype]]: Object
```

```
Object.getPrototypeOf(producto1)
▼ {constructor: f} ⓘ
  ► constructor: f Producto(nombre, precio)
  ► [[Prototype]]: Object
```

```
producto1.__proto__
▼ {constructor: f} ⓘ
  ► constructor: f Producto(nombre, precio)
  ► [[Prototype]]: Object
```

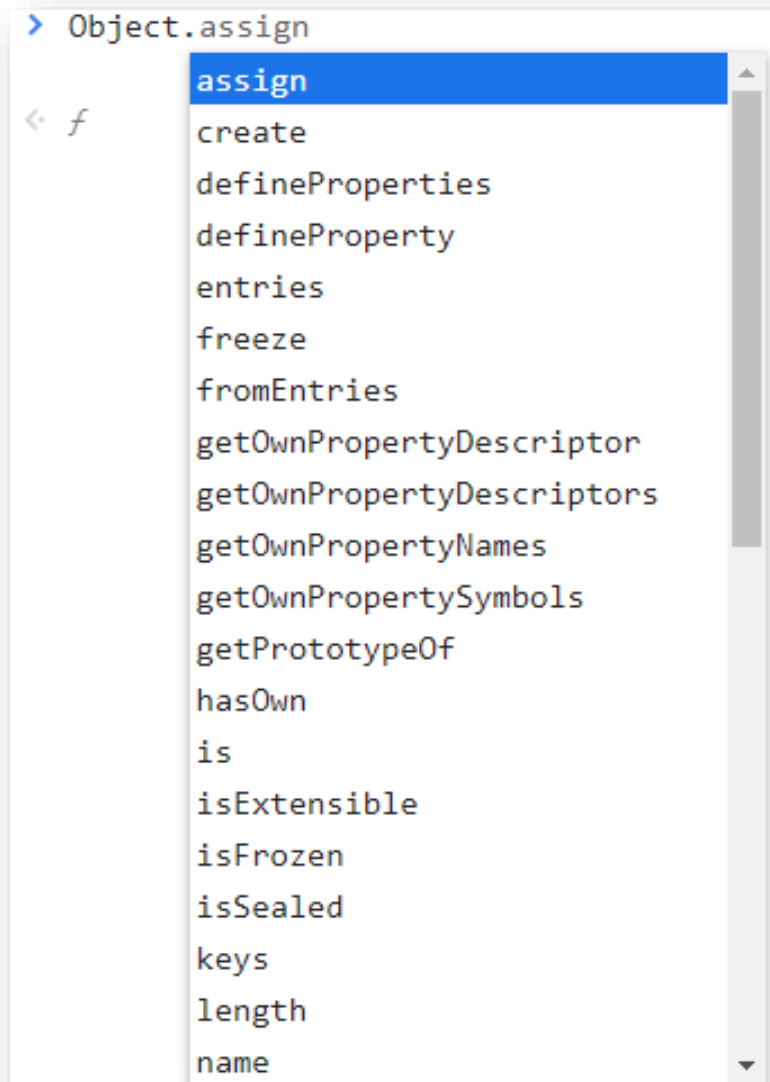
prototype

Como se ha visto, los objetos literales heredan de un objeto llamado *Object*, un objeto estándar incluido en JS.

Sin embargo, si se consulta este objeto se aprecia que contiene un gran número de propiedades y métodos, pero no todos son heredados por los objetos que lo utilizan como prototipo.

Esto se debe a que solo se heredan las propiedades y métodos definidos en la propiedad *prototype*, esto es, solo los que empiezan con *Object.prototype* y no los que empiezan solo con *Object*.

El valor de la propiedad del prototipo es un objeto, que es básicamente un repositorio para almacenar propiedades y métodos que se queremos que sean heredados por los objetos en la cadena del prototipo.



(prototype)

Los objetos almacenan su función o método constructor en una propiedad llamada *constructor*.

Esta propiedad también se almacena dentro de la propiedad *prototype* y, como ya se ha comentado que las propiedades definidas en la propiedad *prototype* del objeto son las que se heredan o están disponibles a todas las instancias de objetos creados, el constructor también estará disponible a los objetos que hereden del objeto.

Aunque no es recomendado, ya que puede fallar debido a herencia de prototipos, binding, preprocesadores, etc. por lo que para ejemplos más complejos es preferible usar el operador *instanceof*, mediante la propiedad *constructor.name* se puede obtener el nombre de la función constructor. También, como se puede acceder al *constructor* podemos acceder a su propiedad *prototype*.

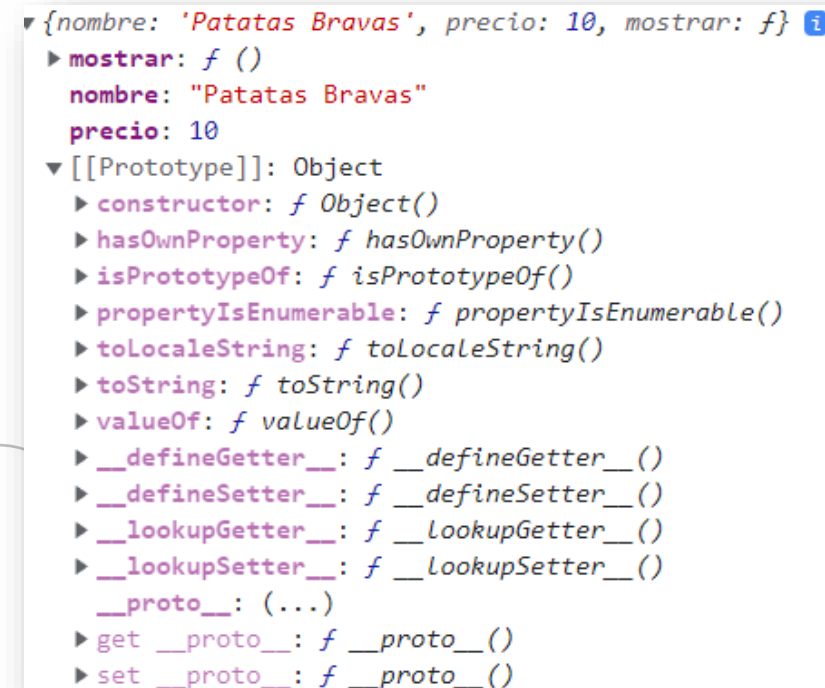
```
> function Producto(nombre, precio){
  this.nombre=nombre;
  this.precio=precio;
  this.mostrar=function(){
    return `${this.nombre} - ${this.precio} euros`;
  }
}
const producto1= new Producto("Patatas Fritas", 10);
< undefined
> producto1
< ▼ Producto {nombre: 'Patatas Fritas', precio: 10, mostrar: f} ⓘ
  ▶ mostrar: f ()
  nombre: "Patatas Fritas"
  precio: 10
  ▼ [[Prototype]]: Object
    ▶ constructor: f Producto(nombre, precio)
    ▶ [[Prototype]]: Object
```

prototype

Cuando se crea un objeto literal su *prototype* coincide con *Object.prototype*.

Ejemplo:

```
const producto1={
  nombre:"Patatas Bravas",
  precio:10,
  mostrar:function(){
    return `${this.nombre} - ${this.precio} euros`;
  }
}
console.log(producto1);
```



```
{nombre: 'Patatas Bravas', precio: 10, mostrar: f} i
  ▶ mostrar: f ()
    nombre: "Patatas Bravas"
    precio: 10
  ▼ [[Prototype]]: Object
    ▶ constructor: f Object()
    ▶ hasOwnProperty: f hasOwnProperty()
    ▶ isPrototypeOf: f isPrototypeOf()
    ▶ propertyIsEnumerable: f propertyIsEnumerable()
    ▶ toLocaleString: f toLocaleString()
    ▶ toString: f toString()
    ▶ valueOf: f valueOf()
    ▶ __defineGetter__: f __defineGetter__()
    ▶ __defineSetter__: f __defineSetter__()
    ▶ __lookupGetter__: f __lookupGetter__()
    ▶ __lookupSetter__: f __lookupSetter__()
    __proto__: (...)
    ▶ get __proto__: f __proto__()
    ▶ set __proto__: f __proto__()
```

prototype

Cuando se crea un nuevo objeto se puede seleccionar cuál será su prototipo, por lo que dos objetos poseerán los mismos métodos y propiedades ya que su cadena de prototipos es la misma.

Para ello, se puede asignar el constructor de un objeto existente a la variable `__proto__` y a continuación invocarlo:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}  
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= new Object();  
producto2.__proto__=producto1.__proto__;  
producto2.constructor("Patatas",2);  
console.log(producto2.mostrar());
```

'Patatas - 2 euros'

prototype

De igual manera se puede utilizar el método *create()* de *Object* para crear un nuevo objeto que utiliza como prototipo el mismo prototipo que otro objeto.

Ejemplo:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}  
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= Object.create(producto1.__proto__);  
producto2.constructor("Patatas",2);  
console.log(producto2.mostrar());
```

'Patatas - 2 euros'

prototype

Observa las diferencias entre ejecutar:

```
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= Object.create(producto1);  
console.Log(producto2.mostrar());
```

'Patatas - 2 euros'

Y ejecutar:

```
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= Object.create(producto1.__proto__);  
console.Log(producto2.mostrar());
```

Uncaught TypeError: producto2.mostrar is not a function
at <anonymous>:3:23

En el primer caso, el objeto *producto2* es un nuevo objeto, copia *producto1*, en el que ya se ha ejecutado su constructor, mientras que, en el segundo caso, el objeto *producto2* no se ha ejecutado el constructor, por lo que no tiene las propiedades y los métodos que se definen en él hasta que se ejecute *producto2.constructor()*.

{prototype}

Pero también puede crearse un nuevo objeto llamando al constructor de un objeto ya creado, mediante el método *constructor*, y almacenar el valor devuelto.

Ejemplo:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}  
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= new producto1.constructor("Patatas",2);  
console.log(producto2.mostrar());
```

'Patatas - 2 euros'

prototype

Por último, se puede crear un objeto que descienda de otro objeto distinto del objeto *Object*, evidentemente al final de la cadena de prototipos estará el objeto *Object*.

Para ello, se asocia la variable `__proto__` al objeto existente distinto de *Object*.

Ejemplo:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
}
```

```
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= new Object;  
producto2.__proto__=producto1;
```

```
producto2  
▼ Producto {} ⓘ  
  ► [[Prototype]]: Producto
```

[modificando prototipos]

Se puede modificar el contenido de la variable el *prototype* de un objeto y todos los objetos que descendan de él se actualizarán automáticamente a las nuevas versiones.

Esto se debe (como ya se ha comentado) a que los métodos y propiedades que se heredan no se guardan en los objetos heredados, sino que, a través de la cadena de prototipos, que actúa como enlace se accede a ellos:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}  
const producto1= new Producto("Patatas Fritas", 10);  
Producto.prototype.cantidad=10;  
console.log("Cantidad:", producto1.cantidad);
```

Cantidad: 10

[modificando prototipos]

Esto es igualmente válido para los métodos.

Ejemplo:

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
}  
const producto1= new Producto("Patatas Fritas", 10);  
Producto.prototype.mostrarCantidad=function(){  
  this.cantidad=10;  
  console.log("Cantidad:", this.cantidad);  
};  
producto1.mostrarCantidad();
```

Cantidad: 10

[herencia prototipos]

De igual manera, un objeto puede sobrescribir, modificando o especializando, un método heredado a través del *prototype*.

Para ello basta con definirlo en el objeto y, debido al funcionamiento de la cadena de prototipos, el método heredado no llegará a ejecutarse salvo que se invoque expresamente mediante la variable *prototype*.

```
function Producto(nombre, precio){  
  this.nombre=nombre;  
  this.precio=precio;  
  this.mostrar=function(){  
    return `${this.nombre} - ${this.precio} euros`;}  
  this.mostrarCantidad=function(){  
    this.cantidad=10;  
    console.log("Cantidad:", this.cantidad);}};
```

```
const producto1= new Producto("Patatas Fritas", 10);  
const producto2= new Object;  
producto2.__proto__=producto1;  
producto2.mostrarCantidad=function(){alert("Hola")};  
producto2.mostrarCantidad();  
producto2.__proto__.mostrarCantidad();
```

herencia prototipos

Otra forma que tiene un objeto de heredar las propiedades y métodos de otro objeto es a través del método *call()*, presente en el *prototype*.

El método *call* sirve para invocar una función, pero añade la funcionalidad que se pasa como primer argumento una referencia a un objeto.

call tiene como primer argumento *this* y como otros argumentos opcionales los parámetros que espera recibir la función que se manda llamar.

Permite que una función o método que pertenece a un objeto sea llamado desde un objeto diferente como si le perteneciera, ya que le aporta su referencia a través del primer argumento. Si, por ejemplo, el método que se llama es un constructor de otro objeto se consigue heredar todas las propiedades y métodos del objeto al que pertenece el constructor.

Con *call* se puede escribir un método una vez y heredarlo en otro objeto, sin tener que reescribir el método en el nuevo objeto.

herencia prototipos

Se puede observar en el siguiente ejemplo:

Se crea una función constructora llamada *Producto*, que permite crear objetos de este tipo.

A continuación, se especializa mediante otra función constructora llamada *Comida*, en la que se invoca mediante el método *call* el constructor *Producto*, heredando de *Producto* sus propiedades y métodos, sin tener que volver a definirlos en *Comida*.

```
✖ ▶ Uncaught Error: No se puede crear el producto VM2612:5
    Patatas Fritas con un precio negativo
    at Comida.Producto (<anonymous>:5:11)
    at new Comida (<anonymous>:11:13)
    at <anonymous>:15:19
```

```
function Producto(nombre, precio) {
  this.nombre=nombre;
  this.precio=precio;
  if (precio < 0)
    throw Error(`No se puede crear el producto
    ${nombre} con un precio negativo`);
  return this;
}
function Comida(nombre, precio) {
  Producto.call(this, nombre, precio);
  this.categoria = 'comida';
}
const producto2=new Comida("Patatas Fritas", -10);
```


Bibliografía y recursos online

- <https://caniuse.com/>
- <https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Inheritance>
- <https://www.arkaitzgarro.com/javascript/capitulo-9.html>
- https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Object_prototypes
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Function/call