

11. OBJETOS (II)

Trabajar con objetos

JavaScript está diseñado en un paradigma simple basado en objetos.

- un **objeto** es una colección de propiedades,
- y una **propiedad** es una asociación entre un **nombre** (o **clave**) y un **valor**.
- el valor de una propiedad puede ser una **función**, en cuyo caso la propiedad se conoce como **método**.

Los objetos en JavaScript, al igual que en muchos otros lenguajes de programación, se pueden comparar con objetos de la vida real.

- en JavaScript, un objeto es una entidad independiente, con propiedades y **tipo**.
- comparémoslo con una taza, por ejemplo.
 - una taza es un objeto, con propiedades.
 - una taza tiene un color, un diseño, un peso, un material del que está hecha, etc.
 - de la misma manera, los objetos de JavaScript pueden tener propiedades, que definen sus características.

Además de los objetos que están predefinidos por el propio navegador, puedes definir tus propios objetos.

En JavaScript, las variables objeto son de tipo **referencia** :

- una referencia es un tipo de dato cuya única función que nos permite acceder a las propiedades y métodos de un objeto dado.
- cada referencia es o bien **null** (o **undefined**) o apunta a un objeto dado en la memoria.

Resumen de objetos

Creación de nuevos objetos

Puedes **crear un objeto**

- utilizando un **inicializador de objetos**.
 - los objetos creados con inicializadores de objetos se denominan **objetos simples**, porque son instancias de **Object**, pero no de ningún otro tipo de objeto.
- como alternativa, primero puede crear una **función constructora** y luego crear una **instancia** de un objeto invocando esa función con el operador **new**.
- algunos tipos de objetos tienen sintaxis especiales de inicialización: por ejemplo
 - **inicializadores de array**
 - **literales de expresiones regulares**.
- las propiedades de los objetos son básicamente lo mismo que las variables, excepto que cada propiedad está asociada con su objeto, no con **alcances/ámbitos** (**scopes**).
- las propiedades de un objeto definen las características del objeto.

Uso de inicializadores de objetos

Un **inicializador de objeto**

- también se denomina **literal objeto**.
- su sintaxis para un objeto que usa un inicializador de objetos es:

```
const objeto = {  
  propiedad1 : valor1,    // el nombre de una propiedad puede ser un identificador ...  
  2          : valor2,    // ... o puede ser un número ...  
  "propiedad n": valor3,  // ... o puede ser un string ... (también puede ser un Symbol)  
};
```

- cada **nombre de propiedad** se escribe antes de dos puntos
 - es un **identificador** (ya sea un nombre, un número o una cadena literal),
 - **el nombre de la propiedad también puede ser una expresión: las claves calculadas deben envolverse entre corchetes.**
- cada **valorN** es una **expresión** cuyo valor se asigna al nombre de la propiedad.
- en este ejemplo, el objeto recién creado se asigna a una variable **objeto**
 - esto es opcional: si no necesitas hacer referencia a este objeto en otro lugar, no necesitas asignarlo a una variable.
 - ten en cuenta que es posible que debas envolver el literal objeto entre paréntesis si el objeto aparece donde se espera una declaración, para que no se confunda el literal objeto con una **declaración de bloque**.
- al igual que las variables de JavaScript, los nombres de propiedad distinguen entre mayúsculas y minúsculas.
- los nombres de propiedades solo pueden ser **cadenas** o **Symbol**:
 - todas las claves se convierten en cadenas a menos que sean **Symbol**.
 - los **índices de array** son, de hecho, propiedades con claves string que contienen números enteros.

Los inicializadores de objetos

- son expresiones,
- cada inicializador de objeto da como resultado la creación de un nuevo objeto cada vez que se ejecuta la instrucción en la que aparece.
- los inicializadores idénticos de objetos crean objetos distintos que no se comparan entre sí como iguales.
- la siguiente declaración crea un objeto y lo asigna a la variable `x` si y solo si la expresión `condicion` es verdadera:

```
let x;  
if (condicion) {  
  x = { saludo: "hola, ¿qué tal?" };  
}
```

- el siguiente ejemplo crea `miHonda` con tres propiedades.
- fíjate que la propiedad del `motor` también es un objeto con sus propias propiedades.

```
const miHonda = {  
  color: "rojo",  
  ruedas: 4,  
  motor: { cilindrados: 4, volumen: 2.2 },  
};
```

Uso de una función constructora

Alternativamente, puede crear un objeto con estos dos pasos:

1. Define el **tipo de objeto** escribiendo una **función constructora**.
 - existe una convención muy establecida, con razón, de usar una letra inicial mayúscula para el identificador del constructor.
2. Crea una **instancia del objeto** con el operador `new`.

Para **definir un tipo de objeto**

- crea una función para el tipo de objeto que especifique su nombre, propiedades y métodos.
- por ejemplo, supón que deseas crear un tipo de objeto para automóviles.
 - quieres que este tipo de objeto se llame `Coche` y que tenga propiedades para marca, modelo y año.
 - para hacer esto, escribirías la siguiente función
 - observa el uso de `this` para asignar valores a las propiedades del objeto en función de los valores pasados a la función.
- también se puede usar la **sintaxis de clase** (como veremos más adelante) en lugar de la **sintaxis de función** para definir una función constructora.

```
function Coche(marca, modelo, anno) {  
  this.marca = marca;  
  this.modelo = modelo;  
  this.anno = anno;  
}
```

Puedes **crear una instancia de objeto** (a veces llamado **ejemplar**) llamada `miCoche` de la siguiente manera:

```
const miCoche = new Coche("Seat", "Panda", 1993);
```

- esta declaración `miCoche` y le asigna los valores especificados para sus propiedades.
- el orden de los **argumentos y parámetros** debe ser el mismo.
- puede crear cualquier número de objetos `miCoche` llamando a `new` : por ejemplo,

```
const cocheJuan = new Coche("Nissan", "300ZX", 1992);  
const cocheMonica = new Coche("Mazda", "Miata", 1990);
```

Un objeto puede tener una propiedad que es en sí mismo otro objeto.

- por ejemplo, supón que defines un objeto llamado `Persona` de la siguiente manera:

```
function Persona(nombre, edad, sexo) {  
  this.nombre = nombre;  
  this.edad = edad;  
  this.sexo = sexo;  
}
```

- y luego creas una instancia de dos nuevos objetos `Persona` de la siguiente manera:

```
const juan = new Persona("Juan López Alcántara", 33, "M");
const beni = new Persona("Benito Malvido Torr  z", 39, "M");
```

- luego, puedes reescribir la definici  n de **Coche** para incluir una propiedad de propietario que tome un objeto **Persona**, de la siguiente manera:

```
function Coche (marca, modelo, anno, propietario) {
  this.marca = marca;
  this.modelo = modelo;
  this.anno = anno;
  this.propietario = propietario;
}
```

- para **instanciar objetos** nuevos, luego usas lo siguiente:

```
const coche1 = new Coche("Eagle", "Talon TSi", 1993, juan );
const auto2 = new Coche("Nissan", "300ZX", 1992, beni );
```

- observa que en lugar de pasar una literal string o un valor entero al crear los nuevos objetos, las declaraciones anteriores pasan los objetos **juan** y **beni** como argumentos para los propietarios.
- luego, si deseas averiguar el nombre del propietario de **auto2**, puedes acceder a la siguiente propiedad:

```
auto2.propietario.nombre;
```

Por otro lado, siempre puedes **agregar una propiedad** a un objeto previamente definido.

- por ejemplo, la declaraci  n

```
coche1.color = "negro";
```

- agrega una propiedad **color** a **coche1** y le asigna un valor de "negro".
- sin embargo, **esto no afecta a ning  n otro objeto**.
- para agregar la nueva propiedad a todos los objetos del mismo tipo, debes agregar la propiedad a la **definici  n del tipo de objeto Coche**.

Uso del m  todo **Object.create()**

Los objetos tambi  n se pueden crear utilizando el m  todo **Object.create()**:

- este m  todo puede ser muy   til, porque te permite elegir el **objeto prototipo** para el objeto que deseas crear, sin tener que definir una **funci  n constructora**.

```
// Propiedades de los animales y encapsulaci  n de m  todos
const Animal = {
  tipo: "Invertebrados", // Default value of properties
  mostrarTipo() { // Method which will display type of Animal
    console.log(this.tipo);
  },
};

// Crear un nuevo tipo de animal llamado animal1
const animal1 = Object.create(Animal);
animal1.mostrarTipo (); // Saca por consola: Invertebrados

// Crear un nuevo tipo de animal llamado pez
const pez = Object.create(Animal);
pez.tipo = "Peces";
pez.mostrarTipo (); // Saca por consola: peces
```

Acceso a las propiedades

Puedes acceder a una propiedad de un objeto por su nombre de propiedad.

- los **accesores de propiedad** (**property accessors**) se pueden utilizar usando 2 sintaxis distintas:
 - **notaci  n de punto**
 - **notaci  n de corchete**.
- por ejemplo, puedes acceder a las propiedades del objeto **miCoche** de la manera siguiente:

```
// Notación de puntos
miCoche.marca = "Ford";
miCoche.modelo = "Mustang";
miCoche.anno = 1969;

// Notación de corchetes
myCar ["marca"] = "Ford";
myCar ["modelo"] = "Mustang";
myCar ["anno"] = 1969;
```

Un **nombre de propiedad** puede ser cualquier string/cadena o un **Symbol**:

- incluyendo una **cadena vacía** "" o ''
 - sin embargo, no puedes usar la notación de puntos para acceder a una propiedad cuyo nombre no es un identificador de JavaScript válido.
 - por ejemplo, un nombre de propiedad que
 - tiene un espacio o un guion
 - comienza con un número
 - se mantiene dentro de una variable
 solo se puede acceder usando la **notación de corchetes**.
 - esta notación también es muy útil cuando los nombres de las propiedades se van a determinar **dinámicamente**, es decir, no se pueden determinar hasta el **tiempo de ejecución**.

Algunos ejemplos de lo anterior:

```
const miObjeto = {};
const cadena = "miCadena";
const aleatorio = Math.random();
const otroObjeto = {};

// Crear propiedades adicionales en miObjeto
miObjeto.tipo = "Sintaxis de puntos para un tipo con clave con nombre";

miObjeto["fecha de creación"] = "Esta clave tiene un espacio";

miObjeto[cadena] = "Esta clave está en la variable cadena";

miObjeto[aleatorio] = "Un número aleatorio es la clave aquí";

miObjeto[otroObjeto] = "Esta clave es objeto otroObjeto";

miObjeto[""] = "Esta clave es una cadena vacía";

console.log(miObjeto);
// {
//   tipo: 'Sintaxis de puntos para un tipo con clave con nombre',
//   'fecha de creación': 'Esta clave tiene un espacio',
//   cadena: 'Esta clave está en la variable cadena',
//   '0.6398914448618778': 'Un número aleatorio es la clave aquí',
//   '[objeto Objeto]': 'Esta clave es objeto otroObjeto',
//   '': 'Esta clave es una cadena vacía'
// }

console.log(miObjeto.miCadena); // se creó en miObjeto[cadena] es decir, miObjeto.miCadena
console.log(miObjeto.cadena); // undefined : no se puede utilizar la notación .
```

- en el código anterior, la clave **otroObjeto** es un objeto, que no es ni una cadena ni un **Symbol**.
 - cuando se agrega a **miObjeto**, JavaScript llama al método **toString()** de **otroObjeto** y usa la cadena resultante como la nueva clave.
- también puede acceder a las propiedades con un valor de cadena/string almacenado en una variable.
 - la variable debe pasarse en notación de corchetes.
 - en el ejemplo anterior, la variable **cadena** contenía "miCadena" y "miCadena" es el nombre de la propiedad.
 - por lo tanto, **miObjeto.cadena** regresará como indefinido.

Poder usar una variable string como propiedad permite acceder a cualquier propiedad determinada **en tiempo de ejecución**:

```
let nombrePropiedad = "marca";
miCoche[nombrePropiedad] = "Ford";

// acceder a diferentes propiedades cambiando el contenido de la variable
nombrePropiedad = "modelo";
miCoche[nombrePropiedad] = "Mustang";
```

```
console.log(miCoche); // { marca: 'Ford', modelo: 'Mustang' }
```

Sin embargo:

- ten cuidado cuando uses corchetes para acceder a propiedades cuyos nombres son dados por una entrada externa (por ejemplo, desde un formulario).
- esto puede hacer que tu código sea susceptible a **ataques de inyección de objetos**.

Las propiedades inexistentes de un objeto tienen un valor **undefined** (en vez de **null**).

```
miCoche.propiedadNoExistente; // = undefined
```

Enumerar propiedades

Hay tres formas nativas de enumerar/recorrer las propiedades de los objetos:

- bucles **for...in**
 - este método atraviesa todas las **propiedades enumerables** y que sean de tipo string de un objeto, así como las de su **cadena de prototipo (prototype chain)**.
- **Object.keys(miObjeto)**: este método devuelve una array con solo las propiedades de **miObjeto** que cumplan:
 - son **propiedades propias (own properties)**: las propiedades que provienen de la cadena de prototipos no son propias
 - son propiedades string (no **Symbol**)
 - son **propiedades enumerables** (cada propiedad tiene asociados una serie de metadatos indicando si es enumerable o no, si es de solo lectura,)
- **Object.getOwnPropertyNames(miObjeto)**: este método devuelve una array que contiene todos los nombres de propiedad de **miObjeto** que cumplan, sean enumerables o **no-enumerables**:
 - son **propiedades propias**
 - son propiedades string (no **Symbol**)

Puedes usar la **notación de cochetes** junto con **for...in** para iterar sobre todas las **propiedades enumerables** de un objeto.

- para ilustrar cómo funciona esto, la siguiente función muestra las propiedades del objeto cuando pasa el objeto y el nombre del objeto como argumentos a la función:

```
function mostrarPropiedades(objeto, nombreObjeto) {
  let resultado = "";
  for (const i in objeto) {
    if (Object.hasOwn(objeto, i)) { // hasOwn() permite excluir las
      resultado += `${nombreObjeto}.${i} = ${objeto[i]}\n`; // prpiedades que proceden de la
    } // cadena de prototipos del objeto
  }
  console.log(resultado);
}
```

- el término "propiedad propia" se refiere a las propiedades del objeto, pero excluyendo las de la cadena prototipo.
- entonces, la llamada a la función **mostrarPropiedades(miCoche, 'miCoche')** imprimiría lo siguiente:

```
miCoche.marca = Ford
miCoche.modelo = Mustang
miCoche.anno = 1969
```

Lo anterior es equivalente a:

```
function mostrarPropiedades(objeto, nombreObjeto) {
  let resultado = "";
  Object.keys(objeto).forEach( (i) => {
    resultado += `${nombreObjeto}.${i} = ${objeto[i]}\n`;
  });
  console.log(resultado);
}
```

No hay una forma nativa de enumerar las propiedades heredadas (a través de la cadena de prototipos) no-enumerables

- sin embargo, esto se puede lograr con la siguiente función:

```
function listarTodasLasPropiedades(miObjeto) {
  let objetoAInspeccionar = miObjeto;
  let resultado = [];

  while ( objetoAInspeccionar !== null) {
    resultado = resultado.concat(Object.getOwnPropertyNames(objetoAInspeccionar));
  }
}
```

```

    objetoAInspeccionar = Object.getPrototypeOf(objetoAInspeccionar);
}

return resultado;
}

```

Eliminación de propiedades

Puede eliminar una propiedad no heredada mediante el **operador de eliminación delete**

- el siguiente código muestra cómo eliminar una propiedad.

```

// Creates a new object, myobj, with two properties, a and b.
const miObjeto = new Object();
miObjeto.a = 5;
miObjeto.b = 12;

delete miObjeto.a;           // Elimina la propiedad a, quedando solo la propiedad b
console.log("a" in miObjeto); // false

```

Herencia

Todos los objetos en JavaScript heredan de al menos otro objeto.

- el objeto del que se hereda se conoce como **prototipo**,
- las **propiedades heredadas** se pueden encontrar en el objeto **prototype** del **constructor**.

Definición de propiedades para todos los objetos de un tipo

Puedes agregar una propiedad a todos los objetos creados a través de un determinado constructor utilizando la propiedad **prototype**.

- esto define una propiedad que comparten todos los objetos del tipo especificado, en lugar de solo una instancia del objeto.
- el siguiente código agrega una propiedad de **color** a todos los objetos de tipo **Coche** y luego lee el valor de la propiedad de una instancia **coche1**.

```

Coche.prototype.color = "rojo";
console.log(coche1.color); // "rojo"

```

Nota:

La anterior modificación sólo afectará a los objetos creados a partir de la última versión del constructor de Coche:

- en los ejemplos anteriores,

```
Coche.prototype.color = "rojo";
```

solo afectará a **coche1** y **auto2**, pero no a **cocheJuan** y **cocheMonica**.

```

// versión 1
function Coche(marca, modelo, anno) {
    this.marca = marca;
    this.modelo = modelo;
    this.anno = anno;
}

const cocheJuan = new Coche("Nissan", "300ZX", 1992);
const cocheMonica = new Coche("Mazda", "Miata", 1990);

// versión 2
function Coche(marca, modelo, anno, propietario) {
    this.marca = marca;
    this.modelo = modelo;
    this.anno = anno;
    this.propietario = propietario;
}

const coche1 = new Coche("Eagle", "Talon TSi", 1993, juan);
const auto2 = new Coche("Nissan", "300ZX", 1992, beni);

```

Definición de métodos

Un **método**

- es una función asociada a un objeto,
- o, dicho de otro modo, un método es una propiedad de un objeto que es una función.

FORMA 1:

Los métodos se definen de la forma en que se definen las funciones normales, excepto que deben asignarse como propiedad de un objeto; un ejemplo es:

```
nombreObjeto.nombreMetodo = nombreFuncion;
```

- donde *nombreObjeto* es un objeto existente, *nombreMetodo* es el nombre que está asignando al nuevo método del objeto y *nombreFuncion* es el nombre de la función.
- luego puede llamar al método en el contexto del objeto de la siguiente manera:

```
nombreObjeto.nombreMetodo(parámetros);
```

FORMA 2:

Otro ejemplo:

```
const miObjeto = {  
  miMetodo: function (parámetros) {  
    // hacer algo  
  },  
  
  // Ejemplo de sintaxis abreviada: no hace falta usar : function  
  miOtroMetodo(parámetros) {  
    // hacer algo más  
  },  
};
```

FORMA 3:

Los métodos normalmente se definen en el objeto **prototype** del **constructor**, de modo que todos los objetos del mismo tipo comparten el mismo método.

- por ejemplo, puedes definir una función que formatee y muestre las propiedades de los objetos **Coche** previamente definidos.

```
Car.prototype.mostrarCoche = function () {  
  const resultado = `Un precioso coche de ${this.anno} ${this.marca} ${this.modelo}`;  
  console.log(resultado);  
};
```

- observa el uso de **this** para referirse al objeto al que pertenece el método.
- luego puedes llamar o invocar al método **mostrarCoche** para cada uno de los objetos de la siguiente manera:

```
coche1.mostrarCoche();  
coche2.mostrarCoche();
```

Uso de **this** para referencias de objetos

JavaScript tiene una palabra clave especial, **this**, que puede usar dentro de un método para referirse al objeto actual.

- por ejemplo, supón que tienes 2 objetos, **Gerente** e **Becario**.
- cada objeto tiene su propio nombre, edad y trabajo.
- en la función **decirHola()**, observa que hay **this.nombre**: gracias a esto, al crearse a los 2 objetos, se les puede llamar e imprimir el mensaje con su respectivo nombre.

```
const Gerente = {  
  nombre: "Karina",  
  edad: 27,  
  trabajo: "Desarrolladora de software",  
};
```

```
const Becario = {  
  nombre: "Lucía",  
  edad: 21,
```

```

    trabajo: "Becaria desarrolladora de software",
  };

function decirHola() {
  console.log(`Hola, mi nombre es ${this.nombre}`);
}

// añadir la función decirHola a ambos objetos
Gerente.decirHola = decirHola;
Bacario.decirHola = decirHola;

Gerente.decirHola(); // Hola, mi nombre es Karina
Bacario.decirHola(); // Hola, mi nombre es Lucía

```

- **this** es un " **parámetro oculto** " de una **llamada de función** que se pasa especificando el objeto antes de la función que se llamó.
- por ejemplo, en `Gerente.decirHola()`, este es el objeto `Gerente`, porque `Gerente` viene antes que la función `decirHola()`.
- si accedes a la misma función desde otro objeto, **this** también cambiará.
- si usas otros métodos para llamar a la función, como `Function.prototype.call()` or `Reflect.apply()`, puedes pasar explícitamente el valor de **this** como un argumento.

Definición de getters y setters

Tenemos que:

- un **getter** es una función asociada con una propiedad "virtual" de un objeto y dicha función se encarga de hacer de intermediaria de forma transparente para asignar un valor a dicha propiedad específica.
- un **setter** es una función asociada con una propiedad "virtual" de un objeto y dicha función se encarga de hacer de intermediaria de forma transparente para obtener el valor de dicha propiedad específica.
- juntos, pueden representar indirectamente el valor de una propiedad.

Getters y setters pueden ser

- definido dentro de un **inicializador de objeto**, o
- añadido más tarde a cualquier objeto existente.

Dentro de los inicializadores de objetos,

- getters y setters se definen como métodos regulares, pero con el prefijo
 - la **palabra clave get**
 - la **palabra clave set**
- el **método getter** no debe esperar un **parámetro**
- el **método setter** espera exactamente un **parámetro** (el nuevo valor a establecer).
- por ejemplo:

```

const miObj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};

console.log(miObj.a); // 7
console.log(miObj.b); // 8, valor devuelto por el método get b(x)
miObj.c = 50; // Invoca o llama al método set c(x)
console.log(miObj.a); // 25

```

- las propiedades del objeto `miObj` son:
 - `myObj.a` : un número
 - `myObj.b` : un **getter** que devuelve `miObj.a` mas 1
 - `miObj.c` : un **setter** que establece el valor de `miObj.a` a la mitad del valor en el que se establece `miObj.c`

Los getters y setters también se pueden agregar a un objeto en cualquier momento después de la creación mediante el método `Object.defineProperty()`

- el primer parámetro de este método es el objeto en el que desea definir el **getter** o **setter**.
- el segundo parámetro es un objeto cuyos nombres de propiedad son los nombres de getter o setter, y cuyos valores de

propiedad son objetos para definir las funciones de getter o setter.

- aquí hay un ejemplo que define el mismo getter y setter usado en el ejemplo anterior:

```
Object.defineProperty(miObj, {
  b: {
    get() { return this.a + 1; },
  },
  c: {
    set(x) { this.a = x / 2; },
  },
});

miObj.c = 10;           // Ejecuta el setter, el cual asigna 10 / 2 (5) a la propiedad a
console.log(miObj.b);   // Ejecuta el getter, el cual produce a + 1 or 6
```

Cuál de las dos formas elegir depende de tu estilo de programación y de la tarea que tengas entre manos:

- si puede trabajar directamente con la definición del objeto original, probablemente definirás getters y setters a través del inicializador original: esta forma es más compacta y natural.
- sin embargo, si necesitas agregar getters y setters más tarde (tal vez porque no escribiste el objeto en particular)
 - entonces la segunda forma es la única forma posible.
 - la segunda forma representa mejor la naturaleza dinámica de JavaScript, pero puede dificultar la lectura y comprensión del código.

Comparación de objetos

En JavaScript, los objetos son un **tipo de referencia**.

- dos objetos nunca son iguales, aunque tengan las mismas propiedades.
- solo comparar dos referencias distintas al mismo objeto da como resultado verdadero.

```
// Dos variables, dos objetos distintos con las mismas propiedades
const fruta = { nombre: "manzana" };
const fruta2 = { nombre: "manzana" };

fruta == fruta2;    // devuelve false
fruta === fruta2;   // devuelve false

// Dos variable que tienen una referencia al mismo objeto
const fruta = { nombre: "manzana" };
const fruta2 = fruta;    // Asignamos a fruta2 la misma referencia que fruta

fruta == fruta2;      // Aquí fruta y fruta2 apuntan al mismo objeto: devuelve true
fruta === fruta2;     // Por lo mismo: devuelve true

fruta.nombre = "uva";
console.log(fruta2); // { nombre: "uva" }; y no { name: "apple" }
```

Prototipos de objetos

Los **prototipos** de objetos

- son el mecanismo por el cual los objetos de JavaScript **heredan características** unos de otros.
- vamos a profundizar en
 - qué es un prototipo,
 - cómo funcionan las **cadenas de prototipos**
 - cómo se puede configurar un prototipo para un objeto.

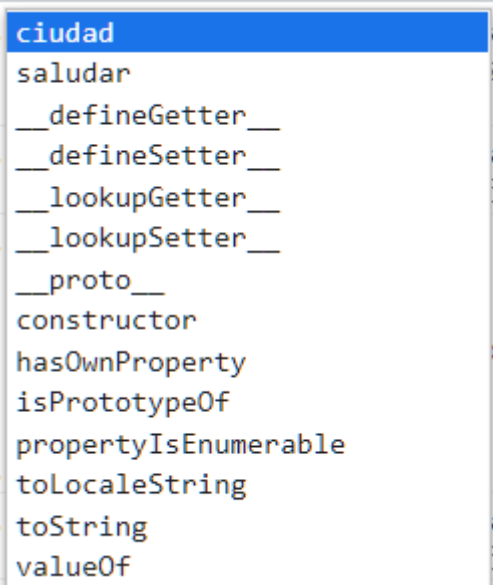
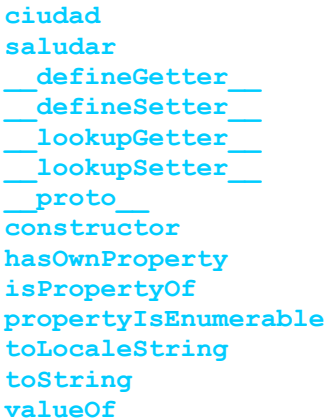
La cadena de prototipos

En la consola del navegador, intenta crear un **literal objeto**:

```
const miObjeto = {  
  ciudad: "Madrid",  
  saludar() {  
    console.log(`Saludos desde ${this.ciudad}`);  
  },  
};
```

```
miObjeto.saludar(); // Saludos desde Madrid
```

- este es un objeto con una propiedad datos, **ciudad**, y una propiedad método, **saludar()**.
- si escribes el nombre del objeto seguido de un punto en la consola del navegador, luego la consola mostrará una lista de todas las propiedades disponibles para este objeto.
- verás que, además de **ciudad** y **saludar**, hay muchas más propiedades

	
--	---

- intenta acceder a uno de ellos; funciona (incluso si no es obvio lo que hace **toString()**).

```
miObjeto.toString(); // '[object Object]'
```

¿Qué son estas propiedades adicionales y de dónde vienen?

- cada objeto en JavaScript tiene una propiedad integrada (*built-in*), que se llama su **prototipo** (**prototype**)
- el prototipo es en sí mismo un objeto, por lo que el prototipo tendrá su propio prototipo y así sucesivamente: formando así lo que se llama una **cadena de prototipos**.
- la cadena termina cuando llegamos a un prototipo que tiene **null** como su propio prototipo.

Nota:

- la propiedad de un objeto que apunta a su prototipo **no se llama prototype**:
 - su nombre no es estándar, pero en la práctica todos los navegadores usan **[[Prototype]]** (**__proto__**)
- la forma estándar de acceder al prototipo de un objeto es el método **Object.getPrototypeOf()**
- además **cada (objeto) función** tiene una propiedad **prototype**, pero no así los objetos que no son una función

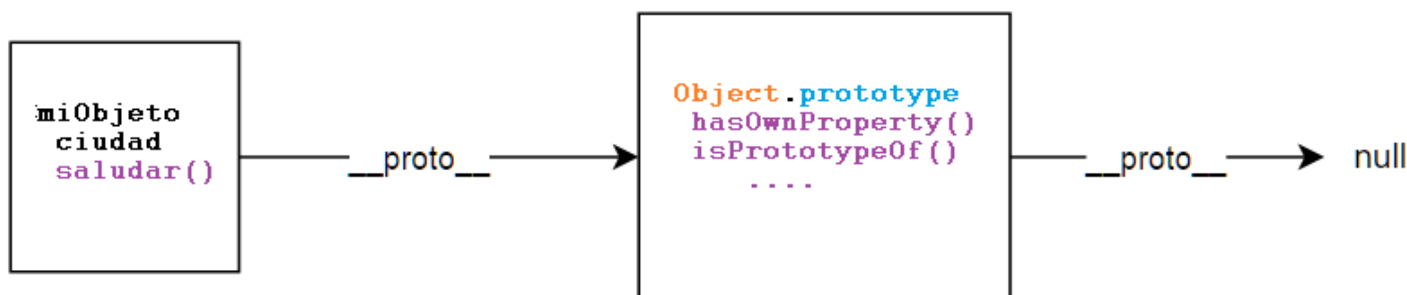
Tenemos que:

- cuando intentas acceder a una propiedad de un objeto:
 - si la propiedad no se puede encontrar en el objeto en sí, se busca la propiedad en su prototipo.
 - si aún así no se puede encontrar la propiedad, se busca el prototipo del prototipo,
 - y así sucesivamente hasta que se encuentre la propiedad o se alcance el final de la cadena de prototipos, `null`, en cuyo caso se devuelve `undefined`.
- entonces cuando llamamos a `miObjeto.toString()`, el navegador:
 - busca `toString` en `miObjeto`
 - no puede encontrarlo allí, así que busca `toString` en el objeto prototipo de `miObjeto`, que es `Object`
 - lo encuentra allí, y lo llama.

¿Cuál es el prototipo de `miObjeto`? Para averiguarlo, podemos usar la función `Object.getPrototypeOf()`:

```
Object.getPrototypeOf(miObjeto); // Objeto { }
```

- este es un objeto llamado `Object.prototype`,
- es el prototipo más básico, que todos los objetos tienen por defecto.
- el prototipo de `Object.prototype` es `null`, por lo que está al final de la cadena de prototipos:

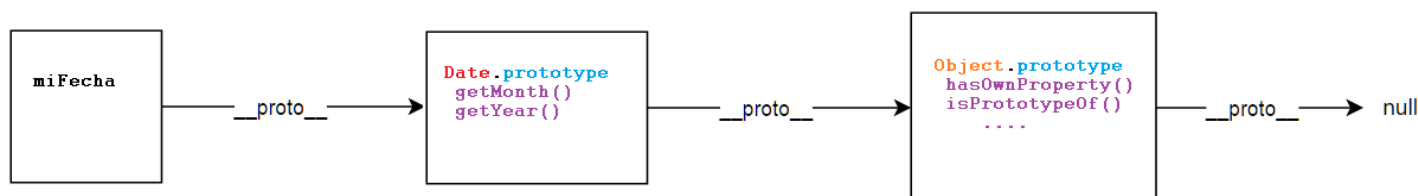


El prototipo de un objeto no siempre es `Object.prototype`. Prueba esto:

```
const miFecha = new Date();  
let objeto = miFecha;  
  
do {  
  objeto = Object.getPrototypeOf( objeto );  
  console.log(objeto);  
} while (objeto);
```

```
// Date.prototype  
// Object { }  
// null
```

- este código crea un objeto `Date`, luego sube por la **cadena de prototipos** y saca por consola el prototipo de cada objeto de la cadena.
- nos muestra que el prototipo de `miFecha` es un objeto `Date.prototype`, (porque `miFecha` y `Date` son funciones, funciones constructoras, por eso tienen la propiedad `prototype`)
- y el prototipo de eso es `Date.prototype`.



- de hecho, cuando llamabas a métodos ya conocidos, como `miFecha2.getMonth()`, estás llamando a un método que está definido en `Date.prototype`.

```
Date.__proto__  
f () { [native code] }
```

Date.prototype

```
▼ {constructor: f, toString: f, toDateString: f, toTimeString: f, toISOString: f, ...} ⓘ  
  ▶ constructor: f Date()  
  ▶ getDate: f getDate()  
  ▶ getDay: f getDay()  
  ▶ getFullYear: f getFullYear()  
  ▶ getHours: f getHours()  
  ▶ getMilliseconds: f getMilliseconds()  
  ▶ getMinutes: f getMinutes()  
  ▶ getMonth: f getMonth()  
  ▶ getSeconds: f getSeconds()  
  ▶ getTime: f getTime()  
  ▶ getTimezoneOffset: f getTimezoneOffset()  
  ▶ getUTCDate: f getUTCDate()  
  
  .....  
  ▶ toUTCString: f toUTCString()  
  ▶ valueOf: f valueOf()  
  ▶ Symbol(Symbol.toPrimitive): f [Symbol]  
  ▼ [[Prototype]]: Object ← Date.prototype.__proto__  
    ▶ constructor: f Object()  
    ▶ hasOwnProperty: f hasOwnProperty()  
    ▶ isPrototypeOf: f isPrototypeOf()  
    ▶ propertyIsEnumerable: f propertyIsE  
    ▶ toLocaleString: f toLocaleString()  
    ▶ toString: f toString()  
    ▶ valueOf: f valueOf()  
    ▶ __defineGetter__: f __defineGetter_  
    ▶ __defineSetter__: f __defineSetter_  
    ▶ __lookupGetter__: f __lookupGetter_  
    ▶ __lookupSetter__: f __lookupSetter_  
    ▼ __proto__: Object  
      ▶ constructor: f Object()  
      ▶ hasOwnProperty: f hasOwnProperty(  
      ▶ isPrototypeOf: f isPrototypeOf()  
      ▶ propertyIsEnumerable: f propertyI  
      ▶ toLocaleString: f toLocaleString(  
      ▶ toString: f toString()  
      ▶ valueOf: f valueOf()  
      ▶ __defineGetter__: f __defineGette  
      ▶ __defineSetter__: f __defineSette  
      ▶ __lookupGetter__: f __lookupGette  
      ▶ __lookupSetter__: f __lookupSette  
      __proto__: null  
      ▶ get __proto__: f __proto__()  
      ▶ set __proto__: f __proto__()  
      ▶ get __proto__: f __proto__()  
      ▶ set __proto__: f __proto__()
```

Date.prototype.__proto__

```
▼ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnPropert  
y: f, __lookupGetter__: f, ...} ⓘ  
  ▶ 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__: null  
  ▶ get __proto__: f __proto__()  
  ▶ set __proto__: f __proto__()
```

Sombreado de propiedades (*shadowing properties*)

¿Qué pasa si defines una propiedad en un objeto, cuando ya está definida una propiedad con el mismo nombre en el **prototipo del objeto**? Probemos:

```
const miFecha = new Date(1995, 11, 17);

console.log(miFecha.getYear()); // 95

miFecha.getYear = function () {
  console.log("¡algo más!");
};

miFecha.getYear(); // '¡algo más!'
```

- esto debería ser predecible, dada la descripción de la cadena prototipo.
- cuando llamamos a `getYear()`, el navegador primero busca en `miFecha` una propiedad con ese nombre, y solo mirará dentro del prototipo si `miFecha` no define dicha función
- entonces, cuando agregamos `getYear()` a `miFecha`, se llama a la versión en `miFecha`.
- esto se llama "sombrear" la propiedad.

Configuración de un prototipo

Hay varias formas de configurar el prototipo de un objeto en JavaScript, y aquí se describen dos:

- `Object.create()`
- constructores

Usando `Object.create`

El método `Object.create`

- crea un nuevo objeto y le permite especificar un objeto que se usará como prototipo del nuevo objeto.
- por ejemplo:

```
const prototipoPersona = {
  saludar() {
    console.log("¡hola!");
  },
};

const carlos = Object.create(prototipoPersona);
carlos.saludar(); // ¡hola!
```

- aquí se crea un objeto `prototipoPersona`, que tiene un método `saludar()`.
- luego se usa `Object.create()` para crear un nuevo objeto con `prototipoPersona` como su prototipo.
- ahora podemos llamar a `saludar()` en el nuevo objeto, y el prototipo proporciona su **implementación**.

Usando un constructor

En JavaScript, todas las funciones tienen una propiedad llamada `prototype`. (no confundir con la propiedad `__proto__` que tiene cada objeto (función o no))

- cuando llamas a una función como constructor (con el operador `new`), esta propiedad se establece como el prototipo del objeto recién construido
- por convención, se añade en la propiedad denominada `__proto__`
- entonces, si configuramos el `prototype` de un constructor, **podemos asegurarnos de que todos los objetos creados con ese constructor reciban ese prototipo**:

```
const prototipoPersona = {
  saludar() {
    console.log(`Hola, mi nombre es ${this.nombre}!`);
  },
};

function Persona(nombre) {
  this.nombre = nombre;
}
```

```
Object.assign(Persona.prototype, prototipoPersona);
// también es válido:
// Persona.prototype.saludar = prototipoPersona.saludar;
```

- aquí creamos:
 - un objeto `prototipoPersona`, que tiene un método `saludar()`
 - una **función constructora** `Persona()` que inicializa el nombre de la persona a crear.
- luego colocamos los métodos definidos en `prototipoPersona` en la propiedad de prototipo de la función `Persona` usando `Object.assign`
- después de este código, los objetos creados con `Persona()` obtendrán `Persona.prototype` como su prototipo, que contiene automáticamente el método de `saludar`.

```
const ruben = new Persona("Rubén");
ruben.saludar(); // ¡Hola, mi nombre es Rubén!
```

- esto también explica por qué se indicó anteriormente que el prototipo de `miFecha` se llama `Date.prototype`: es la propiedad prototipo del constructor `Date`.

Propiedades propias

Los objetos que creamos usando el constructor `Persona` anterior tienen dos propiedades:

- una propiedad de **nombre**, que se establece en el constructor, por lo que aparece directamente en los objetos `Persona`
- un método de `saludar()`, que se establece en el prototipo.
- es común ver este patrón, en el que los métodos se definen en el prototipo, pero las propiedades de los datos se definen en el constructor: eso es porque
 - los métodos suelen ser los mismos para cada objeto que creamos,
 - mientras que a menudo queremos que cada objeto tenga su propio valor para sus propiedades de datos (al igual que aquí, donde cada persona tiene un nombre diferente).

Las propiedades que se definen directamente en el objeto, como el nombre aquí, se denominan **propiedades propias**, y puedes verificar si una propiedad es una propiedad propia utilizando el método static `Object.hasOwnProperty()` :

```
const irma = new Persona("Irma");

console.log( Object.hasOwnProperty(irma, "nombre")); // verdadero
console.log( Object.hasOwnProperty(irma, "saludar")); // falso
```

Nota: también puedes utilizar método no static `Object.hasOwnProperty()` aquí, pero se recomienda usar `Object.hasOwn()` si se puede.

Prototipos y herencia

Los **prototipos**

- son una característica potente y muy flexible de JavaScript, que permite reutilizar código y combinar objetos.
- en particular soportan una versión de herencia.
 - **herencia** es una característica de **los lenguajes de programación orientados a objetos** que permite a los programadores expresar la idea de que algunos objetos en un sistema son **versiones más especializadas** de otros objetos.
 - por ejemplo, si estamos modelando una escuela, podríamos tener **profesores** y **estudiantes** :
 - ambos son personas, por lo que tienen algunas características en común (por ejemplo, ambos tienen **nombres**),
 - pero cada uno podría
 - agregar características adicionales (por ejemplo, los profesores tienen una materia que enseñan),
 - o podría implementar la misma característica de diferentes maneras.
 - en un sistema OOP podríamos decir que tanto los **profesores** como los **estudiantes** heredan de **personas**
 - puedes ver cómo en JavaScript, si los objetos **Profesor** y **Estudiante** pueden tener prototipos de **Persona**, entonces pueden heredar las propiedades comunes, mientras agregan y redefinen aquellas propiedades que deben diferir.

Descriptores de propiedades

Cada propiedad de un objeto tiene asociado un **descriptor de propiedad** que puede ser de uno de los 2 tipos siguientes (o uno u otro, pero no los 2 a la vez):

- **descriptor de datos**: si es una propiedad que tiene un **valor**, que puede o no ser **writable** (mutable)
- **descriptor accesor**: es una propiedad descrita por un par de **funciones getter-setter**

Tanto los **descriptores de datos** como los **descriptores accesor** son objetos y ambos comparten las siguientes claves opcionales (ten en cuenta: los **valores predeterminados** mencionados aquí son en el caso de definir propiedades usando

`Object.defineProperty()`)

- **configurable**
 - cuando esto se establece en **false**
 - el **tipo de la propiedad** no se puede cambiar entre **propiedad de datos** y propiedad accesor, y
 - la propiedad no se puede eliminar, y
 - no se pueden cambiar otros atributos de su descriptor (sin embargo, si es un descriptor de datos con **writable: true**, el valor se puede cambiar y **writable** se puede cambiar a **false**).
 - **por defecto es: false**.
- **enumerable**
 - **true** si y solo si esta propiedad aparece durante la enumeración de las propiedades en el objeto correspondiente.
 - **por defecto es: false**.

Un **descriptor de datos** también tiene las siguientes claves opcionales:

- **value**
 - el valor asociado a la propiedad.
 - puede ser cualquier valor de JavaScript válido (número, objeto, función, etc.).
 - **por defecto es: undefined**.
- **writable**
 - true si el valor asociado con la propiedad se puede cambiar con un operador de asignación.
 - **por defecto es: false**.

Un **descriptor accesor** también tiene las siguientes claves opcionales:

- **get**
 - una función que sirve como **getter** de la propiedad, o **undefined** si no hay getter.
 - cuando se accede a la propiedad, esta función se llama sin argumentos y con **this** igual al objeto a través del cual se accede a la propiedad (éste puede no ser el objeto en el que se define la propiedad debido a la herencia).
 - el valor devuelto se utilizará como el valor de la propiedad.
 - **por defecto es: undefined**
- **set**
 - una función que sirve como **setter** para la propiedad, o **undefined** si no hay setter.
 - cuando se asigna un valor a la propiedad, se llama a esta función con un argumento (el valor que se asigna a la propiedad) y con **this** igual al objeto a través del cual se asigna la propiedad.
 - **por defecto es: undefined**.

Si un **descriptor**

- no tiene ninguna de las claves **value**, **writable**, **get** y **set**, se trata como un **descriptor de datos**.
- tiene las claves tanto [**value** o **writable**] como [**get** o **set**], se lanza una excepción.

Tenga en cuenta que estos **atributos** no son necesariamente propiedades propias del descriptor.

- las propiedades heredadas también serán consideradas.
- para garantizar que se conserven estos valores predeterminados, puedes:
 - congelar (`Object.freeze()`) los objetos existentes en la cadena de prototipos del objeto descriptor por adelantado,
 - especificar todas las opciones explícitamente
 - o apuntar a **null** con `Object.create(null)`.

Métodos que permiten trabajar con descriptores de propiedades:

- `Object.defineProperty()`
- `Object.defineProperties()`
- `Object.getOwnPropertyDescriptor()`
- `Object.getOwnPropertyDescriptors()`
- `Object.freeze()`
- `Object.isFrozen()`

- `Object.seal()`
- `Object.isSealed()`
- `Object.isExtensible()`

Enumerabilidad y posesión de las propiedades

Cada propiedad en los objetos de JavaScript se puede clasificar por tres factores:

- **enumerables** o **no enumerables**;
- String o **Symbol**;
- **Propiedad propia** o **propiedad heredada** de la cadena de prototipos.

Las **propiedades enumerables**

- son aquellas propiedades cuyo indicador enumerable interno se establece en verdadero, que es el valor predeterminado para las propiedades creadas mediante una **asignación simple** o mediante un **inicializador de propiedad**.
- las propiedades definidas a través de `Object.defineProperty` y otras no se pueden enumerar de forma predeterminada.
- la mayoría de los medios de iteración (como los bucles `for...in` loops y `Object.keys`) solo visitan claves enumerables.

La posesión de las propiedades

- está determinada por si la propiedad pertenece directamente al objeto y no a su cadena de prototipos.
- se puede acceder a todas las propiedades, enumerables o no, cadenas o símbolos, propias o heredadas, con notación de puntos o corchetes.

Vamos a realizar los métodos de JavaScript que visitan un grupo de propiedades de objetos una por una.

Consulta de propiedades de objetos

Hay cuatro formas integradas de consultar una propiedad de un objeto.

- todos admiten claves de cadena y de símbolo.
- la siguiente tabla resume cuándo cada método devuelve true.

	enumerable, propia	Enumerable, heredada	No enumerable, propia	No enumerable, heredado
<code>propertyIsEnumerable()</code>	cierto ✓	falso ✗	falso ✗	falso ✗
<code>hasOwnProperty()</code>	cierto ✓	falso ✗	cierto ✓	falso ✗
<code>Object.hasOwn()</code>	cierto ✓	falso ✗	cierto ✓	falso ✗
<code>in</code>	cierto ✓	cierto ✓	cierto ✓	cierto ✓

Recorrido de las propiedades de un objeto

Hay muchos métodos en JavaScript que atraviesan un grupo de propiedades de un objeto.

- a veces, estas propiedades se devuelven como una array; a veces, se iteran uno por uno en un bucle;
- a veces, se utilizan para construir o **mutar** otro objeto.

La siguiente tabla resume cuándo se puede visitar una propiedad.

- los métodos que solo visitan propiedades de cadenas o solo propiedades de **Symbol** tendrán una nota adicional.
 - ✓ significa que se visitará una propiedad de este tipo;
 - ✗ significa que no lo hará.

	enumerable, propio	Enumerable, heredado	No enumerable, propio	No enumerable, heredado
<code>Object.keys</code> <code>Object.values</code> <code>Object.entries</code>	✓ (strings)	✗	✗	✗
<code>Object.getOwnPropertyNames</code>	✓ (strings)	✗	✓ (strings)	✗
<code>Object.getOwnPropertySymbols</code>	✓ (Symbol)	✗	✓ (Symbol)	✗
<code>Object.getOwnPropertyDescriptors</code>	✓	✗	✓	✗
<code>Reflect.ownKeys</code>	✓	✗	✓	✗
<code>for...in</code>	✓ (strings)	✓ (strings)	✗	✗

	enumerable, propio	Enumerable, heredado	No enumerable, propio	No enumerable, heredado
<code>Object.assign</code> (Después del primer parámetro)	☑	✗	✗	✗
<code>Object.spread</code>	☑	✗	✗	✗

Obtención de propiedades por enumerabilidad/propiedad

Ten en cuenta que este no es el algoritmo más eficiente para todos los casos, pero es útil para una demostración rápida.

- la detección de una propiedad se puede conseguir con:

```
RecuperadorDePropiedadesSimple.usarElMetodoNecesario(obj).includes(prop)
```

- la iteración puede lograrse mediante

```
RecuperadorDePropiedadesSimple.usarElMetodoNecesario(obj).forEach((valor, prop) => {});  
(o user filter(), map(), etc.)
```

```
const RecuperadorDePropiedadesSimple = {

  getPropiasEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, false, this._enumerables);
    // También podríamos usar for...in filtrando con Object.hasOwn
    // o también simplemente: return Object.keys(obj);
  },

  getPropiasNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, false, this._enumerablesYNoEnumerables);
  },

  getPropiasEnumerablesYNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, false, this._enumerablesYNoEnumerables);
    // O simplemente usar: return Object.getOwnPropertyNames(obj);
  },

  getPrototipoEnumerables(obj) {
    return this._getNombresPropiedades(obj, false, true, this._enumerables);
  },

  getPrototipoNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, false, true, this._noEnumerables);
  },

  getPrototipoEnumerablesYNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, false, true, this._enumerablesYNoEnumerables);
  },

  getPropiasYPrototipoEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, true, this._enumerables);
    // O se podría usar sin filtrar for...in
  },

  getPropiasYPrototipoNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, true, this._noEnumerables);
  },

  getPropiasYPrototipoEnumerablesYNoEnumerables(obj) {
    return this._getNombresPropiedades(obj, true, true, this._enumerablesYNoEnumerables);
  },

  // Callbacks para comprobar propiedades static privadas
  _enumerables(obj, prop) {
    return Object.prototype.propertyIsEnumerable.call(obj, prop);
  },

  _noEnumerables(obj, prop) {
    return !Object.prototype.propertyIsEnumerable.call(obj, prop);
  },
}
```

```

_enumerablesYNoEnumerables(obj, prop) {
    return true;
},

// Inspirado por http://stackoverflow.com/a/8024294/271577
_getNombresPropiedades(obj, iterarSobreYoMismo, iterarSobrePrototipo, debeIncluir) {
    const props = [];
    do {
        if (iterarSobreYoMismo) {
            Object.getOwnPropertyNames(obj).forEach( (prop) => {
                if (props.indexOf(prop) === -1 && debeIncluir(obj, prop)) {
                    props.push(prop);
                }
            });
        }
        if (!iterarSobrePrototipo) {
            break;
        }
        iterarSobreYoMismo = true;
        obj = Object.getPrototypeOf(obj);
    } while (obj);
    return props;
}
};

```