13. Herencia y cadena de prototipos

JavaScript es un poco confuso para los desarrolladores con experiencia en lenguajes basados en clases (como Java o C++), ya que es dinámico y no tiene tipos estáticos.

Cuando se trata de **herencia**, JavaScript solo tiene una construcción: objetos.

- cada objeto tiene una **propiedad privada** que mantiene un vínculo con otro objeto: su **prototipo**.
- ese **objeto prototipo** también tiene un prototipo propio, y así sucesivamente hasta que se llega a un objeto con `null` como prototipo.
- por definición, `null` no tiene prototipo y actúa como el eslabón final en esta **cadena de prototipos**.
- es posible **mutar** cualquier miembro de la cadena de prototipos o incluso intercambiar el prototipo **en tiempo de ejecución**, por lo que conceptos como **el despacho estático** (***static dispatching***) no existen en JavaScript.

Si bien esta confusión a menudo se considera una de las debilidades de JavaScript, el **modelo de herencia prototípico** en sí mismo es, de hecho, más poderoso que el modelo clásico.

- es, por ejemplo, bastante trivial construir un modelo clásico sobre un modelo prototípico, que es cómo se implementan las clases.

Aunque las clases ahora se adoptan ampliamente y se han convertido en un nuevo paradigma en JavaScript

- las clases no traen un nuevo patrón de herencia.
- si bien las clases abstraen la mayor parte del mecanismo prototípico, sigue siendo útil comprender cómo funcionan los prototipos bajo el capó.

Como se implementa la herencia usando la cadena prototipos

Heredar propiedades

Los objetos de JavaScript son "bolsas" dinámicas de propiedades (referidas como **propiedades propias** (***own properties***)).

- cada objeto JavaScript tienen un enlace a un **objeto prototipo**.
- al intentar acceder a una propiedad de un objeto, la propiedad no solo se buscará en el objeto sino en el prototipo del objeto, el prototipo del prototipo, y así sucesivamente hasta que se encuentre una propiedad con un nombre coincidente o se alcance el final de la cadena prototipos.

Nota:
Siguiendo el estándar ECMAScript, la notación

```
algunObjeto.[[Prototype]]
```

se utiliza para designar el prototipo de `algunObjeto`.

Se puede acceder a la "ranura interna" `[[Prototype]]` con las funciones:

- `Object.getPrototypeOf()`
- `Object.setPrototypeOf()`

Esto es equivalente al **campo accesor** de JavaScript `__proto__` el cual:

- no es estándar pero está **implementado de facto** por muchos motores de JavaScript.
- para evitar confusiones y mantener las posteriores explicaciones lo más sucintas posible, en nuestra notación evitaremos usar

  ***objeto*.__proto__**

  y usaremos el término:

  ***objeto*.[[Prototype]]**

  Esto equivale a `Object.getPrototypeOf( objeto ).`

`[[Prototype]]` no debe confundirse con la propiedad que tiene toda función ***nombreFuncion*.`prototype` :**

- ***nombreFuncion*.`prototype`:** especifica el `[[Prototype]]` que se asignará a todas las instancias de los objetos creados por la función ***nombreFuncion*** cuando se use como constructor (es decir, invocada con **`new`**)

Es un tema un tanto lioso, pero es así.

Hay varias formas de especificar el `[[Prototype]]` de un objeto, que se enumeran en una sección posterior.

- en esta primera explicación, usaremos la sintaxis *objeto*.`__proto__` por razones didácticas.
- también vale la pena señalar que la sintaxis de declarar un literal objeto **`{ __proto__: ... }`** es estándar y no obsoleta (a diferencia de la sintaxis de acceso al campo *objeto*.**`__proto__`**).

Estructura del objeto base de la cadena de prototipos: `Object.prototype`

CHROME

```
Object.prototype;
{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__
perty: ƒ, __lookupGetter__: ƒ, …}
  constructor: ƒ Object()
  hasOwnProperty: ƒ hasOwnProperty()
  isPrototypeOf: ƒ isPrototypeOf()
  propertyIsEnumerable: ƒ propertyIsEnumerable()
  toLocaleString: ƒ toLocaleString()
  toString: ƒ toString()
  valueOf: ƒ valueOf()
  __defineGetter__: ƒ __defineGetter__()
  __defineSetter__: ƒ __defineSetter__()
  __lookupGetter__: ƒ __lookupGetter__()
  __lookupSetter__: ƒ __lookupSetter__()
  __proto__: null
  get __proto__: ƒ __proto__()
  set __proto__: ƒ __proto__()
```

FIREFOX

```
Object.prototype;
Object { … }
  __defineGetter__: function __defineGetter__()
  __defineSetter__: function __defineSetter__()
  __lookupGetter__: function __lookupGetter__()
  __lookupSetter__: function __lookupSetter__()
  __proto__: null
  constructor: function Object()
  hasOwnProperty: function hasOwnProperty()
  isPrototypeOf: function isPrototypeOf()
  propertyIsEnumerable: function propertyIsEnumerable()
  toLocaleString: function toLocaleString()
  toString: function toString()
  valueOf: function valueOf()
  <get __proto__()>: function __proto__()
  <set __proto__()>: function __proto__()
```

En un **literal objeto** como `{ a: 1, b: 2 }`

- las `a` y `b` ***propiedades propias*** del objeto.
- el `[[Prototype]]` es `Object.prototype`.

CHROME

```
const o = {
  a: 1,
  b: 2,};
undefined

o;
{a: 1, b: 2}
  a: 1
  b: 2
  [[Prototype]]: Object
    constructor: ƒ Object()
    hasOwnProperty: ƒ hasOwnProperty()
    isPrototypeOf: ƒ isPrototypeOf()
    propertyIsEnumerable: ƒ propertyIsEnumerable()
    toLocaleString: ƒ toLocaleString()
    toString: ƒ toString()
    valueOf: ƒ valueOf()
    __defineGetter__: ƒ __defineGetter__()
    __defineSetter__: ƒ __defineSetter__()
    __lookupGetter__: ƒ __lookupGetter__()
    __lookupSetter__: ƒ __lookupSetter__()
    __proto__: Object
      constructor: ƒ Object()
      hasOwnProperty: ƒ hasOwnProperty()
      isPrototypeOf: ƒ isPrototypeOf()
      propertyIsEnumerable: ƒ propertyIsEnumerable()
      toLocaleString: ƒ toLocaleString()
      toString: ƒ toString()
      valueOf: ƒ valueOf()
      __defineGetter__: ƒ __defineGetter__()
      __defineSetter__: ƒ __defineSetter__()
      __lookupGetter__: ƒ __lookupGetter__()
      __lookupSetter__: ƒ __lookupSetter__()
      __proto__: null
      get __proto__: ƒ __proto__()
      set __proto__: ƒ __proto__()
    get __proto__: ƒ __proto__()
    set __proto__: ƒ __proto__()
```

FIREFOX

```
const o = {
  a: 1,
  b: 2,};
undefined

o;
Object { a: 1, b: 2 }
    a: 1
    b: 2
  <prototype>: Object { … }
    __defineGetter__: function __defineGetter__()
    __defineSetter__: function __defineSetter__()
    __lookupGetter__: function __lookupGetter__()
    __lookupSetter__: function __lookupSetter__()
    __proto__: Object { … }
      __defineGetter__: function __defineGetter__()
      __defineSetter__: function __defineSetter__()
      __lookupGetter__: function __lookupGetter__()
      __lookupSetter__: function __lookupSetter__()
      __proto__: null
      constructor: function Object()
      hasOwnProperty: function hasOwnProperty()
      isPrototypeOf: function isPrototypeOf()
      propertyIsEnumerable: function propertyIsEnumerable()
      toLocaleString: function toLocaleString()
      toString: function toString()
      valueOf: function valueOf()
      <get __proto__()>: function __proto__()
      <set __proto__()>: function __proto__()
    constructor: function Object()
    hasOwnProperty: function hasOwnProperty()
    isPrototypeOf: function isPrototypeOf()
    propertyIsEnumerable: function propertyIsEnumerable()
    toLocaleString: function toLocaleString()
    toString: function toString()
    valueOf: function valueOf()
    <get __proto__()>: function __proto__()
    <set __proto__()>: function __proto__()
```

En un **literal objeto** como `{ a: 1, b: 2, __proto__: c }`

- el valor `c` (que tiene que ser `null` u otro objeto) se convertirá en el `[[Prototype]]` del objeto representado por el literal objeto `{ a: 1, b: 2, __proto__: c }`
- mientras que las otras claves como `a` y `b` se convertirán en ***propiedades propias*** del objeto.
- esta sintaxis se lee muy naturalmente, ya que `[[Prototype]]` es solo una " **propiedad interna** " del objeto.

Esto es lo que sucede al intentar acceder a una propiedad:

```
const o = {
  a: 1,
  b: 2,
  // __proto__ configura el [[Prototype]]. Aparece especificado aquí como otro literal objeto
  __proto__ : {
    b: 3,
    c: 4,
  },
};

//  o.[[Prototype]] tiene las propiedades b y c
//
//  o.[[Prototype]].[[Prototype]] is Object.prototype
//
//  Finalmente, o.[[Prototype]].[[Prototype]].[[Prototype]] is null.
//    hemos llegado al final de la cadena de prototipos (null) que
//    por definición no tiene [[Prototype]].
//
// Por lo tanto, y para resumir, la cadena completa de prototipos es:
//    { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> Object.prototype ---> null

console.log(o.a);
// 1
// ¿Tiene o una propiedad a? Sí, y su valor es 1.

console.log(o.b);
// 2
// ¿Tiene o una propiedad b? Sí, y su valor es 2.
// El [[Prototype]] también tiene una propiedad 'b', pero no llegamos a e ella:
//  esto es lo que se denomina sombreado de propiedad (Property Shadowing)

console.log(o.c);
// 4
// ¿Tiene o una propiedad c? No, por lo tanto tenemos que comprobar su [[Prototype]].
// ¿Tiene o.[[Prototype]] una propiedad c? Sí, su valor es 4.

console.log(o.d);
// undefined
// ¿Tiene o una propiedad d? No, por lo tanto tenemos que comprobar su [[Prototype]]
//
// ¿Tiene o.[[Prototype]] una propiedad d? No, por lo tanto tenemos que comprobar su
//      o.[[Prototype]].[[Prototype]]
//
// ¿Tiene o.[[Prototype]].[[Prototype]] una propiedad d?
//    o.[[Prototype]].[[Prototype]] es lo mismo que Object.prototype, que no tiene la propiedad d
//   Por lo que tenemos que comprobar su [[Prototype]]
//
// ¿Tiene o.[[Prototype]].[[Prototype]].[[Prototype]] una propiedad c? No, pues es null
//   se acabó la búsqueda: al no haber encontrado la propiedad devolvemos undefined
```

Establecer una propiedad en un objeto crea una **propiedad propia**.

- la única excepción a las reglas de obtener (get) y configurar (set) el comportamiento es cuando es interceptado por un **getter** o **setter**.

Veamos esto con las DevTools de 2 navegadores actuales:

CHROME:

```
o;
▼{a: 1, b: 2} i
    a: 1
    b: 2
  ▶[[Prototype]]: Object
```

FIREFOX:

```
o;
▼ Object { a: 1, b: 2 }
    a: 1
    b: 2
  ▶ <prototype>: Object { b: 3, c: 4 }
```

Más detelladamente:

CHROME:

```
o;
{a: 1, b: 2}
  a: 1
  b: 2
  [[Prototype]]: Object
    b: 3
    c: 4
    [[Prototype]]: Object
      constructor: ƒ Object()
      hasOwnProperty: ƒ hasOwnProperty()
      isPrototypeOf: ƒ isPrototypeOf()
      propertyIsEnumerable: ƒ propertyIsEnumerable()
      toLocaleString: ƒ toLocaleString()
      toString: ƒ toString()
      valueOf: ƒ valueOf()
      __defineGetter__: ƒ __defineGetter__()
      __defineSetter__: ƒ __defineSetter__()
      __lookupGetter__: ƒ __lookupGetter__()
      __lookupSetter__: ƒ __lookupSetter__()
      __proto__: Object
        b: 3
        c: 4
        [[Prototype]]: Object
          constructor: ƒ Object()
          hasOwnProperty: ƒ hasOwnProperty()
          isPrototypeOf: ƒ isPrototypeOf()
          propertyIsEnumerable: ƒ propertyIsEnumerable
          toLocaleString: ƒ toLocaleString()
          toString: ƒ toString()
          valueOf: ƒ valueOf()
          __defineGetter__: ƒ __defineGetter__()
          __defineSetter__: ƒ __defineSetter__()
          __lookupGetter__: ƒ __lookupGetter__()
          __lookupSetter__: ƒ __lookupSetter__()
          __proto__: Object
            constructor: ƒ Object()
            hasOwnProperty: ƒ hasOwnProperty()
            isPrototypeOf: ƒ isPrototypeOf()
            propertyIsEnumerable: ƒ propertyIsEnumerabl
            toLocaleString: ƒ toLocaleString()
            toString: ƒ toString()
            valueOf: ƒ valueOf()
            __defineGetter__: ƒ __defineGetter__()
            __defineSetter__: ƒ __defineSetter__()
            __lookupGetter__: ƒ __lookupGetter__()
            __lookupSetter__: ƒ __lookupSetter__()
            __proto__: null
            get __proto__: ƒ __proto__()
            set __proto__: ƒ __proto__()
          get __proto__: ƒ __proto__()
          set __proto__: ƒ __proto__()
        get __proto__: ƒ __proto__()
        set __proto__: ƒ __proto__()
```

FIREFOX:

```
o;
Object { a: 1, b: 2 }
  a: 1
  b: 2
  <prototype>: Object { b: 3, c: 4 }
    b: 3
    c: 4
    <prototype>: Object { … }
      __defineGetter__: function __defineGetter__()
      __defineSetter__: function __defineSetter__()
      __lookupGetter__: function __lookupGetter__()
      __lookupSetter__: function __lookupSetter__()
      __proto__: Object { b: 3, c: 4 }
        b: 3
        c: 4
        <prototype>: Object { … }
          __defineGetter__: function __defineGetter__()
          __defineSetter__: function __defineSetter__()
          __lookupGetter__: function __lookupGetter__()
          __lookupSetter__: function __lookupSetter__()
          __proto__: Object { … }
            __defineGetter__: function __defineGetter__()
            __defineSetter__: function __defineSetter__()
            __lookupGetter__: function __lookupGetter__()
            __lookupSetter__: function __lookupSetter__()
            __proto__: null
            constructor: function Object()
            hasOwnProperty: function hasOwnProperty()
            isPrototypeOf: function isPrototypeOf()
            propertyIsEnumerable: function
          propertyIsEnumerable()
            toLocaleString: function toLocaleString()
            toString: function toString()
            valueOf: function valueOf()
            <get __proto__()>: function __proto__()
            <set __proto__()>: function __proto__()
          constructor: function Object()
          hasOwnProperty: function hasOwnProperty()
          isPrototypeOf: function isPrototypeOf()
          propertyIsEnumerable: function propertyIsEnumerab
          toLocaleString: function toLocaleString()
          toString: function toString()
          valueOf: function valueOf()
          <get __proto__()>: function __proto__()
          <set __proto__()>: function __proto__()
        constructor: function Object()
        hasOwnProperty: function hasOwnProperty()
        isPrototypeOf: function isPrototypeOf()
        propertyIsEnumerable: function propertyIsEnumerable()
        toLocaleString: function toLocaleString()
        toString: function toString()
        valueOf: function valueOf()
        <get __proto__()>: function __proto__()
        <set __proto__()>: function __proto__()
```

Observamos que los navegadores actuales muestran:

- el primer `[[Prototype]]` que aparece (remarcado en amarillo), que confunde un poco, es en realidad el campo `__proto__` (si nos fijamos, no tiene los métodos heredados de `Object.prototype`, ni un campo `__proto__` propio)
- cada `[[Prototype]]` posterior, al ser en sí mismo un objeto, tiene también su propio objeto `[[Prototype]]`, que son las propiedades "heredadas", contenidas en el campo `__proto__` del dicho `[[Prototype]]`
- una pista importante para entender bien esto es fijarse que hay campos `__proto__` o `[[Prototype]]` que tienen los métodos de `Object.prototype` y otros que no
- los campos **`a`** y **`b`**, son **propiedades propias** (***own properties***) de este objeto concreto y no forman parte de prototipo

Podemos verificar lo anterior:

```
o.__proto__;
▼ {b: 3, c: 4} i
    b: 3
    c: 4
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
    ▼ __proto__: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
        __proto__: null
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

```
o.__proto__;
▼ Object { b: 3, c: 4 }
    b: 3
    c: 4
  ▼ <prototype>: Object { … }
    ▶ __defineGetter__: function __defineGetter__()
    ▶ __defineSetter__: function __defineSetter__()
    ▶ __lookupGetter__: function __lookupGetter__()
    ▶ __lookupSetter__: function __lookupSetter__()
    ▼ __proto__: Object { … }
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
        __proto__: null
      ▶ constructor: function Object()
      ▶ hasOwnProperty: function hasOwnProperty()
      ▶ isPrototypeOf: function isPrototypeOf()
      ▶ propertyIsEnumerable: function propertyIsEnumerable()
      ▶ toLocaleString: function toLocaleString()
      ▶ toString: function toString()
      ▶ valueOf: function valueOf()
      ▶ <get __proto__()>: function __proto__()
      ▶ <set __proto__()>: function __proto__()
    ▶ constructor: function Object()
    ▶ hasOwnProperty: function hasOwnProperty()
    ▶ isPrototypeOf: function isPrototypeOf()
    ▶ propertyIsEnumerable: function propertyIsEnumerable()
    ▶ toLocaleString: function toLocaleString()
    ▶ toString: function toString()
    ▶ valueOf: function valueOf()
    ▶ <get __proto__()>: function __proto__()
    ▶ <set __proto__()>: function __proto__()
```

```
Object.getPrototypeOf(o);
▼ {b: 3, c: 4} i
    b: 3
    c: 4
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
    ▼ __proto__: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
        __proto__: null
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

```
Object.getPrototypeOf(o);
▼ Object { b: 3, c: 4 }
    b: 3
    c: 4
  ▼ <prototype>: Object { … }
    ▶ __defineGetter__: function __defineGetter__()
    ▶ __defineSetter__: function __defineSetter__()
    ▶ __lookupGetter__: function __lookupGetter__()
    ▶ __lookupSetter__: function __lookupSetter__()
    ▼ __proto__: Object { … }
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
        __proto__: null
      ▶ constructor: function Object()
      ▶ hasOwnProperty: function hasOwnProperty()
      ▶ isPrototypeOf: function isPrototypeOf()
      ▶ propertyIsEnumerable: function propertyIsEnumerable()
      ▶ toLocaleString: function toLocaleString()
      ▶ toString: function toString()
      ▶ valueOf: function valueOf()
      ▶ <get __proto__()>: function __proto__()
      ▶ <set __proto__()>: function __proto__()
    ▶ constructor: function Object()
    ▶ hasOwnProperty: function hasOwnProperty()
    ▶ isPrototypeOf: function isPrototypeOf()
    ▶ propertyIsEnumerable: function propertyIsEnumerable()
    ▶ toLocaleString: function toLocaleString()
    ▶ toString: function toString()
    ▶ valueOf: function valueOf()
    ▶ <get __proto__()>: function __proto__()
    ▶ <set __proto__()>: function __proto__()
```

Del mismo modo, puedes crear cadenas de prototipos más largas, y el motor de JavaScript buscará una propiedad yendo de uno en uno de todos ellos.

```
const o = {
  a: 1,
  b: 2,
  // __proto__ configura el [[Prototype]]. Aparece especificado aquí como otro literal objeto
  __proto__: {
      b: 3,
      c: 4,
      __proto__: {
          d: 5,
      },
  },
};

// Por lo tanto, y para resumir, la cadena completa de prototipos es:
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> { d: 5 } ---> Object.prototype ---> null

console.log(o.d);
// 5
```

```
o;
▼{a: 1, b: 2} ℹ
    a: 1
    b: 2
  ▼[[Prototype]]: Object
      b: 3
      c: 4
    ▼[[Prototype]]: Object
        d: 5
      ▼[[Prototype]]: Object
        ▶constructor: ƒ Object()
        ▶hasOwnProperty: ƒ hasOwnProperty()
        ▶isPrototypeOf: ƒ isPrototypeOf()
        ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
        ▶toLocaleString: ƒ toLocaleString()
        ▶toString: ƒ toString()
        ▶valueOf: ƒ valueOf()
        ▶__defineGetter__: ƒ __defineGetter__()
        ▶__defineSetter__: ƒ __defineSetter__()
        ▶__lookupGetter__: ƒ __lookupGetter__()
        ▶__lookupSetter__: ƒ __lookupSetter__()
        ▼__proto__: Object
            b: 3
            c: 4
          ▼[[Prototype]]: Object
              d: 5
            ▼[[Prototype]]: Object
              ▶constructor: ƒ Object()
              ▶hasOwnProperty: ƒ hasOwnProperty()
              ▶isPrototypeOf: ƒ isPrototypeOf()
              ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
              ▶toLocaleString: ƒ toLocaleString()
              ▶toString: ƒ toString()
              ▶valueOf: ƒ valueOf()
              ▶__defineGetter__: ƒ __defineGetter__()
              ▶__defineSetter__: ƒ __defineSetter__()
              ▶__lookupGetter__: ƒ __lookupGetter__()
              ▶__lookupSetter__: ƒ __lookupSetter__()
              ▼__proto__: Object
                  d: 5
                ▼[[Prototype]]: Object
                  ▶constructor: ƒ Object()
                  ▶hasOwnProperty: ƒ hasOwnProperty()
                  ▶isPrototypeOf: ƒ isPrototypeOf()
                  ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
                  ▶toLocaleString: ƒ toLocaleString()
                  ▶toString: ƒ toString()
                  ▶valueOf: ƒ valueOf()
                  ▶__defineGetter__: ƒ __defineGetter__()
                  ▶__defineSetter__: ƒ __defineSetter__()
                  ▶__lookupGetter__: ƒ __lookupGetter__()
                  ▶__lookupSetter__: ƒ __lookupSetter__()
                  ▼__proto__: Object
                    ▶constructor: ƒ Object()
                    ▶hasOwnProperty: ƒ hasOwnProperty()
                    ▶isPrototypeOf: ƒ isPrototypeOf()
                    ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
                    ▶toLocaleString: ƒ toLocaleString()
                    ▶toString: ƒ toString()
                    ▶valueOf: ƒ valueOf()
                    ▶__defineGetter__: ƒ __defineGetter__()
                    ▶__defineSetter__: ƒ __defineSetter__()
                    ▶__lookupGetter__: ƒ __lookupGetter__()
                    ▶__lookupSetter__: ƒ __lookupSetter__()
                      __proto__: null
                    ▶get __proto__: ƒ __proto__()
                    ▶set __proto__: ƒ __proto__()
                  ▶get __proto__: ƒ __proto__()
                  ▶set __proto__: ƒ __proto__()
              ▶get __proto__: ƒ __proto__()
              ▶set __proto__: ƒ __proto__()
        ▶get __proto__: ƒ __proto__()
        ▶set __proto__: ƒ __proto__()
```

```
o;
▼ Object { a: 1, b: 2 }
    a: 1
    b: 2
  ▼ <prototype>: Object { b: 3, c: 4 }
      b: 3
      c: 4
    ▼ <prototype>: Object { d: 5 }
        d: 5
      ▼ <prototype>: Object { … }
        ▶ __defineGetter__: function __defineGetter__()
        ▶ __defineSetter__: function __defineSetter__()
        ▶ __lookupGetter__: function __lookupGetter__()
        ▶ __lookupSetter__: function __lookupSetter__()
        ▼ __proto__: Object { b: 3, c: 4 }
            b: 3
            c: 4
          ▼ <prototype>: Object { d: 5 }
              d: 5
            ▼ <prototype>: Object { … }
              ▶ __defineGetter__: function __defineGetter__()
              ▶ __defineSetter__: function __defineSetter__()
              ▶ __lookupGetter__: function __lookupGetter__()
              ▶ __lookupSetter__: function __lookupSetter__()
              ▼ __proto__: Object { d: 5 }
                  d: 5
                ▼ <prototype>: Object { … }
                  ▶ __defineGetter__: function __defineGetter__()
                  ▶ __defineSetter__: function __defineSetter__()
                  ▶ __lookupGetter__: function __lookupGetter__()
                  ▶ __lookupSetter__: function __lookupSetter__()
                  ▼ __proto__: Object { … }
                    ▶ __defineGetter__: function __defineGetter__()
                    ▶ __defineSetter__: function __defineSetter__()
                    ▶ __lookupGetter__: function __lookupGetter__()
                    ▶ __lookupSetter__: function __lookupSetter__()
                      __proto__: null
                    ▶ constructor: function Object()
                    ▶ hasOwnProperty: function hasOwnProperty()
                    ▶ isPrototypeOf: function isPrototypeOf()
                    ▶ propertyIsEnumerable: function propertyIsEnumerable()
                    ▶ toLocaleString: function toLocaleString()
                    ▶ toString: function toString()
                    ▶ valueOf: function valueOf()
                    ▶ <get __proto__()>: function __proto__()
                    ▶ <set __proto__()>: function __proto__()
                  ▶ constructor: function Object()
                  ▶ hasOwnProperty: function hasOwnProperty()
                  ▶ isPrototypeOf: function isPrototypeOf()
                  ▶ propertyIsEnumerable: function propertyIsEnumerable()
                  ▶ toLocaleString: function toLocaleString()
                  ▶ toString: function toString()
                  ▶ valueOf: function valueOf()
                  ▶ <get __proto__()>: function __proto__()
                  ▶ <set __proto__()>: function __proto__()
              ▶ constructor: function Object()
              ▶ hasOwnProperty: function hasOwnProperty()
              ▶ isPrototypeOf: function isPrototypeOf()
              ▶ propertyIsEnumerable: function propertyIsEnumerable()
              ▶ toLocaleString: function toLocaleString()
              ▶ toString: function toString()
              ▶ valueOf: function valueOf()
              ▶ <get __proto__()>: function __proto__()
              ▶ <set __proto__()>: function __proto__()
        ▶ constructor: function Object()
        ▶ hasOwnProperty: function hasOwnProperty()
        ▶ isPrototypeOf: function isPrototypeOf()
        ▶ propertyIsEnumerable: function propertyIsEnumerable()
        ▶ toLocaleString: function toLocaleString()
        ▶ toString: function toString()
        ▶ valueOf: function valueOf()
        ▶ <get __proto__()>: function __proto__()
        ▶ <set __proto__()>: function __proto__()
```

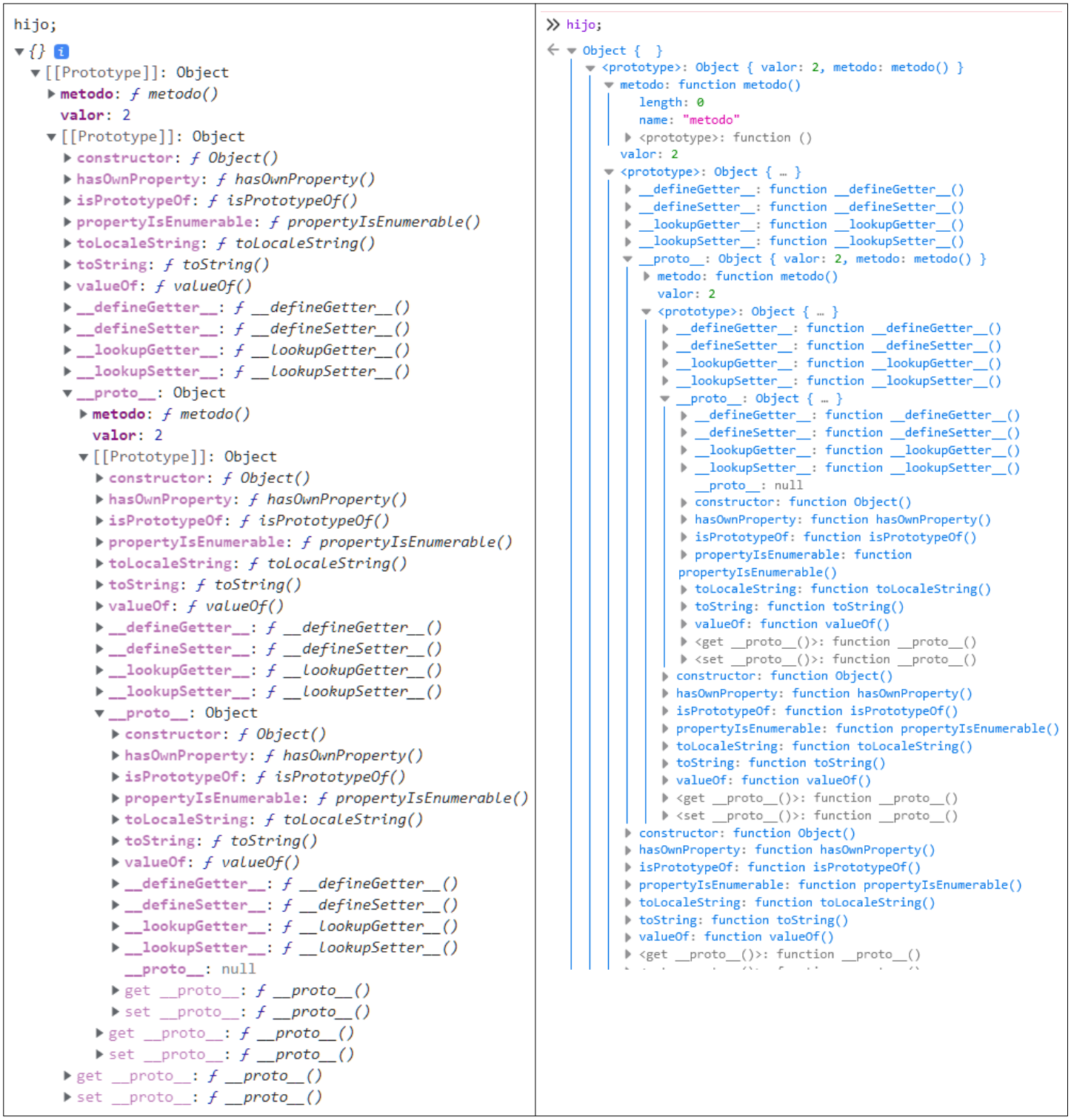
Heredar "métodos"

JavaScript no tiene "métodos" en la forma en que los definen los lenguajes basados en clases.

- en JavaScript, cualquier función se puede agregar a un objeto en forma de propiedad.
- una función heredada actúa como cualquier otra propiedad, incluido **el sombreado de propiedades** como se muestra arriba (en este caso, una forma de ***sobreescribir métodos*** (***method overriding***)).

Cuando se ejecuta una función heredada, el valor de `this` apunta al objeto heredado, no al objeto prototipo donde la función es una propiedad propia.

```
const madre = {
  valor: 2,
  metodo() {
    return this.valor + 1;
  }
};

console.log(madre.metodo());
// 3
// cuando llamamos al método madre.metodo, 'this' hace referencia al metodo madre.metodo()
// hijo es un objeto que hereda de madre
const hijo = {
  __proto__: madre,
};
console.log(hijo.metodo());
// 3
// Cuando invocamos el método hijo.metodo, 'this' hace referencia a hijo.
// Por lo tanto, cuando el hijo hereda el método metodo() la propiedad 'valor'
// se busca primero en el hijo. Sin embargo, puesto que hijo no tiene una propiedad
// 'valor', se buscará la propiedad en el [[Prototype]], y será madre.valor

hijo.valor = 4;
// al asignar 4 a la propiedad valor de hijo, estmos crendo una
//  propiedad propia valor en el hijo: esto sombreará la propiedad 'valor' en madre
//
// Ahora el objeto hijo tendrá la estructura:
// { valor: 4, __proto__: { valor: 2, metodo: [Function] } }

console.log(hijo.metodo());
// 5
// Puesto que ahora el hijo tiene una propiedad propia 'valor' this.valor significa
//  hijo.valor

console.log(hijo.__proto__.valor);
// 3
```


hijo;
{} i
[[Prototype]]: Object
metodo: ƒ metodo()
valor: 2
[[Prototype]]: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: Object
metodo: ƒ metodo()
valor: 2
[[Prototype]]: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: null
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
hijo;
Object { }
<prototype>: Object { valor: 2, metodo: metodo() }
metodo: function metodo()
length: 0
name: "metodo"
<prototype>: function ()
valor: 2
<prototype>: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: Object { valor: 2, metodo: metodo() }
metodo: function metodo()
valor: 2
<prototype>: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: null
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable()
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()
<set __proto__()>: function __proto__()
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable()
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()

```
hijo2 = Object.create(hijo);
▶ {}
hijo2;
▼ {} i
  ▼ [[Prototype]]: Object
    ▼ [[Prototype]]: Object
      ▶ metodo: ƒ metodo()
        valor: 2
      ▼ [[Prototype]]: Object
        ▶ constructor: ƒ Object()
        ▶ hasOwnProperty: ƒ hasOwnProperty()
        ▶ isPrototypeOf: ƒ isPrototypeOf()
        ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
        ▶ toLocaleString: ƒ toLocaleString()
        ▶ toString: ƒ toString()
        ▶ valueOf: ƒ valueOf()
        ▶ __defineGetter__: ƒ __defineGetter__()
        ▶ __defineSetter__: ƒ __defineSetter__()
        ▶ __lookupGetter__: ƒ __lookupGetter__()
        ▶ __lookupSetter__: ƒ __lookupSetter__()
        ▼ __proto__: Object
          ▼ [[Prototype]]: Object
            ▶ metodo: ƒ metodo()
              valor: 2
            ▼ [[Prototype]]: Object
              ▶ constructor: ƒ Object()
              ▶ hasOwnProperty: ƒ hasOwnProperty()
              ▶ isPrototypeOf: ƒ isPrototypeOf()
              ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
              ▶ toLocaleString: ƒ toLocaleString()
              ▶ toString: ƒ toString()
              ▶ valueOf: ƒ valueOf()
              ▶ __defineGetter__: ƒ __defineGetter__()
              ▶ __defineSetter__: ƒ __defineSetter__()
              ▶ __lookupGetter__: ƒ __lookupGetter__()
              ▶ __lookupSetter__: ƒ __lookupSetter__()
              ▼ __proto__: Object
                ▶ metodo: ƒ metodo()
                  valor: 2
                ▶ [[Prototype]]: Object
              ▶ get __proto__: ƒ __proto__()
              ▶ set __proto__: ƒ __proto__()
        ▶ get __proto__: ƒ __proto__()
        ▶ set __proto__: ƒ __proto__()
```

```
hijo2 = Object.create(hijo);
▶ Object {  }
hijo2;
▼ Object {  }
  ▼ <prototype>: Object {  }
    ▼ <prototype>: Object { valor: 2, metodo: metodo() }
      ▶ metodo: function metodo()
        valor: 2
      ▼ <prototype>: Object { … }
        ▶ __defineGetter__: function __defineGetter__()
        ▶ __defineSetter__: function __defineSetter__()
        ▶ __lookupGetter__: function __lookupGetter__()
        ▶ __lookupSetter__: function __lookupSetter__()
        ▼ __proto__: Object {  }
          ▼ <prototype>: Object { valor: 2, metodo: metodo() }
            ▶ metodo: function metodo()
              valor: 2
            ▼ <prototype>: Object { … }
              ▶ __defineGetter__: function __defineGetter__()
              ▶ __defineSetter__: function __defineSetter__()
              ▶ __lookupGetter__: function __lookupGetter__()
              ▶ __lookupSetter__: function __lookupSetter__()
              ▼ __proto__: Object { valor: 2, metodo: metodo()
                ▶ metodo: function metodo()
                  valor: 2
                ▼ <prototype>: Object { … }
                  ▶ __defineGetter__: function
                  __defineGetter__()
                  ▶ __defineSetter__: function
                  __defineSetter__()
                  ▶ __lookupGetter__: function
                  __lookupGetter__()
                  ▶ __lookupSetter__: function
                  __lookupSetter__()
                  ▼ __proto__: Object { … }
                    ▶ __defineGetter__: function
                    __defineGetter__()
                    ▶ __defineSetter__: function
                    __defineSetter__()
                    ▶ __lookupGetter__: function
                    __lookupGetter__()
                    ▶ __lookupSetter__: function
                    __lookupSetter__()
                      __proto__: null
                    ▶ constructor: function Object()
                    ▶ hasOwnProperty: function
                    hasOwnProperty()
                    ▶ isPrototypeOf: function isPrototypeOf()
                    ▶ propertyIsEnumerable: function
                    propertyIsEnumerable()
                    ▶ toLocaleString: function
                    toLocaleString()
                    ▶ toString: function toString()
                    ▶ valueOf: function valueOf()
                    ▶ <get __proto__()>: function __proto__()
                    ▶ <set __proto__()>: function __proto__()
                  ▶ constructor: function Object()
                  ▶ hasOwnProperty: function hasOwnProperty()
                  ▶ isPrototypeOf: function isPrototypeOf()
                  ▶ propertyIsEnumerable: function
                  propertyIsEnumerable()
                  ▶ toLocaleString: function toLocaleString()
                  ▶ toString: function toString()
                  ▶ valueOf: function valueOf()
                  ▶ <get __proto__()>: function __proto__()
                  ▶ <set __proto__()>: function __proto__()
                ▶ constructor: function Object()
                ▶ hasOwnProperty: function hasOwnProperty()
                ▶ isPrototypeOf: function isPrototypeOf()
                ▶ propertyIsEnumerable: function
                propertyIsEnumerable()
                ▶ toLocaleString: function toLocaleString()
                ▶ toString: function toString()
                ▶ valueOf: function valueOf()
                ▶ <get __proto__()>: function __proto__()
                ▶ <set __proto__()>: function __proto__()
              ▶ constructor: function Object()
              ▶ hasOwnProperty: function hasOwnProperty()
              ▶ isPrototypeOf: function isPrototypeOf()
              ▶ propertyIsEnumerable: function propertyIsEnumerable()
              ▶ toLocaleString: function toLocaleString()
              ▶ toString: function toString()
              ▶ valueOf: function valueOf()
              ▶ <get __proto__()>: function __proto__()
              ▶ <set __proto__()>: function __proto__()
```

Constructores

El poder de los prototipos es que podemos reutilizar un conjunto de propiedades si fuera útil que estén presentes en cada instancia, especialmente para los métodos.

Supongamos que vamos a crear una serie de cajas, donde cada caja es un objeto que contiene un valor al que se puede acceder a través de una función `getValor`.

Una implementación ingenua sería:

```
const cajas = [
{ valor: 1, getValor() { return this.valor; } },
{ valor: 2, getValor() { return this.valor; } },
{ valor: 3, getValor() { return this.valor; } },
];
```

Esto es una solución pobre, porque cada instancia tiene su propia propiedad de función que hace lo mismo, lo cual es redundante e innecesario.

En su lugar, podemos mover `getValor` al `[[Prototype]]` de todas las cajas:

```
const prototipoDeCaja = {
  getValor() { return this.valor; },
};

const cajas = [
  { valor: 1, __proto__ : prototipoDeCaja },
  { valor: 2, __proto__ : prototipoDeCaja },
  { valor: 3, __proto__ : prototipoDeCaja },
];
```

el método `getValor` de todas las cajas se referirá a la misma función, reduciendo el uso de memoria.

Sin embargo, **vincular manualmente** el `__proto__` para cada creación de objetos sigue siendo muy incómodo:.

- aquí es cuando usaríamos una **función constructora**, que establece automáticamente el `[[Prototype]]` para cada objeto fabricado.
- un **constructor** es una función llamada con el operador `new`.

```
// A constructor function
function Caja(valor) {
  this.valor = valor;
}

// Propiedades que tendrán todas la cajas creadas con el constructor (new Caja())
Caja.prototype.getValor = function () {
  return this.valor ;
};

const cajas = [
  new Caja(1),
  new Caja(2),
  new Caja(3),
];
```

Decimos que `new Caja`(1) es una **instancia** creada a partir de la **función constructora** `Caja`.

- `Caja.prototype` no es muy diferente del objeto `prototipoDeCaja` que creamos anteriormente: es solo un objeto simple.
- cada instancia creada a partir de una función de constructora tendrá automáticamente la propiedad de prototipo del constructor como su `[[Prototype]]` es decir,

  ```
  Object.getPrototypeOf(new Caja()) === Caja.prototype. constructor.prototype
  ```

  por defecto tiene una propiedad propia: `constructor`, que hace referencia a la propia función constructora, es decir,

  ```
  Caja.prototype.constructor === Caja
  ```

  Esto permite acceder al constructor original desde cualquier instancia.

Nota:
Si se devuelve un valor no primitivo de la función constructora, ese valor se convertirá en el resultado de la expresion `new`:

- en este caso, es posible que el `[[Prototype]]` no esté enlazado correctamente, pero esto no debería suceder mucho en la práctica.

CHROME

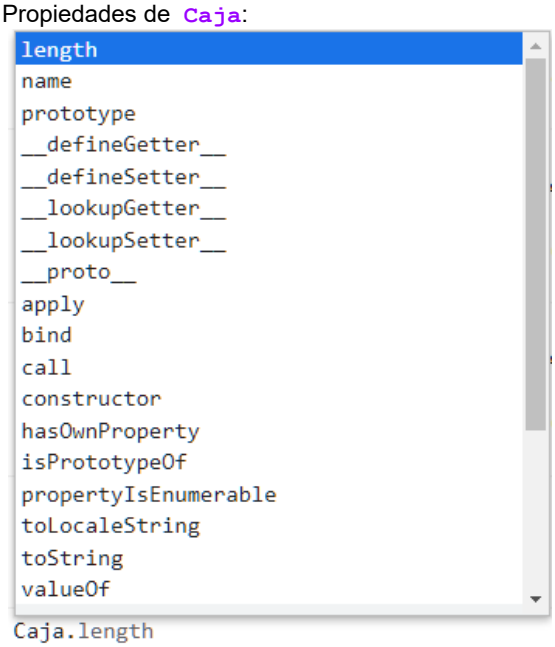
```
const caja = new Caja(1);
undefined

caja;
▼ Caja {valor: 1}
    valor: 1
  ▼ [[Prototype]]: Object
    ▶ getValor: ƒ ()
    ▶ constructor: ƒ Caja(valor)
    ▶ [[Prototype]]: Object

caja;
▼ Caja {valor: 1}
    valor: 1
  ▼ [[Prototype]]: Object
    ▶ getValor: ƒ ()
    ▶ constructor: ƒ Caja(valor)
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
      ▼ __proto__: Object
        ▶ getValor: ƒ ()
        ▶ constructor: ƒ Caja(valor)
        ▼ [[Prototype]]: Object
          ▶ constructor: ƒ Object()
          ▶ hasOwnProperty: ƒ hasOwnProperty()
          ▶ isPrototypeOf: ƒ isPrototypeOf()
          ▶ propertyIsEnumerable: ƒ propertyIsEnumerable
          ▶ toLocaleString: ƒ toLocaleString()
          ▶ toString: ƒ toString()
          ▶ valueOf: ƒ valueOf()
          ▶ __defineGetter__: ƒ __defineGetter__()
          ▶ __defineSetter__: ƒ __defineSetter__()
          ▶ __lookupGetter__: ƒ __lookupGetter__()
          ▶ __lookupSetter__: ƒ __lookupSetter__()
          ▼ __proto__: Object
            ▶ constructor: ƒ Object()
            ▶ hasOwnProperty: ƒ hasOwnProperty()
            ▶ isPrototypeOf: ƒ isPrototypeOf()
            ▶ propertyIsEnumerable: ƒ propertyIsEnumerabl
            ▶ toLocaleString: ƒ toLocaleString()
            ▶ toString: ƒ toString()
            ▶ valueOf: ƒ valueOf()
            ▶ __defineGetter__: ƒ __defineGetter__()
            ▶ __defineSetter__: ƒ __defineSetter__()
            ▶ __lookupGetter__: ƒ __lookupGetter__()
            ▶ __lookupSetter__: ƒ __lookupSetter__()
              __proto__: null
            ▶ get __proto__: ƒ __proto__()
            ▶ set __proto__: ƒ __proto__()
          ▶ get __proto__: ƒ __proto__()
          ▶ set __proto__: ƒ __proto__()
        ▶ get __proto__: ƒ __proto__()
        ▶ set __proto__: ƒ __proto__()
```

FIREFOX

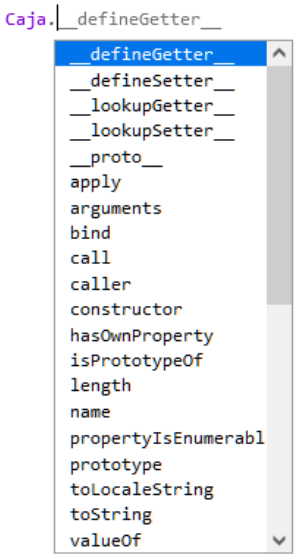
```
const caja = new Caja(1);

» caja;
← ▼ Object { valor: 1 }
    valor: 1
  ▼ <prototype>: Object { getValor: getValor(), … }
    ▶ constructor: function Caja(valor)
    ▶ getValor: function getValor()
    ▼ <prototype>: Object { … }
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
      ▼ __proto__: Object { getValor: getValor(), … }
        ▶ constructor: function Caja(valor)
        ▶ getValor: function getValor()
        ▼ <prototype>: Object { … }
          ▶ __defineGetter__: function __defineGetter__()
          ▶ __defineSetter__: function __defineSetter__()
          ▶ __lookupGetter__: function __lookupGetter__()
          ▶ __lookupSetter__: function __lookupSetter__()
          ▼ __proto__: Object { … }
            ▶ __defineGetter__: function
              __defineGetter__()
            ▶ __defineSetter__: function
              __defineSetter__()
            ▶ __lookupGetter__: function
              __lookupGetter__()
            ▶ __lookupSetter__: function
              __lookupSetter__()
              __proto__: null
            ▶ constructor: function Object()
            ▶ hasOwnProperty: function hasOwnProperty()
            ▶ isPrototypeOf: function isPrototypeOf()
            ▶ propertyIsEnumerable: function
              propertyIsEnumerable()
            ▶ toLocaleString: function toLocaleString()
            ▶ toString: function toString()
            ▶ valueOf: function valueOf()
            ▶ <get __proto__()>: function __proto__()
            ▶ <set __proto__()>: function __proto__()
          ▶ constructor: function Object()
          ▶ hasOwnProperty: function hasOwnProperty()
          ▶ isPrototypeOf: function isPrototypeOf()
          ▶ propertyIsEnumerable: function
            propertyIsEnumerable()
          ▶ toLocaleString: function toLocaleString()
          ▶ toString: function toString()
          ▶ valueOf: function valueOf()
          ▶ <get __proto__()>: function __proto__()
          ▶ <set __proto__()>: function __proto__()
        ▶ constructor: function Object()
        ▶ hasOwnProperty: function hasOwnProperty()
        ▶ isPrototypeOf: function isPrototypeOf()
        ▶ propertyIsEnumerable: function
          propertyIsEnumerable()
        ▶ toLocaleString: function toLocaleString()
        ▶ toString: function toString()
        ▶ valueOf: function valueOf()
        ▶ <get __proto__()>: function __proto__()
        ▶ <set __proto__()>: function __proto__()
```


caja.__proto__;
{getValor: ƒ, constructor: ƒ}
getValor: ƒ ()
constructor: ƒ Caja(valor)
[[Prototype]]: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: null
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
caja.__proto__;
Object { getValor: getValor(), … }
constructor: function Caja(valor)
getValor: function getValor()
<prototype>: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: null
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable()
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()
<set __proto__()>: function __proto__()
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable()
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()
Object.getPrototypeOf(caja);
{getValor: ƒ, constructor: ƒ}
getValor: ƒ ()
constructor: ƒ Caja(valor)
[[Prototype]]: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: Object
constructor: ƒ Object()
hasOwnProperty: ƒ hasOwnProperty()
isPrototypeOf: ƒ isPrototypeOf()
propertyIsEnumerable: ƒ propertyIsEnumerable()
toLocaleString: ƒ toLocaleString()
toString: ƒ toString()
valueOf: ƒ valueOf()
__defineGetter__: ƒ __defineGetter__()
__defineSetter__: ƒ __defineSetter__()
__lookupGetter__: ƒ __lookupGetter__()
__lookupSetter__: ƒ __lookupSetter__()
__proto__: null
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
get __proto__: ƒ __proto__()
set __proto__: ƒ __proto__()
Object.getPrototypeOf(caja);
Object { getValor: getValor(), … }
constructor: function Caja(valor)
getValor: function getValor()
<prototype>: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: Object { … }
__defineGetter__: function __defineGetter__()
__defineSetter__: function __defineSetter__()
__lookupGetter__: function __lookupGetter__()
__lookupSetter__: function __lookupSetter__()
__proto__: null
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()
<set __proto__()>: function __proto__()
constructor: function Object()
hasOwnProperty: function hasOwnProperty()
isPrototypeOf: function isPrototypeOf()
propertyIsEnumerable: function propertyIsEnumerable()
toLocaleString: function toLocaleString()
toString: function toString()
valueOf: function valueOf()
<get __proto__()>: function __proto__()
<set __proto__()>: function __proto__()

Propiedades de `Caja`:



Propiedades de `Caja`:

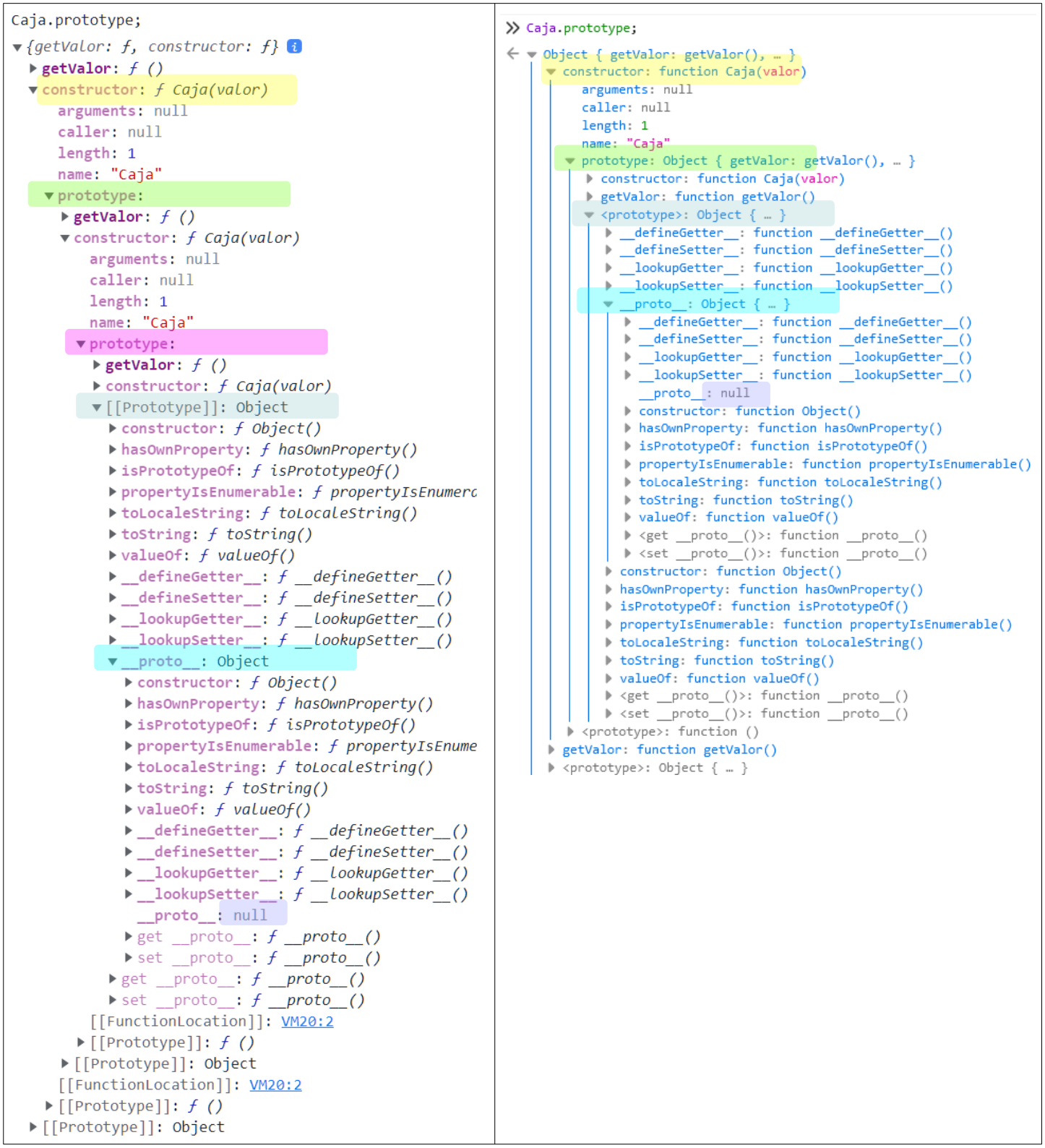



```
Caja.__proto__;
ƒ () { [native code] }
```

Observaciones:

- `Caja` tiene una propiedad `prototype` porque es una función
  - esa propiedad especifica el prototipo por defecto de un objeto creado por esa función cuando esta se usa como constructor.
- ojo: `__proto__` y `prototype` **NO SON LO MISMO**:
  - `prototype:`
    - es una propiedad **solo de funciones**
    - significado: cada vez que se invoca esa función como constructor (con `new`) se crea un objeto y se le asigna como prototipo (*objeto*`.__proto__`) el objeto al que apunta `prototype`.
  - `__proto__:` es una propiedad de **cualquier** objeto (función o cualquier otro) que indica su estructura o composición (propiedades que contiene)

Nos fijamos ahora en lo que hay dentro de **`constructor`**:



La función constructora anterior se puede reescribir usando clases como:

```
class Caja {
  constructor(valor) {
    this.valor = valor;
  }

  // Methods are created on Box.prototype
  getValor () {
    return this.valor;
  }
}
```

Las clases son **azúcar sintáctico** (no añaden más prestaciones ni potencia al lenguaje, pero hacen más cómodo o legible su uso) sobre funciones de constructor,

- lo que significa que aún puedes manipular `Caja.prototype` para cambiar el comportamiento de todas las instancias de `Caja`.
- de todas formas, puesto que las clases fueron diseñadas como una capa de abstracción sobre el mecanismo subyacente de cadena de prototipos, usaremos la sintaxis de función constructora en los siguientes ejemplos, pues es más sencilla para este propósito.

Debido a que `Caja.prototype` hace referencia al mismo objeto que el `[[Prototype]]` de todas las instancias, podemos cambiar el comportamiento de todas las instancias al **mutar** `Caja.prototype`.

```
function Caja(valor) {
  this.valor = valor;
}

// Aquí cambia el método getValue de la clase Box
Caja.prototype.getValor = function () {
  return this.valor ;
};

const caja = new Caja(1);

// Mutate Box.prototype after an instance has already been created
Caja.prototype.getValor = function () {
  return this. valor + 1;
};
caja.getValor(); // 2
```

Una consecuencia que se deriva de lo anterior es que es una mala idea reasignar el prototipo de una función constructora, por dos razones:

`funciónConstructora.prototype`

- el `[[Prototype]]` de las instancias creadas antes de la reasignación ahora hace referencia a un objeto diferente del `[[Prototype]]` de las instancias creadas después de la reasignación:
  - mutar el `[[Prototype]]` de uno ya no muta al otro.
- a menos que restablezca manualmente la propiedad del constructor, la función constructora ya no se puede rastrear desde la ***`instancia`*`.prototype.constructor`**, que puede confundir a los clientes del objeto.
  - algunas operaciones integradas (*built-in*) también leerán la propiedad `constructor` y, si no está configurada, es posible que no funcionen como se esperaba.

`Constructor``.prototype` solo es útil cuando se construyen instancias.

- no tiene nada que ver con *`Constructor`*`.[[Prototype]]`, que es el propio prototipo de la función constructora, que es `Function.prototype`, es decir,

```
Object.getPrototypeOf( Constructor ) === Function.prototype.
```

OBSERVACIÓN:

https://stackoverflow.com/questions/9267157/why-is-it-impossible-to-change-constructor-function-from-prototype

¿Por qué es imposible cambiar la función constructora del prototipo?

Hechémosle un vistazo al ejemplo siguiente:

```
function Conejo() {
    var salta = "Sí";
};
var conejo = new Conejo();
alert(conejo.salta);                    // undefined
alert(Conejo.prototype.constructor);    // La salida es exactamente el código de Conejo();
```

Quiero cambiar el código en `Conejo()` para que la variable `var salta` sea pública y lo hago de esta manera:

```
Conejo.prototype.constructor = function Conejo() {
    this.salta = "no";
};

alert(Conejo.prototype.constructor);    // La salida es exactamente el código de Conejo();
                                        //  y con d with new this.salta = "no";
var conejo2 = new Conejo();             // create new object with new constructor
alert(conejo2.salta);                   // but still outputs undefined
```

¿Por qué no es posible cambiar el código en la función constructora de esta manera?

Respuesta 1:

No puede cambiar un constructor realizando una reasignación `prototype.constructor`
Lo que sucede es que `Conejo.prototype.constructor` es un puntero al constructor original (`function Conejo(){...}`), de modo que los usuarios de la 'clase' pueden detectar el constructor desde una instancia. Por lo tanto, cuando intentas hacer:

```
Conejo.prototype.constructor = function Conejo() {
    this.salta = "no";
};
```

Solo afectará el código que se basa en `prototype.constructor` para instanciar dinámicamente objetos de instancias, de la manera siguiente.

Cuando llamas a `new X`, el motor JS no hace referencia a `X.prototype.constructor`, usa

- la `X` como función constructora, ignorando para esto `X.prototype.constructor`
- `X.prototype` como prototipo del objeto recién creado

Una buena manera de explicar esto es implementar el operador `new` nosotros mismos.

```
// ------------------------------------------
// emulador de `new`
// ------------------------------------------
// No utiliza una referencia a '.constructor' para demostrar que prototype.constructor
//  no se utiliza al instanciar objetos al usar new
function construir(funcionConstructora, arrayArgumentos) {
  // 1) Creamos nueva instancia ligada al prototype pero el constructor,
  //    pero todavía no se habrá invocado la función constructora sobre la instancia
  const nuevaInstancia = Object.create(funcionConstructora.prototype);

  // 2) Invocamos la función constructora sobre la instancia
  funcionConstructora.apply(nuevaInstancia, arrayArgumentos);
  return nuevaInstancia;
}


// If you create a utility function to create from instance, then it uses the
// inherited `constructor` property and your change would affect that.
// En este ejemplo creamos una función de utilidad que permite crear un objeto utilizando
// como plantilla
function construirDesdeInstancia(instancia, arrayArgumentos) {
```

```
  return construir(instancia.constructor, arrayArgumentos);
}

// ------------------------------------------------
// Ahora comprobamos el código anterior:
// ------------------------------------------------
function X(salta) {
  this.salta = salta;
}

// Ahora cambiamos la propiedad constructor para luego ver si tiene efectos:
X.prototype.constructor = function(salta) {
  this.salta = !salta;
}

const xDesdeConstructor = construir(X, [true]);    // equivale a new X(true)
const xDesdeInstancia   = construirDesdeInstancia(xDesdeConstructor, [true]);

console.log({
  xDesdeConstructor,      // salta: true
  xDesdeInstancia         // salta: false
});
```

Como redefinir un constructor correctamente

Si realmente necesitas redefinir un constructor, puedes hacer lo siguiente:

```
// si Conejo tiene cualquier propiedad propia (o métodos static)
//  el siguiente método no los copia: en ese caso también tienes que utilizar:
//  getOwnPropertyNames()

var prototipoAntiguo = Conejo.prototype;
Conejo = function() {...};                  // Nuevo constructor
Conejo.prototype = prototipoAntiguo;        // Restauramos el prototipo antiguo
```

Constructores implícitos en los literales

Algunas sintaxis de literal en JavaScript crean instancias que configuran implícitamente el `[[Prototype]]`. Por ejemplo:

```
// Los literales objeto (que no tengan la clave `__proto__`) automáticamente
// tienen `Object.prototype` como `[[Prototype]]`
const object = { a: 1 };
Object.getPrototypeOf(object) === Object.prototype;
// true

// Los literales array automáticamente tienen `Array.prototype` como su `[[Prototype]]`
const array = [1, 2, 3];
Object.getPrototypeOf(array) === Array.prototype;
// true

// Los literales RegExp automáticamente tienen `RegExp.prototype` como su `[[Prototype]]`
const regexp = /abc/;
Object.getPrototypeOf(regexp) === RegExp.prototype;
// true
```

Podemos "desazucararlos" en su forma de constructor.

```
const array = new Array(1, 2, 3);
const regexp = new RegExp("abc");
```

Por ejemplo, los "métodos de array" como `map()` son simplemente métodos definidos en `Array.prototype`, por lo que están disponibles automáticamente en todas las instancias de array.

Advertencia:
Hay un fallo por parte de programadores que solía ser frecuente: **extender** `Object.prototype` o uno de los otros prototipos integrados:

- un ejemplo de este vicio de programamción consistía en definir `Array.prototype.miMetodo = function () {...}` y luego usar `miMetodo` en todas las instancias de Array.
- este vicio de programación se denomina parcheo a lo mono (***monkey patching***).
  - hacer parcheo a lo mono pone en riesgo la compatibilidad hacia adelante, porque si el lenguaje agrega este método en el futuro pero con una **firma** diferente, el código cliente no funcionará o lo hará incorrectamente.
  - ha llevado a incidentes como el SmooshGate, y puede ser una gran incoveniento para el avance del lenguaje ya que JavaScript intenta "no romper la web".
- la **única** buena razón para extender un prototipo incorporado es hace utilizables las funciones de los motores de JavaScript más nuevos en otros más antiguos como:

  `Array.prototype.forEach`.

Es de interés recordar que debido a razones históricas, la propiedad `prototype` de algunos constuctores incorporados (built-in) son literales:

- `Number.prototype` es un número 0
- `Array.prototype` es un array vacío,
- `RegExp.prototipo` es `/(?:)/`

```
Number.prototype + 1
// 1

Array.prototype.map((x) => x + 1)
// []

String.prototype + "a"
// "a"

RegExp.prototype.fuente
// "(?:)"

Function.prototype()
// Function.prototype es en sí misma una función que no hace nada
```

Sin embargo, este no es el caso de los constructores definidos por el usuario, ni de los constructores modernos como `Map`.

```
Map.prototype.get(1)
// Uncaught TypeError: get method called on incompatible Map.prototype
```

Construcción de cadenas de herencia más largas

La propiedad *`Constructor`*`.prototype` se convertirá en el `[[Prototype]]` de las instancias creadas por el constructor, tal cual

- incluyendo el propio `[[Prototype]]` de *`Constructor`*`.prototype`.
- por defecto, es un objeto simple, es decir,
  `Object.getPrototypeOf(`*`Constructor`*`.prototype) === Object.prototype`.
- la única excepción es `Object.prototype` en sí, cuyo `[[Prototipo]]` es nulo, es decir
  `Object.getPrototypeOf(Object.prototype) === null`

Por lo tanto, un constructor típico construirá la siguiente **cadena de prototipos**:

```
function Constructor() {}

const obj = new Constructor();
// obj ---> Constructor.prototype ---> Object.prototype ---> null
```

Para construir cadenas de prototipos más largas, podemos establecer el `[[Prototype]]` de `Constructor.prototype` **a través de** `Object.setPrototypeOf()` .

```
function Base() {}
function Derivada() {}

// Configurar el `[[Prototype]]` of `Derivada.prototype` a `Base.prototype`
Object.setPrototypeOf(Derivada.prototype, Base.prototype,);

const obj = new Derivada();
// obj ---> Derivada.prototype ---> Base.prototype ---> Object.prototype ---> null
```

En términos de clase, esto es equivalente a usar la sintaxis `extends`.

```
class Base {}
class Derivada extends Base {}

const obj = new Derivada();
// obj ---> Derivada.prototype ---> Base.prototype ---> Object.prototype ---> null
```

También también te puedes encontrar código legado (antiguo) que usa `Object.create()` para construir la cadena de herencia.

- sin embargo, debido a que esto reasigna la propiedad `prototype` y elimina la propiedad `constructor`, puede ser más propenso a errores, mientras que **las ganancias de rendimiento** pueden no ser evidentes si los constructores aún no han creado ninguna instancia.

```
function Base() {}
function Derivada() {}

// Reasigna a `Derivada.prototype` un nuevo objeto
//  `Base.prototype` como su `[[Prototype]]`
// NO HACER ESTO — para realizar cambios, usar Object.setPrototypeOf en su lugar
Derivada.prototype = Object.create(Base.prototype);
```

Inspección de prototipos: una inmersión más profunda

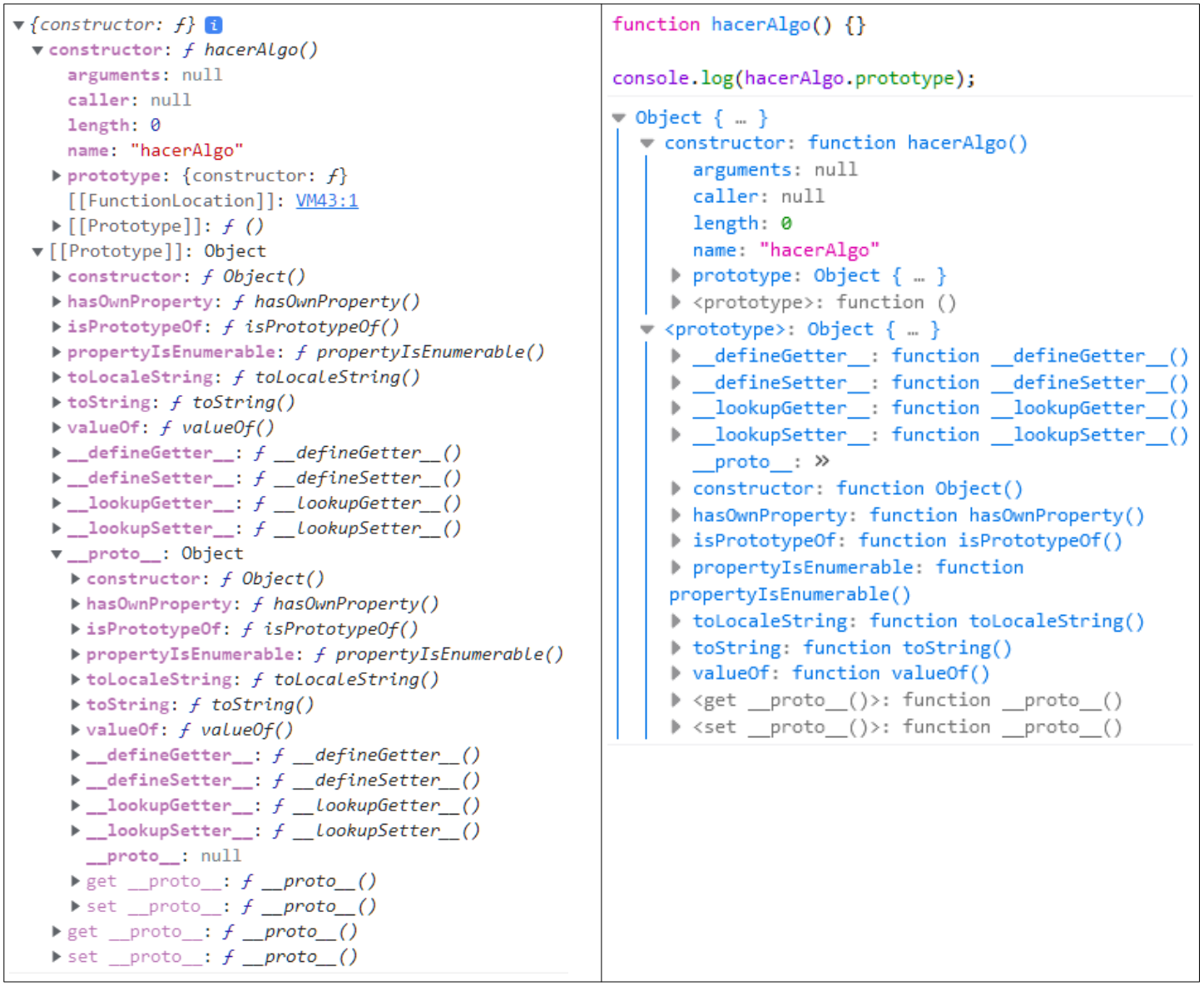
Veamos lo que sucede detrás de escena con un poco más de detalle.

En JavaScript,

- como se mencionó anteriormente, las funciones pueden tener propiedades.
- todas las funciones tienen una propiedad especial llamada `prototype`.
- ten en cuenta que el código que aparece a continuación supone que no hay otro código en el script que pueda interferir

```
function hacerAlgo() {}

console.log(hacerAlgo.prototype);
// {constructor: f}

// It does not matter how you declare the function; a function in JavaScript will
// always have a default prototype property — with one exception:
// una función flecha no tiene por defecto una propiedad default:

const hacerAlgoConFuncionFlecha = () => {};
console.log(hacerAlgoConFuncionFlecha.prototype);
// undefined
```

Como se vio anteriormente, `hacerAlgo()` tiene una propiedad `prototype` predeterminada, como lo demuestra la consola. Después de ejecutar este código, la consola debería haber mostrado un objeto similar a este.



```
hacerAlgoConFuncionFlecha;
() => {}
hacerAlgoConFuncionFlecha.prototype;
undefined
```

```
hacerAlgoConFuncionFlecha;
▼ function hacerAlgoConFuncionFlecha()
    length: 0
    name: "hacerAlgoConFuncionFlecha"
  ▼ <prototype>: function ()
    ▶ apply: function apply()
      arguments: »
    ▶ bind: function bind()
    ▶ call: function call()
      caller: »
    ▶ constructor: function Function()
      length: 0
      name: ""
    ▶ toString: function R()
    ▶ Symbol(Symbol.hasInstance): function Symbol.hasInstance()
    ▶ <get arguments()>: function arguments()
    ▶ <set arguments()>: function arguments()
    ▶ <get caller()>: function caller()
    ▶ <set caller()>: function caller()
    ▶ <prototype>: Object { … }
hacerAlgoConFuncionFlecha.prototype;
```

Podemos agregar propiedades al prototipo de **`hacerAlgo`**(), como se muestra a continuación.

```
function hacerAlgo() {}
hacerAlgo.prototype.foo = 'bar';
console.log(hacerAlgo.prototype);
```

Esto resulta en:

```
hacerAlgo;
ƒ hacerAlgo() {}
hacerAlgo.prototype;
▼ {foo: 'bar', constructor: ƒ}
    foo: "bar"
  ▼ constructor: ƒ hacerAlgo()
      arguments: null
      caller: null
      length: 0
      name: "hacerAlgo"
    ▶ prototype: {foo: 'bar', constructor: ƒ}
      [[FunctionLocation]]: VM21:1
    ▶ [[Prototype]]: ƒ ()
  ▼ [[Prototype]]: Object
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
    ▼ __proto__: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
        __proto__: (...)
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

```
hacerAlgo;
▼ function hacerAlgo()
    arguments: null
    caller: null
    length: 0
    name: "hacerAlgo"
  ▼ prototype: Object { foo: "bar", … }
    ▶ constructor: function hacerAlgo()
      foo: "bar"
    ▶ <prototype>: Object { … }
  ▶ <prototype>: function ()
hacerAlgo.prototype;
▼ Object { foo: "bar", … }
  ▶ constructor: function hacerAlgo()
    foo: "bar"
  ▼ <prototype>: Object { … }
    ▶ __defineGetter__: function __defineGetter__()
    ▶ __defineSetter__: function __defineSetter__()
    ▶ __lookupGetter__: function __lookupGetter__()
    ▶ __lookupSetter__: function __lookupSetter__()
    ▼ __proto__: Object { … }
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
        __proto__: null
      ▶ constructor: function Object()
      ▶ hasOwnProperty: function hasOwnProperty()
      ▶ isPrototypeOf: function isPrototypeOf()
      ▶ propertyIsEnumerable: function propertyIsEnumerable()
      ▶ toLocaleString: function toLocaleString()
      ▶ toString: function toString()
      ▶ valueOf: function valueOf()
      ▶ <get __proto__()>: function __proto__()
      ▶ <set __proto__()>: function __proto__()
    ▶ constructor: function Object()
    ▶ hasOwnProperty: function hasOwnProperty()
    ▶ isPrototypeOf: function isPrototypeOf()
    ▶ propertyIsEnumerable: function propertyIsEnumerable()
    ▶ toLocaleString: function toLocaleString()
    ▶ toString: function toString()
    ▶ valueOf: function valueOf()
    ▶ <get __proto__()>: function __proto__()
    ▶ <set __proto__()>: function __proto__()
```

Ahora podemos usar el operador **new** para crear una instancia de **hacerAlgo()** basada en este prototipo.

- para usar el operador **new**, invocamos la función prototipo precedida del operador **new**
- llamar a una función con el operador **new** devuelve un objeto que es una instancia de la función.
- luego se pueden agregar propiedades a este objeto.

Prueba el siguiente código:

```
function hacerAlgo() {}

hacerAlgo.prototype.foo = 'bar'; // agrega una propiedad al prototipo
```

```
const hacerAlgunaInstancia = new hacerAlgo();
hacerAlgunaInstancia.prop = 'algún valor'; // agrega una propiedad al objeto
console.log(hacerAlgunaInstancia);
```

Esto da como resultado una salida similar a la siguiente:

```
hacerAlgunaInstancia;
▼ hacerAlgo {prop: 'algún valor'}
    prop: "algún valor"
  ▼ [[Prototype]]: Object
      foo: "bar"
    ▶ constructor: ƒ hacerAlgo()
    ▼ [[Prototype]]: Object
      ▶ constructor: ƒ Object()
      ▶ hasOwnProperty: ƒ hasOwnProperty()
      ▶ isPrototypeOf: ƒ isPrototypeOf()
      ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶ toLocaleString: ƒ toLocaleString()
      ▶ toString: ƒ toString()
      ▶ valueOf: ƒ valueOf()
      ▶ __defineGetter__: ƒ __defineGetter__()
      ▶ __defineSetter__: ƒ __defineSetter__()
      ▶ __lookupGetter__: ƒ __lookupGetter__()
      ▶ __lookupSetter__: ƒ __lookupSetter__()
      ▼ __proto__: Object
          foo: "bar"
        ▼ constructor: ƒ hacerAlgo()
            arguments: null
            caller: null
            length: 0
            name: "hacerAlgo"
          ▶ prototype: {foo: 'bar', constructor: ƒ}
            [[FunctionLocation]]: VM19:1
          ▼ [[Prototype]]: ƒ ()
            ▶ apply: ƒ apply()
              arguments: (...)
            ▶ bind: ƒ bind()
            ▶ call: ƒ call()
              caller: (...)
            ▶ constructor: ƒ Function()
              length: 0
              name: ""
            ▶ toString: ƒ toString()
            ▶ Symbol(Symbol.hasInstance): ƒ [Symbol.hasInstance]()
            ▶ get arguments: ƒ ()
            ▶ set arguments: ƒ ()
            ▶ get caller: ƒ ()
            ▶ set caller: ƒ ()
              [[FunctionLocation]]:
            ▶ [[Prototype]]: Object
        ▼ [[Prototype]]: Object
          ▶ constructor: ƒ Object()
          ▶ hasOwnProperty: ƒ hasOwnProperty()
          ▶ isPrototypeOf: ƒ isPrototypeOf()
          ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
          ▶ toLocaleString: ƒ toLocaleString()
          ▶ toString: ƒ toString()
          ▶ valueOf: ƒ valueOf()
          ▶ __defineGetter__: ƒ __defineGetter__()
          ▶ __defineSetter__: ƒ __defineSetter__()
          ▶ __lookupGetter__: ƒ __lookupGetter__()
          ▶ __lookupSetter__: ƒ __lookupSetter__()
          ▼ __proto__: Object
            ▶ constructor: ƒ Object()
            ▶ hasOwnProperty: ƒ hasOwnProperty()
            ▶ isPrototypeOf: ƒ isPrototypeOf()
            ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
            ▶ toLocaleString: ƒ toLocaleString()
            ▶ toString: ƒ toString()
            ▶ valueOf: ƒ valueOf()
            ▶ __defineGetter__: ƒ __defineGetter__()
            ▶ __defineSetter__: ƒ __defineSetter__()
            ▶ __lookupGetter__: ƒ __lookupGetter__()
            ▶ __lookupSetter__: ƒ __lookupSetter__()
              __proto__: null
            ▶ get __proto__: ƒ __proto__()
            ▶ set __proto__: ƒ __proto__()
          ▶ get __proto__: ƒ __proto__()
          ▶ set __proto__: ƒ __proto__()
      ▶ get __proto__: ƒ __proto__()
      ▶ set __proto__: ƒ __proto__()
```

```
hacerAlgunaInstancia;
▼ Object { prop: "algún valor" }
    prop: "algún valor"
  ▼ <prototype>: Object { foo: "bar", … }
    ▼ constructor: function hacerAlgo()
        arguments: null
        caller: null
        length: 0
        name: "hacerAlgo"
      ▼ prototype: Object { foo: "bar", … }
        ▶ constructor: function hacerAlgo()
          foo: "bar"
        ▶ <prototype>: Object { … }
      ▶ <prototype>: function ()
      foo: "bar"
    ▼ <prototype>: Object { … }
      ▶ __defineGetter__: function __defineGetter__()
      ▶ __defineSetter__: function __defineSetter__()
      ▶ __lookupGetter__: function __lookupGetter__()
      ▶ __lookupSetter__: function __lookupSetter__()
      ▼ __proto__: Object { foo: "bar", … }
        ▶ constructor: function hacerAlgo()
          foo: "bar"
        ▼ <prototype>: Object { … }
          ▶ __defineGetter__: function __defineGetter__()
          ▶ __defineSetter__: function __defineSetter__()
          ▶ __lookupGetter__: function __lookupGetter__()
          ▶ __lookupSetter__: function __lookupSetter__()
          ▼ __proto__: Object { … }
            ▶ __defineGetter__: function __defineGetter__()
            ▶ __defineSetter__: function __defineSetter__()
            ▶ __lookupGetter__: function __lookupGetter__()
            ▶ __lookupSetter__: function __lookupSetter__()
              __proto__: null
            ▶ constructor: function Object()
            ▶ hasOwnProperty: function hasOwnProperty()
            ▶ isPrototypeOf: function isPrototypeOf()
            ▶ propertyIsEnumerable: function propertyIsEnumerable()
            ▶ toLocaleString: function toLocaleString()
            ▶ toString: function toString()
            ▶ valueOf: function valueOf()
            ▶ <get __proto__()>: function __proto__()
            ▶ <set __proto__()>: function __proto__()
          ▶ constructor: function Object()
          ▶ hasOwnProperty: function hasOwnProperty()
          ▶ isPrototypeOf: function isPrototypeOf()
          ▶ propertyIsEnumerable: function propertyIsEnumerable()
          ▶ toLocaleString: function toLocaleString()
          ▶ toString: function toString()
          ▶ valueOf: function valueOf()
          ▶ <get __proto__()>: function __proto__()
          ▶ <set __proto__()>: function __proto__()
      ▶ constructor: function Object()
      ▶ hasOwnProperty: function hasOwnProperty()
      ▶ isPrototypeOf: function isPrototypeOf()
      ▶ propertyIsEnumerable: function propertyIsEnumerable()
      ▶ toLocaleString: function toLocaleString()
      ▶ toString: function toString()
      ▶ valueOf: function valueOf()
      ▶ <get __proto__()>: function __proto__()
      ▶ <set __proto__()>: function __proto__()
```

Como se vio anteriormente, el `[[Prototype]]` de `hacerAlgunaInstancia` es `hacerAlgo.prototype`.

Pero, ¿qué hace esto?

- cuando accede a una propiedad de `hacerAlgunaInstancia`, el **entorno de ejecución** primero busca si `hacerAlgunaInstancia` tiene esa propiedad.
- si `hacerAlgunaInstancia` no tiene la propiedad, entonces el entorno de ejecución busca la propiedad en `hacerAlgunaInstancia.[[Prototype]]` (que es igual a `hacerAlgo.prototype`).
  - si tiene la propiedad que se busca, entonces se usa esa propiedad en `hacerAlgunaInstancia.[[Prototype]]`
  - de lo contrario, si no tiene la propiedad, se **comprueba si** `hacerAlgunaInstancia.[[Prototype]].[[Prototype]]` tiene la propiedad.
  - así hará sucesivamente hasta encontrar la propiedad o llegar a `Object.prototype.[[Prototype]]` que es **`null`**
  - luego, y solo entonces, después de revisar toda la cadena de prototipos de `[[Prototype]]`, el entorno de ejecución afirma que la propiedad no existe y concluye que el valor de la propiedad no está definido.

Intentemos ingresar más código en la consola:

```
function hacerAlgo() {}
hacerAlgo.prototype.foo = 'bar';

const hacerAlgunaInstancia = new hacerAlgo();
hacerAlgunaInstancia.prop = 'algún valor';
console.log('hacerAlgunaInstancia.prop:      ', hacerAlgunaInstancia.prop);
console.log('hacerAlgunaInstancia.foo:       ', hacerAlgunaInstancia.foo);
console.log('hacerAlgo.prop:                  ', hacerAlgo.prop);
console.log('hacerAlgo.foo:                   ', hacerAlgo.foo);
console.log('hacerAlgo.prototype.prop:        ', hacerAlgo.prototype.prop);
console.log('hacerAlgo.prototype.foo:         ', hacerAlgo.prototype.foo);
```

El resultado de lo anterior es:

```
hacerAlgunaInstancia.prop:      algún valor
hacerAlgunaInstancia.foo:       bar
hacerAlgo.prop:                 undefined
hacerAlgo.foo:                  undefined
hacerAlgo.prototype.prop:       undefined
hacerAlgo.prototype.foo:        bar
```

Distintas formas de crear y mutar cadenas de prototipos

Hemos encontrado muchas formas de crear objetos y cambiar sus cadenas de prototipos: vamos a resumir sistemáticamente las diferentes formas, comparando los pros y los contras de cada enfoque.

Objetos creados con construcciones de sintaxis

```
const o = {a: 1};
// El objeto recién creado o tiene Object.prototype como su [[Prototype]]
// Object.prototype tiene null como su prototipo.
// o ---> Object.prototype --->  null

const b = ['yo', 'que pasa', '?'];
// Los arreglos heredan de Array.prototype
// (que tiene métodos indexOf, forEach, etc.)
// La cadena de prototipo se parece a:
// b ---> Array.prototype ---> Object.prototype ---> null

function f() {
  return 2;
}
// Las funciones heredan de Function.prototype
// (que tiene métodos call, bind, etc.)
// f ---> Function.prototype ---> Object.prototype ---> null

const p = { b: 2, __proto__: o };
// Es posible apuntar el [[Prototype]] del objeto recién creado a
// otro objeto a través de la propiedad literal __proto__.
// (No confundir lo anterior con accesores get/set Object.prototype.__proto__)
// p ---> o ---> Object.prototype ---> null
```

Pros y contras de usar la clave **`__proto__` en** inicializadores objeto

Pros	Compatible con todos los motores modernos. Apuntar la clave **`__proto__`** a algo que no es un objeto solo falla silenciosamente sin generar una excepción. Contrariamente a la **`Object.prototype.__proto__`** setter, __proto__ en los inicializadores de objetos literales está estandarizado y optimizado, e incluso puede tener más rendimiento que **`Object.create`**. Declarar propiedades propias adicionales al crear un objeto es más ergonómico que **`Object.create`**.
Contras	No compatible con IE10 y versiones anteriores. Es probable que se confunda con métodos accesores de **`Object.prototype.__proto__`** para personas que desconocen la diferencia.

Con funciones constructoras

```
function Grafico() {
  this.vertices = [];
  this.aristas  = [];
}

Grafico.prototype.anexarVertice = function (v) {
  this.vertices.push(v);
}

const g = new Grafico();
// g es un objeto con propiedades propias vertices y aristas
// g.[[Prototype]] es lo mismo que Grafico.prototype when new Grafico() is executed.
```

Pros y contras de usar funciones constructoras

Pros	Compatible con todos los motores, desde IE 5.5. Además, es muy rápido, muy estándar y muy optimizable para JIT.
Contras	Para utilizar este método, la función en cuestión debe estar inicializada. • durante esta inicialización, el constructor puede almacenar información única que debe generarse por parte del objeto. • esta información única solo se generaría una vez, lo que podría generar problemas. • la inicialización del constructor puede colocar métodos no deseados en el objeto. En la práctica, lo anterior no suele ser un problema.

Con `Object.create()`

Invocar `Object.create()` crea un nuevo objeto.

El `[[Prototype]]` de este objeto es el primer argumento de la función:

```
const a = { a: 1 };
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(ba); // 1 (heredado)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);
// d ---> null (d es un objeto que tiene null directamente como su prototipo)
console.log(d.hasOwnProperty());
// indefinido, porque d no hereda de Object.prototype
```

Pros y contras de `Object.create`

Pros	Compatible con todos los motores modernos. Permite configurar directamente el `[[Prototype]]` de un objeto en el momento de la creación, lo que permite que el tiempo de ejecución optimice aún más el objeto. También permite la creación de objetos sin `prototype`, usando `Object.create(null)`
Contras	No compatible con IE8 y versiones anteriores. Sin embargo, como Microsoft suspendió el soporte extendido para los sistemas que ejecutan IE8 y versiones anteriores, eso no debería ser una preocupación para la mayoría de las aplicaciones. Además, la inicialización lenta del objeto puede ser un **agujero negro en el rendimiento** si se usa el segundo argumento, porque cada **propiedad del descriptor de objeto** tiene su propio objeto descriptor independiente. Cuando se trata de cientos de miles de descriptores de objetos en forma de objetos, ese tiempo de retraso (***lag***) puede convertirse en un problema serio.

Con clases

```
class Poligono {
  constructor(altura, ancho) {
    this.altura = altura;
    this.ancho = ancho;
  }
}

class Cuadrado extends Poligono {
  constructor(longitudLado) {
    super(longitudLado, longitudLado);
  }

  get area() {
    return this.altura * this.ancho;
  }

  set longitudLado(nuevaLongitud) {
    this.altura = nuevaLongitud;
    this.ancho = nuevaLongitud;
  }
}

const cuadrado = new Cuadrado(2);
// cuadrado ---> Cuadrado.prototype ---> Poligono.prototype ---> Object.prototype ---> null
```

Pros y contras de las clases.

Pros	Compatible con todos los motores modernos. Muy alta **legibilidad** y **mantenibilidad**. Las propiedades privadas son una característica que no tiene un reemplazo trivial en la herencia prototípica.
Contras	Las clases, especialmente con propiedades privadas, están menos optimizadas que las tradicionales (aunque los implementadores del motor están trabajando para mejorar esto). No se admite en entornos más antiguos y, por lo general, se necesitan **transpiladores** para usar clases en producción.

Con `Objeto.setPrototypeOf()`

Si bien todos los métodos anteriores establecerán la cadena de prototipos en el momento de la creación del objeto, `Object.setPrototypeOf()` permite mutar la propiedad interna `[[Prototype]]` de un objeto existente.

```
const obj = { a: 1 };
const otroObj = { b: 2 };
Object.setPrototypeOf(obj, otroObj);
// obj ---> otroObj ---> Object.prototype ---> null
```

Pros y contras de `Object.setPrototypeOf`

Pros	Compatible con todos los motores modernos. Permite la **manipulación dinámica** del prototipo de un objeto e incluso puede forzar un prototipo en un objeto sin prototipo creado con `Object.create(null)`
Contras	Rendimiento malo: debe evitarse si es posible configurar el prototipo en el momento de la creación del objeto. Muchos motores optimizan el prototipo e intentan adivinar la ubicación del método en la memoria cuando llaman a una instancia por adelantado: pero establecer el prototipo de forma dinámica causa problemas para todas esas optimizaciones. Puede hacer que algunos motores vuelvan a compilar tu código para desoptimizarlo, para que funcione de acuerdo con las especificaciones. No compatible con IE8 y versiones anteriores.

Con el descriptor accesor `__proto__`

Todos los objetos heredan el setter `Object.prototype.__proto__`, que se puede usar para establecer el `[[Prototype]]` de un objeto existente (si la clave `__proto__` no se anula en el objeto).

Advertencia:
`Los accesores Object.prototype.__proto__`, **no** son **estándar** y están en desuso.
Casi siempre deberías usar `Object.setPrototypeOf en su` lugar.

```
const obj = {};
// NO USAR ESTE SISTEMA: es un ejemplo sólo para ilustrar el problema.
obj.__proto__ = { barProp: 'bar val' };
obj.__proto__.__proto__ = { fooProp: 'foo val' };
console.log(obj.fooProp);
console.log(obj.barProp);
```

Pros y contras de establecer la propiedad `__proto__`

Pros	Compatible con todos los motores modernos. Ajuste __proto__ a algo que no es un objeto solo falla silenciosamente. No lanza una excepción.
Contras	Sin rendimiento y en desuso. Muchos motores optimizan el prototipo e intentan adivinar la ubicación del método en la memoria cuando llaman a una instancia por adelantado; pero establecer el prototipo interrumpe dinámicamente todas esas optimizaciones e incluso puede obligar a algunos motores a volver a compilar para desoptimizar su código, para que funcione de acuerdo con las especificaciones. No compatible con IE10 y versiones anteriores. Él `__proto__` setter es normativamente opcional, por lo que es posible que no funcione en todas las plataformas. Casi siempre debería usar `Object.setPrototypeOf en su` lugar.

```
obj;
▼{} ℹ
  ▼[[Prototype]]: Object
     barProp: "bar val"
    ▼[[Prototype]]: Object
       fooProp: "foo val"
      ▼[[Prototype]]: Object
        ▶constructor: ƒ Object()
        ▶hasOwnProperty: ƒ hasOwnProperty()
        ▶isPrototypeOf: ƒ isPrototypeOf()
        ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
        ▶toLocaleString: ƒ toLocaleString()
        ▶toString: ƒ toString()
        ▶valueOf: ƒ valueOf()
        ▶__defineGetter__: ƒ __defineGetter__()
        ▶__defineSetter__: ƒ __defineSetter__()
        ▶__lookupGetter__: ƒ __lookupGetter__()
        ▶__lookupSetter__: ƒ __lookupSetter__()
        ▼__proto__: Object
           barProp: "bar val"
          ▼[[Prototype]]: Object
             fooProp: "foo val"
            ▼[[Prototype]]: Object
              ▶constructor: ƒ Object()
              ▶hasOwnProperty: ƒ hasOwnProperty()
              ▶isPrototypeOf: ƒ isPrototypeOf()
              ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
              ▶toLocaleString: ƒ toLocaleString()
              ▶toString: ƒ toString()
              ▶valueOf: ƒ valueOf()
              ▶__defineGetter__: ƒ __defineGetter__()
              ▶__defineSetter__: ƒ __defineSetter__()
              ▶__lookupGetter__: ƒ __lookupGetter__()
              ▶__lookupSetter__: ƒ __lookupSetter__()
              ▼__proto__: Object
                 fooProp: "foo val"
                ▼[[Prototype]]: Object
                  ▶constructor: ƒ Object()
                  ▶hasOwnProperty: ƒ hasOwnProperty()
                  ▶isPrototypeOf: ƒ isPrototypeOf()
                  ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
                  ▶toLocaleString: ƒ toLocaleString()
                  ▶toString: ƒ toString()
                  ▶valueOf: ƒ valueOf()
                  ▶__defineGetter__: ƒ __defineGetter__()
                  ▶__defineSetter__: ƒ __defineSetter__()
                  ▶__lookupGetter__: ƒ __lookupGetter__()
                  ▶__lookupSetter__: ƒ __lookupSetter__()
                  ▼__proto__: Object
                    ▶constructor: ƒ Object()
                    ▶hasOwnProperty: ƒ hasOwnProperty()
                    ▶isPrototypeOf: ƒ isPrototypeOf()
                    ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
                    ▶toLocaleString: ƒ toLocaleString()
                    ▶toString: ƒ toString()
                    ▶valueOf: ƒ valueOf()
                    ▶__defineGetter__: ƒ __defineGetter__()
                    ▶__defineSetter__: ƒ __defineSetter__()
                    ▶__lookupGetter__: ƒ __lookupGetter__()
                    ▶__lookupSetter__: ƒ __lookupSetter__()
                     __proto__: null
                    ▶get __proto__: ƒ __proto__()
                    ▶set __proto__: ƒ __proto__()
                  ▶get __proto__: ƒ __proto__()
                  ▶set __proto__: ƒ __proto__()
              ▶get __proto__: ƒ __proto__()
              ▶set __proto__: ƒ __proto__()
        ▶get __proto__: ƒ __proto__()
        ▶set __proto__: ƒ __proto__()
```

```
obj;
▼Object {  }
  ▼<prototype>: Object { barProp: "bar val" }
     barProp: "bar val"
    ▼<prototype>: Object { fooProp: "foo val" }
       fooProp: "foo val"
      ▼<prototype>: Object { … }
        ▶__defineGetter__: function __defineGetter__()
        ▶__defineSetter__: function __defineSetter__()
        ▶__lookupGetter__: function __lookupGetter__()
        ▶__lookupSetter__: function __lookupSetter__()
        ▼__proto__: Object { barProp: "bar val" }
           barProp: "bar val"
          ▼<prototype>: Object { fooProp: "foo val" }
             fooProp: "foo val"
            ▼<prototype>: Object { … }
              ▶__defineGetter__: function __defineGetter__()
              ▶__defineSetter__: function __defineSetter__()
              ▶__lookupGetter__: function __lookupGetter__()
              ▶__lookupSetter__: function __lookupSetter__()
              ▼__proto__: Object { fooProp: "foo val" }
                 fooProp: "foo val"
                ▼<prototype>: Object { … }
                  ▶__defineGetter__: function __defineGetter__()
                  ▶__defineSetter__: function __defineSetter__()
                  ▶__lookupGetter__: function __lookupGetter__()
                  ▶__lookupSetter__: function __lookupSetter__()
                  ▼__proto__: Object { … }
                    ▶__defineGetter__: function __defineGetter__()
                    ▶__defineSetter__: function __defineSetter__()
                    ▶__lookupGetter__: function __lookupGetter__()
                    ▶__lookupSetter__: function __lookupSetter__()
                     __proto__: null
                    ▶constructor: function Object()
                    ▶hasOwnProperty: function hasOwnProperty()
                    ▶isPrototypeOf: function isPrototypeOf()
                    ▶propertyIsEnumerable: function propertyIsEnumerable()
                    ▶toLocaleString: function toLocaleString()
                    ▶toString: function toString()
                    ▶valueOf: function valueOf()
                    ▶<get __proto__()>: function __proto__()
                    ▶<set __proto__()>: function __proto__()
                  ▶constructor: function Object()
                  ▶hasOwnProperty: function hasOwnProperty()
                  ▶isPrototypeOf: function isPrototypeOf()
                  ▶propertyIsEnumerable: function propertyIsEnumerable()
                  ▶toLocaleString: function toLocaleString()
                  ▶toString: function toString()
                  ▶valueOf: function valueOf()
                  ▶<get __proto__()>: function __proto__()
                  ▶<set __proto__()>: function __proto__()
              ▶constructor: function Object()
              ▶hasOwnProperty: function hasOwnProperty()
              ▶isPrototypeOf: function isPrototypeOf()
              ▶propertyIsEnumerable: function propertyIsEnumerable()
              ▶toLocaleString: function toLocaleString()
              ▶toString: function toString()
              ▶valueOf: function valueOf()
              ▶<get __proto__()>: function __proto__()
              ▶<set __proto__()>: function __proto__()
        ▶constructor: function Object()
        ▶hasOwnProperty: function hasOwnProperty()
        ▶isPrototypeOf: function isPrototypeOf()
        ▶propertyIsEnumerable: function propertyIsEnumerable()
        ▶toLocaleString: function toLocaleString()
        ▶toString: function toString()
        ▶valueOf: function valueOf()
        ▶<get __proto__()>: function __proto__()
        ▶<set __proto__()>: function __proto__()
```

Rendimiento

El **tiempo de búsqueda** de las propiedades que ocupan un lugar en la parte de arriba de la cadena de prototipos

- puede tener un impacto negativo en el rendimiento,
- esto puede ser significativo en el código donde el **rendimiento** es crítico.
- además, intentar acceder a propiedades inexistentes siempre atravesará la cadena completa del prototipo.

Además, al iterar sobre las propiedades de un objeto, se enumerarará cada **propiedad enumerable** que estea en la **cadena de prototipos**.

- para verificar si un objeto tiene una propiedad definida en sí mismo y no en algún lugar de su cadena de prototipos, es necesario usar los métodos `hasOwnProperty` u `Object.hasOwn`.
- todos los objetos, excepto aquellos con `null` como `[[Prototype]]`, heredan `hasOwnProperty` de `Object.prototype`, a menos que se haya sobreescrito por encima del objeto actual en la cadena de prototipos.

Para darte un ejemplo concreto, tomemos el código de ejemplo del gráfico anterior para ilustrarlo:

```
console.log(g.hasOwnProperty('vertices'));
// true

console.log(Object.hasOwn(g, 'vertices'));
// true

console.log(g.hasOwnProperty('no'));
// false

console.log(Object.hasOwn(g, 'no'));
// false

console.log(g.hasOwnProperty('anexarVertice'));
// false

console.log(Object.hasOwn(g, 'anexarVertice'));
// false

console.log(Object.getPrototypeOf(g).hasOwnProperty('anexarVertice'));
// true
```

Nota: no basta con comprobar si una propiedad no está definida; es muy posible que la propiedad exista, pero su valor simplemente se establece en `undefined`.

Conclusión

JavaScript puede ser un poco confuso para los desarrolladores que vienen de Java o C++, ya que

- todo es dinámico, todo en tiempo de ejecución y no tiene ningún tipo estático.
- todo es un objeto (instancia) o una función (constructor), e incluso las propias funciones son instancias del constructor de funciones.
- Incluso las "clases" como construcciones de sintaxis son solo funciones de constructor en tiempo de ejecución.

Todas las **funciones constructoras** en JavaScript tienen una propiedad especial llamada `prototype`, que funciona con el operador `new`.

- la referencia al objeto prototipo se copia en la propiedad interna `[[Prototype]]` de la nueva instancia.
- por ejemplo, cuando escribes `const a1 = new A()`, JavaScript
  - después de crear el objeto en la memoria y antes de ejecutar la función `A()` con `this` apuntando al objeto recien creado
  - establece `a1.[[Prototype]] = A.prototype`.

- cuando accedes a las propiedades de la instancia, JavaScript
  - primero verifica si existen en ese objeto directamente, y si no, busca en `[[Prototype]]`
  - `[[Prototype]]` se mira recursivamente, es decir
    - `a1.hacerAlgo`
    - `Object.getPrototypeOf(a1).hacerAlgo`
    - `Object.getPrototypeOf(Object.getPrototypeOf(a1)).hacerAlgo`
    - etc.,

    hasta que se encuentra la propiedad o `Object.getPrototypeOf` devuelve `null`.

- esto significa que todas las **instancias** comparten efectivamente todas las propiedades definidas en el prototipo, e incluso puedes cambiar más tarde partes del `prototype` y hacer que los cambios aparezcan en todas las instancias existentes.
  - Si, en el ejemplo anterior, haces

```
const a1 = new A();
const a2 = new A();
```

entonces `a1.hacerAlgo` realmente haría referencia a `Object.getPrototypeOf(a1).hacerAlgo`: que el lo mismo que el `A.prototype.hacerAlgo` que definiste; es decir:

```
Object.getPrototypeOf(a1).hacerAlgo ===  Object.getPrototypeOf(a2).hacerAlgo
                                    ===  A.prototype.hacerAlgo
```

Es esencial:

- comprender el **modelo de herencia prototípico** antes de escribir código complejo que haga uso de él.
- además, ten en cuenta la longitud de las **cadenas de prototipos** en tu código y divídelas si es necesario para evitar posibles problemas de rendimiento.
- Además, los **prototipos nativos** **nunca** deben ampliarse a menos que sea por motivos de compatibilidad con las funciones de JavaScript más nuevas.