JavaScript from ES5 to ESNext

FLAVIO COPES

Tabla de contenido

- Propiedades de reposo/extensión
- Iteración asíncrona
- Promesa.prototipo.finalmente()
- Mejoras en expresiones regulares

ESSiguiente

- Array.prototype.{flat,flatMap}
- Encuadernación de captura opcional
- Objeto.fromEntries()
- Cadena.prototipo.{trimStart,trimEnd}
- Símbolo.prototipo.descripción
- Mejoras JSON
- JSON bien formado.stringify()
- Función.prototipo.toString()

Prefacio

¡Bienvenidos!

Escribí este libro para ayudarlo a pasar del conocimiento de JavaScript anterior a ES6 y ponerlo rápidamente al día con los avances más recientes del lenguaje.

JavaScript hoy está en la posición privilegiada de ser el único lenguaje que puede ejecutarse de forma nativa en el navegador, y está altamente integrado y optimizado para eso.

El futuro de JavaScript va a ser brillante. Mantenerse al día con los cambios no debería ser más difícil de lo que ya es, y mi objetivo aquí es brindarle una descripción general rápida pero completa.
de las cosas nuevas disponibles para nosotros.

Gracias por recibir este libro electrónico. Espero que su contenido te ayude a conseguir lo que quieres.

flavio

Puede comunicarse conmigo por correo electrónico a flavio@flaviocopes.com, en Twitter @flaviocopes.

Mi sitio web es flaviocopes.com.

Introducción a ECMAScript

Cada vez que lea sobre JavaScript, inevitablemente verá uno de estos términos:

- ES3
- ES5
- ES6
- ES7
- ES8
- ES2015
- ES2016
- ES2017
- ECMAScript 2017
- ECMAScript 2016
- ECMAScript 2015

¿Qué quieren decir?

Todos se refieren a un **estándar,** llamado ECMAScript.

ECMAScript es **el estándar en el que se basa JavaScript** y, a menudo, se abrevia como **ES.**

Además de JavaScript, otros lenguajes implementan (ed) ECMAScript, que incluyen:

- *ActionScript* (el lenguaje de secuencias de comandos de Flash), que está perdiendo popularidad desde que Flash se discontinuará oficialmente en 2020
- *JScript* (el dialecto de secuencias de comandos de Microsoft), dado que en ese momento JavaScript solo era compatible con Netscape y la guerra de los navegadores estaba en su apogeo, Microsoft tuvo que crear su propia versión para Internet Explorer.

pero, por supuesto, JavaScript es la implementación de ES **más popular** y ampliamente utilizada.

¿Por qué este extraño nombre? Ecma International es una asociación suiza de estándares que se encarga de definir los estándares internacionales.

Cuando se creó JavaScript, fue presentado por Netscape y Sun Microsystems a Ecma y le dieron el nombre ECMA-262 alias **ECMAScript.**

Este comunicado de prensa de Netscape y Sun Microsystems (el creador de Java) podría ayudar a determinar la elección del nombre, que podría incluir problemas legales y de marca por parte de Microsoft, que estaba en el comité, según Wikipedia.

Después de IE9, Microsoft dejó de calificar su compatibilidad con ES en los navegadores como JScript y comenzó a llamarlo JavaScript (al menos, ya no pude encontrar referencias a él)

Entonces, a partir de 201x, el único lenguaje popular que admite la especificación ECMAScript es JavaScript.

Versión actual de ECMAScript

La versión actual de ECMAScript es **ES2018.**

Fue lanzado en junio de 2018.

¿Qué es TC39?

TC39 es el comité que desarrolla JavaScript.

Los miembros de TC39 son empresas involucradas en JavaScript y proveedores de navegadores, incluidos Mozilla, Google, Facebook, Apple, Microsoft, Intel, PayPal, SalesForce y otros.

Toda propuesta de versión estándar debe pasar por varias etapas, las cuales se explican aquí.

ES Versions

Me resultó desconcertante por qué a veces se hace referencia a una versión ES por número de edición y a veces por año, y estoy confundido por el año por casualidad siendo -1 en el número, que se suma a la confusión general sobre JS/ES ÿÿ

Antes de ES2015, las especificaciones de ECMAScript se llamaban comúnmente por su edición. Entonces ES5 es el nombre oficial de la actualización de la especificación ECMAScript publicada en 2009.

¿Por qué pasó esto? Durante el proceso que condujo a ES2015, se cambió el nombre de ES6 a ES2015, pero como esto se hizo tarde, la gente aún lo mencionaba como ES6, y el comunidad no ha dejado atrás el nombre de la edición: *el mundo todavía está llamando a los lanzamientos de ES por número de edición.*

Esta tabla debería aclarar un poco las cosas:

Edición	Nombre oficial	Fecha de publicación
ES9	ES2018	junio 2018
ES8	ES2017	junio 2017
ES7	ES2016	junio 2016
ES6	ES2015	junio 2015
ES5.1	ES5.1	junio de 2011
ES5	ES5	diciembre 2009
ES4	ES4	Abandonado
ES3	ES3	diciembre de 1999
ES2	ES2	junio de 1998
ES1	ES1	junio de 1997

Profundicemos en las funciones específicas agregadas a JavaScript desde ES5.

ES2015

let y const

Hasta ES2015, var era la única construcción disponible para definir variables.

```
var a = 0
```

Si olvida agregar var , estará asignando un valor a una variable no declarada y los resultados pueden variar.

En entornos modernos, con el modo estricto habilitado, obtendrá un error. En entornos más antiguos (o con el modo estricto deshabilitado), esto inicializará la variable y la asignará al objeto global.

Si no inicializa la variable cuando la declara, tendrá el valor indefinido hasta que asignarle un valor.

```
var a //tipo de a === 'indefinido'
```

Puede volver a declarar la variable muchas veces, anulándola:

```
var a = 1
var a = 2
```

También puede declarar múltiples variables a la vez en la misma declaración:

```
var a = 1, b = 2
```

El **alcance** es la parte del código donde la variable es visible.

Una variable inicializada con var fuera de cualquier función se asigna al objeto global, tiene un alcance global y es visible en todas partes. Una variable inicializada con var dentro de una función se asigna a esa función, es local y solo es visible dentro de ella, como un parámetro de función.

Cualquier variable definida en una función con el mismo nombre que una variable global tiene prioridad sobre la variable global, eclipsándola.

Es importante entender que un bloque (identificado por un par de llaves) no define un nuevo ámbito. Solo se crea un nuevo alcance cuando se crea una función, porque var no tiene alcance de bloque, sino alcance de función.

Dentro de una función, cualquier variable definida en ella es visible en todo el código de la función, incluso si la variable se declara al final de la función, aún se puede hacer referencia al principio, porque JavaScript antes de ejecutar el código en realidad *mueve todas las variables en la parte superior .* (alguna cosa

eso se llama **izar).** Para evitar confusiones, declare siempre las variables al comienzo de una función.

usando let

`let` es una nueva característica introducida en ES2015 y es esencialmente una versión de ámbito de `bloque` de var . Su alcance se limita al bloque, sentencia o expresión donde se define, y todos los contenía bloques interiores.

Los desarrolladores modernos de JavaScript pueden optar por usar solo `let` y descartar por completo el uso de `var` .

> Si `let` parece un término oscuro, simplemente lea `let color = 'red'` como *let the color be red* y todo tiene mucho mas sentido

Definir `let` fuera de cualquier función, al contrario de var , no crea una variable global.

usando const

Las variables declaradas con `var` o `let` pueden cambiarse más adelante en el programa y reasignarse.
Una vez que se inicializa una `const` , su valor nunca se puede cambiar de nuevo y no se puede reasignar a un valor diferente.

```
constante a = 'prueba'
```

No podemos asignar un literal diferente a a const. Sin embargo, podemos mutar a si es un objeto que proporciona métodos que mutan su contenido.

`const` no proporciona inmutabilidad, solo se asegura de que la referencia no se pueda cambiar.

`const` tiene alcance de bloque, igual que `let` .

Los desarrolladores modernos de JavaScript pueden optar por usar siempre `const` para las variables que no necesitan reasignarse más adelante en el programa, porque siempre debemos usar la construcción más simple disponible para evitar cometer errores en el futuro.

Funciones de flecha

Las funciones de flecha, desde su introducción, cambiaron para siempre la apariencia (y el funcionamiento) del código JavaScript.

En mi opinión, este cambio fue tan acogedor que ahora rara vez se ve el uso de la `function` Palabra clave de función en bases de código modernas. Aunque eso todavía tiene su uso.

Visualmente, es un cambio simple y bienvenido, que le permite escribir funciones con una sintaxis más corta, desde:

```
const miFuncion = function() {
  //...
}
```

a

```
const miFuncion = () => {
  //...
}
```

Si el cuerpo de la función contiene solo una declaración, puede omitir los corchetes y escribir todo en una sola línea:

```
const miFuncion = () => hacerAlgo()
```

Los parámetros se pasan entre paréntesis:

```
const miFunción = (param1, param2) => hacerAlgo(param1, param2)
```

Si tiene un parámetro (y solo uno), puede omitir los paréntesis por completo:

```
const miFuncion = param => hacerAlgo(param)
```

Gracias a esta breve sintaxis, las funciones de flecha **fomentan el uso de funciones pequeñas.**

retorno implícito

Las funciones de flecha le permiten tener un retorno implícito: los valores se devuelven sin tener que usar la palabra clave `return` de retorno .

Funciona cuando hay una declaración de una línea en el cuerpo de la función:

```
const miFuncion = () => 'prueba'

miFuncion() //'prueba'
```

Otro ejemplo, al devolver un objeto, recuerde envolver los corchetes entre paréntesis para evitar que se considere el corchete del cuerpo de la función de envoltorio:

```
const myFunction = () => ({ valor: 'prueba' })

myFunction() //{valor: 'prueba'}
```

Cómo funciona esto en funciones de flecha

este es un concepto que puede ser complicado de entender, ya que varía mucho dependiendo de la contexto y también varía según el modo de JavaScript *(modo estricto* o no).

Es importante aclarar este concepto porque las funciones de flecha se comportan de manera muy diferente en comparación con las funciones regulares.

Cuando se define como un método de un objeto, en una función regular esto se refiere al objeto, por lo que puede hacer:

```
const coche =
    { modelo: 'Fiesta',
    fabricante: 'Ford', nombre
    completo: function() {
        devuelve `${este.fabricante} ${este.modelo}`
    }
}
```

llamar a car.fullName() devolverá "Ford Fiesta" .

Las funciones this scope with arrow se **heredan** del contexto de ejecución. Una función de flecha no vincula esto en absoluto, por lo que su valor se buscará en la pila de llamadas, por lo que en este código car.fullName() no funcionará y devolverá la cadena "undefined undefined" :

```
const coche =
    { modelo: 'Fiesta',
    fabricante: 'Ford', fullName:
    () => { return `${este.fabricante}
        ${este.modelo}`
    }
}
```

Debido a esto, las funciones de flecha no son adecuadas como métodos de objetos.

Las funciones de flecha tampoco se pueden usar como constructores, cuando la creación de instancias de un objeto generará un `Error de tipo .`

Aquí es donde se deben usar funciones regulares en su lugar, **cuando el contexto dinámico no es necesario.**

Esto también es un problema cuando se manejan eventos. Los detectores de eventos DOM establecen que este sea el elemento de destino, y si confía en `esto` en un controlador de eventos, es necesaria una función regular:

```
const link = document.querySelector('#link')
link.addEventListener('click', () => {
    // esta === ventana
})
```

```
const link = document.querySelector('#link')
link.addEventListener('click', function() {
    // este enlace ===
})
```

Clases

JavaScript tiene una forma bastante poco común de implementar la herencia: la herencia prototípica. herencia prototípica, aunque en mi opinión es excelente, es diferente a la implementación de herencia de la mayoría de los otros lenguajes de programación populares, que se basa en clases.

Las personas que venían de Java o Python u otros lenguajes tenían dificultades para comprender las complejidades de la herencia prototípica, por lo que el comité de ECMAScript decidió rociar azúcar sintáctica sobre la herencia prototípica para que se asemeje a cómo funciona la herencia basada en clases en otras implementaciones populares.

Esto es importante: JavaScript bajo el capó sigue siendo el mismo y puede acceder a un prototipo de objeto de la forma habitual.

Una definición de clase

Así es como se ve una clase.

```
clase Persona
    { constructor(nombre) {
        este.nombre = nombre
    }

    hola()
        { return 'Hola, soy                '  + este.nombre + '.'
    }
}
```

Una clase tiene un identificador, que podemos usar para crear nuevos objetos usando new `ClassIdentifier()` .

Cuando se inicializa el objeto, se llama al método constructor y se pasan los parámetros.

Una clase también tiene tantos métodos como necesita. En este caso, hello es un método y se puede llamar a todos los objetos derivados de esta clase:

```
const flavio = nueva Persona ('Flavio')
flavio.hola()
```

herencia de clase

Una clase puede extender otra clase y los objetos inicializados usando esa clase heredan todos los métodos de ambas clases.

Si la clase heredada tiene un método con el mismo nombre que una de las clases más altas en la jerarquía, el método más cercano tiene prioridad:

```
El programador de clase extiende a la
    persona { hola () { devuelve super. hola ()
        +                                    Soy un programador.'
    }
}

const flavio = nuevo Programador('Flavio') flavio.hola()
```

(el programa anterior imprime *"Hola, soy Flavio. Soy programador").*

Las clases no tienen declaraciones de variables de clase explícitas, pero debe inicializar cualquier variable en el constructor

Dentro de una clase, puede hacer referencia a la clase principal llamando a `super()` .

Métodos estáticos

Normalmente, los métodos se definen en la instancia, no en la clase.

Los métodos estáticos se ejecutan en la clase en su lugar:

```
clase Persona
    { static genericHello() {
        devolver 'Hola'
    }
}

Persona.genericHello() //Hola
```

Métodos privados

JavaScript no tiene una forma integrada de definir métodos privados o protegidos.

Hay soluciones alternativas, pero no las describiré aquí.

Getters y setters

Puede agregar métodos con el prefijo `get` o `set` para crear un getter y un setter, que son dos fragmentos de código diferentes que se ejecutan según lo que esté haciendo: acceder a la variable o modificar su valor.

```
clase Persona
    { constructor(nombre) {
        este.nombre = nombre
    }

    establecer nombre (valor) {
        este.nombre = valor
    }

    obtener nombre() {
        devolver este.nombre
    }
}
```

Si solo tiene un getter, la propiedad no se puede establecer y se ignorará cualquier intento de hacerlo:

```
clase Persona
    { constructor(nombre) {
        este.nombre = nombre
    }

    obtener nombre() {
        devolver este.nombre
    }
}
```

Si solo tiene un setter, puede cambiar el valor pero no acceder a él desde el exterior:

```
clase Persona
    { constructor(nombre)
        { este.nombre = nombre
    }

    establecer nombre (valor) {
        este.nombre = valor
    }
}
```

Parámetros predeterminados

Los valores de parámetros predeterminados se introdujeron en ES2015 y se implementan ampliamente en navegadores modernos.

Esta es una función doSomething que acepta param1 .

```
const hacerAlgo = (param1) => {

}
```

Podemos agregar un valor predeterminado para param1 si la función se invoca sin especificar un parámetro:

```
const hacerAlgo = (param1 = 'prueba') => {

}
```

Esto también funciona para más parámetros, por supuesto:

```
const hacerAlgo = (param1 = 'prueba', param2 = 'prueba2') => {

}
```

¿Qué sucede si tiene un objeto único con valores de parámetros en él?

Érase una vez, si teníamos que pasar un objeto de opciones a una función, para tener valores predeterminados de esas opciones si una de ellas no estaba definida, había que agregar un poco de código dentro la función:

```
const colorear = (opciones) => {
    if (!opciones)
        { opciones = {}
    }

    const color = ('color' en opciones) ? opciones.color: 'amarillo'
    ...
}
```

Con la desestructuración puede proporcionar valores predeterminados, lo que simplifica mucho el código:

```
const colorear = ({ color = 'amarillo' }) => {
    ...
}
```

Si no se pasa ningún objeto al llamar a nuestra función de `colorear` , de manera similar podemos asignar un objeto vacío por defecto:

```
giro constante = ({ color = 'amarillo' } = {}) => {
    ...
}
```

Literales de plantilla

Los literales de plantilla le permiten trabajar con cadenas de una manera novedosa en comparación con ES5 y versiones anteriores.

La sintaxis a primera vista es muy simple, solo use comillas invertidas en lugar de comillas simples o dobles:

```
const a_string = ` algo`
```

Son únicos porque brindan muchas características que las cadenas normales construidas con comillas no tienen, en particular:

- ofrecen una excelente sintaxis para definir cadenas de
- varias líneas proporcionan una manera fácil de interpolar variables y expresiones en
- cadenas le permiten crear DSL con etiquetas de plantilla (DSL significa lenguaje específico del dominio y se usa, por ejemplo, en React by Styled Components, para definir CSS para un componente)

Vamos a sumergirnos en cada uno de estos en detalle.

Cadenas multilínea

Antes de ES6, para crear una cadena que abarcaba dos líneas, tenía que usar el carácter \ al final de una línea:

```
cadena constante =
    'primera parte \
segunda parte'
```

Esto permite crear una cadena en 2 líneas, pero se representa en una sola línea:

```
primera parte segunda parte
```

Para representar la cadena en varias líneas también, debe agregar explícitamente \n al final de cada línea, así:

```
cadena constante =
    'primera linea\n\
segunda linea'
```

o

```
const string = 'primera línea\n' + 'segunda línea'
```

Los literales de plantilla hacen que las cadenas de varias líneas sean mucho más simples.

Una vez que se abre un literal de plantilla con el acento grave, simplemente presiona Intro para crear una nueva línea, sin caracteres especiales, y se representa tal cual:

```
const string = `Oye esto


cuerda
es impresionante!`
```

Tenga en cuenta que el espacio es significativo, por lo que al hacer esto:

```
const string = `Primero
                Segundo`
```

va a crear una cadena como esta:

```
Primero
                Segundo
```

una manera fácil de solucionar este problema es tener una primera línea vacía y agregar el método trim() justo después del acento grave de cierre, lo que eliminará cualquier espacio antes de la primera personaje:

```
                `
cadena constante =
Primero
Segundo`.trim()
```

Interpolación

Los literales de plantilla proporcionan una manera fácil de interpolar variables y expresiones en cadenas.

Lo hace usando la sintaxis ${...} :

```
constante var = 'prueba'
const string = `algo ${var}` //prueba de algo
```

dentro del ${} puedes agregar cualquier cosa, incluso expresiones:

```
const string = `algo ${1 + 2 + 3}` const string2 = `algo $
{foo() ? 'x' : 'y'}`
```

Etiquetas de plantilla

Las plantillas etiquetadas son una característica que puede sonar menos útil al principio para usted, pero en realidad es utilizada por muchas bibliotecas populares, como Styled Components o Apollo, la biblioteca de cliente/servidor de GraphQL, por lo que es esencial entender cómo funciona.

En la plantilla de componentes con estilo, las etiquetas se utilizan para definir cadenas CSS:

```
const Button = styled.button` font-size:
    1.5em; color de fondo: negro; color
    blanco;
`
```

En Apollo, las etiquetas de plantilla se utilizan para definir un esquema de consulta de GraphQL:

```
const consulta = gql`
    consulta {
        ...
` }
```

Las etiquetas de plantilla styled.button y gql resaltadas en esos ejemplos son solo **funciones:**

```
function gql(literales, ...expresiones) {}
```

esta función devuelve una cadena, que puede ser el resultado de *cualquier* tipo de cálculo.

literals es una matriz que contiene el contenido literal de la plantilla tokenizado por las expresiones interpolaciones.

expressions contiene todas las interpolaciones.

Si tomamos un ejemplo anterior:

```
const string = `algo ${1 + 2 + 3}`
```

literales es una matriz con dos elementos. El primero es algo , la cadena hasta el primero. interpolación, y la segunda es una cadena vacía, el espacio entre el final de la primera interpolación (solo tenemos una) y el final de la cadena.

`expressions` en este caso es una matriz con un solo elemento, `6 .`

Un ejemplo más complejo es:

```
const string = `algo más ${'x'} nueva
línea ${1 + 2 + 3}

prueba`
```

en este caso, los `literales` son una matriz donde el primer elemento es:

```
;`algo
otro `
```

el segundo es:

```
;`
nueva linea `
```

y el tercero es:

```
;`
prueba`
```

`expressions` en este caso es una matriz con dos elementos, `x` y `6 .`

La función a la que se le pasan esos valores puede hacer cualquier cosa con ellos, y este es el poder de esta característica amable.

El ejemplo más simple es replicar lo que hace la interpolación de cadenas, uniendo literales y expresiones :

```
const interpolado = interpolado`pagué ${10}€`
```

y así es como funciona la interpolación :

```
función interpolar(literales, ...expresiones) {
    let string = for
    (const [i, val] de expresiones) { string += literales[i]
        + val

    } cadena += literales[literales.longitud - 1] cadena
    de retorno
}
```

Destrucción de asignaciones

Dado un objeto, puede extraer solo algunos valores y ponerlos en variables con nombre:

```
persona constante = {
    firstName: 'Tom',
    lastName: 'Cruise', actor:
    verdadero, edad: 54, //
    inventado
}

const {firstName: nombre, edad} = persona
```

`name` y `age` contienen los valores deseados.

La sintaxis también funciona en matrices:

```
const a = [1,2,3,4,5] const
[primero, segundo] = a
```

Esta declaración crea 3 nuevas variables al obtener los elementos con índice 0, 1, 4 de la matriz `un` :

```
const [primero, segundo, , , quinto] = a
```

Literales de objetos mejorados

En ES2015 Object Literals ganó superpoderes.

Sintaxis más simple para incluir variables

en lugar de hacer

```
const algo = 'y' const x = {

    algo algo
}
```

tu puedes hacer

```
const algo = 'y' const x =
{ algo

}
```

Prototipo

Un prototipo se puede especificar con

```
const unObjeto = { y: 'y' } const x =
{ __proto__: unObjeto

}
```

súper()

```
const anObject = { y: 'y', test: () => 'zoo' } const x = { __proto__:
anObject, test() { return super.test() + 'x'



    }

} x.prueba() // zoox
```

Propiedades dinámicas

```
constante x = {
```

```
    ['a' + '_' + 'b']: 'z'

} x.a_b //z
```

Bucle for-of

ES5 en 2009 introdujo los bucles forEach() . Si bien eran agradables, no ofrecían forma de romperse, como siempre lo hacían los bucles for .

ES2015 introdujo el bucle **for-of ,** que combina la concisión de forEach con la capacidad de romper:

```
//iterar sobre el valor
for (const v of ['a', 'b', 'c']) { console.log(v);

}

//Obtenga el índice también, usando `entries()` for (const
[i, v] of ['a', 'b', 'c'].entries()) {
    console.log(índice) //índice
    console.log(valor) //valor
}
```

Observe el uso de const . Este ciclo crea un nuevo alcance en cada iteración, por lo que podemos con seguridad use eso en lugar de let .

La diferencia con for...in es:

- for...of **itera sobre los valores de propiedad** for...in **itera**
- **los nombres de propiedad**

promesas

Una promesa se define comúnmente como **un representante de un valor que eventualmente se convertirá disponible.**

Las promesas son una forma de lidiar con el código asincrónico, sin escribir demasiadas devoluciones de llamada en su código.

Las funciones asincrónicas usan la API de promesas como su bloque de construcción, por lo que comprenderlas es fundamental incluso si en el código más nuevo probablemente usará funciones asincrónicas en lugar de promesas.

Cómo funcionan las promesas, en resumen

Una vez que se ha llamado a una promesa, comenzará en **estado pendiente.** Esto significa que la función de la persona que llama continúa la ejecución, mientras espera que la promesa haga su propio procesamiento y le dé alguna retroalimentación a la función de la persona que llama.

En este punto, la función de la persona que llama espera que devuelva la promesa en un **estado resuelto** o en un **estado rechazado,** pero como sabe, JavaScript es asíncrono, por lo que *la función continúa su ejecución mientras la promesa lo hace.*

¿Qué promesas de uso de API JS?

Además de su propio código y el código de la biblioteca, las promesas se utilizan en la Web moderna estándar. API como:

- API de batería
- la API de obtención
- Trabajadores de servicios

Es poco probable que en JavaScript moderno *no* uses promesas, así que comencemos sumergirse directamente en ellos.

Creando una promesa

La API de Promise expone un constructor de Promise, que se inicializa con new Promise() :

```
dejar hecho = verdadero

const isItDoneYet = nueva promesa ((resolver, rechazar) => {
  si (hecho) {
```

```
        const trabajoTerminado = 'Aquí está lo que construí'
        resolve(trabajoTerminado) } else { const por qué = 'Sigo
    trabajando en otra cosa' rechazar(por qué)



} })
```

Como puede ver, la promesa verifica la constante `global` realizada y, si eso es cierto, devolvemos una promesa resuelta; de lo contrario, una promesa rechazada.

Usando `resolver` y `rechazar` podemos comunicar un valor, en el caso anterior solo devolvemos una cadena, pero también podría ser un objeto.

Consumir una promesa

En la última sección, presentamos cómo se crea una promesa.

Ahora veamos cómo se puede *consumir* o usar la promesa.

```
const isItDoneYet = nueva promesa()
//...

const checkIfItsDone = () => { isItDoneYet

        .then(ok =>



        { console.log(ok) }) .catch(err
        => { console.error(err) })
}
```

Ejecutar `checkIfItsDone()` ejecutará la promesa `isItDoneYet()` y esperará a que se resuelva, usando la devolución de llamada, y si hay un error, lo manejará en la captura

llamar de vuelta.

Encadenando promesas

Una promesa se puede devolver a otra promesa, creando una cadena de promesas.

La API Fetch proporciona un gran ejemplo de encadenamiento de promesas , una capa encima de la XMLHttpRequest API, que podemos usar para obtener un recurso y poner en cola una cadena de promesas para se ejecuta cuando se obtiene el recurso.

Fetch API es un mecanismo basado en promesas, y llamar a fetch() es equivalente a definir nuestra propia promesa usando new Promise() .

Ejemplo de encadenamiento de promesas

```
estado constante = respuesta => {
    if (respuesta.estado >= 200 && respuesta.estado < 300) {
        return Promise.resolve(respuesta)

    } return Promise.reject(nuevo Error(response.statusText))
}

const json = respuesta => respuesta.json()

fetch('/


    todos.json') .then(estado) .then(json) .then(datos => {
        console.log('Solicitud exitosa con respuesta JSON', datos) }) .catch(error => {


        console.log('Solicitud fallida', error) })
```

En este ejemplo, llamamos a fetch() para obtener una lista de elementos TODO del archivo todos.json que se encuentra en la raíz del dominio y creamos una cadena de promesas.

Ejecutar fetch() devuelve una respuesta, que tiene muchas propiedades, y dentro de ellas tenemos referencia:

- `estado` , un valor numérico que representa el código de estado HTTP
- `estadoTexto` , un mensaje de estado, que está bien si la solicitud tuvo éxito

`respuesta` también tiene un método json() , que devuelve una promesa que se resolverá con el contenido del cuerpo procesado y transformado en JSON.

Entonces, dadas esas premisas, esto es lo que sucede: la primera promesa en la cadena es una función que definimos, llamada status() , que verifica el estado de la respuesta y si no es un éxito respuesta (entre 200 y 299), rechaza la promesa.

Esta operación hará que la cadena de promesa omita todas las promesas encadenadas enumeradas y salte directamente a la instrucción catch() en la parte inferior, registrando el texto Solicitud fallida junto con el mensaje de error.

Si eso tiene éxito, llama a la función json() que definimos. Dado que la promesa anterior, cuando tuvo éxito, devolvió el objeto de respuesta , lo obtenemos como entrada para la segunda promesa.

En este caso, devolvemos los datos JSON procesados, por lo que la tercera promesa recibe el JSON directamente:

```
.then((datos) => {
  console.log('Solicitud exitosa con respuesta JSON', datos) })
```

y lo registramos en la consola.

Manejo de errores

En el ejemplo anterior, en la sección anterior, teníamos una captura que se agregó a la cadena de promesas.

Cuando algo en la cadena de promesas falla y genera un error o rechaza la promesa, el control va a la instrucción catch() más cercana en la cadena.

```
nueva Promesa((resolver, rechazar) =>
  { lanzar nuevo Error('Error') }).catch(err => {

  consola.error(err) })


// o

nueva Promesa((resolver, rechazar) =>
  { rechazar('Error') }).catch(err => {

  consola.error(err) })
```

Errores en cascada

Si dentro de catch() genera un error, puede agregar un segundo catch() para manejarlo, y pronto.

```
nueva Promesa((resolver, rechazar) =>
  { lanzar nuevo Error('Error')
})
   .catch(err =>
     { lanzar un nuevo
  error('Error') }) .catch(err =>
   { console.error(err) })
```

Orquestando promesas

Promesa.todo()

Si necesita sincronizar diferentes promesas, Promise.all() lo ayuda a definir una lista de promesas, y ejecutar algo cuando están todas resueltas.

Ejemplo:

```
const f1 = buscar('/algo.json') const f2 = buscar('/
algo2.json')

Promise.all([f1, f2]) .then(res
    => { console.log('Array
       de resultados', res) }) .catch(err =>
    { console.error(err) })
```

La sintaxis de asignación de desestructuración de ES2015 también le permite hacer

```
Promise.all([f1, f2]).then(([res1, res2]) => {
    consola.log('Resultados', res1, res2) })
```

Por supuesto, no está limitado a usar fetch , **cualquier promesa es buena.**

Promesa.carrera()

Promise.race() se ejecuta tan pronto como se resuelve una de las promesas que le pasas, y ejecuta el devolución de llamada adjunta solo una vez con el resultado de la primera promesa resuelta.

Ejemplo:

```
const promesaUna = nueva Promesa((resolver, rechazar) => {
    setTimeout(resolver, 500, 'uno') }) const
prometeDos = new Promesa((resolver,
rechazar) => {
    setTimeout(resolver, 100, 'dos') })


Promesa.carrera([promesaUno, promesaDos]).then(resultado => {
    console.log(resultado) // 'dos' })
```

Errores comunes

TypeError no capturado: indefinido no es una promesa

Si obtiene el TypeError no capturado: undefined no es un error de promesa en la consola, asegúrese de usar new Promise() en lugar de solo Promise()

Módulos

ES Modules es el estándar ECMAScript para trabajar con módulos.

Si bien Node.js ha estado usando el estándar CommonJS durante años, el navegador nunca tuvo un sistema de módulos, ya que todas las decisiones importantes, como un sistema de módulos, deben ser estandarizadas primero por ECMAScript y luego implementadas por el navegador.

Este proceso de estandarización se completó con ES2015 y los navegadores comenzaron a implementar este estándar tratando de mantener todo bien alineado, funcionando de la misma manera, y ahora los módulos ES son compatibles con Chrome, Safari, Edge y Firefox (desde la versión 60).

Los módulos son geniales porque te permiten encapsular todo tipo de funcionalidad y exponer esta funcionalidad a otros archivos JavaScript, como bibliotecas.

La sintaxis de los módulos ES

La sintaxis para importar un módulo es:

```
importar paquete desde 'nombre-módulo'
```

mientras que CommonJS usa

```
const paquete = require('nombre-módulo')
```

Un módulo es un archivo JavaScript que **exporta** uno o más valores (objetos, funciones o variables), utilizando la palabra clave `export`. Por ejemplo, este módulo exporta una función que devuelve una cadena mayúsculas:

mayúsculas.js

```
exportar str predeterminado => str.toUpperCase()
```

En este ejemplo, el módulo define una única **exportación predeterminada,** por lo que puede ser una función anónima. De lo contrario, necesitaría un nombre para distinguirlo de otras exportaciones.

Ahora, **cualquier otro módulo de JavaScript** puede importar la funcionalidad que ofrece uppercase.js al importarlo.

Una página HTML puede agregar un módulo usando una etiqueta `<script>` con el tipo especial="módulo" atributo:

```
<script type="module" src="index.js"></script>
```

> Nota: la importación de este módulo se comporta como una carga de secuencia de comandos diferida . Vea cargar JavaScript de manera eficiente con aplazamiento y asíncrono

Es importante tener en cuenta que cualquier script cargado con `type="module"` se carga en modo estricto.

En este ejemplo, el módulo `uppercase.js` define una **exportación predeterminada,** por lo que cuando la importamos, podemos asignarle el nombre que prefiramos:

```
importar a Mayúsculas desde ' ./mayúsculas.js '
```

y podemos usarlo:

```
toUpperCase('prueba') //'PRUEBA'
```

También puede usar una ruta absoluta para la importación del módulo, para hacer referencia a los módulos definidos en otro dominio:

```
importar aUpperCase desde 'https://flavio-es-modules-example.glitch.me/uppercase.js'
```

Esta también es una sintaxis de importación válida:

```
importar {toUpperCase} desde '/uppercase.js' importar
{toUpperCase} desde '../uppercase.js'
```

Esto no es:

```
importar {toUpperCase} desde 'uppercase.js' importar
{toUpperCase} desde 'utils/uppercase.js'
```

Es absoluto o tiene un ./ o / antes del nombre.

Otras opciones de importación/exportación

Vimos este ejemplo arriba:

```
exportar str predeterminado => str.toUpperCase()
```

Esto crea una exportación predeterminada. Sin embargo, en un archivo puede exportar más de una cosa, utilizando esta sintaxis:

```
constante a = 1
constante b = 2
constante c = 3

exportar {a, b, c}
```

Otro módulo puede importar todas esas exportaciones usando

```
importar * desde 'módulo'
```

Puede importar solo algunas de esas exportaciones, utilizando la asignación de desestructuración:

```
importar { a } desde 'módulo'
importar { a, b } desde 'módulo'
```

Puede cambiar el nombre de cualquier importación, para mayor comodidad, utilizando `como` :

```
importar {a, b como dos} desde 'módulo'
```

Puede importar la exportación predeterminada y cualquier exportación no predeterminada por nombre, como en este común Reaccionar importar:

```
importar Reaccionar, {Componente} de 'reaccionar'
```

Puede ver un ejemplo de Módulos ES aquí: https://glitch.com/edit/#!/flavio-es-modules example?path=index.html

CORS

Los módulos se obtienen mediante CORS. Esto significa que si hace referencia a secuencias de comandos de otros dominios, deben tener un encabezado CORS válido que permita la carga entre sitios (como `Access Control-Allow-Origin: * `)

¿Qué pasa con los navegadores que no admiten módulos?

Use una combinación de `type="module"` y `nomodule` :

```
<script type="module" src="module.js"></script> <script
nomodule src="fallback.js"></script>
```

Conclusión

Los módulos ES son una de las características más importantes introducidas en los navegadores modernos. Son parte de ES6 pero el camino para implementarlos ha sido largo.

¡Ya podemos usarlos! Pero también debemos recordar que tener más de unos pocos módulos tendrá un impacto en el rendimiento de nuestras páginas, ya que es un paso más que el navegador debe realizar en tiempo de ejecución.

Es probable que Webpack siga siendo un jugador importante, incluso si los módulos ES aterrizan en el navegador, pero tener una función de este tipo integrada directamente en el lenguaje es enorme para unificar cómo funcionan los módulos del lado del cliente y también en Node.js.

Nuevos métodos de cadena

Cualquier valor de cadena obtuvo algunos métodos de instancia nuevos:

- repetir()
- códigoPuntoEn()

repetir()

Repite las cadenas el número de veces especificado:

```
'Ho'.repetir (3) //' HoHoHo '
```

Devuelve una cadena vacía si no hay ningún parámetro o el parámetro es 0 . Si el parámetro es negativo obtendrá un RangeError.

códigoPuntoEn()

Este método se puede usar para manejar caracteres Unicode que no se pueden representar con un solo Unidad Unicode de 16 bits, pero necesita 2 en su lugar.

Al usar charCodeAt() , debe recuperar el primero y el segundo, y combinarlos. Usando codePointAt() obtienes el carácter completo en una sola llamada.

Por ejemplo, este carácter chino "ÿ" está compuesto por 2 partes UTF-16 (Unicode):

```
"ÿ".charCodeAt(0).toString(16) //d842
"ÿ".charCodeAt(1).toString(16) //dfb7
```

Si crea un nuevo personaje combinando esos caracteres Unicode:

```
"\ud842\udfb7" //"Guau"
```

Puede obtener el mismo resultado usando codePointAt() :

```
"ÿ".codePointAt(0) //20bb7
```

Si crea un nuevo personaje combinando esos caracteres Unicode:

```
"\ tu {20bb7}" // "ÿ"
```

Más sobre Unicode y trabajar con él en mi guía Unicode: https://flaviocopes.com/unicode/

Nuevos métodos de objetos

ES6 introdujo varios métodos estáticos bajo el espacio de nombres Object:

- `Object.is()` determina si dos valores son el mismo valor
- `Object.assign()` utilizado para copiar superficialmente un objeto
- `Object.setPrototypeOf` establece un prototipo de objeto

Objeto.es()

Este método tiene como objetivo ayudar a comparar valores.

Uso:

```
Objeto.es(a, b)
```

El resultado siempre es falso a menos que:

- `a` y `b` son exactamente el mismo objeto
- `a` y `b` son cadenas iguales (las cadenas son iguales cuando están compuestas por los mismos caracteres)
- `a` y `b` son números iguales (los números son iguales cuando su valor es igual)
- `a` y `b` son indefinidos , ambos `nulos` , ambos `NaN` , ambos `verdaderos` o ambos `falsos`

`0` y `-0` son valores diferentes en JavaScript, así que preste atención en este caso especial (convertir todos a `+0` usando el operador unario `+` antes de comparar, por ejemplo).

Objeto.assign()

Introducido en `ES2015` , este método copia todas las **propiedades enumerables propias** de uno o más objetos en otro.

Su caso de uso principal es crear una copia superficial de un objeto.

```
const copiado = Object.assign({}, original)
```

Al ser una copia superficial, los valores se clonan y las referencias de los objetos se copian (no los objetos). ellos mismos), por lo que si edita una propiedad de objeto en el objeto original, eso también se modifica en el objeto copiado, ya que el objeto interno al que se hace referencia es el mismo:

```
constante original = {
    name: 'Fiesta',
```

```
    coche: {
        color: 'azul'
    }

} const copiado = Object.assign({}, original)

original.name = 'Focus'
original.car.color = 'amarillo'

copiado.nombre //Fiesta
copiado.coche.color //amarillo
```

Mencioné "uno o más":

```
const PersonaSabia =
    { esSabia: verdadero

} const tonto = {
    es tonto: cierto

} const PersonaSabiaYTonta = Object.assign({}, PersonaSabia, PersonaTonta)

console.log(personaSabiaYTonta) //{ esSabia: verdadero, esTonta: verdadero }
```

Objeto.setPrototypeOf()

Establecer el prototipo de un objeto. Acepta dos argumentos: el objeto y el prototipo.

Uso:

```
Object.setPrototypeOf(objeto, prototipo)
```

Ejemplo:

```
constante animal = {
    esAnimal: cierto

} const mamífero = {
    esMamífero: cierto
}

mamífero.__proto__ = animal
mamífero.esAnimal //verdadero

const perro = Object.create(animal)

dog.isAnimal //true
console.log(dog.isMammal) //indefinido
```

```
Object.setPrototypeOf(perro, mamífero)

perro.esAnimal //verdadero
perro.isMammal //verdadero
```

El operador de propagación

Puede expandir una matriz, un objeto o una cadena usando el operador de expansión ... .

Comencemos con un ejemplo de matriz. Dado

```
constante a = [1, 2, 3]
```

puedes crear una nueva matriz usando

```
constante b = [...a, 4, 5, 6]
```

También puede crear una copia de una matriz usando

```
constante c = [...a]
```

Esto también funciona para los objetos. Clonar un objeto con:

```
const nuevoObj = {... viejoObj}
```

Usando cadenas, el operador de propagación crea una matriz con cada carácter en la cadena:

```
const hey = 'hey' const
arrayized = [... hey] // ['h', 'e', 'y']
```

Este operador tiene algunas aplicaciones bastante útiles. La más importante es la capacidad de usar una matriz como argumento de función de una manera muy simple:

```
const f = (foo, bar) => {} const a =
[1, 2] f(...a)
```

(en el pasado, podía hacer esto usando f.apply(null, a) pero eso no es tan bueno ni legible)

El **elemento rest** es útil cuando se trabaja con la **desestructuración de matrices:**

```
números constantes = [1, 2, 3, 4, 5]
[primero, segundo, ...otros] = números
```

y **difundir elementos:**

```
números constantes = [1, 2, 3, 4, 5] suma constante
= (a, b, c, d, e) => a + b + c + d + e suma constante = suma (... números)
```

ES2018 introduce propiedades de descanso, que son las mismas pero para los objetos.

Resto de propiedades:

```
const { primero, segundo, ...otros } = {
    primero: 1,
    segundo: 2,
    tercero: 3,
    cuarto: 4,
    quinto: 5
}

primero // 1
segundo // 2
otros // { tercero: 3, cuarto: 4, quinto: 5 }
```

Las propiedades de extensión permiten crear un nuevo objeto combinando las propiedades del objeto pasado después del operador de extensión:

```
elementos constantes = { primero, segundo, ... otros } elementos //
{ primero: 1, segundo: 2, tercero: 3, cuarto: 4, quinto: 5 }
```

Establecer

Una estructura de datos Set permite agregar datos a un contenedor.

Un Conjunto es una colección de objetos o tipos primitivos (cadenas, números o valores booleanos), y puede pensar en él como un Mapa donde los valores se usan como claves de mapa, siendo el valor del mapa siempre un booleano verdadero.

Inicializar un conjunto

Un conjunto se inicializa llamando:

```
const s = nuevo Conjunto()
```

Agregar elementos a un conjunto

Puede agregar elementos al conjunto utilizando el método de agregar :

```
s.add('uno')
s.add('dos')
```

Un conjunto solo almacena elementos únicos, por lo que llamar a s.add('one') varias veces no agregará elementos nuevos. elementos.

No puede agregar varios elementos a un conjunto al mismo tiempo. Necesitas llamar a add() multiple veces.

Comprobar si un elemento está en el conjunto

Una vez que un elemento está en el conjunto, podemos verificar si el conjunto lo contiene:

```
s.has('uno') //verdadero
s.has('tres') //falso
```

Eliminar un elemento de un conjunto por clave

Utilice el método eliminar() :

```
s.delete('uno')
```

Determinar el número de elementos en un conjunto

Utilice la propiedad de tamaño :

```
s. tamaño
```

Eliminar todos los elementos de un conjunto

Usa el método clear() :

```
claro()
```

Iterar los elementos en un conjunto

Use los métodos keys() o values() - son equivalentes:

```
for (const k de s.keys()) {
    consola.log(k)
}

for (const k de s.values()) {
    consola.log(k)
}
```

El métodoentries() devuelve un iterador, que puede usar así:

```
const i = s.entradas()
consola.log(i.siguiente())
```

llamar a i.next() devolverá cada elemento como un objeto { value, done = false } hasta que finalice el iterador,
momento en el cual done es verdadero .

También puede usar el método forEach() en el conjunto:

```
s.forEach(v => consola.log(v))
```

o simplemente puede usar el conjunto en un bucle for..of:

```
for (const k of s)
    { console.log(k)
}
```

Inicializar un conjunto con valores

Puede inicializar un conjunto con un conjunto de valores:

```
const s = nuevo Conjunto ([1, 2, 3, 4])
```

Convertir a matriz

Convierta las teclas Set en una matriz

```
const a = [...s.keys()]

// o

const a = [...s.valores()]
```

Un conjunto débil

Un WeakSet es un tipo especial de Set.

En un conjunto, los elementos nunca se recolectan como basura. En cambio, un WeakSet permite que todos sus elementos se recopilen libremente. Cada clave de un WeakSet es un objeto. Cuando se pierde la referencia a este objeto, el valor se puede recolectar como basura.

Aquí están las principales diferencias:

1. no puedes iterar sobre el WeakSet
2. no puede borrar todos los elementos de un WeakSet 3.

no puede verificar su tamaño

Un WeakSet generalmente se usa en el código de nivel de marco y solo expone estos métodos:

- agregar
- () tiene
- () eliminar ()

Mapa

Una estructura de datos Map permite asociar datos a una clave.

Antes de ES6

Antes de su introducción, la gente generalmente usaba objetos como mapas, asociando algún objeto o valor a un valor clave específico:

```
const coche = {}
coche['color'] = 'rojo'
auto.propietario = 'Flavio'
console.log(auto['color']) //rojo console.log(auto.color) //
rojo console.log(auto.propietario) //Flavio
console.log(auto['propietario']) //Flavio
```

Entrar en el mapa

ES6 introdujo la estructura de datos Map, brindándonos una herramienta adecuada para manejar este tipo de organización de datos.

Un mapa se inicializa llamando:

```
const m = nuevo Mapa()
```

Agregar elementos a un mapa

Puede agregar elementos al mapa usando el método set :

```
m.set('color', 'rojo') m.set('edad', 2)
```

Obtener un elemento de un mapa por clave

Y puede obtener elementos de un mapa usando get :

```
const color = m.get('color') const edad =
m.get('edad')
```

Eliminar un elemento de un mapa por clave

Utilice el método eliminar() :

```
m.delete('color')
```

Eliminar todos los elementos de un mapa

Usa el método clear() :

```
m.claro()
```

Comprobar si un mapa contiene un elemento por clave

Usa el método has() :

```
const hasColor = m.has('color')
```

Encontrar el número de elementos en un mapa

Utilice la propiedad de tamaño :

```
const tamaño = m.tamaño
```

Inicializar un mapa con valores

Puede inicializar un mapa con un conjunto de valores:

```
const m = nuevo Mapa([['color', 'rojo'], ['propietario', 'Flavio'], ['edad', 2]])
```

Claves del mapa

Al igual que cualquier valor (objeto, matriz, cadena, número) se puede usar como el valor de la entrada de clave-valor de un elemento del mapa, **cualquier valor se puede usar como la clave,** incluso los objetos.

Si intenta obtener una clave que no existe usando get() fuera de un mapa, devolverá undefined .

Situaciones extrañas que casi nunca encontrarás en la vida real

```
const m = nuevo Mapa()
m.set(NaN, 'prueba')
m.get(NaN) //prueba
```

```
const m = nuevo Mapa()
m.set(+0, 'prueba')
m.get(-0) //prueba
```

Iterando sobre un mapa

Iterar sobre claves de mapa

Map ofrece el método keys() que podemos usar para iterar en todas las claves:

```
for (const k de m.keys()) {
    consola.log(k)
}
```

Iterar sobre los valores del mapa

El objeto Map ofrece el método de valores () que podemos usar para iterar en todos los valores:

```
for (const v de m.values()) {
    consola.log(v)
}
```

Iterar sobre clave de mapa, pares de valores

El objeto Map ofrece el métodoentries() que podemos usar para iterar en todos los valores:

```
for (const [k, v] de m.entradas()) {
    consola.log(k, v)
}
```

que se puede simplificar a

```
for (const [k, v] de m)
    { console.log(k, v)
```

```
}
```

Convertir a matriz

Convierta las claves del mapa en una matriz

```
const a = [...m.keys()]
```

Convierta los valores del mapa en una matriz

```
const a = [...m.valores()]
```

Mapa débil

Un WeakMap es un tipo especial de mapa.

En un objeto de mapa, los elementos nunca se recolectan como basura. En cambio, un WeakMap permite que todos sus elementos se recolecten libremente como basura. Cada clave de un WeakMap es un objeto. Cuando se pierde la referencia a este objeto, el valor se puede recolectar como basura.

Aquí están las principales diferencias:

1. no puede iterar sobre las claves o valores (o valores-clave) de un WeakMap 2. no puede borrar todos los elementos de un WeakMap 3. no puede verificar su tamaño

Un WeakMap expone esos métodos, que son equivalentes a los de Map:

- `obtener (k)`
- `conjunto (k, v)`
- `tiene (k)`
- `eliminar (k)`

Los casos de uso de un WeakMap son menos evidentes que los de un Mapa, y es posible que nunca encuentre la necesidad de ellos, pero esencialmente se puede usar para construir un caché sensible a la memoria que no interferirá con la recolección de basura, o para el encapsulamiento cuidadoso y la ocultación de información.

Generadores

Los generadores son un tipo especial de función con la capacidad de pausarse y reanudarse más tarde, lo que permite que se ejecute otro código mientras tanto.

Consulte la Guía completa de generadores de JavaScript para obtener una explicación detallada del tema.

El código decide que tiene que esperar, por lo que permite que se ejecute otro código "en la cola" y se reserva el derecho de reanudar sus operaciones "cuando lo que está esperando" haya terminado.

Todo esto se hace con una palabra clave única y simple: `rendimiento` . Cuando un generador contiene esa palabra clave, la ejecución se detiene.

Un generador puede contener muchas palabras clave de `rendimiento` , por lo que se detiene varias veces, y se identifica con la palabra clave `*function` , que no debe confundirse con el operador de desreferencia de puntero utilizado en lenguajes de programación de nivel inferior como C, C++ o Go.

Los generadores permiten paradigmas completamente nuevos de programación en JavaScript, lo que permite:

- Comunicación bidireccional mientras un generador está funcionando
- durante mucho tiempo mientras los bucles no congelan su programa

Aquí hay un ejemplo de un generador que explica cómo funciona todo.

```
function *calculadora(entrada) { var
    dobleEso = 2 * (rendimiento (entrada / 2)) var otro =
    rendimiento (dobleEso) return (entrada * dobleEso *
    otro)
}
```

Lo inicializamos con

```
const calc = calculadora (10)
```

Luego iniciamos el iterador en nuestro generador:

```
calc.siguiente()
```

Esta primera iteración inicia el iterador. El código devuelve este objeto:

```
{
  hecho: falso
  valor: 5
}
```

Lo que sucede es: el código ejecuta la función, con entrada = 10 como se pasó en el constructor de generadores. Funciona hasta que alcanza el rendimiento , y devuelve el contenido de yield : entrada / 2 = 5 . Entonces obtuvimos un valor de 5, y la indicación de que la iteración no ha terminado (la la función está en pausa).

En la segunda iteración pasamos el valor 7 :

```
calc.siguiente(7)
```

y lo que obtuvimos es:

```
{
    hecho: falso
    valor: 14
}
```

7 se colocó como el valor de doubleThat . Importante: puede leer como input / 2 fue el argumento, pero ese es solo el valor de retorno de la primera iteración. Ahora nos saltamos eso, y usamos el nuevo valor de entrada, 7 , y lo multiplicas por 2

Luego alcanzamos el segundo rendimiento, y eso devuelve el doble de eso. , por lo que el valor devuelto es 14 .

En la siguiente y última iteración, pasamos 100

```
calc.next(100)
```

y a cambio obtuvimos

```
{
    hecho: cierto
    valor: 14000
}
```

A medida que finaliza la iteración (no se encuentran más palabras clave de rendimiento) y simplemente devolvemos (input * doubleThat * otro) que asciende a 10 * 14 * 100 _

ES2016

Array.prototipo.incluye()

Esta función introduce una sintaxis más legible para verificar si una matriz contiene un elemento.

Con ES6 e inferior, para verificar si una matriz contenía un elemento, tenía que usar `indexOf`, cual verifica el índice en la matriz y devuelve -1 si el elemento no está allí.

Dado que `-1` se evalúa como un valor verdadero, no podría **hacer ,** por ejemplo

```
si (![1,2].indexOf(3)) {
    consola.log('No encontrado')
}
```

Con esta característica introducida en ES7 podemos hacer

```
si (![1,2].incluye(3)) {
    consola.log('No encontrado')
}
```

Operador de exponenciación

El operador de exponenciación ** es el equivalente de Math.pow() , pero llevado al lenguaje en lugar de ser una función de biblioteca.

```
Math.pow(4, 2) == 4 ** 2
```

Esta característica es una buena adición para las aplicaciones JS intensivas en matemáticas.

El operador ** está estandarizado en muchos lenguajes, incluidos Python, Ruby, MATLAB, Lua, Perl y muchos otros.

ES2017

relleno de cuerdas

El propósito del relleno de cadenas es **agregar caracteres a una cadena, de** modo que **alcance un valor específico. longitud.**

ES2017 presenta dos métodos String : padStart() y padEnd() .

```
padStart(targetLength [, padString])
padEnd (targetLength [, padString])
```

Ejemplo de uso:

padStart ()	
'prueba'.padStart (4)	'prueba'
'prueba'.padStart (5)	' prueba'
'prueba'.padStart (8)	' prueba'
'prueba'.pathStart(8, 'abcd')	'pruebaabcd'

padEnd ()	
'prueba'.padEnd (4)	'prueba'
'prueba'.padEnd (5)	'prueba '
'prueba'.padEnd (8)	'prueba '
'prueba'.padEnd (8, 'abcd')	'testabcd'

Objeto.valores()

Este método devuelve una matriz que contiene todos los valores de propiedad propios del objeto.

Uso:

```
persona constante = { nombre: 'Fred', edad: 87 }
Objeto.valores(persona) // ['Fred', 87]
```

Object.values() también funciona con matrices:

```
gente constante = ['Fred', 'Tony']
Objeto.valores(personas) // ['Fred', 'Tony']
```

Objeto.entradas()

Este método devuelve una matriz que contiene todas las propiedades propias del objeto, como una matriz de pares [clave, valor] .

Uso:

```
persona constante = { nombre: 'Fred', edad: 87 }
Objeto.entradas(persona) // [['nombre', 'Fred'], ['edad', 87]]
```

Object.entries() también funciona con matrices:

```
gente constante = ['Fred', 'Tony']
Objeto.entradas(personas) // [['0', 'Fred'], ['1', 'Tony']]
```

Objeto.getOwnPropertyDescriptors()

Este método devuelve todos los descriptores de propiedades propias (no heredadas) de un objeto.

Cualquier objeto en JavaScript tiene un conjunto de propiedades y cada una de estas propiedades tiene un descriptor.

Un descriptor es un conjunto de atributos de una propiedad y está compuesto por un subconjunto de los siguientes:

- **valor:** el valor de la propiedad
- **escritura:** verdadero la propiedad se puede cambiar
- **obtener:** una función captadora para la propiedad, llamada cuando se lee la propiedad
- **set:** una función de establecimiento para la propiedad, llamada cuando la propiedad se establece
- en un valor **configurable:** si es falso, la propiedad no se puede eliminar ni se puede cambiar ningún atributo, excepto su valor **enumerable:** verdadero si la propiedad es enumerable
-

`Object.getOwnPropertyDescriptors(obj)` acepta un objeto y devuelve un objeto con el conjunto de descriptores.

¿De qué manera es esto útil?

ES6 nos proporcionó `Object.assign()` , que copia todas las propiedades enumerables propias de uno o más objetos y devuelve un nuevo objeto.

Sin embargo, hay un problema con eso, porque no copia correctamente las propiedades con no atributos predeterminados.

Si un objeto, por ejemplo, solo tiene un setter, no se copia correctamente en un nuevo objeto, usando `Objeto.assign()` .

por ejemplo con

```
const persona1 =
    { establecer
        nombre(nuevoNombre) { consola.log(nuevoNombre)
    }
}
```

Esto no funcionará:

```
constante persona2 = {}
Objeto.asignar(persona2, persona1)
```

Pero esto funcionará:

```
constante persona3 = {}
Objeto.defineProperties(persona3,
    Objeto.getOwnPropertyDescriptors(persona1))
```

Como puede ver con una simple prueba de consola:

```
persona1.nombre = 'x'
"X"

persona2.nombre = 'x'

persona3.nombre = 'x'
"X"
```

`person2` pierde el setter, no se copió.

La misma limitación se aplica a los objetos de clonación superficial con **Object.create().**

comas finales

Esta característica permite tener comas finales en las declaraciones de funciones y en las llamadas a funciones:

```
const hacerAlgo = (var1, var2,) => {
    //...
}

hacerAlgo('prueba2', 'prueba2',)
```

Este cambio alentará a los desarrolladores a detener el feo hábito de "coma al comienzo de la línea".

Funciones asíncronas

JavaScript evolucionó en muy poco tiempo de devoluciones de llamada a promesas (ES2015) y, desde ES2017, JavaScript asíncrono es aún más simple con la sintaxis async/await.

Las funciones asíncronas son una combinación de promesas y generadores y, básicamente, son una abstracción de mayor nivel sobre las promesas. Permítanme repetir: **async/await se basa en promesas.**

¿Por qué se introdujeron async/await?

Reducen el texto estándar en torno a las promesas y la limitación de "no romper la cadena" de encadenar promesas.

Cuando se introdujeron Promises en ES2015, estaban destinados a resolver un problema con el código asincrónico, y lo hicieron, pero durante los 2 años que separaron ES2015 y ES2017, quedó claro que las *promesas no podían ser la solución final.*

Las promesas se introdujeron para resolver el famoso problema del *infierno de devolución* de llamada , pero introdujeron complejidad por sí mismas y complejidad de sintaxis.

Eran buenas primitivas en torno a las cuales se podía exponer una mejor sintaxis a los desarrolladores, por lo que cuando llegó el momento adecuado obtuvimos **funciones asíncronas.**

Hacen que el código parezca síncrono, pero detrás es asíncrono y no bloquea.
las escenas.

Cómo funciona

Una función asíncrona devuelve una promesa, como en este ejemplo:

```
const hacerAlgoAsync = () => {
    devolver nueva promesa (resolver => {
        setTimeout(() => resolve('Hice algo'), 3000) })

}
```

Cuando desee **llamar** a esta función, anteponga esperar a que **la promesa** , y **el código de llamada se detendrá hasta que se resuelva o rechace.** Una advertencia: la función de cliente debe definirse como asíncrono _ Aquí hay un ejemplo:

```
const hacerAlgo = asíncrono () => {
    console.log(espera hacerAlgoAsync())
```

```
}
```

Un ejemplo rápido

Este es un ejemplo simple de async/await utilizado para ejecutar una función de forma asíncrona:

```
const hacerAlgoAsync = () => {
    devolver nueva promesa (resolver => {
        setTimeout(() => resolve('Hice algo'), 3000) })

}

const hacerAlgo = asíncrono () => {
    console.log(espera hacerAlgoAsync())
}

consola.log('Antes')
hacerAlgo()
consola.log('Después')
```

El código anterior imprimirá lo siguiente en la consola del navegador:

```
Antes
Después
Hice algo //después de 3s
```

Promete todas las cosas

Anteponer la palabra clave async a cualquier función significa que la función devolverá una promesa.

Incluso si no lo hace explícitamente, internamente hará que devuelva una promesa.

Es por eso que este código es válido:

```
const aFunction = asíncrono () => {
    devolver 'prueba'
}

aFunction().then(alerta) // Esto alertará a 'prueba'
```

y es lo mismo que:

```
const aFunction = asíncrono () => {
    return Promesa.resolve('prueba')
}
```

```
aFunction().then(alerta) // Esto alertará a 'prueba'
```

El código es mucho más simple de leer.

Como puede ver en el ejemplo anterior, nuestro código parece muy simple. Compárelo con el código usando promesas simples, con funciones de encadenamiento y devolución de llamada.

Y este es un ejemplo muy simple, los mayores beneficios surgirán cuando el código sea mucho más complejo.

Por ejemplo, así es como obtendría un recurso JSON y lo analizaría usando promesas:

```
const getFirstUserData = () => {
    return fetch('/users.json') // obtener la lista de usuarios
        .then(response => response.json()) // analiza JSON .then(users =>
        users[0]) // selecciona el primer usuario .then(user => fetch(`/users/$
        {user.name} `)) // obtener datos de usuario .then(userResponse => response.json()) //
        analizar JSON
}

getFirstUserData ()
```

Y aquí está la misma funcionalidad provista usando await/async:

```
const getFirstUserData = asíncrono () => {
    const respuesta = await fetch('/users.json') // obtener la lista de usuarios const users =
    await respuesta.json() // analizar JSON const user = usuarios[0] // seleccionar el primer
    usuario const userResponse = await fetch (` /users/${user.name}`) // obtener datos de
    usuario const userData = esperar usuario.json() // analizar JSON

    return userData
}

getFirstUserData ()
```

Múltiples funciones asíncronas en serie

Las funciones asíncronas se pueden encadenar muy fácilmente, y la sintaxis es mucho más legible que con simples promesas:

```
const prometeHacerAlgo = () => {
    return new Promise(resolve => { setTimeout(()
        => resolve('Hice algo'), 10000) })

}
```

```
const mirarSobreAlguienHaciendoAlgo = asíncrono () => { const algo
    = esperar prometerHacerAlgo() devolver algo +
                                        'y vi'
}

const vigilarSobreAlguienMirandoAlguienHaciendoAlgo = async () => {
    const algo = aguardar verOverAlguienHaciendoAlgo() devolver algo +
                                        'y yo también vi'
}

vigilar a alguien vigilando a alguien haciendo algo().then(res => {
    consola.log(res) })
```

Imprimirá:

```
Hice algo y observé y también observé
```

Depuración más fácil

La depuración de promesas es difícil porque el depurador no pasará por encima del código asíncrono.

Async/await lo hace muy fácil porque para el compilador es como un código síncrono.

Memoria compartida y atómica

Los WebWorkers se utilizan para crear programas de subprocesos múltiples en el navegador.

Ofrecen un protocolo de mensajería vía eventos. Desde ES2017, puede crear una matriz de memoria compartida entre los trabajadores web y su creador, utilizando un SharedArrayBuffer .

Dado que se desconoce cuánto tiempo tarda en propagarse la escritura en una parte de la memoria compartida, **Atomics** es una forma de hacer cumplir que al leer un valor, se completa cualquier tipo de operación de escritura.

Puede encontrar más detalles sobre esto en la propuesta de especificaciones, que desde entonces se ha implementado.

ES2018

Propiedades de reposo/extensión

ES2015 introdujo el concepto de un **elemento de descanso** al trabajar con la **desestructuración de matrices:**

```
números constantes = [1, 2, 3, 4, 5] [primero,
 segundo, ...otros] = números
```

y **difundir elementos:**

```
números constantes = [1, 2, 3, 4, 5] suma constante
= (a, b, c, d, e) => a + b + c + d + e suma constante = suma (... números)
```

ES2018 introduce lo mismo pero para objetos.

Resto de propiedades:

```
const { primero, segundo, ...otros } = { primero: 1, segundo: 2, tercero: 3, cuarto: 4, quinto: 5 }


primero // 1
segundo // 2
otros // { tercero: 3, cuarto: 4, quinto: 5 }
```

Las propiedades de extensión permiten crear un nuevo objeto combinando las propiedades del objeto pasado después del operador de extensión:

```
elementos constantes = { primero, segundo, ... otros } elementos //
{ primero: 1, segundo: 2, tercero: 3, cuarto: 4, quinto: 5 }
```

Iteración asíncrona

La nueva construcción `for-await-of` le permite usar un objeto iterable asíncrono como el bucle iteración:

```
para esperar ( línea constante de readLines (filePath)) {
    consola.log(línea)
}
```

Dado que esto usa `await` , puede usarlo solo dentro de funciones `asíncronas` , como un `await` `normal` .

Promesa.prototipo.finalmente()

Cuando se cumple una promesa, llama con éxito a los métodos `then()` , uno tras otro.

Si algo falla durante esto, los métodos `then()` saltan y el método `catch()` es ejecutado.

`finalmente ()` le permite ejecutar algún código independientemente del éxito o no éxito ejecución de la promesa:

```
fetch('file.json') .then(data
    => data.json()) .catch(error =>
    console.error(error)) .finally(() =>
    console.log('finished'))
```

Mejoras en expresiones regulares

ES2018 introdujo una serie de mejoras con respecto a las expresiones regulares. Recomiendo mi tutorial sobre ellos, disponible en https://flaviocopes.com/javascript-regular-expressions/.

Aquí están las adiciones específicas de ES2018.

RegExp lookbehind aserciones: haga coincidir una cadena dependiendo de lo que la precede

Esta es una búsqueda anticipada: usa `?=` para hacer coincidir una cadena seguida de una subcadena específica:

```
/Roger(?=Aguas)/

/Roger(?= Waters)/.test('Roger es mi perro') //false /Roger(?=
Waters)/.test('Roger es mi perro y Roger Waters es un músico famoso') //true
```

`?!` realiza la operación inversa, haciendo coincidir si una cadena **no** va seguida de una subcadena específica:

```
/Roger Waters)/

/Roger(?! Waters)/.test('Roger es mi perro') //verdadero /Roger(?!
Waters)/.test('Roger Waters es un músico famoso') //falso
```

Las previsiones utilizan el símbolo `?=` . Ya estaban disponibles.

Lookbehinds, una nueva función, utiliza `?<=` .

```
/(?<=Roger) Aguas/

/(?<=Roger) Waters/.test('Pink Waters es mi perro') //false /(?<=Roger)
Waters/.test('Roger es mi perro y Roger Waters es un músico famoso') / /verdadero
```

Una mirada atrás se niega usando `?<!` :

```
/(?<!Roger) Aguas/

/(?<!Roger) Waters/.test('Pink Waters es mi perro') //true /(?<!Roger)
Waters/.test('Roger es mi perro y Roger Waters es un músico famoso') / /falso
```

La propiedad Unicode escapa `\p{...}` y `\P{...}`

En un patrón de expresión regular, puede usar \d para hacer coincidir cualquier dígito, \s para hacer coincidir cualquier carácter que no es un espacio en blanco, \w para coincidir con cualquier carácter alfanumérico, y así sucesivamente.

Esta nueva característica extiende este concepto a todos los caracteres Unicode introduciendo \p{} y es la negación \P{} .

Cualquier carácter Unicode tiene un conjunto de propiedades. Por ejemplo , Script determina la familia de idiomas, ASCII es un valor booleano que se aplica a los caracteres ASCII, etc. Puede poner esta propiedad entre paréntesis del gráfico, y la expresión regular verificará que sea cierto:

```
/^\p{ASCII}+$/u.prueba('abc') //ÿ /^\p{ASCII}+
$/u.prueba('ABC@') //ÿ /^\p{ASCII} +$/
u.prueba('ABCÿÿÿ//                    ('
```

ASCII_Hex_Digit es otra propiedad booleana que verifica si la cadena solo contiene dígitos hexadecimales:

```
/^\p{ASCII_Hex_Digit}+$/u.test('0123456789ABCDEF') //ÿ //ÿ /
^\p{ASCII_Hex_Digit}+$/u.test('h')
```

Hay muchas otras propiedades booleanas, que simplemente verifica agregando su nombre entre paréntesis del gráfico, incluidas Mayúsculas , Minúsculas , White_Space , Alfabético , Emoji y más:

```
/^\p{Minúsculas}$/u.prueba('h') //ÿ /
^\p{Mayúsculas}$/u.prueba('H') //ÿ

/^\p{Emoji}+$/u.prueba ('H') // ÿ / ^ \ p
{Emoji} + $ / u.prueba ('ÿÿ ÿÿÿ // ('
```

Además de esas propiedades binarias, puede verificar cualquiera de las propiedades de caracteres Unicode para que coincida con un valor específico. En este ejemplo, verifico si la cadena está escrita en alfabeto griego o latino:

```
/^\p{Script=Griego}+$/u.test('ÿÿÿÿÿÿÿÿ') //ÿ /^\p{Script=Latín}
+$/u.test('hey') //ÿ
```

Lea más sobre todas las propiedades que puede usar directamente en la propuesta.

Grupos de captura con nombre

En ES2018, se puede asignar un nombre a un grupo de captura, en lugar de solo asignarle una ranura en la matriz de resultados:

```
const re = /(?<año>\d{4})-(?<mes>\d{2})-(?<día>\d{2})/ const resultado =
re.exec('2015- 01-02')

// resultado.grupos.año === '2015'; //
resultado.grupos.mes === '01'; //
resultado.grupos.dia === '02';
```

La bandera s para expresiones regulares

La bandera s , abreviatura de *línea única,* hace que . para que coincida con los caracteres de nueva línea también. Sin él, el punto coincide con los caracteres regulares pero no con la nueva línea:

```
/hola.bienvenido/.test('hola\nbienvenido') // falso /
hola.bienvenido/s.test('hola\nbienvenido') // verdadero
```

ESSiguiente

¿Que sigue? ESSiguiente.

ESNext es un nombre que siempre indica la próxima versión de JavaScript.

La versión actual de ECMAScript es **ES2018.** Fue lanzado en junio de 2018.

Históricamente, las ediciones de JavaScript se han estandarizado durante el verano, por lo que podemos esperar que **ECMAScript 2019** se lance en el verano de 2019.

Entonces, en el momento de escribir este artículo, se lanzó ES2018 y **ESNext es ES2019.**

Las propuestas al estándar ECMAScript se organizan en etapas. Las etapas 1 a 3 son una incubadora de nuevas características, y las características que alcanzan la etapa 4 se finalizan como parte del nuevo estándar.

En el momento de escribir este artículo, tenemos una serie de características en **la Etapa** 4. Las presentaré en esta sección. Las últimas versiones de los principales navegadores ya deberían implementar la mayoría de ellos.

Algunos de esos cambios son principalmente para uso interno, pero también es bueno saber qué está pasando.

Hay otras características en la Etapa 3, que podrían ascender a la Etapa 4 en los próximos meses, y puede consultarlas en este repositorio de GitHub: https://github.com/tc39/proposals.

Array.prototype.{flat,flatMap}

`flat()` es un nuevo método de instancia de matriz que puede crear una matriz unidimensional a partir de una matriz multidimensional.

Ejemplo:

```
['Perro', ['Oveja', 'Lobo']].flat()
//[ 'Perro', 'Oveja', 'Lobo' ]
```

De forma predeterminada, solo se "aplana" hasta un nivel, pero puede agregar un parámetro para establecer la cantidad de niveles a los que desea aplanar la matriz. Configúralo en `Infinity` para tener niveles ilimitados:

```
['Perro', ['Oveja', ['Lobo']]].flat()
//[ 'Perro', 'Oveja', [ 'Lobo' ] ]

['Perro', ['Oveja', ['Lobo']]].flat(2)
//[ 'Perro', 'Oveja', 'Lobo' ]

['Perro', ['Oveja', ['Lobo']]].plano(Infinito)
//[ 'Perro', 'Oveja', 'Lobo' ]
```

Si está familiarizado con el método JavaScript `map()` de una matriz, sabe que al usarlo puede ejecutar una función en cada elemento de una matriz.

`flatMap()` es un nuevo método de instancia de Array que combina `flat()` con `map()` . Es útil cuando se llama a una función que devuelve una matriz en la devolución de llamada `map()`, pero desea que la matriz resultante sea plana:

```
['Mi perro', 'es genial'].map(palabras => palabras.split(' '))
//[ [ 'Mi', 'perro' ], [ 'es', 'impresionante' ] ]

['Mi perro', 'es genial'].flatMap(palabras => palabras.split(' '))
//[ 'Mi', 'perro', 'es', 'impresionante' ]
```

Encuadernación de captura opcional

A veces no necesitamos tener un parámetro vinculado al bloque catch de un try/catch.

Antes teníamos que hacer:

```
prueba {
    //...
} catch (e) { //
    manejar el error
}
```

Incluso si nunca tuviéramos que usar `e` para analizar el error. Ahora podemos simplemente omitirlo:

```
prueba {
    //...
} catch { //
    manejar el error
}
```

Objeto.fromEntries()

Los objetos tienen un método de entradas() , desde ES2017.

Devuelve una matriz que contiene todas las propiedades propias del objeto, como una matriz de pares [clave, valor] :

```
persona constante = { nombre: 'Fred', edad: 87 }
Objeto.entradas(persona) // [['nombre', 'Fred'], ['edad', 87]]
```

ES2019 presenta un nuevo método Object.fromEntries() , que puede crear un nuevo objeto a partir de dicha matriz de propiedades:

```
const persona = { nombre: 'Fred', edad: 87 } const
entradas = Object.entries(persona) const newPerson =
Object.fromEntries(entries)

persona !== nuevaPersona //verdadero
```

Cadena.prototipo.{trimStart,trimEnd}

Esta característica ha sido parte de v8/Chrome durante casi un año y va a ser estandarizado en ES2019.

recortarInicio()

Devuelve una nueva cadena con espacios en blanco eliminados desde el comienzo de la cadena original

```
'Prueba'.trimStart() //'Prueba'
' Prueba'.trimStart() //'Prueba'
' Prueba '.trimStart() //'Prueba '
'Prueba'.trimStart() //'Prueba'
```

recortarFin()

Devuelve una nueva cadena con espacios en blanco eliminados desde el final de la cadena original

```
'Prueba'.trimEnd() //'Prueba'
' Prueba'.trimEnd() //' Prueba'
' Prueba '.trimEnd() //' Prueba'
' Prueba'.trimEnd() //'Prueba'
```

Símbolo.prototipo.descripción

Ahora puede recuperar la descripción de un símbolo accediendo a su propiedad de descripción en lugar de tener que usar el método toString() :

```
const símboloPrueba = Símbolo('Prueba')
símboloPrueba.descripción // 'Prueba'
```

Mejoras JSON

Antes de este cambio, los símbolos de separador de línea (\u2028) y separador de párrafo (\u2029) no estaban permitidos en cadenas analizadas como JSON.

Al usar JSON.parse(), esos caracteres generaron un SyntaxError, pero ahora se analizan correctamente, según lo define el estándar JSON.

JSON bien formado.stringify()

Corrige la salida de JSON.stringify() cuando procesa puntos de código sustitutos UTF-8 (U+D800 a U+DFFF).

Antes de este cambio, llamar a JSON.stringify() devolvería un carácter Unicode mal formado (un "ÿ").

Ahora esos puntos de código sustitutos se pueden representar de manera segura como
JSON.stringify() , cadenas usando y transformando nuevamente a su representación original usando JSON.parse() .

Función.prototipo.toString()

Las funciones siempre han tenido un método de instancia llamado toString() que devuelve una cadena que contiene el código de la función.

ES2019 introdujo un cambio en el valor de retorno para evitar eliminar comentarios y otros caracteres como espacios en blanco, representando exactamente la función tal como se definió.

Si antes tuviéramos

```
función /* esta es la barra */ barra () {}
```

El comportamiento fue este:

```
bar.toString() //'función bar() {}
```

ahora el nuevo comportamiento es:

```
barra.toString(); // 'función /* esta es la barra */ barra () {}'
```