



DECODED

ACADEMY

FULL STACK DEVELOPER BOOTCAMP

Desarrollamos
{ talento }

JS ES6

—

Compatibilidad

- Compatibilidad en navegadores: [Caniuse](#)
- Compatibilidad con NodeJs: [Node Green](#)

let y const

let

let le permite crear declaraciones limitadas a cualquier bloque, que se denomina ámbito de bloque. En lugar de usar var, que proporciona un ámbito de function, se recomienda usar let en ES6.

```
{  
  let a = 3;  
  console.log(a); // 3  
}  
console.log(a); // Error a no defined
```

let

Si utilizamos var podemos caer en problemas de sobreescritura de variables difíciles de rastrear ya que no dan ningún tipo de error pero alteran el funcinamiento del programa.

```
let a = 10;
var b = 10;
{
  let a = 2;
  var b = 2;
  console.log('let: ', a, ', var: ', b); // let: 2 , var: 2
}
console.log('let: ', a, ', var: ', b); // let: 10 , var: 2
```

const

Otra forma de hacer declaración de ámbito de block es usando const, que crea constantes. En ES6, un const representa una referencia constante a un valor. En otras palabras, el valor no está congelado, sólo la asignación de él. Sigue un ejemplo.

```
const PI = 3.14;  
PI = 3.1419; // TypeError
```

contantes con objetos

```
{  
    const ARR = [5, 6];  
    ARR.push(7);  
    console.log(ARR); // [5, 6, 7]  
    ARR = 10; // TypeError  
    ARR[0] = 3; // valor no es inmutable  
    console.log(ARR); // [3, 6, 7]  
}
```

Algunas cosas a recordar:

- "hoisting" de let y const varían de la forma tradicional de "hoisting" de variables y funciones. Ambos están "hoistados", pero no pueden ser accedidos antes de sus declaraciones, debido a la [Temporal Dead Zone](#)
- let y const su alcance es el del bloque más cercanas. (scope)
- Cuando utilice const, utilice CAPITAL_CASING
- const debe definirse en su declaración

Arrow Functions

Arrow Functions son una notación de corto para escribir funciones en ES6. La definición de arrow function consiste en una lista de parámetros (parametro1, parametro2, etc), seguido de un => y el cuerpo de la function.

```
function(a, b) {  
    return a + b;  
};  
// Implementación con Arrow Function  
(a, b) => {  
    return a + b;  
};
```

return directo

```
function(a, b) {  
    return a + b;  
};  
// Implementación con Arrow Function  
(a, b) => a + b ;
```

Parámetro Único

Si sólo vamos a utilizar un parámetro podemos omitir los paréntesis.

```
function(a) {  
    return a * 2;  
};  
// Implementación con Arrow Function  
a => a * 2;
```

Devolviendo un objeto

```
function(nombre) {  
    return { name: nombre};  
};  
// Implementación con Arrow Function  
nombre => { name: nombre} // Error  
  
nombre => {  
    return {name: nombre};  
}  
nombre => ({name: nombre});
```

Ejemplo real

```
let singulares = ['manzana', 'banana', 'naranja'];  
  
let plurales = singulares.map(fruta => fruta + 's');  
  
console.log(plurales); // ['manzanas', 'bananas', 'naranjas']
```

this en Arrow functions

Arrow functions no sólo reducen el tamaño del código. También están relacionados al comportamiento del binding del this

El comportamiento de las arrow funciones con el this es diferente del de las funciones normales. Cada function en JavaScript define su propio contexto de this, pero las Arrow functions capturan el this de su contexto delimitador (scope padre).

Echa un vistazo al código siguiente:

```
function Persona() {  
    // El constructor de Persona () define this como una instancia de él  
    // mismo  
    this.edad = 0;  
  
    setInterval(function crece() {  
        this.edad++;  
        console.log(this.edad); // NaN  
  
        // Cuando no se usa el modo estrictamente, la function crece ()  
        // define this como el objeto global, que es diferente  
        // del this definido por el constructor de Persona ()  
    }, 1000); // .bind(this)  
}  
var p = new Persona();
```

En ECMAScript 3/5, este problema se podía corregir de dos maneras:

- al asignar el valor de `this` a una variable:

```
function Persona() {  
    // El constructor de Persona () define this como una instancia de él  
    // mismo  
    var self = this;  
    this.edad = 0;  
  
    setInterval(function crece() {  
        self.edad++;  
        console.log(self.edad); // Va incrementando  
    }, 1000);  
}  
var p = new Persona();
```

- o enlazando la función a mano con un `.bind(this)`

```
function Persona() {  
    // El constructor de Persona () define this como una instancia de él  
    // mismo  
    this.edad = 0;  
  
    setInterval(  
        function crece() {  
            this.edad++;  
            console.log(this.edad); // Va incrementando  
        }.bind(this),  
        1000  
    );  
}  
var p = new Persona();
```

Como se mencionó anteriormente, Arrow functions capturan el valor de this del contexto delimitador, por lo que el siguiente código funciona como se esperaba.

```
function Persona() {  
    this.edad = 0;  
  
    setInterval(() => {  
        this.edad++; // `this` se refiere correctamente al objeto persona  
        console.log(this.edad);  
    }, 1000);  
}  
  
var p = new Persona();
```

Más información

Más informaciones sobre 'Lexical this' en arrows function

Parámetros predeterminados

ES6 le permite establecer parámetros por defecto en las configuraciones de funciones. A continuación, una pequeña ilustración.

```
let getPrecoFinal = (precio, impuesto = 0.7) => precio + precio *  
impuesto;  
getPrecoFinal(500); // 850
```

Si le indicamos el valor del parámetro no utilizará el predeterminado.

```
let getPrecoFinal = (precio, impuesto = 0.7) => precio + precio *  
impuesto;  
console.log(getPrecoFinal(500, 0.1)); // 550
```

Operador Spread / Rest

El operador . . . se denomina operador spread o rest, dependiendo de cómo y dónde se utiliza.

spread

Cuando se usa con cualquier iterable, actúa como "spread" en elementos individuales.

```
function foo(x, y, z) {  
  console.log(x, y, z);  
}  
  
let arr = [1, '2', 3, 4];  
foo(...arr); // foo(arr[0], foo[1], foo[2], foo[3])  
foo(1, '2', 3, 4); // 1 '2' 3  
let str = 'hola';  
foo(...str); // 'h' 'o' 'l'
```

rest

El otro uso común de ... es unir una serie de valores en una array. En este caso, el operador se denomina "rest".

```
function f(...args) {  
  console.log(args);  
}  
f(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]  
  
function g(arg1, arg2, ...argRest) {  
  console.log(arg1, arg2, argRest);  
}  
g(1, 2, 3, 4, 5); // 1 2 [3,4,5]
```

Extensiones de Objetos Literales

ES6 permite declarar objetos literales con una sintaxis abreviada para iniciar propiedades de variables y definir métodos de las funciones.

También permite tener índices de propiedades calculadas en una definición de objeto literal.

```
function getCarro(fabricante, modelo, valor) {  
    return {  
        fabricante, // el mismo que el fabricante: fabricante  
        modelo, // el mismo que modelo: modelo  
        valor, // igual que valor: valor  
        ['fabricante ' + fabricante]: true,  
        depreciar() {  
            this.valor -= 2500;  
        },  
    };  
}  
  
let coche = getCarro('Kia', 'Sorento', 40000);  
console.log(coche);  
// { fabricante: 'Kia',  
//   modelo: 'Sorento',  
//   valor: 40000,  
//   'fabricante Kia': true,  
//   depreciar: [Function: depreciar] }  
coche.depreciar();  
console.log(coche.valor); // 37500  
console.log(coche['fabricante Kia']); // true
```

Desestructuración de arrays y objetos

El desestructurado ayuda a evitar la necesidad de variables temporales cuando se trata de objetos y arrays.

```
let arr = [1, 2, 3];
let [a, b, c] = arr;
// let a = arr[0];
// let b = arr[1];
// let c = arr[2];
console.log(a, b, c); // 1 2 3
```

```
let obj = { x: 4, y: 5, z: 6 };

let { x: d, y: e, z: f } = obj;
console.log(d, e, f); // 4 5 6
```

```
let { x, y, z } = obj;
// let {x:x, y:y, z:z} = obj;
console.log(x, y, z); // 4 5 6
```

Destructurando parametros:

```
let arr = [1, 2, 3, 4];
let f = ([x, y, z]) => x + y + z;
console.log(f(arr)); // 6
```

```
let f2 = ([x, ...lastest]) => x;
console.log(f2(arr)); // 1, [2,3,4]
```

```
let obj = { nombre: 'juan', signo: 'libra' };  
let f = ({ nombre, signo }) => nombre + signo;  
console.log(f(obj)); // 'juanlibra'  
  
// es lo mismo que:  
let f2 = (nombre, signo, edad) => nombre + signo;  
console.log(f2(obj.nombre, obj.signo, obj.edad)); // 'juanlibra'
```

Strings Literales y Delimitadores

interpolación

ES6 introduce una forma más fácil de agregar la interpolación analizada automáticamente.

- `soy un string` Contra-aspa se utiliza como delimitador.
- `\${nombreVariable}` utilizado para renderizar las variables

```
let user = 'Kevin';
console.log (`Hola ${user}!`); // Hola Kevin!
```

Multilínea

```
let str1 = 'Kevin \n nueva linea';
let str2 = `Kevin
nueva linea`;
console.log(str1);
console.log(str2);
```

```
/* Ambas renderizan lo mismo:
Kevin
nueva linea
*/
```

for of vs for in

for ... of

- itera en objetos iterables, tipo arrays.

```
let apellidos = ['perez', 'garcia', 'gomez'];
apellidos.hola = 3;
for (let apellido of apellidos) {
  console.log(apellido);
}
// perez
// garcia
// gomez

console.log(apellidos.hola); // 3, es ignorado porque no es parte del
array
```

for ... in

- itera sobre todas las propiedades enumerables del objeto, dando el valor de la clave de objeto a nuestra variable.

```
let apellidos = ['perez', 'garcia', 'gomez'];
apellidos.hola = 3;
for (let apellido in apellidos) {
  console.log('apellidos[' + apellido + '] = ' + apellidos[apellido]);
}
// apellidos.0 = perez
// apellidos.1 = garcia
// apellidos.2 = gomez
// apellidos.hola = 3
```

Map y WeakMap

ES6 introduce una nueva serie de estructura de datos denominados Map yWeakMap. Pero en realidad, utilizamos mapas en JavaScript cada hora. Incluso todo objeto puede ser considerado un Map

Un objeto se hace de índices (siempre cadenas) y valores, mientras que en un Map, cualquier valor (tanto objetos como valores primitivos) pueden ser usados como índice o como valor. Echa un vistazo a este código:

Map

```
let miMap = new Map();  
  
let IndiceString = 'una cadena',  
    IndiceObj = {},  
    Indicefunction = function() {};  
  
// asignando valores  
miMap.set(IndiceString, 'valor asociado con una cadena');  
miMap.set(IndiceObj, 'valor asociado con IndiceObj');  
miMap.set(Indicefunction, 'valor asociado con Indicefunction');  
  
miMap.size; // 3  
  
// recibiendo los valores  
console.log(miMap.get(IndiceString)); // "valor asociado con una cadena"  
console.log(miMap.get(IndiceObj)); // "valor asociado con IndiceObj"  
console.log(miMap.get(Indicefunction)); // "valor asociado con  
Indicefunction"
```

WeakMap

Un WeakMap es un Map donde los índices son referenciados de forma débil, lo que no previene que sus índices sean recogidos por el recolector de basura (garbage collector)

Otra cosa a notar aquí: En un Weakmap, a diferencia delMap, *todo índice debe ser un objeto* .

Un WeakMap sólo tiene 4 métodos:delete (índice),has (índice),get (índice) y set (índice, valor)

```
let w = new WeakMap();
w.set('a', 'b');
// Uncaught TypeError: No válido valor utilizado como mapache débil

var o1 = {},
    o2 = function() {},
    o3 = window;

w.set(o1, 37);
w.set(o2, 'azerty');
w.set(o3, undefined);

w.get(o3); // undefined, ya que es el valor establecido

w.has(o1); // true
w.delete(o1);
w.has(o1); // false
```

Set y WeakSet

Set

Los objetos de tipo Set son colecciones de valores únicos. Los valores duplicados se omite, ya que la colección debe tener sólo valores únicos. Los valores pueden ser primitivos o referencias a objetos.

```
let mySet = new Set([1, 1, 2, 2, 3, 3]);  
mySet.size; // 3  
mySet.has(1); // true  
mySet.add('cadena');  
mySet.add({ a: 1, b: 2 });
```

Usted puede iterar sobre un objeto de tipo set por orden de inserción, usando o forEach, o for ... of.

```
mySet.forEach(item => {
  console.log(item);
  // 1
  // 2
  // 3
  // 'strings'
  // Object { a: 1, b: 2 }
});
```



```
for (let value of mySet) {
  console.log(value);
  // 1
  // 2
  // 3
  // 'strings'
  // Object { a: 1, b: 2 }
}
```

Los objetos de tipo set también tienen los métodos delete () y clear ().

WeakSet

Al igual que WeakMap, WeakSet le permite almacenar de forma débil *objetos* en una colección. Un objeto en el WeakSet sólo ocurre una vez; él es único en la colección de WeakSet.

```
var ws = new WeakSet();
var obj = {};
var foo = {};

ws.add(window);
ws.add(obj);

ws.has(window); // true
ws.has(foo); // false, foo no se ha agregado al set

ws.delete(window); // remove window del set
ws.has(window); // false, window se ha eliminado
```

Literales octetos y binarios

ES6 tiene un nuevo soporte para literales octetos y binarios. Prefijar un número con `0o` o `00` lo convertirá en octal. Solo mira:

```
let oValor = 0o10;  
console.log (el valor); // 8
```

```
let bValor = 0b10; // 0b o 0B para binario  
console.log (bValor); // 2
```

Clases

Las clases de javascript son introducidas en el ECMAScript 6 (2015) y son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos a JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

Las funciones definidas con la palabra clave `static` implementan funciones estáticas / funciones de clase en la clase.

```
class Tarea {  
    constructor(texto, id) {  
        console.log('tarea instanciada! con el texto:', texto);  
        this.texto = texto;  
        this.id = id;  
        this.fecha = new Date();  
    }  
    mostrarId() {  
        console.log(this.id);  
    }  
    fechaHoy() {  
        this.fecha = new Date();  
    }  
    static dimeHoy() {  
        console.log(new Date());  
    }  
}  
console.log(typeof Tarea); // function  
let tarea = new Tarea('comprar leche', 1); // "tarea instanciada!"  
console.log(typeof tarea); // object  
console.log(tarea); // Tarea { texto: 'comprar leche', id: 1 }  
tarea.fechaHoy();  
console.log(tarea.fecha); // 'comprar leche'  
tarea.mostrarId(); // 1  
Tarea.dimeHoy(); // date
```

extends y super en clases

Considere el siguiente código:

```
class Coche {  
    constructor() {  
        console.log('Creación de un nuevo coche');  
    }  
}  
  
class Porsche extends Coche {  
    constructor() {  
        super();  
        console.log('Creación de un Porsche');  
    }  
}  
  
let c = new Porsche();  
// Creación de un nuevo coche  
// Creando un Porsche
```

`extends` permite que las clases hijas hereden de la clase principal en ES6. Es importante notar que el constructor derivado debe invocar `super()`.

Usted también puede llamar al método de la clase primaria en la clase hija usando `super.nombreDelMetodoPadre ()`

[Lea más acerca de las clases aquí](#)

Algunas cosas para recordar:

- La declaración de las clases no están hechas. Primero usted necesita declarar sus clases y luego acceder a ellas. De lo contrario, va a tirar en un ReferenceError.
- No hay necesidad de usar `function` al definir funciones dentro de una definición de clase.

super en objetos

ES6 permite utilizar el método super en objetos (sin clase) en prototipos. Un ejemplo simple a seguir.

```
var padre = {  
  foo: function() {  
    console.log('hola padre');  
  },  
};  
  
var hijo = {  
  bar() {  
    super.foo();  
    console.log('hola hijo');  
  },  
};  
  
Object.setPrototypeOf(hijo, padre);  
hijo.bar(); // hola padre  
// hola hijo
```

Documentación

- [Más información](#)

Símbolo (Symbol)

Un símbolo es un tipo único e inmutable de datos introducidos en el ES6. La utilidad del símbolo es generar un identificador único, que nunca tendrá acceso a él.

Cómo crear un símbolo:

```
var sym = Symbol('descripción de cualquier cosa');  
console.log(typeof sym); // symbol
```

Observe que no se utiliza new en `Symbol (...)`.

Si un símbolo se utiliza como propiedad / índice de un objeto, se almacena de una forma especial en la que la propiedad no aparecerá en las enumeraciones normales de las propiedades de un objeto.

```
var o = {  
    val: 10,  
    [Symbol('random')]: 'Hola, soy un símbolo',  
};  
  
console.log(Object.getOwnPropertyNames(o));  
Español;
```

Para recibir las propiedades de un símbolo, utilice
`Object.getOwnPropertySymbols (o)`

Iteradores

Un iterador accede a los elementos de una colección uno a la vez, mientras mantiene el control de su posición actual en la secuencia. Tiene un método `next ()` que devuelve el siguiente elemento de la secuencia. Este método devuelve un objeto con dos propiedades: `done` y `value`

ES6 tiene `Symbol.iterator` que especifica el iterador predeterminado de un objeto. Cada vez que un objeto necesita ser iterado (como al principio de un `for ... of`), su método `@@ iterator` se llama sin argumentos, y el iterador devuelto se utiliza para obtener los valores a ser iterados.

Vamos a echar un vistazo a una `array`, que es un iterable, y el iterador que se puede producir para consumir sus valores.

```
var arr = [11, 12, 13];
var itr = arr[Symbol.iterator]();

itr.next(); // {value: 11, done: false}
itr.next(); // {value: 12, done: false}
itr.next(); // {value: 13, done: false}

itr.next(); // {value: undefined, done: true}
```

Percibe que usted puede escribir iteradores personalizados definiendo obj [Symbol.iterator] () en la definición del objeto.

Generadores

Las funciones generadoras son una nueva característica que permite que una function genere cuántos valores son necesarios devolviendo un objeto que puede ser iterado para extraer más valores de la function a la vez.

Una function generadora devuelve un **objeto iterable** cuando se llama. Se escribe utilizando la nueva sintaxis * además de la nueva palabra clave `yield` introducida en ES6.

```
function * numerosInfinitos () {  
    var n = 1;  
    while (true) {  
        rendimiento n ++;  
    }  
}  
  
var numbers = numerosInfinitos () // devuelve un objeto iterable  
  
numbers.next () // {value: 1, done: false}  
numbers.next () // {value: 2, done: false}  
numbers.next () // {value: 3, done: false}
```

Cada vez que se conoce el rendimiento, el valor producido se convierte en el siguiente valor en la secuencia.

Se percibe también que los generadores calculan sus valores producidos bajo demanda, lo que permite que representen eficientemente secuencias que son costosas de producirse, o incluso secuencias infinitas.

Promises

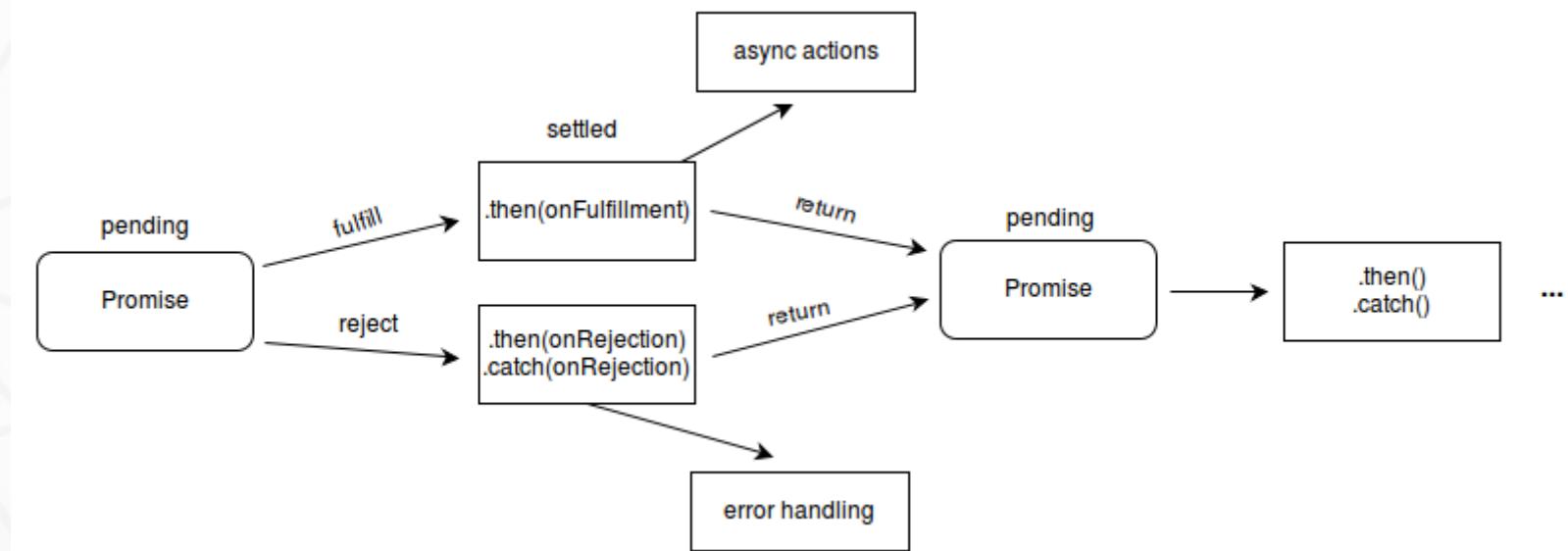
ES6 tiene soporte nativo para las promesas. Una promesa es un objeto que espera que una operación asincrónica sea completada, cuando se completa la promesa se vuelve resuelta o rechazada.

La forma estándar de crear una Promise es usando el constructor `new Promise ()` que acepta un controlador que se da 2 funciones como parámetros. El primer manipulador (generalmente nombrado `resuelve`) es una function para llamar con el futuro valor cuando está listo; y el segundo manipulador (generalmente llamado `reject`) es una function para llamar si la Promise no puede resolver el valor futuro y se rechaza.

```
var p = new Promise (function (resolve, reject) { }
if /* condition */ {
    resolve /* value */; // resuelto con éxito
} else {
    reject /* reason */; // error, rechazado
}
});
```

Una Promesa se encuentra en uno de los siguientes estados:

- pendiente (**pending**): estado inicial, no cumplida o rechazada.
- cumplida (**fulfilled**): significa que la operación se completó satisfactoriamente.
- rechazada (**rejected**): significa que la operación falló.



Una promesa pendiente puede ser cumplida con un valor, o rechazada con una razón (error). Cuando cualquiera de estas dos opciones sucede, los métodos asociados, encolados por el método `then` de la promesa, son llamados. (Si la promesa ya ha sido cumplida o rechazada en el momento que es anexado su correspondiente manejador, el manejador será llamado, de tal manera que no exista una condición de carrera entre la operación asíncrona siendo completada y los manejadores siendo anexados)

Como los métodos `Promise.prototype.then()` y `Promise.prototype.catch()` retornan promesas, éstas pueden ser encadenadas.

then and catch

Cada promise tiene un método llamado `then`, que recibe un par de callbacks. La primera devolución de llamada se llama si se resuelve la promesa, mientras que el segundo se llama si se rechaza.

```
p.then(  
  val => console.log('Promise resuelta', val),  
  err => console.log('Promise rechazado', err)  
)  
// ○  
p.then(val => console.log('Promise resuelta', val)).catch(err =>  
  console.log('Promise rechazado', err)  
)
```

El valor devuelto de then se pasará como valor al siguiente then.

```
var hello = new Promise(function(resuelve, reject) {  
    resuelve('Hola');  
});  
  
hello  
.then(str => `\$ ${str} Mundo`)  
.then(str => `\$ ${str}!`)  
.then(str => console.log(str)); // Hello World!
```

El valor resuelto de la promesa se pasará al siguiente callback para encadenarlos efectivamente. Esta es una forma sencilla de evitar el "callback hell"

```
var p = new Promise(function(resolve, reject) {  
    resolve(1);  
});  
  
var EventualmenteAdicional = val => {  
    return new Promise(function(resolve, reject) {  
        resolve(val + 1);  
    });  
};  
  
p.then(EventualmenteAdicional)  
    .then(EventualmenteAdicional)  
    .then(val => console.log(val)); // 3
```

finally

El método `finally()` puede ser útil si desea realizar algún proceso o limpieza una vez que se ha resuelto la promesa, independientemente de su resultado.

El método `finally(onFinally)` es muy similar a invocar `.then (onFinally, onFinally)`, sin embargo, hay un par de diferencias:

- Al crear una función en línea, puede pasarla una vez, en lugar de verse obligado a declararla dos veces o crear una variable para ella.
- Una devolución de llamada definitiva no recibirá ningún argumento, ya que no hay medios confiables para determinar si la promesa se cumplió o rechazó. Este caso de uso es precisamente cuando no le importa el motivo de rechazo o el valor de cumplimiento, por lo que no es necesario proporcionarlo.

Promise.all

El método `Promise.all(iterable)` devuelve una promesa que termina correctamente cuando todas las promesas en el argumento iterable han sido concluídas con éxito, o bien rechaza la petición con el motivo pasado por la primera promesa que es rechazada.

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

Promise.race

El método `Promise.race(iterable)` retorna una promesa que se cumplirá o no tan pronto como una de las promesas del argumento iterable se cumpla o se rechace, con el valor o razón de rechazo de ésta.

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'uno');
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'dos');
});

Promise.race([p1, p2]).then(value => {
  console.log(value); // "dos"
  // Ambas se resuelven, pero la p2 antes.
});
```

```
// Ejemplo con un resolve y un reject en el mismo método race.  
var p3 = new Promise((resolve, reject) => {  
    setTimeout(resolve, 100, 'tres');  
});  
var p4 = new Promise((resolve, reject) => {  
    setTimeout(reject, 500, 'cuatro');  
});  
  
Promise.race([p3, p4]).then(  
    value => {  
        console.log(value); // "tres"  
        // p3 es mas rápida, así que se resuelve el race  
    },  
    reason => {  
        // No es llamado  
    }  
);
```

```
var p5 = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'cinoc');
});
var p6 = new Promise((resolve, reject) => {
  setTimeout(reject, 100, 'seis');
});

Promise.race([p5, p6]).then(
  value => {
    // No es llamado
  },
  reason => {
    console.log(reason); // "seis"
    // p6 es mas rápida, así que se rechaza
  }
);
```

async await

Cuando usamos la palabra reservada `async` al declarar una función, suceden dos cosas:

- Podemos usar la palabra `await` dentro de esa función para acceder directamente a los valores que devolverían métodos que devuelven promesas.
- La propia función que estamos declarando devuelve su valor de retorno como una promesa.

Cuando usamos la palabra reservada await al invocar una función:

- Si la función sin await hubiera devuelto una promesa satisfecha, la llamada devolverá el valor de esa promesa.
- Si la función sin await hubiera devuelto una promesa rechazada, lanzará un error con la razón del rechazo.
- Si la función sin await hubiera devuelto un valor que no es una promesa, la llamada devolverá ese mismo valor (esto incluye undefined en llamadas a funciones sin valor de retorno).

```
let getFilm = () =>
  new Promise((res, rej) => {
    res('Matrix');
  });
let getMain = film =>
  new Promise((res, rej) => {
    film === 'Matrix' ? res('Neo') : rej('404');
  });
async function queue() {
  var film = await getFilm(); //Supongamos que toca 'Matrix'
  var main = await getMain(film); //Neo
  console.log(main);
}
queue(); //escribirá 'Neo' en la consola.
```

async function devuelve un promesa

```
async function get() {  
    return 100;  
}  
  
get().then(console.log); //100
```

await fuera de async

Intentar utilizar await en cualquier lugar que no sea una función declarada como **async** **resultará en un error**. Por el contrario, se puede utilizar la palabra reservada **async** para declarar funciones sin usar await en ningún momento; no es que sea muy útil, sin embargo.

Gestión de errores

Si hay algún error dentro de una función declarada como `async`, éste es automáticamente atrapado y devuelto como una promesa rechazada.

```
async function throwError() {  
  throw new Error(  
    'Esto no aparecerá como un Error sino como una promesa rechazada'  
  );  
}
```

```
throwError().catch(console.log); //Error: Esto no aparecerá como un error  
sino como una promesa rechazada...
```

De forma inversa, cuando una llamada precedida por await devuelve una promesa rechazada, ésta se convierte en un error:

```
function rejectedPromise() {  
    return new Promise(function(resolve, reject) {  
        reject('promesa rechazada');  
    });  
}  
  
async function get() {  
    try {  
        await rejectedPromise(); // al rechazarse la promesa, esta linea  
        lanza un error.  
    } catch (error) {  
        console.log(error);  
    }  
}  
  
get(); //Promesa rechazada
```

Resumen async await

- Cuando se llama a una función async, esta devuelve un elemento Promise.
- Cuando la función async devuelve un valor, Promise se resolverá con el valor devuelto.
- Si la función async genera una excepción, la Promise se rechazará con el Error generado.
- Una función async puede contener una expresión await, la cual pausa la ejecución de la función asíncrona y espera la resolución de la Promise pasada y, a continuación, reanuda la ejecución de la función async y devuelve el valor rechazado.

Soporte actual

- await