ES6. El remozado JavaScript.

JavaScript ha ido incorporando muchas novedades al lenguaje a través de las últimas especificaciones del estándar EcmaScript. En este tutorial le doy un repaso a las principales novedades, desde un punto de vista eminentemente práctico, ilustrando cada caso con ejemplos.

Juan Antonio Jiménez Torres

Fecha: 17/04/2018

Índice

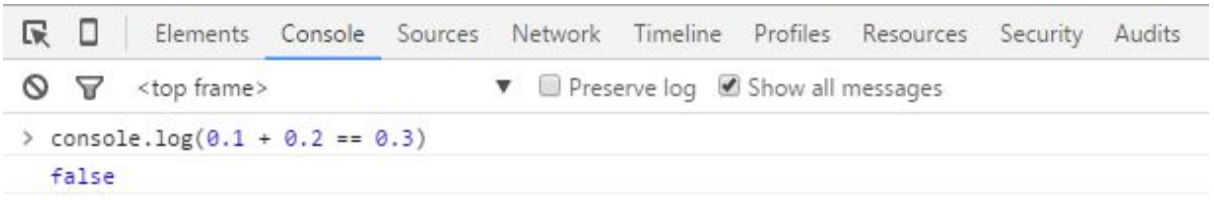
1. Introducción

Ya hace tiempo que la nueva especificación que rige JavaScript viera la luz, y llegó pertrechada de novedades. Sin embargo, hay quien opina que su venida se demoró demasiado, y ahora tiene que ganar terreno frente a otros lenguajes, que parecen estarle ganando por la mano. Pero tiene la ventaja y virtud de que es un estándar, así que acabará sobreponiéndose.

Ningún programador que hubiera trabajado con JavaScript se mostraba indiferente: o bien eran auténticos defensores a ultranza de dicho lenguaje, o bien sus más fervientes detractores. Yo me encuentro en el primer grupo, pero comprendo perfectamente al segundo.

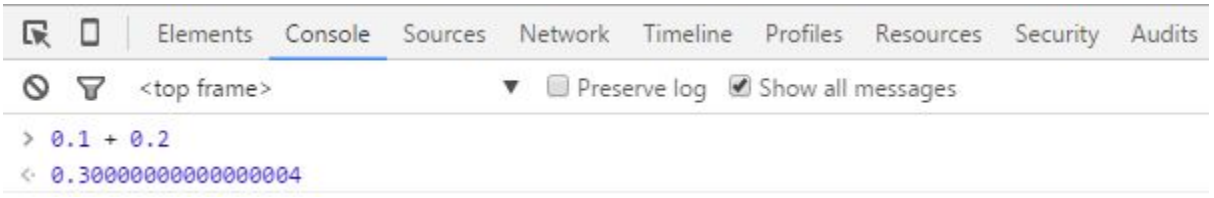
El mayor punto de frustración con el que se topa un desarrollador bregado en programación orientada a objetos, es que JavaScript le parece un "quiero y no puedo", que carece de herencia, de polimorfismo y de tipos, e intenta trasladar sus hábitos, a veces mediante ingeniosas triquiñuelas, a un lenguaje que puede instanciar un objeto sin ni siquiera haber definido su clase. Y es que JavaScript usa un paradigma de programación orientada a prototipos, que está pensado para ejecutarse y no consumir grandes recursos. Simplemente es diferente. Es otra cosa. No es Java, ni se comporta como Java. Pero en cuanto lo entiendes, empiezas a apreciarlo, y a disfrutar de sus matices.

Claro, que a veces tiene cosas que exasperan a cualquiera. Por ejemplo, abra la consola de su navegador y compruebe cómo 0.1 + 0.2 no es 0.3...



- *¿¿¿Cómo???*

A ver, a ver, vamos más despacio, ¿cómo que es falso?



¡Ahhhhhhhh! ¿y eso? ...

Es como el chiste: *¿Qué prefieres? ¿velocidad o precisión?*

En fin, eso es porque implementa el estándar definido en la aritmética de punto flotante IEEE-754, y es inherente a muchos lenguajes de programación, incluidos todos los lenguajes EcmaScript *(JavaScript, ActionScript, etc...)*. Si quieres conocer más detalles de esta imprecisión, en el post *"Why is 0.1+0.2 not equal to 0.3 in most programming*

languages?" lo explica muy bien.

Pero vamos a empezar con las novedades de esta nueva implementación de JavaScript. El nombre oficial es ECMA 262, 6ª Edición, pero es conocido como ES6, ECMAScript 6, o ES2015, dado que su especificación se liberó en Junio de 2015.

Su principal cambio, respecto a versiones anteriores de la especificación, es que introduce una nueva sintaxis manteniendo la retrocompatibilidad, es modular, y esta vez sí que tiene una forma directa de definir clases y herencia.

2. Declaración de variables.

Una de las razones por las que JavaScript no gusta a muchos programadores, es que el ámbito de las variables no es el mismo que en otros lenguajes, siendo en muchos casos variables globales.

```
function foo(){
  for (var i=0; i<10; i++){
    console.log(i);
  }

  console.log(i + 3); //devuelve 13
}
```

En el ejemplo anterior vemos que la variable i sigue existiendo fuera del bucle y además conserva su último valor 10.

Desde la concepción de JavaScript, las variables declaradas dentro de un bloque se mueven al principio del bloque en tiempo de interpretación. Eso hace que siempre las podamos referenciar sin que dé error. Este proceder se llama **hoisting**. En el caso que nos ocupa sería:

```
function foo(){
  var i;
  for (i=0; i<10; i++){
    console.log(i);
  }

  console.log(i + 3); //devuelve 13
}
```

En ES6 sí introduce una nueva forma de declarar variables, que es mediante la palabra reservada *let*. El ámbito de las variables así definidas está limitado al bloque donde se han declarado y no hacen *hoisting*.

```
function foo(){
  var a = 3;
  if (a < 5){
    let b = 10;
    console.log(a + b); //devuelve 13
  }

  console.log(b); //b no está definida fuera del if
}
```

La variable *b* no está viva fuera del bloque definido por las llaves del *if*. Si se ejecuta está función el intérprete de JavaScript debería dar un *ReferenceError* al intentar referirse a una variable que no existe.

```
> foo()
  13
✖ ▶ Uncaught ReferenceError: b is not defined
      at foo (<anonymous>:8:15)
      at <anonymous>:1:1
```

Esto es especialmente útil en los bucles que llaman a funciones. Recordemos que JavaScript permite la programación funcional, y pasar otra función como argumento de una función. Un caso particular de este hecho son los *callbacks*.

```
let vocales = ['a','e','i','o','u'];

function foo(){
  for (var i=0; i < 5; i++){

    bar(function(){
      console.log('La vocal es :' + vocales[i]);
    });
  }
}
```

En este caso es evidente y casi inmediato, pero imaginemos por un momento que la función *bar()* es muy compleja. Cuando se empiece a ejecutar el primer *callback* es muy probable que el bucle haya terminado, y como *i* es una variable que sigue existiendo al salir del bucle puede acabar llamando al *callback* con el valor equivocado.

Antaño, esto se solucionaba pasando la variable como parámetro al *callback*. Pero ahora podemos usar let, y el ámbito estará definido en la ejecución de dicho callback.

```
let vocales = ['a','e','i','o','u'];

function foo(){
  for (let i=0; i < 5; i++){

    bar(function(){
      console.log('La vocal es :' + vocales[i]);
    });
  }
}
```

Las variables declaradas con let aunque puede ser sobreescritas, no pueden volver a ser declaradas en el mismo bloque.

```
> let version = '';
  version = '1.2';

  let version = '1.3';
Uncaught SyntaxError: Identifier 'version' has already been declared
```

Entonces, ¿por qué el siguiente código no da error?

```
> let version = '';
  version = '1.2';

  if (version != ""){
      let version = '1.3';
  }
```

Porque se pueden declarar una variable con el mismo nombre usando *let* siempre que estén declaradas en distintos bloques. Al tener distinto ámbito, se trata de variables diferentes aunque tengan el mismo nombre. Otra cosa, es que por claridad y legibilidad sea conveniente.

```
let version = '';
version = '1.2';
console.log(version); //version vale 1.2

if (version != ""){
  let version = '1.3';
  console.log(version); //version vale 1.3
}
```

```
console.log(version); //version vale 1.2
```

3. Constantes

Por fin se pueden definir constantes en *JavaScript* mediante la palabra reservada *const*. Al declararla es obligatorio definir su valor, y luego sólo podrá ser consultada.

```
const MAX_SIZE = 4;

for (let i=0; i < MAX_SIZE; i++){
  console.log(i);
}
```

Una vez declarada no podrá ser modificada, y arrojará un error si se intenta.

```
> const MAX_SIZE = 4;
  MAX_SIZE = 10;
✖ ▶ Uncaught TypeError: Assignment to constant variable.
      at <anonymous>:2:10
```

Por otro lado, el ámbito de las constantes se limita al bloque donde se han declarado.

```
const MAX_SIZE = 4;

for (let i=0; i < MAX_SIZE; i++){
  const MSG = "La variable es: "
  console.log(MSG + i);
}
console.log(MSG); //daría error porque MSG aquí no está definida
```

El resultado de ejecutar el código anterior es:

```
  La variable es: 0
  La variable es: 1
  La variable es: 2
  La variable es: 3
✖ ▶ Uncaught ReferenceError: MSG is not defined
      at <anonymous>:7:13
```

Hay que tener en cuenta que las constantes también pueden ser objetos y como constantes, si intentamos volver a asignarles un valor nos dará el correspondiente error:

```
> const GAME = {lives: 3, width: 500, height: 300};
  GAME = {lives: 5, width: 500, height: 300};
✖ ▶ Uncaught TypeError: Assignment to constant variable.
      at <anonymous>:2:6
```

Sin embargo, y aunque no podemos asignar un nuevo objeto a esa constante, sí que podemos modificar las propiedades de dicho objeto. Hay que entender cómo funciona por debajo: con la palabra reservada *const* estamos asignando un nombre a una dirección de memoria. Y a ese nombre no le podremos asignar otra dirección de memoria distinta al tratarse de una constante. Pero al tratarse de un objeto, si podemos modificar sus propiedades sin modificar su dirección de memoria. Así, podríamos hacer lo siguiente y no daría error:

```
const GAME = {lives: 3, width: 500, height: 300};
console.log(GAME.lives);  //el resultado será 3

GAME.lives = 5;
console.log(GAME.lives);  //el resultado será 5
```

Si lo que queremos es definir un objeto inmutable como constante sería utilizando *Object.freeze()*.

```
const GAME = Object.freeze({lives: 3, width: 500, height: 300});

//es GAME inmutable?
Object.isFrozen(GAME);
```

Y comprobaríamos que dicho objeto es inmutable con *Object.isFrozen()*.

4. Funciones

Otro de los puntos donde se han introducido bastantes cambios es en las funciones.

4.1. Valores por defecto

Una cosa que siempre ha tenido JavaScript, y que ha sido fruto de controversia entre los programadores de otros lenguajes, es que se puede definir una función con más

parámetros que con los que se la llama.

```
function ejemplo(nombre, apellidos) {
  if (apellidos != undefined){
    console.log("Hola " + nombre + " " + apellidos);
  } else {
    console.log("Hola " + nombre);
  }
}

ejemplo("Manuel");
ejemplo("Pepito", "Grillo");
```

Pero ahora, como pasaba en otros lenguajes *EcmaScript* como Flex, se pueden declarar valores por defecto para esos parámetros.

```
function ejemplo(nombre, apellidos = "García") {
  console.log("Hola " + nombre + " " + apellidos);
}

ejemplo("Manuel");                //Hola Manuel García
ejemplo("Pepito", "Grillo");      //Hola Pepito Grillo
```

A las funciones en JavaScript que pueden recibir un número variable de parámetros se les llama *funciones variádicas*. Y es una de los puntos que aporta potencia al lenguaje, pero que lo convierte en incómodo a aquellos acostumbrados a otros lenguajes de programación, que para nada esperan este comportamiento.

4.2. Parámetro REST y operador SPREAD

Ahora en una función se permite que el último argumento que se le pasa, sea especial y se denota con tres puntos suspensivos. Eso nos indica que este último argumento, recibirá todos los parámetros de más de la función en forma de array.:

```
function ejemplo (param1, param2, ...restParams) {
  console.log(restParams);
}
ejemplo('a','b','c','d','e','f');
//el resultado es el array ["c", "d", "e", "f"]
```

¿Y la operación contraria? Imaginemos que lo que tenemos es una función variática que espera un número variable de parámetros en lugar de un array. Podemos convertir

fácilmente el array a una lista de parámetros con el operador SPREAD, que también se denota con tres puntos suspensivos.

Esto puede parecer que da lugar a equívocos, pero fijémonos que uno se utiliza para definir una función, y el otro para invocarla.

```
let aVocales = ["a", "e", "i", "o", "u"];

function bar(){
  let txt = '';
  for(let i in arguments){
    txt += arguments[i];
  }
  console.log(txt); // la salida por consola es aeiou
}

function foo(a){
  console.log(a.length);  //a es un array
  bar(...a);    //esto es como llamar a bar('a','e','i','o','u')
}

foo(aVocales);
```

4.3. Funciones Arrow

Se denominan funciones arrow, se denotan por el símbolo => y se utilizan de la siguiente manera. Las siguientes expresiones son equivalentes

```
let multiplicar = function (a,b) {
                    return a * b;
                  }

let multiplicar = (a,b) => a * b;

console.log(multiplicar(3,5)); //15
```

Se utiliza mucho en las funciones anónimas que se definen como callbacks de otras funciones porque la función arrow sí preserva el contexto del *this*, al contrario de lo que pasa con las funciones estándar.

Por ejemplo, muchas veces habremos visto código de terceros que hacen cosas como *var that = this* y luego se refieren a ese *that*. Eso lo hacen porque la palabra *this* hace referencia al contexto del callback y no de la función externa. Veamos un ejemplo

```
let ImageHandler = {
```

```
  id: "",

  init: function (id) {
      this.id = id;
      let that = this;
    document.addEventListener("click", function (event) {
      that.show(event);   // se usa that aquí, porque this haría
referencia al contexto de la función anónima
    }, false);
  },

  show: function (event) {
    console.log("Se ha disparado el evento de typo " + event.type);
  }
};
```

Sin embargo, con una función Arrow quedaría:

```
let ImageHandler = {
  id: "",

  init: function (id) {
      this.id = id;
    document.addEventListener("click", (event) => this.show(event),
false);
  },

  show: function (event) {
    console.log(event.type + " has been fired for " + this.id);
  }
};
```

Cuando la función arrow sólo recibe un argumento, no es necesario escribir paréntesis:

```
event => this.show()
```

Pero sí cuando no hay argumentos o son más de uno.

```
() => "Soy Íñigo Montoya";
(a,b) => a * b;
```

5. Objetos y Arrays

5.1. Forma abreviada para quitar repeticiones

En numerosas ocasiones nos hemos encontrado devolviendo un objeto con nombres y valores repetidos, donde el primero hace referencia a la propiedad, y el segundo es la variable con su valor

```
return {width: width, height: height};
```

Pero ahora es posible devolver una versión abreviada, donde la propiedad será como el nombre de la variable que contiene el valor. En el siguiente ejemplo, ambas funciones son equivalentes:

```
function getSizes(){
  let width = window.innerWidth;
  let height = window.innerHeight - 200;
  return {width: width, height: height};
}

function getSizes(){
  let width = window.innerWidth;
  let height = window.innerHeight - 200;
  return {width, height};
}
```

Y también funciona en la asignación de variables:

```
  let sizes = {width, height};
  console.log(sizes.width);
```

En lugar de

```
  let sizes = {width: width, height: height};
```

5.2. Asignación por Destructuring

Se pueden asignar variables recorriendo las propiedades de un objeto de una en una, o bien se puede usar la asignación por destructuring, donde se asigna a cada variable la propiedad correspondiente.

```
let sizes = getSizes();
let width = sizes.width;
let height = sizes.height;

let {width, height} = getSizes();
console.log(width);
console.log(height);
```

No es obligatorio usar todas las propiedades del objeto. Se puede utilizar sólo una parte para hacer la asignación por destructuring.

```
let {width} = getSizes();
```

La asignación por destructuring también funciona para *arrays*.

```
let aVocales = ["a", "e", "i", "o", "u"];

let [z, y, x, w, v] = aVocales;
console.log(z);    //a
console.log(y);    //e
console.log(x);    //i
console.log(w);    //o
console.log(v);    //u
```

¿y funcionará con el operador Spread?

```
let aVocales = ["a", "e", "i", "o", "u"];

let [z, y, ...x] = aVocales;
console.log(z);    //a
console.log(y);    //e
console.log(x);    //["i", "o", "u"]
```

Pues sí que funciona, al menos en Google Chrome, aunque según la especificación, eso es de la próxima versión de *EcmaScript* ES7.

5.3. Iteraciones: *for... in* vs *for... of*

En las versiones previas de JavaScript ya se podía recorrer un array con bucles de tipo *for .... in*

Realmente, recorre todas las propiedades enumerables de un objeto en un orden indeterminado.

```
let jugador = {
  name: "Sergio Callejas",
  posicion: "4",
  esTitular: true,
  dorsal: 33
}

for (propiedad in jugador){
  console.log(propiedad, eval('jugador.' + propiedad));
}

for (propiedad in jugador){
  console.log(propiedad, jugador[propiedad]);
}
```

De los dos bucles for, el primero nos obliga a acceder a la propiedad mediante la triquiñuela del eval(), porque si accedemos directamente mediante jugador.propiedad, no evalúa el valor de "propiedad" en cada vuelta del bucle, y busca literalmente el atributo llamado "propiedad", y nos devolvería undefined.

Pero todos los objetos pueden recorrerse como arrays asociativos. Es decir, podemos acceder al nombre de ambas maneras.

```
console.log(propiedad.name)
console.log(propiedad["name"])
```

Y esto es lo que usamos en el segundo bucle.

Pero ¿y qué pasa si lo que tenemos es un array de objetos?

```
let equipo = [
  {name: "David Gómez", posicion: "5"},
  {name: "Sergio Callejas", posicion: "4"},
  {name: "Javier Rodríguez", posicion: "3"},
  {name: "Laura Fernández", posicion: "2"},
  {name: "Manuel Fernández", posicion: "1"}
]

//recorremos por posición que ocupa en el array
for (i in equipo){
  console.log(equipo[i]);
}
```

Pero en ES6 existe una nueva forma de recorrer con *for...of*, que en lugar de por índice, recorre directamente los objetos que contiene:

```
//recorremos directamente los objetos del array
for (jugador of equipo){
  console.log(jugador.name);
}
```

Esto de for...of lo podemos hacer sobre los arrays porque son objetos iterables. Pero no es lo habitual en JavaScript. Si definimos un objeto, e intentamos recorrer sus propiedades, nos dará un error similar al siguiente: *"TypeError: jugador[Symbol.iterator] is not a function".*

```
> let jugador = {
    name: {
      nombre: "David",
      primerApellido: "Goméz",
      segundoApellido: "García"
    },
    posicion: {
      tipo: "pivot",
      numero: 5
    },
    esTitular: true
  };

  for (prop of jugador){
      console.log(prop);
  }
✖ ▶ Uncaught TypeError: jugador[Symbol.iterator] is not a function
      at <anonymous>:14:14
```

Pero siempre podemos crearlo.

5.4. Entendiendo Iteradores, y como crearlos

Hay algunos tipos de objetos que tienen implementado el iterador. Vamos a ver qué pinta tiene este iterador

```
> let equipo = [
    {name: "David Gómez", posicion: "5", esTitular: true, dorsal: "00"},
    {name: "Sergio Callejas", posicion: "4", esTitular: true, dorsal: "33"},
    {name: "Javier Rodríguez", posicion: "3", esTitular: true, dorsal: "23"},
    {name: "Laura Fernández", posicion: "2", esTitular: true, dorsal: "05"},
    {name: "Manuel Fernández", posicion: "1", esTitular: true, dorsal: "17"},
  ]

  let iterador = equipo[Symbol.iterator]();

  console.log(iterador.next());
▶ Object {value: Object, done: false}
```

Lo primero que se puede observar, es que la variable *equipo*, que es un array, sí implementa *Symbol.iterator*. Y al preguntar por *iterador.next()* nos devuelve un objeto con dos propiedades: *value*, que tiene el primer elemento del array, y *done* un booleano que nos indica si ya hemos recorrido todos los elementos.

Pero si usamos un objeto en lugar de un array, esto no funcionará porque no implementa un iterador. Pero podemos definirlo nosotros. Imaginemos el siguiente objeto no iterable:

```
let equipo = {
  posteBajo: {name: "David Gómez", dorsal: "00"},
  posteAlto: {name: "Sergio Callejas", dorsal: "33"},
  alaPivot: {name: "Javier Rodríguez", dorsal: "23"},
  escolta: {name: "Laura Fernández", dorsal: "05"},
  base: {name: "Manuel Fernández", dorsal: "17"}
};
```

Y queremos iterar sobre sus atributos. Podríamos definir el siguiente iterador:

```
equipo[Symbol.iterator] = function(){
  let listaPosiciones = Object.keys(this);
  let i = 0;

  let next = () => {
    let posicion = listaPosiciones[i];
    return {
      value : this[posicion],
      done: (i++ >= listaPosiciones.length)
    };
  };

  return {next};
};
```

Una vez ya definido, podríamos recorrer las propiedades del objeto con un bucle de tipo *for... of*:

```
for (miembro of equipo){
  console.log(miembro);
};
```

```
> for (miembro of equipo){
    console.log(miembro);
  };
  ▶ Object {name: "David Gómez", dorsal: "00"}
  ▶ Object {name: "Sergio Callejas", dorsal: "33"}
  ▶ Object {name: "Javier Rodríguez", dorsal: "23"}
  ▶ Object {name: "Laura Fernández", dorsal: "05"}
  ▶ Object {name: "Manuel Fernández", dorsal: "17"}
```

5.5. Generadores

En esta nueva versión de JavaScript se introducen los generadores, que tienen todo el aspecto de una función, pero con algunas particularidades. La primera, y que es la que lo define, es que tras la palabra reservada *function* hay un asterisco *. Y en lugar de *return* usa la palabra reservada *yield*. Además, todo generador implementa un iterador.

```
function* miGenerador(){
      yield "uno";
      yield "dos";
      yield "tres";
}

let ejemplo = miGenerador();
```

Cuando se invoca a un generador, como en el ejemplo anterior, no hace nada de nada.

Como los iteradores, funciona cuando se llama a su método *next()*, entonces devuelve el primer *yield*, y se guarda el estado a la espera de que en algún momento se vuelva a llamar a *next()*. Cuando se llame a next() por segunda vez, devolverá el segundo *yield*. Y así hasta que no haya más.

```
> ejemplo.next()
◂ ▶ Object {value: "uno", done: false}
> ejemplo.next()
◂ ▶ Object {value: "dos", done: false}
> ejemplo.next()
◂ ▶ Object {value: "tres", done: false}
> ejemplo.next()
◂ ▶ Object {value: undefined, done: true}
```

Esto tiene múltiples implicaciones, ligadas a la preservación del estado y la asincronía. Una que se me ocurre es para implementar un asistente, wizard o sistema de pasos:

```
let wizardBox = {
      step1 : function(){
            console.log("paso 1 de 3");
      },
      step2 : function(){
            console.log("paso 2 de 3");
      },
      step3 : function(){
            console.log("paso 3 de 3");
      }
}

function* wizard(){
      yield wizardBox.step1();
      yield wizardBox.step2();
      yield wizardBox.step3();
}

let w = wizard();

w.next();
//ejecutamos el primer paso y hacemos lo que sea

w.next();
//se ejecuta el segundo paso continuado donde lo dejamos

w.next();
//se ejecuta el tercer paso continuando desde el punto anterior
```

Y otra de las más utilizadas es para escribir el iterador que poníamos de ejemplo en el punto anterior, pero de una forma mucho más simple:

```
equipo[Symbol.iterator] = function *(){
  let listaPosiciones = Object.keys(this);

  for (let posicion of listaPosiciones){
      yield this[posicion];
  }
};
```

5.6. Los nuevos tipos Map y WeakMap

En esta versión de JavaScript se introducen cuatro tipos nuevos de objetos: Map, WeakMap, Set y WeakSet.

Los mapas son un conjunto de pares clave, valor que implementan algunas operaciones:

- para añadir elementos al mapa
- para obtener un elemento del mapa
- para consultar si está un elemento en el mapa
- para quitar un elemento del mapa
- para vaciar el mapa

```
let equipacion = new Map();
equipacion.set('camiseta',{size:'XXL', dorsal:'02'});
equipacion.set('zapatillas',46);
equipacion.set('pantalones','54');

equipacion.get('zapatillas');     //46
equipacion.get('camiseta').size;  //XXL
equipacion.size;                  //3

equipacion.has('pantalones');     //true
equipacion.delete('pantalones')   //quitamos el elemento pantalones
equipacion.has('pantalones');     //false
equipacion.clear();               //vaciamos el mapa
equipacion.size;                  //0
```

En el caso concreto de los mapas, son iterables, y admiten pares de objetos, tanto en las claves como en los valores:

```
let equipacion = new Map();
```

```
equipacion.set(equipo.posteBajo, {camiseta: 'XXXL', pie: 46});
equipacion.get(equipo.posteBajo);
```

Por el contrario los *WeakMap* son un tipo de mapas "*débiles*". La primera diferencia es que las claves sólo pueden ser objetos. Nunca strings. Tampoco son iterables, y como tal no puedes consultar sus claves con *Object.keys()*. Además, hay que tener en cuenta que sus objetos se encuentran referenciados de forma "*weak*" (*débil*). Esto quiere decir, que el objeto que forma parte de la clave, si no es referenciado por nadie más que el WeakMap, es susceptible de que el *Garbage Collector* lo elimine.

Por tanto, el uso de los *WeakMap* tienen sus pros y sus contras.

En contra tienen que no son colecciones que se puedan recorrer con bucles de tipo *for...of* , hay que conocer la clave a priori para recuperar su valor. Tiene el peligro de que si nadie referencia a la clave, se pierda el valor asociado. A cambio, tiene la ventaja de no permitir *memory leaks*.

```
let equipacion = new WeakMap();

equipacion.set("posteBajo", {camiseta: 'XXXL', pie: 46});
//TypeError: la clave sólo puede ser un objeto

equipacion.set(equipo.posteBajo, {camiseta: 'XXXL', pie: 46});

console.log(equipacion.get(equipo.posteBajo));
//{camiseta: 'XXXL', pie: 46}

equipo.posteBajo = undefined;      //borramos el objeto posteBajo
console.log(equipacion.get(equipo.posteBajo));  //undefined
```

5.7. Los nuevos tipos Set y WeakSet

Al igual que los mapas, los conjuntos también se introducen en esta especificación de ES6. Consisten en colecciones de objetos o valores primitivos y no se pueden repetir.

```
let vocales = new Set();
vocales.add('a');
vocales.add('e');
vocales.add('i');
vocales.add('o');
vocales.add('u');
vocales.add('a');
```

```
console.log(vocales.size); //5

for (let vocal of vocales){
      console.log(vocal);
}
```

Los WeakSet son como los Set pero con la misma línea argumental que los WeakMap:

- No implementan un iterador
- Sólo pueden contener objetos
- Y su referencia a sus valores es débil

6. Clases

En javaScript clásico hay dos corrientes principales para tratar con objetos:

a) una definir directamente el objeto sin preocuparnos de cuál es su clase, y si esta nos es necesaria, preguntar por su prototipo y modificarlo a conveniencia.
b) Definir una función que será el constructor del ejemplo.

Si vamos a tratar con un objeto, que tendrá una única instancia, se define el objeto directamente:

```
let tablero = {
  filas: 20,
  columnas: 20,
  bombas: 15

  render: function(){
    ...
  }
};
```

Pero cuando podemos tener varias instancias será necesario definir una forma de crear objetos. Hasta ahora se hacía mediante una función constructora con la palabra reservada new.

```
function Pelota(id, punto, direccion){
  this.id = id;
  this.punto = punto;
  this.direccion = direccion;
```

```
  this.render = function(){
    ...
  };
}

let pelota1 = new Pelota(1, {x:0, y:0}, {deltaX: 4, deltaY: 3} );
let pelota2 = new Pelota(2, {x:0, y:0}, {deltaX: -1, deltaY: 2} );
```

Pero imaginemos que en lugar de dos instancias del objeto "pelota" hay miles de instancias, todas moviéndose, y rebotando por la pantallita.

Cada método que hemos añadido a la clase Pelota, tiene una dirección de memoria distinta apuntándole a él. En otros lenguajes como Java ésto es normal, pero en los navegadores históricamente se ha controlado mucho la memoria, ya que la aplicación se ejecuta en el navegador, que hasta no hace mucho, tenía unos recursos muy limitados.

Entonces, ¿es necesario que haya miles de funciones iguales que renderizan la posición de la pelota? ¿o sería necesario con definir un única función externa?

```
function Pelota(id, punto, direccion){
  this.id = id;
  this.punto = punto;
  this.direccion = direccion;
}

function renderPelota(pelota){
  ...
}
```

Lo único malo de hacerlo así es que la función parece que no tiene nada que ver con el objeto. Lo que se hacía era obtener el prototipo del objeto, y añadirle la función en cuestión.

```
Pelota.prototype.render = function(){
  ...
}


let pelota1 = new Pelota(1, {x:0, y:0}, {deltaX: 4, deltaY: 3} );
let pelota2 = new Pelota(2, {x:0, y:0}, {deltaX: -1, deltaY: 2} );

pelota1.render();
pelota2.render();
```

Y con los prototipos llega uno de los mayores fracasos de JavaScript en cuanto a su popularidad. Y es que ha fracasado totalmente a la hora de comunicar a programadores de otros lenguajes de programación su funcionamiento. Durante años, los programadores de Java han intentado programar JavaScript como si fuera un lenguaje de Programación Orientado a Objetos, como el que ellos conocen, como Java. Y no. JavaScript es un lenguaje orientado a objetos sin clases. En su lugar todo objeto tiene un prototipo, que no es sino otro objeto.

6.1. Entendiendo los prototipos

El prototipo de un objeto es un objeto que nos devuelve su constructor y sus métodos. Con lo cual se puede recrear el objeto a partir de su prototipo.

```
Rectangulo = function(alto, ancho){
  this.alto = alto;
  this.ancho = ancho;
}
Rectangulo.prototype.area = function(){
  return this.alto * this.ancho;
}

let rectangulo1 = new Rectangulo(3,5);
console.log(rectangulo1.area());  //15
```

Si preguntamos por el prototipo de la clase *Rectangulo* nos devuelve su constructor y el método *área*.

```
> Rectangulo.prototype
<· ▼Object i
    ▶area: ()
    ▶constructor: (alto, ancho)
    ▶__proto__: Object
```

Y podemos definir la clase *Cuadrado* en base al prototipo de *Rectángulo*:

```
Cuadrado = function(lado){
  this.lado = lado;
  this.alto = lado;
  this.ancho = lado;
}
Cuadrado.prototype = Object.create(Rectangulo.prototype);
let cuadrado1 = new Cuadrado(2);
```

Aunque no hayamos definido el área del cuadrado, está definida por la herencia del rectángulo:

```
console.log(cuadrado1.area()); //4
```

Y vemos que cuadrado1 es una instancia de *Cuadrado*, pero también es una instancia de *Rectángulo*. En este caso se dice que está en la *cadena de prototipos*.

```
console.log(cuadrado1 instanceof Cuadrado); //true
console.log(cuadrado1 instanceof Rectangulo); //true
```

Y ésto es uno de los mayores motivos de desasosiego para los programadores de otros lenguajes, que esperan que JavaScript se comporte como ellos esperan siguiendo el paradigma de POO que conocen. Es por eso, que la comunidad de desarrolladores, en lugar de aprender el lenguaje, ha ejercido presión para que éste evolucionara por unos caminos donde no les fallara la intuición. Y por eso, en ES6 se incluyen clases y modularización, que per sé no es malo, y así se acerca a una comunidad mucho más amplia.

6.2. Clases

Esta necesidad se ha visto satisfecha en la versión ES6. Y así, ahora tenemos clases para definir objetos.

```
class Rectangulo {

  constructor(alto, ancho){
    this.alto = alto;
    this.ancho = ancho;
  };

  area(){
    return this.alto * this.ancho;
  };

};

let r = new Rectangulo(2,3);
console.log(r.area()); //6
```

Realmente JavaScript no ha cambiado su paradigma orientado a prototipos. Esta forma

de escribir clases es meramente azúcar sintáctico que por debajo se traduce en lo visto en el punto anterior, pero al menos, es una notación con la que la mayoría de desarrolladores están más familiarizados.

Y nos permite definir una herencia de una forma mucho más amigable.

```
class Cuadrado extends Rectangulo {

  constructor(lado){
    super(lado,lado);
    this.lado = lado;
  };

};

let c = new Cuadrado(3);
console.log(c.area());  //9
```

6.3. Módulos

En ES6 se introduce el concepto de módulos que no es otro que permitir trocear el código en distintos ficheros, de forma que se permita la exportación de clases, funciones, variables o constantes e importarlos desde otro módulo.

Así, nuestro ejemplo anterior quedaría guardado en un fichero *poligonos.js* con el siguiente código:

```
export class Rectangulo {

  constructor(alto, ancho){
    this.alto = alto;
    this.ancho = ancho;
  };

  area(){
    return this.alto * this.ancho;
  };

};


export class Cuadrado extends Rectangulo {

  constructor(lado){
    super(lado,lado);
```

```
    this.lado = lado;
  };

};
```

O si preferimos exportar separar la definición de las clases, de cuáles exportamos:

```
class Rectangulo {
  ...
};


class Cuadrado extends Rectangulo {
  ...
};

export {Rectangulo, Cuadrado};
```

Y para importarlas desde otro fichero y hacer uso de ellas:

```
import { Rectangulo, Cuadrado } from 'poligonos.js';

let c = new Cuadrado(2);
```

Y esto se puede hacer tanto a nivel de clases, como de funciones, variables o constantes. Todo muy útil, y muy esperado por una comunidad de desarrolladores acostumbrados a modularizar su código. Las ventajas son incontestables, empezando desde el principio de responsabilidad única, a una simple organización de código. Pero aquí es donde viene la mala noticia, y es que aunque está definido en el estándar, no se ponen de acuerdo cómo implementarlo en los navegadores. Y ninguno de ellos lo implementa de forma nativa.

De todas formas, vamos a meditar qué significa implementar ésto en el navegador: la realidad es que la página html importaría un fichero JavaScript, que a su vez importaría a otros ficheros JavaScripts resolviendo todas sus dependencias e importando para ello más ficheros JavaScript. La resolución en cascada podría traducirse en miles de llamadas al servidor para traerse pequeños módulos JavaScript. Y aquí es donde viene uno de los cambios que han revolucionado el mundo front, y es que antes para ejecutar JavaScript te valía un navegador, y ahora se requiere un proceso de "compilación", que normalmente realiza un *TaskRunner* tipo *grunt*, *gulp*, *parcel* o *webpack*, que empaqueta todo nuestro código diseminado por módulos en un único fichero *(bundle)* entendible por el navegador, que a veces ha sufrido procesos de minificación, de validación, transpilación a ES5 o varias cosas más, y que veremos más adelante.

Hay un pollyfill para cargar módulos de forma estándar.

7. Otras novedades en el lenguaje

Son muchas las novedades que se incluyen en esta nueva especificación, y que vienen heredadas de lo que pasa en otros lenguajes.

7.1. Promesas

A medida que las interfaces son más atractivas, requerimos que la interacción con las mismas sea más inmediata, consiguiendo una buena experiencia de usuario, que de otro modo sería pobre. Pero al contrario de lo que ocurre en otros lenguajes, donde se pueden ejecutar varios hilos de ejecución concurrentemente, JavaScript tiene un único hilo de ejecución de forma que hasta que no ejecuta una instrucción, no pasa a la siguiente. Eso puede llegar a producir retrasos indeseables, e incluso pérdida de control por parte del usuario.

```
instruccion1();
instruccion2();
```

Hay que entender el hilo de ejecución como una cola. Cuando vinculamos una acción a un evento, lo que estamos haciendo es encolar la acción al final de la cola de instrucciones a ejecutar, y hasta que no se termine, no devolverá el control. Por ejemplo, definimos que al pulsar una tecla se ejecute una acción. Ésta no es inmediata y se encola.

A veces nos interesa cierta asincronía adrede, cuando queremos que se ejecute cierta instrucción pero nos interesa devolver el control inmediatamente, para no dejar bloqueada la pantalla.

```
instruccion1(function(){
  instruccion2();
});
```

El código anterior ejecuta la instruccion1, encola la instruccion2 para que se ejecute cuando pueda, y devuelve el control a la línea siguiente.

Este tipo de llamadas se llaman callbacks y se ejecutan cuando sea, pero la instruccion1 ya ha devuelto el control. Introduce la asincronía, y es la forma clásica en las versiones anteriores de JavaScript.

Pero tiene un problema bastante engorroso de lidiar con él, y es que los callbacks se pueden anidar con un nivel de complejidad que es el llamado *Infierno de Callbacks.*

```
getData(url, function (error, data) {
  if (error) {
```

```
    console.log('Error: ' + error);
  } else {
    return fetchData(data, function(erro,dat){
      if(erro) {
        console.log('Error: ' + erro);
      } else {
        return getFetchData(dat, function(err,da){
          if (err) {
            console.log(err);
          } else {
            return getFetchGetDta(da, function(er,d){
              if (er) {
                console.log(er);
              } else {
                return d;
              }
            });
          }
        });
      }
    });
  }
});
```

Una promesa es una nueva forma de implementar esta asincronía pero sin usar callbacks, lo que hace que se lea mucho más fácil.

```
getData(url)
  .then(fetchData)
  .then(getFethcData)
  .then(getFetchGetData)
  .catch(function(error){
    console.log('Error: ' + error);
  });
```

El ciclo de vida de una promesa es muy sencillo: cuando se crea se queda en estado *pending* esperando a que se resuelva o se rechace.

```
function getData(url){
  return new Promise(function(resolve, reject){
    ...
    if (data) {
      resolve(data);
```

```
    } else {
      reject(error);
    }

  });
}
```

Y podríamos invocar así:

```
getData(url)
  .then(instruccion1)
  .catch(function(error){
    console.log('Error: ' + error);
  });
```

Una promesa tiene 3 posibles estados: pendiente, resuelta y rechazada. Cuando la invocamos queda en estado pendiente, y se inicia la asincronía, y cuando se obtiene un resultado, se invoca con éste a *resolve()* o si se produce un error a *reject()*. El caso es que desde fuera no podemos consultar el estado de una promesa, pero podemos saber cuando cambia su estado mediante el *then()*.

```
new Promise(function(resolve, reject) {
    console.log("Promesa pendiente");
    resolve();
}).then(function() {
    console.log("Promesa resuelta");
});

console.log("Hola mundo!");
```

Y el resultado de ejecutar el código anterior es:

```
Promesa pendiente
Hola mundo!
Promesa resuelta
```

Lo que ha pasado es lo siguiente: lo primero que se ha lanzado es la función que se le pasa a la promesa, y pinta *"Promesa pendiente"*. Luego se devuelve el control a la línea de ejecución y pasa a la siguiente instrucción y escribe en la consola *"Hola Mundo!"*, y acto seguido se resuelve la promesa, en nuestro caso cuando invocamos a *resolve()*, y entonces se lanza lo que hay en el *then()* y escribe *"Promesa resuelta"* en la consola.

El argumento que se le pasa a la promesa se llama función ejecutora. Cualquier error que se produzca en un bloque *try ... catch* se resuelve la promesa lanzando el método

reject().

```
let miPromesa = new Promise(function(resolve, reject) {
  throw new Error("Se rechaza sin llamar al siguiente resolve");
  resolve();
});
```

Cuando se rechaza una promesa se puede ejecutar de dos formas: como la segunda función de un then() o como un catch(). El siguiente código sería equivalente:

```
miPromesa.then(function() {
  console.log("Promesa resuelta");
}, function(error){
  console.log("Error: " + error.message);
});

miPromesa.then(function() {
  console.log("Promesa resuelta");
}).catch(function(error){
  console.log("Error: " + error.message);
});
```

Y escribiría por consola: *"Error: Se rechaza sin llamar al siguiente resolve".*

Se pueden tener varias promesas, y querer que se lance algo cuando se cumplan todas o que se rechace con que una falle.

```
Promise.all([miPromesa1, miPromesa2]).then(function() {
  console.log("Todas mis promesas resueltas");
}).catch(function(error){
  console.log("Error: " + error.message);
});
```

7.1.1. Async y await

Esta funcionalidad aún no está disponible en ES6, aunque muchos transpiladores ya permiten que la usemos. Se espera que esté en la próxima liberación del estándar ES7, o ES2017.

Básicamente es lo mismo que las promesas, para introducir asincronía, pero con una sintaxis ligeramente distinta y a mi parecer más sencilla. Comienza con preceder una función de la palabra *async* para indicar que en esa función hay alguna parte que es asíncrona. Y luego, en la parte que debe esperar a resolver una promesa, se precede de la palabra reservada *await* indicando que debe esperar hasta que ésta queda resuelta.

```
async function(url){
  let miElemento = document.getElementById("texto");
  let texto = await getData(url);
  miElemento.innerHTML = texto;
}
```

7.2. Proxies e Interceptores

Esta nueva versión de JavaScript nos aporta una funcionalidad existente en otros lenguajes y que se echaba en falta y es la posibilidad de definir Proxies e Interceptores sobre objetos. Como no se trata de azúcar sintáctico, sino que es algo totalmente nuevo de ES6, es una característica que no puede ser emulada por un polyfill, ni transpilada a ES5.

En esencia un Proxy es un objeto que recubre a otro ya existente, y que permite interceptar algunas operaciones sobre él. Veámoslo con un ejemplo:

```
let cuadrado = {
  lado: 4,
  area: function() {
      return this.lado * this.lado;
  }
}

let cuadradoHandlers = {
  get: function(target,prop){
    console.log("Se consulta el valor de la propiedad: " + prop);
  }
}

let pCuadrado = new Proxy(cuadrado,cuadradoHandlers);
```

¿Qué pasa cuando consultamos el valor del lado del nuevo objeto?

```
> console.log(pCuadrado.lado);
  Se consulta el valor de la propiedad: lado
```

¿Y cómo es que no ha devuelto 4? Sólo ha interceptado el get, y no hemos devuelto la respuesta de lo que vale la propiedad *cuadrado.lado*

```
let cuadradoHandlers = {
  get: function(target,prop){
```

```
    console.log("Se consulta el valor de la propiedad: " + prop);
    return target[prop];
  }
}
```

Ahora sí. Si repetimos la prueba obtendremos el valor del lado. ¿Y si consultamos el valor del área, que no es una propiedad sino un método?

```
> console.log(pCuadrado.lado);
  Se consulta el valor de la propiedad: lado
  4
<· undefined
> console.log(pCuadrado.area());
  Se consulta el valor de la propiedad: area
  Se consulta el valor de la propiedad: lado
  Se consulta el valor de la propiedad: lado
  16
<· undefined
```

Cuando consultamos el área nos retorna el valor correcto, sin embargo vemos que se han escrito por consola tres consultas a propiedades: una para el área, y como para calcular el *area* hay que hacer *lado * lado*, se accede dos veces más a la propiedad lado. De ahí que salga tres veces.

Aquí estamos usando el interceptor de forma inocente para mostrar un mensaje por consola, pero se puede utilizar para alterar el resultado de métodos y operaciones, para formatear campos numéricos, fechas, etc... o incluso para internacionalizar mensajes de error.

A todos los efectos es como si fuese el mismo objeto, que cuando accedemos a través de su versión proxy intercepta la consulta de sus propiedades. Pero centrémonos en esa afirmación: "es como si fuese el mismo objeto". No es exactamente cierta, pero se parece mucho. Si cambiamos la propiedad en el objeto proxy, cambia en el objeto al que se refiere, y viceversa.

```
pCuadrado.lado = 27;
console.log(cuadrado.lado);  //27

cuadrado.lado = 3;
console.log(pCuadrado.lado);  //3
```

Se puede utilizar para hacer validaciones al momento de asignar un valor a una propiedad. En el ejemplo que nos ocupa, se podría validar que el lado fuera siempre un número entero positivo:

```
let cuadradoHandlers = {
  set: function(target, prop, value){
    if (prop == "lado"){
      if (!Number.isInteger(value) || value<0) {
        throw new TypeError("El lado debe ser un número positivo");
      }
      target["lado"] = value;
    }
  }
}
```

```
> cuadrado.lado = "A" // nos deja poner un valor no numérico
"A"
> pCuadrado.lado = "A" // valida, y arroja un error
▶ Uncaught TypeError: El lado debe ser un número positivo
    at Object.set (<anonymous>:12:11)
    at <anonymous>:1:16
```

Hay muchos métodos de los objetos que se pueden interceptar. La documentación del objeto Proxy y sus handler es muy exhaustiva.

7.3. Strings y Literales

Otra de las novedades que se echaban en falta era poder definir cadenas que ocuparan varias líneas. Los *templates strings* vienen a solventar ese problema sin necesidad de reemplazar /n por
 ni similares. Sólo hay que encerrar el texto entre comillas invertidas.

```
let saludo = `Hola a todos.

Os deseo un buen fin de semana.

Y este testo está escrito
en varias líneas`;

console.log(saludo);
```

También sirven para interpolar resultados:

```
let precio = 2950;
const IVA = 1.21

let cadena = `El precio es ${precio*IVA} euros`;
console.log(cadena);
```

Aquí se mostrará por consola *El precio es 3569.5 euros*. Hay que tener en cuenta, que estas expresiones se evalúan sólo cuando se definen, no cuando se consultan.

```
precio = 10;
console.log(cadena);
```

Si asignamos a precio 10 y volvemos a mostrar por consola la cadena nos seguirá mostrando el mismo mensaje anterior de *El precio es 3569.5 euros*. No cambia aunque ahora el precio sea 10.

Si queremos hacer que se evalúen cada vez se pueden usar las *plantillas de texto con postprocesado* (tagged template literals), que no son otra cosa que definir una función con la plantilla que se quiere que se devuelva, y por otro llamar a esa función pasándole los parámetros con los que debe interpretarse. En el ejemplo que tenemos entre manos:

```
let precio = 2950;
const IVA = 1.21


function cadenaPrecio(cadenas, ...values){
  return `El precio es ${values[0]} euros`;
}

console.log(cadenaPrecio `${precio*IVA}`);  //El precio es 3569.5 euros

precio = 10;
console.log(cadenaPrecio `${precio*IVA}`);  //El precio es 12.1 euros
```

Con los template literals los caracteres como retornos de carro /n y tabuladores /t se convierten en retornos de carro y tabuladores de verdad.

```
> let precio = 2950;
  const IVA = 1.21;
  let saludo = `Gracias por su visita.\n\tEl precio es:\n\t${precio*IVA} euros`;
  console.log(saludo);
  Gracias por su visita.
      El precio es:
      3569.5 euros
```

Pero a veces, nos interesa evaluar una interpolación, pero preservar estos caracteres. Existe un nuevo método dentro del objeto String que nos permite preservar el texto tal y como se escribió, pero evaluando las interpolaciones: *String.raw*.

```
String.raw`Gracias por su visita.\n\tEl precio es:\n\t${precio*IVA}
euros`;
```

Y por consola saldrá *Gracias por su visita.\n\tEl precio es:\n\t3569.5 euros.*

7.3.1. Nuevos métodos de String

Se añaden nuevos métodos, que si bien se podían hacer antes con el *indexof* de ES5, facilitan la vida al programador:

- startsWith: comprueba si una cadena comienza por otra
- endsWith: comprueba si una cadena finaliza por otra
- Includes: comprueba si una cadena está incluida en otra
- repeat: repite una cadena n veces

```
'abcdef'.startsWith('a');    //true
'abcdef'.startsWith('abc');  //true
'abcdef'.startsWith('bc');   //false
'abcdef'.includes('bc');     //true
'abcdef'.endsWith('bc');     //false
'abcdef'.endsWith('ef');     //true
'abc'.repeat(3)               //abcabcabc
```

7.4. Novedades en los objetos Math y Number

La primera novedad que incluye ES6 es la posibilidad de definir enteros por su forma octal o binaria. Hasta ahora se podía hacer en hexadecimal, usando el prefijo *0x* para indicar que lo que seguía era una representación hexadecimal. Ahora se puede usar el prefijo *0b* para indicar que lo que sigue es la representación binaria de un número y *0o* para indicar que lo que sigue es la representación octal de un número.

```
let a = 0xFF;        //255 en hexadecimal
let b = 0o377;       //255 en octal
```

```
let c = 0b11111111;  //255 en binario

console.log(a === b);      //true
console.log(a === c);      //true
```

Se introducen muchos métodos en *Number*, la mayoría ya viejos conocidos. Funciones existentes, que ahora se engloban bajo el objeto *Number* para darle una perspectiva más propia de un paradigma orientado a objetos. Así la función *isNaN()* se puede usar como *Number. isNaN()*. Lo mismo con *isFinite()*, que ahora se puede usar como *Number.isFinite()*.

Se introduce *Number.isInteger(value)* y *Number.isSafeInteger(value).*

Pero para mi, uno de las novedades introducidas más importantes, es *Number.EPSILON*. Una constante, que nos sirve para comparar con el error de imprecisión por la operativa de coma flotante. La descripción exacta de este valor es: la distancia mínima entre 1 y el siguiente número representado en como flotante, que es *2,22e-16*. Una unidad muy pequeña. No confundir con el número e.

Y de alguna forma nos viene a recordar el problema de comparar números en JavaScript. Ya vimos que *0.1 + 0.2* no es *0.3* y sin embargo *0.25 + 0.125* sí es *0.375*. Vamos a profundizar en el problema.

```
console.log(0.1 + 0.2 == 0.3);       //false
console.log(0.25 + 0.125 == 0.375);  //true
```

La representación de *0.1* y *0.2* en coma flotante es eso, una representación. Una aproximación al número real más cercano que se puede representar. En este caso los decimales más cercanos con representación son:

```
> let a = 0.1;
  console.log(a.toExponential(20));

  let b = 0.2;
  console.log(b.toExponential(20));

  let c = 0.3
  console.log(c.toExponential(20));

  console.log((a+b-c).toExponential(20));
  1.00000000000000005551e-1
  2.00000000000000011102e-1
  2.99999999999999988898e-1
  5.55111512312578270212e-17
```

Vemos que la distancia entre la representación del resultado de sumar *0.1 + 0.2* y la representación de *0.3* es del orden de 5.55×10^{-17}

Sin embargo, *¿por qué no pasa con 0.25 y 0.125?* Pues porque tienen representación binaria exacta al poderse representar su denominador como potencias de 2. Uno es 2^{-2} y el otro 2^{-3}

Eso quiere decir que siempre que hagamos aritmética en coma flotante, ya sea en JavaScript, como en cualquier otro lenguaje que use este estándar para representar números, debemos tener en cuenta la precisión.

7.4.1 La clase Math

Incluye numerosas constantes y funciones matemáticas entre las que en esta versión se han añadido todas las razones trigonométricas hiperbólicas:

```
Math.cosh(1)     //coseno hiperbolico
Math.sinh(1)     //seno hiperbólico
Math.tanh(1)     //tangente hiperbólica
Math.acosh(1.5430806348152437)    //arco coseno hiperbólico
Math.asinh(1.1752011936438014)    //arco seno hiperbólico
Math.atanh(0.7615941559557649)    //arco tangente hiperbólica
```

```
> Math.cosh(1)     //coseno hiperbolico
< 1.5430806348152437
> Math.sinh(1)     //seno hiperbólico
< 1.1752011936438014
> Math.tanh(1)     //tangente hiperbólica
< 0.7615941559557649
> Math.acosh(1.5430806348152437)    //arco coseno hiperbólico
< 1
> Math.asinh(1.1752011936438014)    //arco seno hiperbólico
< 1
> Math.atanh(0.7615941559557649)    //arco tangente hiperbólica
< 0.9999999999999999
```

Nuevas funciones para manejar logaritmos que no estén en base e:

- *Math.log10()*: logaritmo en base 10
- *Math.log2()*: logaritmo en base 2

Mención aparte merecen

- *Math.log1p(x)* es el logaritmo neperiano de x+1. Se suele usar para valores muy pequeños que tienen problemas de precisión.
- *Math.expm1()* es e^x *-1*

Y otras funciones son:

- Math.*hypot(x,y):* es la hipotenusa. Es decir la raiz cuadrada de la suma del cuadrado de sus catetos.

```
Math.hypot(3,4) // el resultado es 5
```

ya que $\sqrt{3^2 + 4^2} = 5$

Pero además, es el módulo del vector libre (3,4). De hecho, podemos usar esta función para obtener el módulo de un vector de n-dimensiones, ya que admite n argumentos.

```
Math.hypot(a1, a2 ,..., an) // el resultado es √(a1^2 + a2^2 + ... + an^2)
```

- *Math.trunc(x):* devuelve la parte entera del número x
- *Math.sign(x):* devuelve si un número es positivo (1), negativo (-1) o cero (0).
- *Math.imul(x,y):* devuelve el resultado de una multiplicación de 32 bits
- *Math.fround(x):* devuelve la representación en coma flotante más cercana del número x
- *Math.cbrt(x):* devuelve la raíz cúbica de x
- *Math.clz32(x):* devuelve el número de ceros por la izquierda en la representación binaria del número x con 32 bits. Aún no se me ocurre en qué casos puede ser útil este método.

8. Epílogo

Durante muchos años, JavaScript se ha mantenido estable y sin evolucionar, pero el redescubrimiento de sus posibilidades por parte de los desarrolladores ha empujado una evolución, que lo han rejuvenecido ampliando sus posibilidades. Sin duda es un lenguaje que está vivo y que a día de hoy es cambiante.

Recomiendo emplear el estándar, que para eso está. Y antes de profundizar en otros frameworks que se apoyan en ES6, conocer de primera mano la tecnología que subyace a estos frameworks.

Muchos desarrolladores se refieren despectivamente a desarrollar con JavaScript a pelo como "*Vainilla JS*" en referencia a la carestía del lenguaje. Siento discrepar, y amo profundamente este lenguaje, y su versatilidad con el tipado dinámico. Mientras pueda, defenderé el uso del estándar frente a frameworks que se pueden quedar obsoletos o desmantenidos en cualquier momento.

9. Referencias

- Este tutorial online: "ES6: el remozado JavaScript"
- ECMAScript 6 — New Features: Overview & Comparison
- Especificación de ECMAScript 2019 (borrador)
- Tutorial de Mozilla sobre JavaScript
- Tabla de compatibilidad de los navegadores con ES6