# ARRAYS

|  |
| --- |
| **for ... in**  no nos preocupamos de actulizar el index, pues cambia de forma automática  let names = [ 'Pepe', 'Juan', 'Mateo'];  for( i in names){  console.log(i, names[i]);  } |
| **for ... in con objetos**  let book = {  title: 'how to make a bomb',  author: 'Pepito Grillo',  ISBN: '134342'  }  for (b in book){  console.log(b,book[b]);  } |
| **for...of**  se usa para hacer bucles con cualquier objeto que sea iterable  for( name of names ){  console.log(name);  } |
| **forEach**  recorre el array, y aplica un método para cada elemento del bucle  const numbers = [4,5,6,3,2];  numbers.forEach( number => {  console.log(number);  })  // forEach obteniendo index  numbers.forEach( (number,index)=>{  console.log('index =' + index,number);  }) |
| **map**  Retorna un array cuyo resultado es la aplicación de una función sobre cada elemento del array original  let dup = numbers.map( n => n\*2 );  console.log(dup); // [8,10,12,6,4] |
| **filter**  Retorna un array. Aplica una condición y extrae los elementos que la cumplan  Si no se encuentra nada, se retorna []  let filtered = numbers.filter( n => n%2 === 0 );  console.log(filtered); // [4,6,2]  // es lo mismo  filtered = numbers.filter( (n)=>{  if(n%2===0){  return (true);  }  });  console.log(filtered); |
| **find**  busca un elemento del array que cumpla con la condición que se indica.  sólo devuelve 1 resultado  si no encuentra nada, retorna undefined  let found = numbers.find( n => n===6);  console.log(found); // 6 |
| **findIndex**  como find, pero retorna el índice del elemento en el array  si no existe, retorna -1  let indexFound = numbers.findIndex( n=> n===6 );  console.log(indexFound); // 2 |
| **reduce**  extrae un único valor aplicando una función a los elementos del array  se suele definir un valor inicial que, al aplicarse el reduce irá variando  se suele usar para sumar todos los valores de un array de números  let arrayNum = [4,5,6,3,2];  let sum = arrayNum.reduce( (a,b)=> a+b );  console.log(sum); // 20  sum = arrayNum.reduce( (a,b)=> a+b,5 );  console.log(sum); // 25 |
| **includes**  retorna true o false si el array contiene un elemento que cumpla la condición  let included = arrayNum.includes( 2 );  console.log(included); // true |
| **sort**  ordena los elementos de un array según el criterio que se le de en la función  para ordenar de forma ascendente:  const sorted = arrayNum.sort( (a,b)=> a>b);  console.log(sorted); |

# CLASES

Aunque se permita usar la palabra class. JS sigue gestionando todo usando funciones

**constructor**

- método especial que se llama al crearse una instancia de la clase

- se usa para definir las variables de instancia usando 'this'

**geters, setters**

Podemos declarar métodos get/set para los atributos, lo cual obliga a que estos sean

declarados con un nombre distinto, habitualmente se añáde un \_

**static**

pueden crearse métodos definidos static.

permite llamar a estos métodos sin necesidad de crear una instancia

class Device {

constructor( name, price = 25, age){

this.name = name;

this.price = price;

this.\_age = age;

console.log('Device was created');

}

get age(){

return this.\_age;

}

set age(age){

this.\_age = age;

}

start(){

console.log('Starting...')

}

doStuff(){

console.log(`${this.name} doing some stuf`);

}

}

const myPhone = new Device('Peter',900);

myPhone.age = 2;

myPhone.doStuff();

console.log(myPhone.age); //2

// static

class Logger {

static prefix = "Logger>";

static log(msg){

console.log(this.prefix + msg);

}

}

Logger.log('Hola'); 'Logger>Hola';

console.log(Logger.prefix); 'Logger>'

# SYMBOL

Es un nuevo dato primitivo (como number, String, boolean, null, undefined)

Permite identificar de forma inequívoca e inmutable la propiedad de un objeto

const something = Symbol();

const other = Symbol('just a description');

Imaginemos un objeto que representa una mochila.

En esa mochila vamos a ir metiendo otros objetos

backpacks = {

'bottle': {brand:'laken', color:'silver'},

'helmet': {brand:'petzl', color:'blue'}

}

Si necesitamos introducir dos objetos con el mismo identificador, uno machaca al otro.

Para evitar esto se usan los símbolos, pues crean un identificador único

backpacks = {

[Symbol('bottle')]:{brand:'laken', color:'silver'},

[Symbol('helmet')]:{brand:'petzl', color:'blue'},

[Symbol('helmet')]:{brand:'petzl', color:'yellow'},

}

Symbol permite crear un tipo de dato que representa un identificador único

El parámetro que se le pasa a Symbol no es más que una descripción

const helmet1 = Symbol('helmet');

const helmet2 = Symbol('helmet');

console.log(helmet1==helmet2); // false

# ITERADORES

ES6 introduce dos interfaces o protocolos para que cualquier objeto pueda ser iterable, es decir, que se pueda recorrer pasando por todos sus elementos. --> iterable, iterator

Para que un objeto sea iterable ha de implementar el método next()

Se puede añadir este método usando la constante:

[Symbol.iterator]

**Symbol.iterator** es una función sin argumetos que genera un iterator.

Gracias a este iterator, ya se dispondrá del método **next**

Cuando se llama al método next() de un iterator, se devuelve un objeto

con dos valores:

value: el siguiente valor del conjunto

done: true,false, según si quedan o no más elementos

const numbers = [5,42,15];

const iterator = numbers[Symbol.iterator]();

console.log(iterator.next()); // {value:5, done:false}

console.log(iterator.next()); // {value:42, done:false}

console.log(iterator.next()); // {value:15, done:false}

console.log(iterator.next()); // {value:undefined, done:true}

# SETS

Nueva estructura de datos en los que los elementos NO SE PUEDEN REPETIR

**Es como un array sin valores repetidos**

WeakSet

es un caso específico de Set

sólo puede contener objetos

no es iterable

no tienen método clear

si se elimina uno de sus elementos, el GarbageCollector de JS lo borra

const players = new Set();

const bestPlayers = new Set([13,99,15,4]);

**// add**

players.add(34);

players.add(45);

players.add(45);

players.add(6);

players.add(99);

console.log(players); // Set(4){ 34, 45, 6, 99}

**// delete**

players.delete(6);

console.log(players); // Set(3){ 34, 45, 99 }

**// has**

console.log(players.has(99)); // true

**// clear**

bestPlayers.clear();

console.log(bestPlayers); // Set(0){}

**// obtención de los valores de forma tradicional**

**// values**

**// keys**

for(p of players){

console.log(p);

}

for(p of players.values()){

console.log(p);

}

for(p of players.keys()){

console.log(p);

}

**// con el método values o keys, se crea un objeto SetIterator**

console.log(players.values()); // [Set Iterator] { 34, 45, 99 }

console.log(players.keys()); // [Set Iterator] { 34, 45, 99 }

**// podemos ahora llamar al método next**

**const playerIterator = players.values();**

console.log(**playerIterator.next()**); // { value: 34, done: false }

console.log(playerIterator.next()); // { value: 45, done: false }

console.log(playerIterator.next()); // { value: 99, done: false }

console.log(playerIterator.next()); // { value: undefined, done: true }

# MAP

Estructura de **almacenamiento clave/valor**. Similares a los Hashtables o arrays relacionales

**// No se pueden inicializar con datos, se han de agregar con .set**

const swords = new Map();

swords.set('Orcrist','Thorin');

swords.set('Foehammer','Gandalf');

swords.set('Sting','Bilbo');

console.log(swords);

/\*

Map(3) {

'Orcrist' => 'Thorin',

'Foehammer' => 'Gandalf',

'Sting' => 'Bilbo'

}

\*/

**// get**

console.log(swords.get('Sting')); // Bilbo

**// delete** - retorna true,false si existía o no

console.log(swords.delete('Orcrist')); // true

console.log(swords.delete('tizona')); // false

**// has**

console.log(swords.has('Orcrist')); // false

**// size**

console.log(swords.size); // 2

**// clear** - Borra todo

//swords.clear();

**// Iteraciones con for...of**

for(s of swords){

console.log( s );

}

/\*

[ 'Foehammer', 'Gandalf' ]

[ 'Sting', 'Bilbo' ]

\*/

for(sk of swords.keys()){

console.log(sk);

}

/\*

Foehammer

Sting

\*/

for(sv of swords.values()){

console.log(sv);

}

/\*

Gandalf

Bilbo

\*/

**// Iteraciones con forEach**

swords.forEach( (value,key)=>{

console.log(key,value);

});

**// Se pueden usar los iteradores keys y values**

const **swordValuesIterator = swords.values();**

console.log(**swordValuesIterator.next()**); // { value: 'Gandalf', done: false }

console.log(swordValuesIterator.next()); // { value: 'Bilbo', done: false }

console.log(swordValuesIterator.next()); // { value: undefined, done: true }

const swordKeysIterator = swords.keys();

console.log(swordKeysIterator.next()); // { value: 'Foehammer', done: false }

console.log(swordKeysIterator.next()); // { value: 'Sting', done: false }

console.log(swordKeysIterator.next()); // { value: undefined, done: true }

# PROMESAS

Permiten **crear código asíncrono** fácil de gestionar.

Consiste en que **una función, en vez de retornar un resultado, retorna una Promesa**

const hello = ( msg ) => {

// return a new Promise

return new Promise( ( resolve, reject ) => {

setTimeout(()=>{

if(msg.length>0){

resolve(msg);

}else{

reject(Error('msg not valid'));

}

},1000);

});

}

la cláusula then acepta dos parámetros

- un método que recoge el resultado de la promesa al llamar a **resolve**

- un método que recoge el valor de **reject**

hello('hola')

.then( (msg)=>{

console.log('ok, msg is: ' + msg);

},(error)=>{

console.log(error);

});

**cláusula catch para recoger errores**

hello('')

.then((msg)=>{

console.log('ok, msg is: ' + msg);

})

.catch((err)=>{

console.log('Error under control:' + err);

});

## async, await

JS permite la palabra clave await para expresiones, generalmente asíncronas.

El programa no avanzará hasta que termine esa llamada

**El uso de await, obliga a que el código con una cláusula await deba estar dentro de una función con una cláusula async**

la función getIpAddress se declara como asíncrona.

las llamadas a fetch y json son asíncronas, por tanto usamos await para que

la función no avance hasta que haya terminado estas operaciones

const getIpAddress = **async**() => {

const data = await fetch('https://api6.ipify.org?format=json');

const result = **await** data.json();

return result;

}

llamamos a getIpAddresss desde el método main,el cual ha de declarase como async y usar await para la llamada

const main = **async**() => {

console.log('ready?');

const result = **await** getIpAddress();

console.log('Finished: ' + result.ip);

}

main();

# SPREAD ...

## Spread en llamadas de función

|  |  |
| --- | --- |
| *function* foo( *x*, *y*, *z* ){    console.log( x + y + z );  }  *let* array1 = [ 0, 1, 2 ];  foo( ...array1 );  printa ‘3’  separa valores de un array | *function* foo( ...*args* ){    console.log( args.length );    console.log( args );  }  foo( 3, 4, 5 );  printa [3 ,4 ,5 ]  junta valores en un array |

## Spread en Array literales

**Crear un array usando un array existente sin usar push, splice, concat**

*var* parts = [ 'shoulders', 'knees' ];

*var* lyrics = [ 'head', ...parts, 'and', 'toes' ];

// ['head','shoulders','knees','and','toes']

**Copiar array**

*var* arr = [ 1, 2, 3 ];

*var* arr2 = [ ...arr ]; //same as arr.slice();

arr2.push(4);

  //arr = [ 1, 2, 3 ]

  //arr2 = [1 ,2 ,3 ,4 ]

**Concatenar arrays**

*var* arr1 = [ 0, 1, 2 ];

*var* arr2 = [ 3, 4, 5 ];

arr1 = [ ...arr1, ...arr2 ]; // arr1 = arr1.concat( arr2 );

## Spread en literales tipo Objeto

*let* obj1 = { foo: 'bar', x: 42 };

*let* obj2 = { foo: 'baz', y: 13 };

*let* clonedObj = { ...obj1 }; // { foo:'bar,x:42 }

*let* mergedObj = { ...obj1, ...obj2 } // { foo:'baz',x:42,y:13}

# Paso de parámetros por Referencia / Valor

**Valores Primitivos** ( Boolean, null, undefined, Number, String, Symbol ) -> Por **valor**

el valor asignado es una copia del valor que estamos asignando

**Valores NO Primitivos** ( Object, Array, Function ) -> Por **referencia**

no pasamos el valor en sí, sino una referencia