ECMASCRIPT)

http://es6-features.org/

Références

Array

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/find

Préambule

JavaScript versus ECMAScript

JavaScript est ce que tout le monde appelle le langage, mais ce nom est une marque (par Oracle, qui a hérité de la marque de Sun). Par conséquent, le nom officiel de JavaScript est ECMAScript.

Ce nom vient de l'organisation de normalisation Ecma, qui gère la norme du langage. Depuis la création d'ECMAScript, le nom de l'organisation est passé de l'acronyme «ECMA» (**European Computer Manufacturers Association)** au nom propre «Ecma».

Les versions de JavaScript sont définies par des spécifications portant le nom officiel du langage. Par conséquent, la première version standard de JavaScript est ECMAScript 1, qui signifie «ECMAScript Language Specification, Edition 1». ECMAScript x est souvent abrégé ESx.

TC39 (Comité technique 39 de l'Ecma)

Est le comité qui fait évoluer JavaScript. Ses membres sont des entreprises (entre autres, tous les principaux fournisseurs de navigateurs). Le TC39 se réunit régulièrement. Les délégués et les experts invités y assistent. Les procès-verbaux des réunions sont disponibles en ligne et vous donnent une bonne idée du fonctionnement du TC39. Voir ici.

Les versions de JavaScript : ES6, ES2019…

ES6 désigne la nouvelle version de JavaScript.

Le standard JavaScript

« ES » est l'abréviation d'ECMAScript, le standard sur lequel repose **JavaScript**.

Pendant longtemps (l'époque jQuery), il s'agissait de l'**ES5**, la version 5 de l'ECMAScript.

Depuis juin 2015, une nouvelle version est disponible : l'**ES6**.

Terminologie

La mise à jour du standard JavaScript est désormais annuelle. Afin de mieux s'y repérer, vous verrez donc également une appellation basée sur l'année :

ES6 = ES2015

ES7 = ES2016

ES8 = ES2017

ES9 = ES2018

ES10 = ES2019

Cependant, alors que l'ES6 fût une profonde transformation pour JavaScript, les versions suivantes apportent peu de nouveautés (la fréquence de mise à jour étant plus élevée). **On se réfère donc encore souvent à l'ES6 pour parler de la nouvelle version de JavaScript**, même si elle a déjà légèrement évolué depuis.

Compatibilité

Tous les navigateurs modernes supportent l'ES6 depuis un moment, et les frameworks majeurs (Angular, React, Vue…) utilisent tous cette nouvelle version de JavaScript.

6th Edition - ECMAScript 2015 [edit]

The 6th edition, initially known as ECMAScript 6 (**ES6**) then and later renamed to ECMAScript 2015, was finalized in June 2015.[11][29] This update adds significant new syntax for writing complex applications, including class declarations (`class Foo { ... }`), ES6 modules like `import * as moduleName from "...";` `export const Foo` , but defines them semantically in the same terms as ECMAScript 5 strict mode. Other new features include iterators and `for...of` loops, Python-style generators, arrow function expression (`() => {...}`), `let` keyword for local declarations, `const` keyword for constant variable declarations, binary data, typed arrays, new collections (maps, sets and WeakMap), promises, number and math enhancements, reflection, proxies (metaprogramming for virtual objects and wrappers) and template literals for strings.[30][31] The complete list is extensive.[32][33] As the first "ECMAScript Harmony" specification, it is also known as "ES6 Harmony."

7th Edition - ECMAScript 2016 [edit]

The 7th edition, officially known as ECMAScript 2016, was finalized in June 2016.[12] The major standard language features include block-scoping of variables and functions, destructuring patterns (of variables), proper tail calls, exponentiation operator `**` for numbers, `await` , `async` keywords for asynchronous programming.[12][34]

8th Edition - ECMAScript 2017 [edit]

The 8th edition, officially known as ECMAScript 2017, was finalized in June 2017.[13] Includes `async/await` constructions, which work using generators and promises. [35] ECMAScript 2017 (ES2017), the eighth edition, includes features for concurrency and atomics, syntactic integration with promises (async/await).[35][13]

9th Edition - ECMAScript 2018 [edit]

The 9th edition, officially known as ECMAScript 2018, was finalized in June 2018.[14] New features include rest/spread operators for variables (three dots: `...identifier`), asynchronous iteration, `Promise.prototype.finally()` and additions to RegExp.[14]

10th Edition - ECMAScript 2019 [edit]

The 10th edition, officially known as ECMAScript 2019, was published in June 2019.[9] Added features include, but is not limited to, `Array.prototype.flat` , `Array.prototype.flatMap` , changes to `Array.sort` and `Object.fromEntries` .[9]

ES.Next [edit]

ES.Next is a dynamic name that refers to whatever the next version is at the time of writing. ES.Next features are more correctly called *proposals* because, by definition, the specification has not been finalized yet.[*citation needed*]

ES6

Table de compatibilité : https://kangax.github.io/compat-table/es6/

Constantes et mode strict

```
const PI = 3.141593;
```

La directive "use strict"

La directive «use strict» était nouvelle dans la version ECMAScript 5.

Le but de "use strict"est d'indiquer que le code doit être exécuté en mode « strict ».

Avec le mode strict, vous ne pouvez pas, par exemple, utiliser des variables non déclarées.

Exemple :

```
x = 3.14;       // Pas d'erreur.
maFonction();

function maFonction() {
  "use strict";
  y = 3.14;   // Cause une erreur
}
```

Le mot clé let

Le mot clé let permet de déclarer une variable limitée à la portée d'un bloc, c'est-à-dire qu'elle ne peut être utilisée que dans le bloc où elle a été déclarée, ce qui n'est pas le cas avec var.

```
function swap(x, y) {
   if (x != y) {
      var old = x;
      let tmp = x;
      x = y;
      y = tmp;
   }

   console.log(typeof(old));   // number
   console.log(typeof(tmp));   // undefined
}
swap(2,3);
```

Portée

```
for (let i = 0; i < a.length; i++) {
    let x = a[i]
    ...
}
for (let i = 0; i < b.length; i++) {
    let y = b[i]
    ...
}

let callbacks = []
for (let i = 0; i <= 2; i++) {
    callbacks[i] = function () { return i * 2 }
}
callbacks[0]() === 0
callbacks[1]() === 2
callbacks[2]() === 4
```

Portée fonctions

```
{
    function foo () { return 1 }
    foo() === 1
    {
        function foo () { return 2 }
        foo() === 2
    }
    foo() === 1
}
```

Templates et chaînes de caractères

```
<script>
    let moi = "Yannick";
    let mAge = 29;
    let result = `Je suis ${moi} et j'ai ${mAge} ans`;
    alert(result);
</script>
```

a=2; b=3;
On peut ${a+5} affichera 5

Suite

ES6

```
<script>
var client = { nom: "Pierre" }
var panier = { quantite: 7, produit: "Chocolats", prixunitaire: 42 }
var message = `Salut ${client.nom},
vous voulez acheter ${panier.quantite} ${panier.produit} pour
un total de ${panier.quantite * panier.prixunitaire} $?`
alert(message)
</script>
```

ES5

```
var client = { nom: "Pierre" }; var panier = { montant: 7, produit: "Bar", prixunitaire: 42 }; var message
= "Salut " + customer.nom + ",\n" + "vous voulez acheter" + panier.montant + " " + panier.produit + "\n"
+ « pour un total de " + (panier.montant * panier.prixunitaire) + " bucks?";
alert(message);
```

Les paramètres par défaut

Ancienne méthode	Nouvelle méthode

var add = function (x, y) {
  var x = typeof(x) === "number" ? x : 0;
  var y = typeof(y) === "number" ? y : 0;
  return x + y;
};
 	
alert(add(2,3));
alert(add(undefined,5));
	let add = function (x = 0, y = 0) {
	return x + y; };
	
	alert(add(2,3));
	alert(add(undefined,5));

Arrow functions

```
<script>
impaires=[3,5,7,9,11,13,15];
// Expression bodies
var paires = impaires.map(v => v + 1);alert(paires);
var nums = impaires.map((v, i) => v + i);alert(nums);
var pairs = impaires.map(v => ({impaires: v, paires: v + 1}));alert(JSON.stringify(pairs));

// Statement bodies
fives=[];
impaires.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});
alert(fives);
// Lexical this
var bob = {
  _name: "Bob",
  _friends: ["AA","BB","CC"],
  printFriends() {
    this._friends.forEach(f =>
      alert(this._name + " knows " + f));
  }
}
bob.printFriends();
</script>
```

```
var tableau = ["abc", "de", "fgjijk"];
var resultat = [];
for(var i=0 ; i<tableau.length ; i++){
  var element = tableau[i];
  resultat.push(element.length);
}
```

```
var resultat = ["abc", "de", "fgjijk"].map( element => element.length );
```

ECMAScript 6 est également connu comme ES6 et ECMAScript 2015.

Quelques nouvelles fonctionnalités ES6

let JavaScript

JavaScript const

Exponentiation (**)

les valeurs des paramètres par défaut

Array.find ()

Array.findIndex ()

Exponentiation

```
var x = 5;
var z = x ** 2;          // vaut 25

var x = 5;
var z = Math.pow(x,2);   // vaut 25
```

Array.find()

```
<script>
let tab = [4, 9, 16, 25, 29];
let premier = tab.find(f1);
document.write("<br><b>premier =
"+premier+"</b><br><br>");

function f1(value, index, array) {
document.write(value+" <br>  "+index+" <br>
"+JSON.stringify(array)+"<br>");
return value > 18;
}
</script>
```

```
4
0
[4,9,16,25,29]
9
1
[4,9,16,25,29]
16
2
[4,9,16,25,29]
25
3
[4,9,16,25,29]

premier = 25
```

```
<script>
let tab = ['a','b','c'];
let rep=tab.find(k =>
{document.write("<br>k="+k);return k=='b'; });
document.write("<br>rep = "+rep);
</script>
```

```
k=a
k=b
rep = b
```

```
let tab = [3, 5, 21, 8, 9, 10, 12];
function estPaire(i) {
  return i % 2 == 0;
}

tab.find(estPaire);
// 8
```

Array.findIndex()

Au lieu de retourner la valeur retourne la position. Si pas trouvé retourne -1.

```
let tab = ['a','b','c'];
tab.findIndex(k => k=='b');
// 1

arr.findIndex(k => k=='z');
// -1
```

```
let tab = [3, 5, 21, 8, 9, 10, 12];
function estImpaire(i) {
  return i % 2 !== 0;
}

tab.findIndex(estImpaire);

//0
```

Opérateur spread et rest

spread

Étendre un argument en multiple paramètres. Cas d'un Array.

```
var monArray = [5, 10, 50]; //pour vider un tableau monArray.length=0 ou monArray=[]
Math.max(...monArray);    // 50
```

Autre situation

```
function maFonction() {
  for(var i in arguments){
    console.log(arguments[i]);
  }
}
var params = [10, 15];
maFonction(5, ...params, 20, ...[25]);    // 5 10 15 20 25
```

Opérateur spread et rest

rest

Le paramètre rest a la même syntaxe que l'opérateur spread, mais au lieu de développer un tableau en paramètres, il collecte les paramètres et les transforme en tableau.

```
function maFonction(...options) {
    return options;
}
maFonction('a', 'b', 'c');      // ["a", "b", "c"]
```

```
<script>
function testSubstrings(chaine, ...cles) {
  for (var cle of cles) {
    if (chaine.indexOf(cle) === -1) {
      return false;
    }
  }
  return true;
}
alert(testSubstrings('ceci est une chaine', 'une', 'ceci'));
</script>
```

Limites de rest

1. Doit être le dernier paramètre dans la liste des paramètres

```
function logArguments(a, ...params, b) {
      console.log(a, params, b);
}
logArguments(5, 10, 15);    // SyntaxError: parameter after rest parameter
```

2. Un seul paramètre rest

```
function logArguments(...param1, ...param2) {
}
logArguments(5, 10, 15);    // SyntaxError: parameter after rest parameter
```

Opérateur spread et rest

```
<script>
var params = [ "hello", true, 7 ]
var autre = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]
alert(autre);
function f (x, y,...a) {alert(a[0])
    return (x + y) * a.length
}
alert(f(1, 2,4,5,7,8))// === 12

var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
alert(chars);
</script>
```

Utilisé par un constructeur

```
new Date(...[2016, 5, 6]);    // Mon Jun 06 2016 00:00:00 GMT-0700 (Pacific Daylight Time)
```

Array.reduce

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/reduce

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Boucle for – of

```
let langages = [ "C", "C++", "C#", "JavaScript" ];

for (let langage of langages) {
    console.log(langage);
}
```

Il est aussi possible d'utiliser un système clé/valeur en spécifiant un tableau en paramètre et en exploitant la fonction Array::entries() qui va renvoyer un objet de type Iterator contenant le couple clé/valeur pour chaque élément du tableau.

```
for (let [index, langage] of langages.entries()) {
    console.log(`Index: ${index} => Valeur: ${langage}`);
}
```

Évolution des objets littéraux

Les objets littéraux peuvent être vus comme une structure, ils permettent de stocker des variables et des méthodes. Il est désormais possible d'avoir une écriture beaucoup plus légère pour les déclarer, en se passant du mot clé function.

```
let Engine = {
    initialize() {
        this.display("Init") ;
    },
    display(message) {
        alert(message);
    }
};

Engine.initialize();
```

arguments

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
  var i;
  var max = -Infinity;
  for (i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
```

PAUSE EXERCICES

Exercice ES6-1

Écrivez un programme JavaScript pour convertir un nombre spécifié en tableau de chiffres.

Remarque: convertissez le nombre en chaîne en utilisant l'opérateur spread (...) pour construire un tableau.

Exercice ES6-2

Écrivez un programme JavaScript pour filtrer les valeurs spécifiées d'un tableau. Renvoie le tableau d'origine sans les valeurs filtrées.

Aide : utiliser les méthodes «filter» et « include ».

Exercice ES6-3

Écrivez un programme JavaScript pour renvoyer la valeur minimum-maximum d'un tableau, après avoir appliqué la fonction fournie comme règle de comparaison.

Classes

```
const p = new Rectangle(); // ReferenceError

class Rectangle {}
```

```
class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

foo(); // fonctionne `foo` est hoisted (scope de sa définiton)

function foo() {}

Une classe existe quand on a trouvé sa definition en exécution.

new Foo(); // ReferenceError

class Foo {}

```
<script>
let f1=(re)=>{
    k= new re(5,3,"k");
    alert("OK");
}
let Rect = class  {
  constructor(hauteur, largeur, de) {
    this.hauteur = hauteur;
    this.largeur = largeur; alert(de);
  }
};

r=new Rect(12,5,"r");
f1(Rect);
</script>
```

```
class Rectangle {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.largeur * this.hauteur;
  }
}

const carré = new Rectangle(10, 10);

console.log(carré.area);
```

Méthodes statiques

```
<script>
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.abs(dx+dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));
</script>
```

Attributs privés

```
<script>
class MyClass {
  a = 1;           // .a est publique
  #b = 2;          // .#b est privé
  static #c = 3;   // .#c est privé et statique
  incB() {
    this.#b++;
  }
  get b(){
    return this.#b;
  }
  set b(autreB){
    this.#b+=autreB;
  }
}
const m = new MyClass();
m.incB(); // OK
alert(m.b);
m.b=2; //appel set b
alert(m.b);
//m.#b = 0; // erreur attribut privé
</script>
```

Il n'existe aucun moyen de définir des méthodes privées, des getters et des setters

Héritage

```
<script>
class Animal {
  constructor(nom) {
    this.nom = nom;
  }

  parle() {
    alert(this.nom + ' ne sait pas parler.');
  }
}

class Chien extends Animal {
  constructor(nom) {
    super(nom); // appelle le constructeur parent avec le paramètre
  }
  parle() {
    alert(this.nom + ' fait ouf,ouf.');
  }
}
let chien = new Chien("Ringo");
let animal = new Animal("Milou");
chien.parle();
animal.parle();
</script>
```

Suite

```
<script>
class Animal {
  constructor(nom) {
    this.nom = nom;
  }

  parle() {
    alert(this.nom + ' ne sait pas parler.');
  }
}

class Chien extends Animal {
  constructor(nom) {
    super(nom); // appelle le constructeur parent avec le paramètre
  }
  parle() {
    alert(this.nom + ' fait ouf,ouf.');
  }
}
let tabChiens=[new Chien("Ringo"),new Chien("Milou"),new Chien("Zorro")];
for(unChien of tabChiens){
    unChien.parle();
}
</script>
```

Autres cas

Propriétés partagées

ES6	ES5
var x = 0, y = 0 obj = { x, y }	var x = 0, y = 0; obj = { x: x, y: y };

```
<script>
    var x = 0, y = 0
    obj1 = { x, y }
    alert(obj1.x)
    obj1.x=20
    alert(obj1.x)
    obj1.y=30
    alert(obj1.y)
    obj2 = { x, y }
    alert(obj2.x)
    alert(obj2.y)
</script>
```

Propriétés calculées

ES6

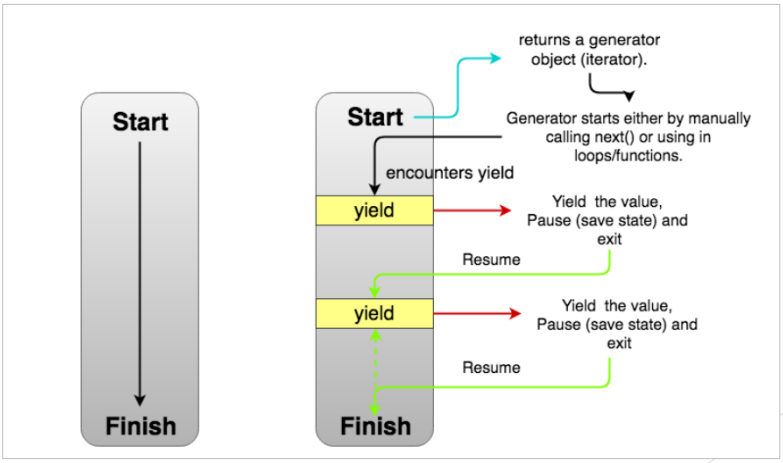
```
1  <script>
2      f1=()=>1
3      let obj = {
4          foo: "bar",
5          [ "baz" + f1() ]: 42
6      }
7      alert(obj.baz1)
8  </script>
```

ES5

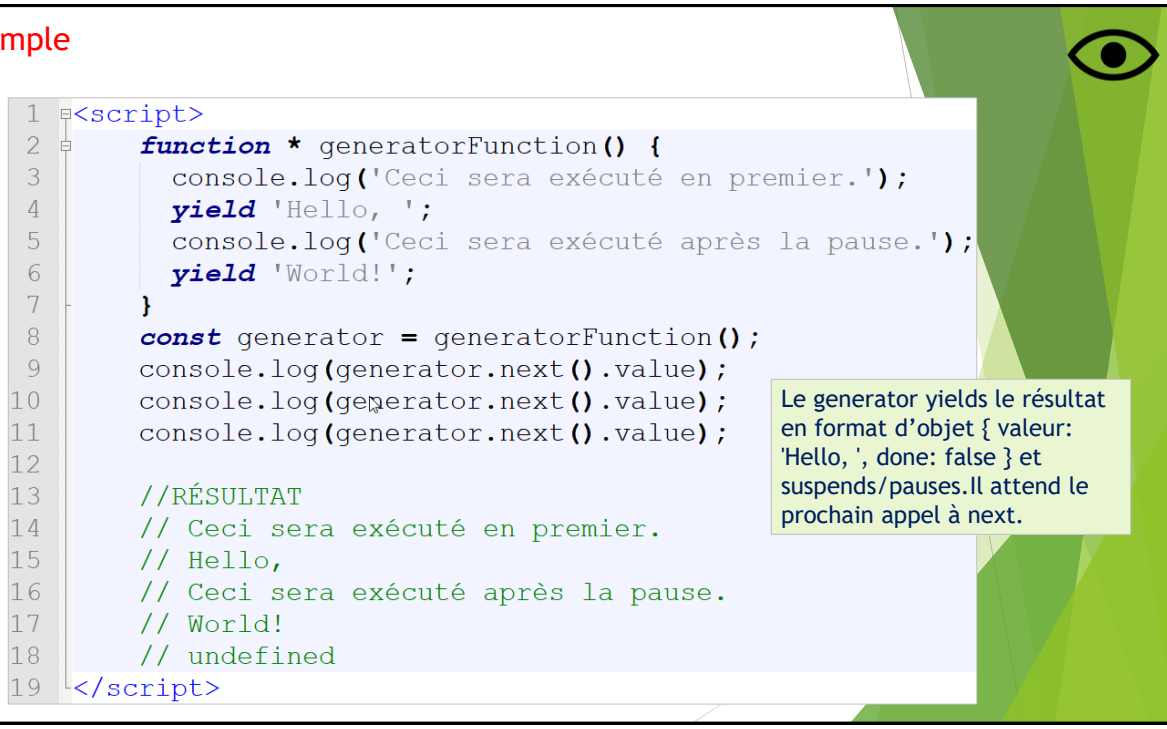
```
var obj = { foo: "bar" };
obj[ "baz" + f1() ] = 42;
```

Generator Functions

Schéma général



Exemple



```
<script>
    function * generatorFunction() {
      console.log('Ceci sera exécuté en premier.');
      yield 'Hello, ';
      console.log('Ceci sera exécuté après la pause.');
      yield 'World!';
    }
    const generator = generatorFunction();
    console.log(generator.next().value);
    console.log(generator.next().value);
    console.log(generator.next().value);

    //RÉSULTAT
    // Ceci sera exécuté en premier.
    // Hello,
    // Ceci sera exécuté après la pause.
    // World!
    // undefined
</script>
```

Le generator yields le résultat en format d'objet { valeur: 'Hello, ', done: false } et suspends/pauses.Il attend le prochain appel à next.

Pas de return

```
function * generatorFunc() {
    yield 'a';
    return 'b'; // Generator termine ici.
    yield 'c'; // Ne sera jamais exécuté.
}
```

Utilisation comme itérateur

```
function * nombres() {
  let nb = 1;
  while (true) {
    yield nb;
    nb = nb + 1
  }
}
const nbs = nombres();
console.log(nbs.next().valeur)
console.log(nbs.next().valeur)
// 1
// 2
```

Envoyer des valeurs par next

https://exploringjs.com/es6/ch_generators.html#sec_generators-as-observers

Generators as observers (data consumption)

Comme consommateur de données, generator objects se conforment à l'interface

```
interface Observer {
   next(valeur? : any) : void;
   return(valeur? : any) : void;
   throw(error) : void;
}
```

A generator attend jusq'à ce qu'elle reçoi un input. Trois types input, transmises par les methods de l'interface.

next() envoi normal input.

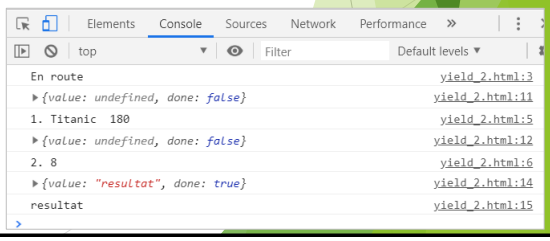
return() termine le generator.

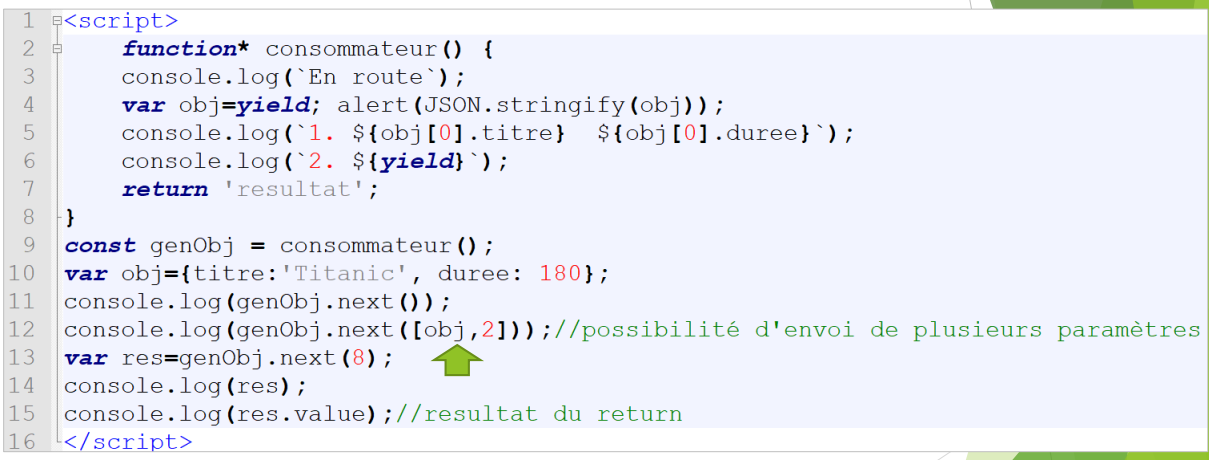
throw() signal une error.

```
<script>
    function* consommateur() {
    console.log(`En route`);
    var n=yield; n=n+2;
    alert(n);
    console.log(`1. ${n}`);
    console.log(`2. ${yield}`);
    return 'resultat';
}
const genObj = consommateur();
console.log(genObj.next());
console.log(genObj.next(3));
var res=genObj.next(8);
console.log(res);
console.log(res.value);//resultat du return
</script>
```

Lors de l'utilisation d'un générateur en tant qu'observateur, il est important de noter que la première invocation de next () a pour seul but de démarrer l'observateur. Il n'est prêt à recevoir des entrées, car cette première invocation avance l'exécution au premier rendement(yield). Par conséquent, toute entrée que vous envoyez via le premier next () est ignorée.

```
<script>
    function* consommateur() {
    console.log(`En route`);
    var obj=yield;
    console.log(`1. ${obj.titre}  ${obj.duree}`);
    console.log(`2. ${yield}`);
    return 'resultat';
}
const genObj = consommateur();
var obj={titre:'Titanic', duree: 180};
console.log(genObj.next());
console.log(genObj.next(obj));
var res=genObj.next(8);
console.log(res);
console.log(res.value);//resultat du return
</script>
```





```
<script>
    function* consommateur() {
    console.log(`En route`);
    var obj=yield; alert(JSON.stringify(obj));
    console.log(`1. ${obj[0].titre}  ${obj[0].duree}`);
    console.log(`2. ${yield}`);
    return 'resultat';
}
const genObj = consommateur();
var obj={titre:'Titanic', duree: 180};
console.log(genObj.next());
console.log(genObj.next([obj,2]));//possibilité d'envoi de plusieurs paramètres
var res=genObj.next(8);
console.log(res);
console.log(res.value);//resultat du return
</script>
```

Nouveautés avec Array

Bonne manière: parcours Array

1. Uns simple boucle classique

```
for (var i=0; i<arr.length; i++) {
    console.log(arr[i]);
}
```

2. *Par* exemple, forEach():

```
arr.forEach(function (elem) {
    console.log(elem);
});
```

Array.from(…)

```
const arrayLike = { length: 2, 0: 'a', 1: 'b' };//Comme un Array mais sans ses méthodes

// for-of only works with iterable valeurs
for (const x of arrayLike) { // TypeError  avec for…in OK
    console.log(x);
}

const arr = Array.from(arrayLike);
for (const x of arr) { // OK, iterable
    console.log(x);
}
// Output:
// a
// b
```

map.call (map retourne un résultat : tableau)

```
var map = Array.prototype.map;
var a = map.call('Hello World', function(x)
{alert(x); return x.charCodeAt(0);});

// tableau des codes ASCII [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100]
```

Array.from() généralement est une bonne alternative à l'utilisation de map() :

```
<span class='name'>AAA</span>
<span class='name'>BBB</span>
<span class='name'>CCC</span>
<script>
const spans = document.querySelectorAll('span.name');

// map(), generically:
const names1 = Array.prototype.map.call(spans, s => s.textContent);
alert(names1)
// Array.from():
const names2 = Array.from(spans, s => s.textContent);
alert(names2)
</script>
```

```
<script>
    class MonArray extends Array {
   //.....
}

const instanceDeMonArray = MonArray.from([1, 2, 3], x => x * x);
alert(instanceDeMonArray)

//Toujours instance de Array
const instanceDeArray   = [1, 2, 3].map(x => x * x);
alert(instanceDeArray)
</script>
```

Array.of(...items)

Array.of(item_0, item_1, ⋯) créer un Array dont les éléments sont item_0, item_1, …

```
class MonArray extends Array {
    ...
}
console.log(MonArray.of(3, 11, 8) instanceof MonArray); // true
console.log(MonArray.of(3).length); // 1
```

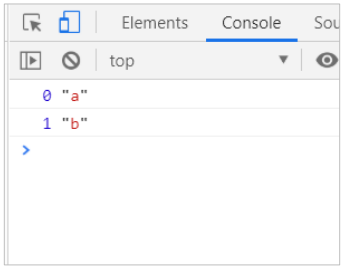
Itérater sur des Arrays

Méthodes

- Array.prototype.entries()
- Array.prototype.keys()
- Array.prototype.valeurs()

```
> Array.from(['a', 'b'].keys())
[ 0, 1 ]
> Array.from(['a', 'b'].valeurs())
[ 'a', 'b' ]
> Array.from(['a', 'b'].entries())
[ [ 0, 'a' ],
  [ 1, 'b' ] ]
```

```
for (const [index, element] of ['a', 'b'].entries()) {
   console.log(index, element);
}
```



find(), findIndex()

Trouve la première occurrence

Question : afficher tous les nombres négatifs en utilisant map.

```
> [6, -5, 8].find(x => x < 0)
-5
> [6, 7, 8].find(x => x < 0)
Undefined

> [6, -5, 8].findIndex(x => x < 0)
1
> [6, 7, 8].findIndex(x => x < 0)
-1
```

Array.prototype.copyWithin()

Signature :

Array.prototype.copyWithin(target : number, start : number, end = this.length) : this

Nota : [start, end)

```
> const arr = [0,1,2,3];
> arr.copyWithin(2, 0, 2)
[ 0, 1, 0, 1 ]
> arr
[ 0, 1, 0, 1 ]
```

Array.prototype.fill()

Signature :

Array.prototype.fill(valeur : any, start=0, end=this.length) : this

Nota : [start, end)

```
> const arr = ['a', 'b', 'c'];
> arr.fill(7)
[ 7, 7, 7 ]
> arr
[ 7, 7, 7 ]

> ['a', 'b', 'c'].fill(7, 1, 2)
[ 'a', 7, 'c' ]
```

Trous dans les Array

Cas ES5

```
> const arr = ['a',,'b'] //tableau avec un trou à l'indice 1
'use strict'
> 0 in arr
true
> 1 in arr
false
> 2 in arr
true
> arr[1]
undefined
```

Autre façon de créer des trous :

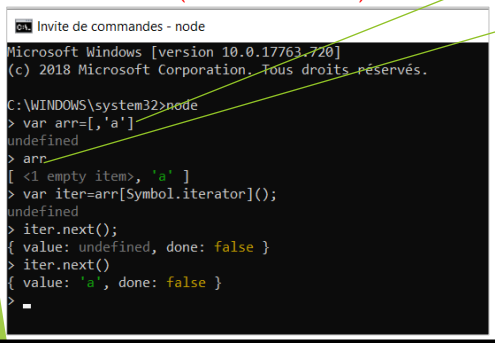
```
> var arr = [];
> arr[0] = 'a';
> arr[2] = 'c';
> 1 in arr  // trou à indice 1
false
```

Cas ES6

Traite les trous comme undefined (n'aime pas trop les trous)

```
> Array.from(['a',,'b'])
[ 'a', undefined, 'b' ]
> [,'a'].findIndex(x => x === undefined)
0
```

Avec itération (testé avec node)



Écrit toujours puisque la console attend une valeur de retour.

Ici comme il a eu une valeur de retour alors il n'y a pas de undefined.

```
for (const x of [, 'a']) {
  console.log(x);
}
// Sortie:
// undefined
// a
```

The following table describes how `Array.prototype` methods handle holes.

Method	Holes are	
`concat`	Preserved	`['a',,'b'].concat(['c',,'d']) → ['a',,'b','c',,'d']`
`copyWithin`ES6	Preserved	`[,'a','b',,].copyWithin(2,0) → [,'a',,'a']`
`entries`ES6	Elements	`[...[,'a'].entries()] → [[0,undefined], [1,'a']]`
`every`	Ignored	`[,'a'].every(x => x==='a') → true`
`fill`ES6	Filled	`new Array(3).fill('a') → ['a','a','a']`
`filter`	Removed	`['a',,'b'].filter(x => true) → ['a','b']`
`find`ES6	Elements	`[,'a'].find(x => true) → undefined`
`findIndex`ES6	Elements	`[,'a'].findIndex(x => true) → 0`
`forEach`	Ignored	`[,'a'].forEach((x,i) => log(i)); → 1`
`indexOf`	Ignored	`[,'a'].indexOf(undefined) → -1`
`join`	Elements	`[,'a',undefined,null].join('#') → '#a##'`
`keys`ES6	Elements	`[...[,'a'].keys()] → [0,1]`
`lastIndexOf`	Ignored	`[,'a'].lastIndexOf(undefined) → -1`
`map`	Preserved	`[,'a'].map(x => 1) → [,1]`
`pop`	Elements	`['a',,].pop() → undefined`
`push`	Preserved	`new Array(1).push('a') → 2`
`reduce`	Ignored	`['#',,undefined].reduce((x,y)=>x+y) → '#undefined'`
`reduceRight`	Ignored	`['#',,undefined].reduceRight((x,y)=>x+y) → 'undefined#'`
`reverse`	Preserved	`['a',,'b'].reverse() → ['b',,'a']`
`shift`	Elements	`[,'a'].shift() → undefined`
`slice`	Preserved	`[,'a'].slice(0,1) → [,]`
`some`	Ignored	`[,'a'].some(x => x !== 'a') → false`
`sort`	Preserved	`[,undefined,'a'].sort() → ['a',undefined,,]`
`splice`	Preserved	`['a',,].splice(1,1) → [,]`
`toString`	Elements	`[,'a',undefined,null].toString() → ',a,,'`
`unshift`	Preserved	`[,'a'].unshift('b') → 3`
`values`ES6	Elements	`[...[,'a'].values()] → [undefined,'a']`

Créer des tableaux avec des valeurs

> new Array(3).fill(7)

[7, 7, 7]

> [...new Array(3).keys()]

[0, 1, 2]

> Array.from(new Array(5), (x,i) => i*2)

[0, 2, 4, 6, 8]

Enlever les trous

> ['a',,'c'].filter(() => true)

['a', 'c']

La taille des indices

Même régles que en ES5

Array taille T dans l'interval $0 \leq T \leq 2^{32}-1$.

Array indices I dans l'interval $0 \leq I < 2^{32}-1$.

Généralités sur les paramètres

```
function getParam() {
    alert("En getParam");
    return 3;
}

function multiplier(param1, param2 = getParam()) {
    return param1 * param2;
}
multiplier(2, 5);     // 10
multiplier(2);     // 6 (affiche aussi le alert)
```

```
function maFonction(a=10, b=a) {
    console.log('a = ' + a + '; b = '  + b);
}

maFonction();     // a=10; b=10
maFonction(22);    // a=22; b=22
maFonction(2, 4);    // a=2; b=4
```

```
function maFonction(a, b = ++a, c = a*b) {
    console.log(c);
}

maFonction(5);    // 36
```

Tous les paramètres sont passés par valeur sauf les objets et Array

```
function foo(param){
    param.bar = 'nouvelle valeur';
}
obj = {
    bar : 'valeur'
}
console.log(obj.bar);    // valeur
foo(obj);
console.log(obj.bar);    // nouvelle valeur
```

Paramètres obligatoires

```
function throwError() {
    throw new Error('Paramètre manquant');
}

function foo(param1 = throwError(), param2 = throwError()) {
    // faire quelque chose
}
foo(10, 20);    // ok
foo(10);    // Erreur: Paramètre manquant
```

```
function verifieParams(param1) {
    console.log(param1);    // 2
    console.log(arguments[0], arguments[1]);    // 2 3
    console.log(param1 + arguments[0]);    // 2 + 2
}

verifieParams(2, 3);
```

Sans nommer les paramètres

```
function verifieParams() {
    console.log(arguments[1], arguments[0], arguments[2]);
}
verifieParams(2, 4, 6);  // 4 2 6
```

```
function verifieParams(...params) {
    console.log(params[1], params[0], params[2]);    // 4 2 6
    console.log(arguments[1], arguments[0], arguments[2]);    // 4 2 6
}
verifieParams(2, 4, 6);
```

alert(JSON.stringify(arguments)) ont obtient

{"0":40,"1":20,"2":50,"3":30}

```
<script>
function trier1() {
    var a = Array.from(arguments);
    return a.sort();
}
alert(trier1(40, 20, 50, 30));    // [20, 30, 40, 50]
function trier2(...a) {
        return a.sort();
}
alert(trier2(40, 20, 50, 30));    // [20, 30, 40, 50]
</script>
```

Destructuring

Permet d'extraire des valeurs de tableaux et d'objets et de les affecter à des variables. La syntaxe est claire et facile à comprendre et est particulièrement utile pour passer des arguments à une fonction.

ES5

```
<script>
    function initialiserTransfert(options) {
        var  protocol = options.protocol,
            port = options.port,
            delay = options.delay,
            retries = options.retries,
            timeout = options.timeout,
            log = options.log;
        // code pour initialiser le transfert
    }
    options = {
      protocol: 'http',
      port: 8080,
      delay: 150,
      retries: 10,
      timeout: 500,
      log: true
    };
    initialiserTransfert(options);
</script>
```

ES6

```
<script>
    function initialiserTransfert({protocol, port, delay, retries, timeout, log}) {
     // code pour initialiser le transfert
    alert(port);
    };
    var options = {
      protocol: 'http',
      port: 8080,
      delay: 150,
      retries: 10,
      timeout: 500,
      log: true
    }
    initialiserTransfert(options);

</script>
```

Si dans options un paramètre manque alors dans la fonction il sera undefined. S'il manque dans la fonction alors si tentative d'y accéder alors erreur. Peuvent avoir des valeurs par défaut. Si aucun paramètre est envoyé alors déclarer comme suit :

```
function initialiserTransfert({protocol, port, delay, retries, timeout, log}={}) {
```

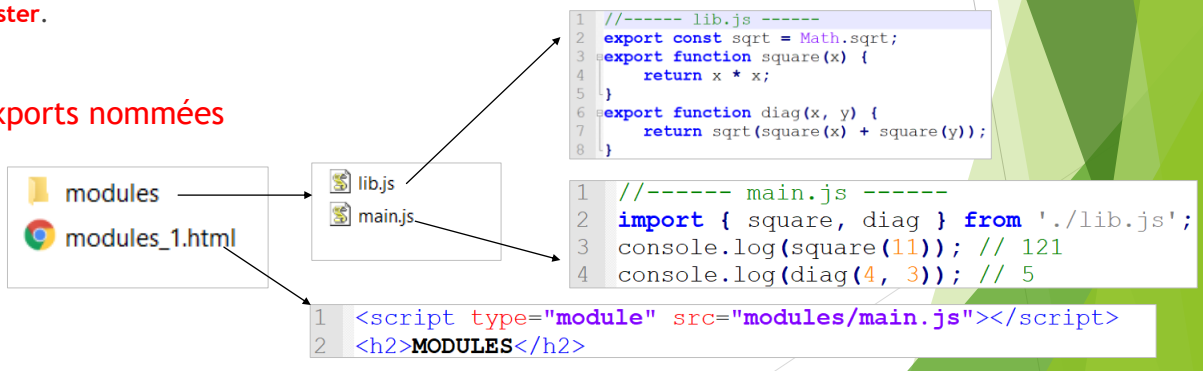
Modules

Démarrer le serveur

Il existe deux types d'exportations: les exportations **nommées** (plusieurs par module) et les exportations par **défaut** (une par module). Comme expliqué plus tard, il est possible d'utiliser les deux en même temps, mais il est généralement préférable de les séparer.

Attention aux tests sur un environnement local : si vous chargez le fichier HTML directement depuis le système de fichier dans le navigateur (en double-cliquant dessus par exemple, ce qui donnera une URL file://), vous rencontrerez des erreurs CORS pour des raisons de sécurité. **Il faut donc un serveur local afin de pouvoir tester**.

Exports nommées



Aussi

```
1  //------ main2.js ------
2  import * as lib from './lib.js';
3  console.log(lib.square(11)); // 121
4  console.log(lib.diag(4, 3)); // 5
```

Default exports (un par module)

```
//------ mod1.js ------
export default function () {} // pas de ;

//------ main1.js ------
import maFonc from 'mod1';
maFonc();

//------ maClass.js ------
export default class {} // pas de ;

//------ main2.js ------
import maClass from 'maClass';
const instance = new maClass();
```

Exporter des valeurs

```
export default 'abc';
export default foo();
export default /^xyz$/;
export default 5 * 7;
export default { no: false, yes: true };
```

Imports et exports doivent être au niveau supérieur

```
if (Math.random()) {
    import 'foo'; // SyntaxError
}

// Pas `import` et `export` intérieur d'un bloc
{
    import 'foo'; // SyntaxError
}
```

```
foo();
import { foo } from 'mon_module';
```

Ce code fonctionne puisque à l'interne le module a été déplacé au début du «scope» (portée).

On peut renommer des imports:

```
// Renommer: import `nom1` as `localNom1`
import { nom1 as localNom1, nom2 } from 'src/ma_lib';

// Renaming: import the default export as `foo`
import { default as foo } from 'src/my_lib';
```

```
export var maVar1 = ···;
export let maVar2 = ···;
export const ma_CONST = ···;

export function maFunc() {
    ···
}
export function* maGeneratorFunc() {
    ···
}
export class maClass {
    ···
}
```

Reste encore beaucoup à voir. À vous de continuer.