



**ECMAScript**

# ECMAScript

[JavaScript ES6 \(w3schools.com\)](https://www.w3schools.com/js/)

**JS**

# Objectifs

---

- Connaître les fonctionnalités syntaxiques qui rendent le code JavaScript plus lisible, concis et expressif

## Pourquoi ?

Ces fonctionnalités, telles que les classes, les fonctions fléchées, la déstructuration, les littéraux de modèle etc. simplifient la syntaxe et améliorent la lisibilité du code React.

# Qu'est-ce que *ECMAScript* ?

---

ECMAScript est la norme qui sous-tend le langage de script JavaScript, et il joue un rôle crucial dans l'assurance de la cohérence et de la compatibilité des implémentations du langage sur différents navigateurs et environnements d'exécution.

# ES6 (ou javascript 2015)

---

- Impact significatif sur le développement JavaScript.
- Entre autres :
  - les variables let et const
  - les fonctions fléchées
  - les classes
  - la déstructuration
  - les littéraux de modèles
  - les opérateurs de propagation
  - les promesses
  - ...
- Ces fonctionnalités d'ES6 ont grandement contribué à rendre le code React plus lisible, maintenable et efficace. Elles sont largement adoptées dans la communauté React et font partie intégrante des bonnes pratiques de développement React.

# ES6 (ou javascript 2015)

---

## Browser Support for ES6 (2015)

ES6 is fully supported in all modern browsers since June 2017:

				
Chrome 51	Edge 15	Firefox 54	Safari 10	Opera 38
May 2016	Apr 2017	Jun 2017	Sep 2016	Jun 2016

# Variables let et const (ES6)

---

- let et const ont uniquement une portée de bloc !
- Var a une "portée de fonction"

```
var x = 10;
console.log("1:",x)

{
  let x = 2;
  console.log("2:",x)

  var y = 5;
  const z = 6
}

console.log("3:",x)
console.log("4: ", y)
console.log(["5: ", z])
```

```
1: 10
2: 2
3: 10
4: 5
ReferenceError: z is not defined
```

# Les fonctions fléchées (ES6)

---

- Avant ES6: une fonction s'écrivait obligatoirement avec la syntax suivante:

```
function helloName(name){  
    return 'Hello ' + name;  
}
```

- Depuis ES6: on peut utiliser des fonctions fléchées:

```
const helloName = (name) => {  
    return 'Hello ' + name  
}
```

Avec accolade

```
const helloName = (name) => 'Hello ' + name
```

Sans accolade: le 'return' est implicite

**Les trois fonctions des captures d'écran ci-dessus font exactement la même chose**

# Les tableaux

- Les tableaux ES6



# Les tableaux: parcourir avec `.forEach()` (ES5)

- Syntax: `array.forEach(function(currentValue, index, arr), thisValue)`

Paramètres	Description	Obligatoire / optionnel
<code>function()</code>	fonction qui s'exécute sur chacun des éléments du tableau	Obligatoire
<code>currentValue</code>	valeur de l'élément parcouru	Obligatoire <b>Optionnel fonction fléchée</b>
<code>index</code>	valeur de l'index parcouru	Optionnel
<code>arr</code>	tableau de l'élément parcouru	Optionnel
<code>ThisValue</code>	valeur à utiliser comme 'this' lors de l'exécution de la fonction de rappel.	Optionnel

# Les tableaux: Parcourir avec `.forEach()`

---

- Exemples:

- Avec uniquement le paramètre currentValue:

```
myArray2.forEach((element) => console.log(element))
```

- Sans paramètres:

```
// Exemple = je veux compter le nbr d'éléments d'un tableau
const nbrElements = (array) => {
  let y = 0;
  array.forEach(() => y++);
  return y;
}
```

- Avec l'ensemble des paramètres:

```
myArray = ['Riri', 'Fifi', 'Loulou']

myArray.forEach((element, index, arr) =>
  console.log(index + ": " + element + " présent dans le tableau '" + arr + "'));
```

```
0: Riri présent dans le tableau 'Riri,Fifi,Loulou'
1: Fifi présent dans le tableau 'Riri,Fifi,Loulou'
2: Loulou présent dans le tableau 'Riri,Fifi,Loulou'
```

# Les tableaux: '*for (variable of iterable)*' (ES6)

---

- Syntax:

```
Nom que je donne à l'élément parcouru      Nom du tableau que je parcours  
↑                               ↗  
for (variable of iterable) {  
    // code block to be executed  
}
```

- Exemple: j'affiche dans la console chacun des éléments de mon tableau:

```
myArray = ['Riri', 'Fifi', 'Loulou']  
for (element of myArray){  
    console.log(element)  
}
```

# Les tableaux: les méthodes map() et flatMap()

---

- **Syntax:** `array.forEach(function(currentValue, index, arr), thisValue)`
- **crée un nouveau tableau** en exécutant une fonction sur chaque élément du tableau
- n'exécute pas la fonction pour les éléments du tableau sans valeurs
- **ne modifie pas le tableau d'origine**
- la méthode flatMap() *map* d'abord tous les éléments d'un tableau, puis crée un nouveau tableau en aplatisant le tableau.

Paramètres	Description	Obligatoire / optionnel
function()	fonction qui s'exécute sur chacun des éléments du tableau	Obligatoire
currentValue	valeur de l'élément parcouru	Obligatoire <b>Optionnel fonction fléchée</b>
index	valeur de l'index parcouru	Optionnel
arr	tableau de l'élément parcouru	Optionnel
ThisValue	valeur à utiliser comme 'this' lors de l'exécution de la fonction de rappel.	Optionnel

# Les tableaux: les méthodes map() et flatMap()

---

- Exemple:

```
const array = [1, 2, [4, 5], 6, 7, [8]];
console.log(array.flatMap((element) => element));

// Équivalent avec un .map
console.log(array.map((element) => element).flat());

// return => [1, 2, 4, 5, 6, 7, 8]
```

- Voir les exemples du slide .forEach() (même logique sauf que le map() duplique le tableau)

# Les tableaux: la méthode filter()

---

- Permet de **duplicer un tableau en appliquant un filtre lié à une condition.**
- Syntax: `arr.filter(callback);`

Paramètres	Description	Obligatoire / optionnel
<code>function()</code>	La fonction à appliquer à chaque élément du tableau sur lequel on fait le <code>.filter()</code> . Cette fonction est appelée avec les arguments ci-dessous	Obligatoire
<code>currentValue</code>	Elément à traîter	Obligatoire <b>Optionnel fonction fléchée</b>
<code>index</code>	valeur de l'index de l'élément parcouru	Optionnel
<code>arr</code>	Tableau complet de l'élément parcouru	Optionnel

# Les tableaux: la méthode filter()

- Exemple:

```
const words = ['spray', 'elite', 'exuberant', 'destruction', 'present'];

const wordsFilter = words.filter(word => word.length > 6);
console.log("wordsFilter: ", wordsFilter) → wordsFilter: [ 'exuberant', 'destruction', 'present' ]

const filtreTexte = (entreeUtilisateur) => {
  return words.filter (
    element => element.indexOf(entreeUtilisateur) !== -1
  );
}
console.log('filtreTexte: ', filtreTexte('es')) → filtreTexte: [ 'destruction', 'present' ]
```

# La méthode reduce() (ES5)

---

- Exécute une fonction de réduction pour l'élément du tableau.
- **Renvoie une valeur unique : le résultat cumulé de la fonction.**
- N'exécute pas la fonction pour les éléments vides du tableau.
- **Ne modifie pas le tableau d'origine**, on le duplique

# La méthode reduce() (ES5)

---

- Syntax: `array.reduce(function(total, currentValue, currentIndex, arr), initialValue)`

Paramètres	Description	Obligatoire / optionnel
<code>function()</code>	fonction qui s'exécute sur chacun des éléments du tableau	Required
<code>total</code>	Total actuel de la fonction (InitialValue ou la valeur précédemment renvoyée de la fonction)	Required
<code>currentValue</code>	Valeur de l'élément actuel	Required
<code>currentIndex</code>	L'index de l'élément actuel	Optionnel
<code>arr</code>	Le tableau auquel appartient l'élément actuel	Optionnel
<code>initialValue</code>	Valeur à transmettre à la fonction comme valeur initiale	Optionnel

# La méthode reduce() (ES5)

- Exemple:

```
const arrayToReduce = [1,2,3,4]
```

```
// Somme de tous les chiffres du tableau
const sumArray = arrayToReduce.reduce((accumulator, currentValue) =>
|  accumulator + currentValue, 0
);
console.log(sumArray) -----> 10
```

```
// Somme de tous les chiffres au carré
const reduce = arrayToReduce.reduce(
  (accumulator, currentValue, index) => {
    console.log(`Index: ${index}, Carré: ${currentValue * currentValue}`);
    return accumulator + currentValue * currentValue;
  }, 0
);
console.log(reduce) -----> 30
```

```
Index: 0, Carré: 1
Index: 1, Carré: 4
Index: 2, Carré: 9
Index: 3, Carré: 16
```

# Les littéraux de modèle (ES6)

---

- Permettent de créer des chaînes de caractères de manière plus lisible et flexible en incorporant des expressions variables directement dans la chaîne.

```
const prenom = 'John';
const nom = 'Doe';

// Utilisation des backticks (`) pour délimiter le littéral de modèle
// Utilisation de ${} pour incorporer des expressions variables
const message = `Bonjour ${prenom} ${nom}!`;
console.log(message);
```

Bonjour John Doe!

- On peut aussi exécuter des méthodes ou fonctions fléchées

```
const array = [1,2,3,4]

const sommeValeursArray = `La somme de l'ensemble des éléments de mon tableau est ${array.reduce((accumulator, currentValue) => accumulator + currentValue)}`

console.log(sommeValeursArray)
```

La somme de l'ensemble des éléments de mon tableau est 10

# Les objets

---

- Les objets sont aussi des variables. Mais les objets peuvent contenir plusieurs valeurs.
- Les valeurs des objets sont écrites sous forme de paires nom : valeur (nom et valeur séparés par deux points).

```
let person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

- **Un objet JavaScript est une collection de valeurs nommées**
- Les valeurs nommées, dans les objets JavaScript, sont appelées propriétés.
- Les propriétés d'objet peuvent être à la fois des **valeurs primitives, d'autres objets et des fonctions**.

# Les objets: créer un objet javascript

---

Il existe différentes manières de créer de nouveaux objets :

- Créez un seul objet, en utilisant un littéral d'objet.
- Créez un seul objet, avec le mot-clé new.
- Définir un constructeur d'objet, puis créez des objets de type '*constructed*' -> via une **fonction** OU une **classe**
- Créez un objet en utilisant Object.create()

# Les objets: créer un objet javascript

---

- **Créer un objet**

```
const person =  
{firstName:"John",  
lastName:"Doe",  
age:50,  
eyeColor:"blue"};
```

- On peut créer un objet JavaScript vide, puis ajouter des propriétés:

```
const person = {};  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

```
const person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";
```

# Les objets: créer un objet javascript

---

- On crée un objet avec `Object.create(Object.prototype);`

```
const personneViaCreate = Object.create(Object.prototype);
```

OU

- En utilisant `Object.create(null);`

```
const personneViaCreate = Object.create(null);
```

Utilisation du prototype par défaut (`Object.prototype`)  
-> permet l'héritage des méthodes et propriétés standard de tous les objets JavaScript.

**La seule façon standard de créer un objet qui n'hérite pas du prototype par défaut `Object.prototype` est `object.create(null)`**

# Les objets: les 'class' (ES6)

---

## Pourquoi utiliser les classes ?

- Syntaxe plus concise, logique plus simple pour de la POO (programmation orientée objet) notamment via la définition des objets et des constructeurs
- Offrent un moyen plus structuré et facile à comprendre pour créer des objets et gérer l'héritage
- En résumé, les avantages:
  - Encapsulation et clarté: c'est un moyen de regrouper les données (propriétés) et les méthodes associées dans une seule entité
  - Héritage: les classes facilitent la mise en œuvre de l'héritage
  - Constructeur et initialisation: les classes permettent de définir un constructeur qui est appelé lors de la création d'une instance
  - Regrouper des méthodes statiques
  - ...

# Les objets: les 'class' (ES6)

- Syntax:

```
class ClassName {  
    constructor() { ... }  
}
```

- Exemple:

```
class Car {  
    constructor(name, year) {  
        this.name = name;  
        this.year = year;  
    }  
}
```

Définition du model

On crée l'*instance* d'un objet en utilisant le mot clé *new* et en utilisant les paramètres attendus par le constructeur

```
const myCar1 = new Car("Ford", 2014);  
const myCar2 = new Car("Audi", 2019);
```

Création selon le model

# Les objets: les 'class' (ES6)

---

- Les méthodes de classes:

```
class Car {  
    constructor(name, year) {  
        this.name = name;  
        this.year = year;  
    }  
    age() {  
        const date = new Date();  
        return date.getFullYear() - this.year;  
    }  
}  
  
car = new Car('Peugeot', 2010);  
  
console.log(car.age());
```

# Les objets: copier un objet

---

- Si je copie un objet, je dois utiliser 'Object.assign()'

```
monObjetCopié = Objet.assign(monObjet)
```

- Lorsqu'on utilise un simple égal (ex: '*monObjetCopié = monObjet*') en JavaScript, on ne crée pas réellement une nouvelle copie de l'objet. Au lieu de cela, vous créez une référence à l'objet existant. Les deux variables, *monObjetCopié* et *monObjet*, pointent vers le même emplacement mémoire où l'objet est stocké.
- **Il existe un autre méthode très utilisé en JS, en utilisant le 'spread operator' + la destructuration**

# Le spread operator (ES6)

---

- L'opérateur de propagation JavaScript (...) nous permet de copier rapidement tout ou partie **d'un tableau ou d'un objet** existant dans un autre tableau ou objet.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];

console.log(numbersCombined) —————> [ 1, 2, 3, 4, 5, 6 ]
```

- On peut attribuez le premier et le deuxième éléments des nombres aux variables et placez le reste dans un tableau :

```
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;

console.log(`one: ${one}, two: ${two}, rest: ${rest}`)
console.log(rest)                                one: 1, two: 2, rest: 3,4,5,6
                                                [ 3, 4, 5, 6 ]
```

# La destructuration: les tableaux (ES6)

---

- Assigner des variables avec la destructuration:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;

console.log(`car: ${car} __ truck: ${truck} __ suv: ${suv}`)
```

```
car: mustang __ truck: f-150 __ suv: expedition
```

```
const [car, , suv] = vehicles
console.log(`car: ${car} __ suv: ${suv}`)
```

```
car: mustang __ suv: expedition
```

# La destructuration: les tableaux (ES6)

---

- Exemple d'une fonction qui retourne un tableau de valeur:

```
const calculate = (a, b) => {
    const add = a + b;
    const subtract = a - b;
    const multiply = a * b;
    const divide = a / b;

    return [add, subtract, multiply, divide];
}

const [add, subtract, multiply, divide] = calculate(4, 7);

const [somme] = calculate(4,7)
const [,,division] = calculate(4,7)
```

# La destructuration: les objets (ES6)

---

- Permet d'extraire des valeurs d'objets et de les assigner à des variables en une seule ligne de code

```
const person = { name: 'John', age: 30, city: 'New York' };

// Destructuration
const { name, age, city } = person;

console.log(name); // 'John'
console.log(age); // 30
console.log(city); // 'New York'
```

```
const {age} = person
console.log(age) // 30
```

→ Pour assigner une seule des propriétés de 'person'

```
// Destructuration pour extraire la propriété 'name' et
l'assigner à une constante 'nom'
const { name: nom } = person;
console.log(nom); // 'John'
```

# La destructuration: les objets (ES6)

---

- Je peux utiliser, en paramètre de fonction uniquement les propriétés d'un objet

```
const person = { name: 'John', age: 30, city: 'New York' };

const personInfo = ({name, age}) => {
    console.log(` ${name}, ${age} ans`)
}
```

# Copier un objet: 'spreed operator' + la destructuration

---

```
const person = { name: 'John', age: 30, city: 'New York' };
const person2 = { ...person}
```

# La mémoire en JS

---

- Le cycle de vie d'un espace mémoire



- La gestion de la mémoire en JavaScript est effectuée par le moteur d'exécution JavaScript de chaque navigateur ou environnement d'exécution. Les deux principaux composants liés à la gestion de la mémoire sont le **tas (heap)** et la **pile d'appels (call stack)**, appelé **Stack**

# La mémoire en JS

---

## La stack

- Mémoire dite statique, utilisé pour l'allocation de mémoire dont on connaît la taille au moment de la compilation. Cela inclut:
  - Les valeurs primitives (number, string, boolean, undefined et null)
  - Les références à des objets ou des fonctions
- Liste élément qui répond au principe FILO (first-in, last-out) --> le premier élément ajouté sera le dernier à être effacé

# La mémoire en JS

---

## Heap

- Mémoire dynamique
- Utilisé pour l'allocation de mémoire dont on ne connaît la taille qu'à l'exécution
  - --> objets, tableaux et fonctions (object, array et function)
- A la différence de la stack, la heap n'alloue pas un espace mémoire de taille défini. L'espace peut évoluer lors de l'exécution. ( => donc mémoire dynamique)
- Le heap ne fonctionne pas seule !
  - Chaque espace alloué par le moteur JavaScript possède une référence à cette espace dans la stack.

# La mémoire en JS

```
let person =  
{firstName:"John",  
lastName:"Doe",  
age:50,  
eyeColor:"blue"};
```

