Ces trois déclarations peuvent sembler similaires mais elles ont des utilisations bien distinctes.

On va donc faire un tour rapide des 3 manières de déclarer une variable / constante en Javascript, sans trop rentrer dans les détails, juste pour comprendre les différences majeures entre les 3 instructions :

L'instruction let :

Elle permet de déclarer une variable d'une manière simple :

```
let maVar = true;
```

Il n'est pas obligatoire de déclarer une variable avec sa valeur initiale. Cela peut-être fait ensuite. Par exemple :

```
let maVar;
```

Comme c'est une variable, on peut modifier sa valeur après sa déclaration :

```
let maVar = 0;
console.log(maVar)
// Retourne 0;maVar = 12;
console.log(maVar)
// Retourne 12
```

Sa principale caractéristique est sa portée : elle est limité à celle du bloc courant. Pour rappel, un bloc en Javascript, c'est ce qu'on retrouve entre accolades : une comparaison en *if*, une boucle *while* etc...

```
function test(){
  if( 1 === 1 ){
    let maVar = true;
    console.log(maVar);
    // Retourne true
```

```
  }console.log(maVar);
  // Retourne une erreur, let n'existe pas hors du bloc "if"
}
```

L’instruction const :

Cette instruction permet de déclarer une constante disponible uniquement en lecture.

A l’inverse de *let*, *const* a besoin d’être déclaré avec une valeur initiale.

```
const CONFIG;
// Retourne une erreur, il manque la valeur
```

La bonne déclaration est donc :

```
const CONFIG = 'maConfig';
```

La portée de *const* est celle du bloc, comme la déclaration *let*. Pour la rendre globale, il faut simplement la définir hors de toute fonction.

```
function test(){
  if( 2 === 2 ){
    const CONFIG = 12;
    console.log(CONFIG);
    // Retourne 12
  }  console.log(CONFIG);
  // Retourne une erreur, config n'existe que dans le bloc
courant
}
```

Comme *const* déclare une constante, on ne peut pas en re-déclarer une qui partage le même nom, dans la même portée :

```
const CONFIG = true;const CONFIG = false;
// Retourne une erreur, on ne peut pas déclarer une autre
constante avec le même nomlet CONFIG = false;
// Retourne aussi une erreur, cela ne fonctionne pas non plus en
déclarant une variable avec le même nom
```

De la même manière, on ne peut pas changer sa valeur après déclaration :

```
const CONFIG = true;
CONFIG = false;
// Retourne une erreur car ne peut pas changer la valeur d'une constante
```

Ce dernier point est à nuancer : c'est plutôt la référence à la valeur qui ne peut pas être changée. On peut très bien ajouter un élément à un tableau déclaré via *const* (dans ce cas, on ne fait que changer la valeur référencée) :

```
const CONFIG = [];
CONFIG.push(12);
console.log(CONFIG);
// Retourne [12], car je peux ajouter un élément à mon tableau.CONFIG = true;
// Retourne une erreur, je ne peux pas changer la valeur de CONFIG
```

L'instruction var :

C'est la plus connue, celle qui existe depuis Javascript 1.0. On peut résumer, même si c'est plus subtile que ça, que *var* est similaire à *let*, mais avec une portée de fonction (là ou *let* a une portée de bloc).

Pour déclarer une variable, il suffit d'écrire :

```
var maVar = 'valeur';
```

Comme *let*, on peut aussi ne pas définir de valeur initiale, la variable aura donc une valeur *undefined*.

```
var maVariable;
console.log(maVariable);
// Retourne undefined
```

La portée de la variable est celle du contexte dans lequel elle est déclarée :

- Si elle est déclarée dans une fonction, la portée est celle de la fonction, qu'importe le bloc dans lequel elle se trouve.

- Si elle est déclarée hors d'une fonction, la portée sera celle du contexte global.

La déclaration sans instruction *var* (exemple : *maVar = 'test')* revient à écrire var *maVar = 'test'* dans un contexte global. La variable devient donc une propriété de l'objet global (Objet *window* en javascript).

```
function test(){
  maVar = 'valeur';
}
test();
console.log(maVar);
// Retourne 'valeur'
console.log(window.maVar);
// Retourne 'valeur'
```

Pour conclure

En 2017, on retrouve plus souvent *const* et *let* que *var*. Ca ne veut pas dire que *var* est une mauvaise pratique, mais *let* et *const* sont maintenant préférés à *var* grâce à leurs portées par bloc, plus subtile.

Les tableaux

Qu’est-ce qu’un tableau en JavaScript

Rapidement, regardons ce qu’est un tableau en réalité : Un tableau JS est un objet qui hérite de l’objet global standard ***Array***, qui lui même hérite l’objet fondamental ***Object***.

```
let tableau = new Array;
// tableau > Array.prototype > Object.prototype
```

Pour une définition plus accessible, on peut dire qu’un tableau contient une liste d’éléments, qu’ils soient du même type ou non.

Créer un tableau

En JavaScript, un tableau vide peut se créer de 2 manières :

```
let tableau = new Array;
// Ici c'est explicite, on crée une instance de l'objet Array
let tableau = [];
// Mais on peut aussi le déclarer de manière littérale avec []
```

Bien-sûr, on peut aussi créer des tableaux avec des valeurs initiales :

```
let tableau = new Array(5, 10, 15, 20);
let tableau = ['A', 'B', 'C'];
```

Il est aussi possible d’initialiser un tableau avec un certain nombre d’éléments vides, il suffit dans ce cas de mettre un entier en paramètre de ***Array*** :

```
let tableau = new Array(3);
console.log(tableau);
// Retourne [undefined × 3] soit un tableau de 3 éléments vides
```

Accéder aux éléments du tableau

Maintenant que nous avons un tableau, on va avoir besoin de récupérer les éléments dedans. Pour cela, on peut utiliser l'indice. Attention, pour rappel les indices d'un tableau débute à 0 et non à 1 :

```
let tableau = ['A', 'B', 'C'];
console.log(tableau[1]);
// Retourne 'B'
```

A l'inverse, on peut avoir besoin de récupérer l'indice à partir de l'élément associé. C'est la méthode ***indexOf()*** qui fait cela :

```
let tableau = ['A', 'B', 'C'];
console.log(tableau.indexOf('C'));
// Retourne 2
```

indexOf() a l'avantage d'avoir un deuxième paramètre optionnel, qui permet de choisir l'indice à partir duquel on débute la recherche. Par défaut, ce deuxième paramètre est à 0, mais on peut le modifier de cette manière :

```
let tableau = ['A', 'B', 'C', 'B'];
console.log(tableau.indexOf('B'));
// Retourne 1, l'indice du premier B
trouvéconsole.log(tableau.indexOf('B', 2));
// Retourne 3, l'indice du premier B trouvé après l'indice 2
```

Récupérer la taille un tableau

Pour obtenir le nombre d'éléments d'un tableau, c'est très simple, on utilise la propriété ***length*** de l'objet ***Array*** :

```
let tableau = ['A', 'B', 'C'];
let nbElements = tableau.length;
console.log(nbElements);
// Retourne 3 (et non 2, ici length compte le nombre d'éléments,
il ne se passe pas sur les indices)
```

Parcourir un tableau

Pour parcourir le contenu d'un tableau, l'objet ***Array*** dispose de sa propre méthode de boucle dans son prototype.

Il s'agit de ***forEach()***, et permet d'exécuter une fonction (et donc des instructions) pour chaque élément du tableau. Par exemple :

```
let tableau = ['A', 'B'];
tableau.forEach(function(element) {
  console.log(element);
});
// Retourne :
// 'A';
// 'B';
```

Vous remarquerez que l'on passe en paramètre de la fonction l'élément, ce qui nous permet de le récupérer et de l'utiliser dans nos instructions.

Ajouter un élément dans un tableau

Si l'on souhaite ajouter un élément, deux possibilités :

```
let tableau = ['A', 'B'];
tableau[] = 'C';
console.log(tableau);
// La manière littérale, retourne ['A', 'B', 'C']
tableau.push('D');
console.log(tableau);
// La manière objet, Retourne ['A', 'B', 'C', 'D']
```

Bien sûr, on peut ajouter n'importe quel type d'élément à un tableau : Un nombre, un booléen, une chaîne de caractère, un objet et même un autre tableau. Il n'est pas nécessaire que tous les éléments d'un tableau soient du même type :

```
let tableau = ['A', 'B'];
tableau.push(22);
```

```
console.log(tableau);
// Retourne ['A', 'B', 22];tableau.push( {id: 1, name: 'Nom'} );
console.log(tableau);
// Retourne ['A', 'B', 22, {id: 1, name: 'Nom'}];
```

Comme vous le voyez, la méthode ***push()*** ajoute notre élément à la fin du tableau. Si vous souhaitez l'ajouter au début, vous pouvez utiliser ***unshift()*** :

```
let tableau = ['A', 'B'];
tableau.unshift(22);
console.log(tableau);
// Retourne [22, 'A', 'B'];
```

Modifier un élément du tableau

Pour modifier la valeur d'un élément, on peut directement utiliser son indice :

```
let tableau = ['A', 'B', 'C'];
tableau[1] = 'D';
console.log(tableau);
// Retourne ['A', 'D', 'C'];
```

Supprimer un élément d'un tableau

Si vous avez besoin de supprimer le dernier élément d'un tableau, ***pop()*** est fait pour vous :

```
let tableau = ['A', 'B', 'C'];
tableau.pop();
console.log(tableau);
// Retourne ['A', 'B'];
```

Même principe avec ***shift()*** pour supprimer le premier élément du tableau :

```
let tableau = ['A', 'B', 'C'];
tableau.shift();
console.log(tableau);
// Retourne ['B', 'C'];
```

Trier un tableau

Il existe plusieurs méthodes de ***Array*** pour vous permettre de trier vos tableaux. Si vous souhaitez les classer par ordre alphabétique, utilisez ***sort()*** :

```
let tableau = ['E', 'A', 'D'];
tableau.sort();
console.log(tableau);
// Retourne ['A', 'D', 'E']
```

Si on souhaite simplement retourner le tableau en ayant les derniers éléments en premier, c'est ***reverse()*** qu'il faut utiliser :

```
let tableau = ['A', 'B', 'C'];
tableau.reverse();
console.log(tableau);
// Retourne ['C', 'B', 'A'];
```

Recherche un élément dans le tableau

Souvent, on a besoin d'un ou plusieurs éléments de notre tableau qui remplissent une condition.

Une première approche est d'utiliser la méthode ***findIndex()*** qui permet de remonter l'indice du premier élément de notre tableau qui rempli une condition. Par exemple :

```
function findC(element) {
  return element === 'C';
}

let tableau = ['A', 'B', 'C', 'D', 'C'];
tableau.findIndex(findC);
// Retourne 2, l'indice de notre premier C
```

This en javascript

En utilisant Javascript, on se rend rapidement compte de l'importance du this . Son utilisation est puissante, et il est nécessaire de bien comprendre son fonctionnement pour l'utiliser correctement.

this est un opérateur qui permet de retourner une valeur en fonction de son contexte. Deux éléments peuvent influer dessus :

- **Le contexte** dans lequel *this* est appelé : global, fonction, fonction fléchée, méthode d'un objet…
- **Le mode utilisé :** *strict* ou *non-strict*.

On va prendre chaque contexte un par un et regarder ce que retourne *this* selon le mode utilisé.

this dans un contexte global

Le contexte global, c'est lorsque l'on exécute du code en dehors de toute fonction, de tout bloc. Si on utilise *this* **dans ce contexte**, il retourne l'objet global :

```
console.log(this);
// Dans un navigateur, cela retourne l'objet DOM
windowthis.alert('Message');
// Ouvre une boite de dialogue avec comme contenu :
"Message".window.alert('Message');
// C'est donc le même résultat que window.alert()
```

En *mode strict*, *this* retourne exactement la même chose qu'en *mode non-strict* :

```
"use strict";
console.log(this);
// Retourne l'objet DOM window dans le navigateur
```

this dans une fonction classique

Lorsque l'on utilise *this* dans une fonction classique, il retourne un objet, l'objet global (comme si on l'utilise hors de toute fonction) :

```
function maFonction() {
  return this;
}maFonction();
// Dans un navigateur, cela retourne l'objet global window
```

En mode strict, c'est un peu différent, this retourne *undefined* :

```
function maFonction() {
  "use strict";
  return this;
}maFonction();
// Retourne undefined
```

Ce n'est pas plus compliqué que cela. Attention par contre, dans nos exemples nous avons utilisé *this* sans qu'il soit défini à son appel.

Grâce à *apply()* et *call()*, il est en effet possible de définir une valeur de *this* au sein de la fonction, ce qui change forcément l'utilisation que l'on peut en faire.

this dans une fonction fléchée (arrow function)

Le cas des fonctions fléchées est un peu particulier car elles ne créent pas de nouveau contexte. Du coup, la valeur

de *this* reprend celle du contexte dans laquelle la fonction est définie. Par exemple :

```
const maFonction = () => { return this };
maFonction();
// Retourne window dans mon navigateur, this ayant la valeur de
l'objet global.
```

Maintenant, si on utilise this dans une fonction fléchée définie dans une méthode d'un objet, this retourne bien notre objet :

```
const commentaire = {
  getAuthor: function(){
    const value = (() => this);
    return value;
  },
};const test = commentaire.getAuthor();console.log(test());
// Retourne bien notre objet complet
```

En mode strict, le comportement est exactement le même :

```
const maFonction = () => { "use strict"; return this };
maFonction();
// Retourne window dans mon navigateur, this ayant la valeur de
l'objet global.
```

this dans une méthode d'un objet

En POO, il est souvent nécessaire d'accéder aux propriétés et méthodes de l'objet courant pour les utiliser ailleurs. *this* est parfait pour y faire référence.

Dans ce contexte, *this* retourne l'objet dans lequel la méthode est définie. Par exemple :

```
const commentaire = {
  id: 7,
  author: 'Jean',
  content: 'Juste un simple commentaire',
  getAuthor: function(){
    return this.author
  },
```

```
};commentaire.getAuthor();
// Retourne 'Jean', this faisant référence à notre objet
commentaire, this.author correspond à l'auteur de commentaire.
```

En mode strict, le comportement est exactement le même :

```
"use strict";const article = {
  title: 'Un article',
  content: 'Contenu de mon article',
};article.getTitle = function(){
  return this.title;
};article.getTitle();
// Retourne bien "Un article"
```

Dans le cas de l'utilisation de *this* dans une méthode de type constructeur, c'est le même principe : this fait référence à l'objet nouvellement créé :

```
function Jeu(){
  this.points = 120;
}

var j = new Jeu();
console.log(Jeu.points);
// Retourne 120
```

En mode strict, la valeur de *this* est similaire au mode non-strict.