

# Javascript

## Scopes

### 1) Un peu de théorie – le Scope :

En JavaScript, les variables ont une portée qu'on appelle **scope**. En effet, en fonction de l'emplacement où tu déclare ta variable, sa portée peut être différente. Par exemple : si tu la declares dans une fonction (à l'intérieur de ses accolades), la variable ne peut être utilisée que par cette fonction.

Ca te semble abstrait ? On va illustrer ça. Pour comprendre le scope, tu dois savoir qu'il existe deux types de variables : **les variables locales et les variables globales**.

Variable globale	Console.log
<pre>var tutu = "tutu"; // variable globale  function maFonction() {     return tutu; // elle peut être utilisée n'importe où dans le script, y compris dans une fonction. } maFonction() ;</pre>	<pre>&gt;&gt; ▶ var tutu = "tutu";        function maFonction() {           return tutu;       }... ← "tutu"</pre>

Variable locale	Console.log
<pre>function maFonction() {     var toto = "toto"; // variable locale, elle ne peut pas être utilisée en dehors de sa fonction.     return toto; }  console.log(toto) ; // si je fais un console.log de cette variable à l'extérieur de la fonction, elle n'est pas reconnue</pre>	<pre>&gt;&gt; ▶ function maFonction() {       var toto = "toto";       return toto;     }     ... ❗ ▶ Uncaught ReferenceError: toto is not defined     &lt;anonymous&gt; debugger eval code:6     [En savoir plus]</pre>

## Récap :

Variable globale	Variable locale
<pre>// le code ici ne peut PAS utiliser carName  function myFunction() {     var carName = "Volvo";     // le code ici peut utiliser     carName }</pre>	<pre>var carName = "Volvo";     // le code ici peut utiliser     carName function myFunction() {     // le code ici aussi peut     utiliser carName }</pre>



Les **variables globales** sont stockées en mémoire dans votre navigateur. Cela signifie qu'elles prennent plus de mémoire que les variables locales, et augmente aussi le risque de conflits de nommage avec d'autres scripts. C'est pourquoi, tu dois privilégier l'utilisation des variables locales autant que possible.

## Exemple 1 :

```
//On déclare deux variables globales  
let x = 5;  
var y = 10;  
//Première fonction qui utilise les variables globales x et y  
function portee1(){  
    console.log('Portée 1 : x = ' + x + ' y = ' + y) ;}  
//Deuxième fonction qui utilise les variables locales a et b  
function portee2(){  
    let a = 1;  
    var b = 2;  
    console.log('Portée 2 : a = ' + a + ' b = ' + b) ;}  
//Troisième fonction qui utilise également des variables locales x et y  
function portee3(){  
    let x = 20;  
    var y = 40;  
    console.log('Portée 3 : x = ' + x + ' y = ' + y) ;}  
//On pense bien à exécuter nos fonctions !  
portee1();  
portee2();  
portee3();  
//On console.log des variables globales puis locales depuis l'espace global  
console.log('Espace global : x = ' + x + ' y = ' + y) ;  
console.log('Espace global : a = ' + a + ' b = ' + b) ; //ces variables sont  
hors portée à partir de la portée globale
```

## Console.log :

```
>> ▶ var x = 5;
      var y = 10;

      function portee1(){
        console.log('Portée 1 : x = ' + x + ' y = ' + y) ;}..
```

Portée 1 : x = 5 y = 10

Portée 2 : a = 1 b = 2

Portée 3 : x = 20 y = 40

Espace global : x = 5 y = 10

❗ ▶ Uncaught ReferenceError: a is not defined  
<anonymous> debugger eval code:23  
[\[En savoir plus\]](#)



Vu que les variables a et b ont été déclarées  
à l'intérieur de la fonction « portée2 »,  
elles ne sont pas accessibles  
à partir de la portée globale

Lorsqu'une variable est globale,  
elle sera accessible **par son nom**  
ou par l'objet global **window**.  
Exemple :

```
var monNom = "Dominic";
console.log(monNom); //"Dominic"
console.log(window.monNom); //"Dominic"
```

## Exemple 2

<pre>var monNom = "Dominic";  function mettre_a_jour_nom() {   var monNom = "Sara"; }  mettre_a_jour_nom(); console.log(monNom); console.log(window.monNom);</pre>	<p>// Le console.log est dans la portée globale, donc, il va ignorer monNom à l'intérieur de la fonction qui a une portée locale</p> <pre>//"Dominic" //"Dominic"</pre>
<pre>var monNom = "Dominic";  function mettre_a_jour_nom() {   var monNom = "Sara";   console.log(monNom);   console.log(window.monNom); }  mettre_a_jour_nom();</pre>	<p>// Un console.log est dans la portée locale, donc, il va appeler monNom à l'intérieur de la fonction. Le deuxième console.log fait appel à l'objet window, donc, il va chercher la variable globale</p> <pre>//"Sara" //"Dominic"</pre>

<pre> var monNom = "Dominic";  function mettre_a_jour_nom() {   monNom = "Sara";   console.log(monNom);   console.log(window.monNom); } mettre_a_jour_nom(); </pre>	<p>// Si on enlève l'instruction "var" de la variable à l'intérieur de la fonction, on re-initialise la variable globale monNom = "Dominique" et on remplace sa valeur par monNom = "Sara". Nous pouvons donc modifier les variables globales avec l'instruction var même à l'intérieur d'une fonction (mais pas les variables avec les instructions <b>let</b> – qui ont toujours une portée locale – ni <b>const</b> – qui ne peuvent pas être initialisées)</p> <pre> // "Sara" // "Sara" </pre>
<pre> var monNom = "Dominic";  function mettre_a_jour_nom() {   var monNom = "Lucas";   console.log(monNom);   console.log(window.monNom); } mettre_a_jour_nom();  function mettre_a_jour_nom2() {   monNom = "Sara";   console.log(monNom);   console.log(window.monNom); } mettre_a_jour_nom2(); </pre>	<p>// à l'intérieur de la fonction mettre_a_jour_nom2 on écrase "Dominic" par "Sara" car nous avons enlevé l'instruction « var »</p> <pre> // "Lucas" // "Dominic" // "Sara" // "Sara" </pre>
<pre> function mettre_a_jour_nom() {   monNom = "Sara"; }  mettre_a_jour_nom(); console.log(monNom); console.log(window.monNom); </pre>	<p>// monNom va être créé dans la portée globale, à cause de l'absence de l'instruction var.</p> <pre> // "Sara" // "Sara" </pre>
<pre> function mettre_a_jour_nom() {   var monNom = "Sara";    function mettre_a_jour_nom2() {     var monNom = "Dominic";     console.log(monNom);   }    mettre_a_jour_nom2()   console.log(monNom); }  mettre_a_jour_nom(); </pre>	<p>// Ici, même si monNom est déclaré à l'intérieur d'une fonction pour les deux cas, JavaScript va chercher la portée de fonction la plus proche. Donc il va considérer var monNom = "Sara" et var monNom = "Dominic" comme étant deux variables différentes.</p> <pre> // "Dominic" // "Sara" </pre>

<pre>function mettre_a_jour_nom() { var monNom = "Sara";  function mettre_a_jour_nom2() { console.log(monNom); }  mettre_a_jour_nom2() console.log(monNom); }  mettre_a_jour_nom();</pre>	<p><i>// Ici JavaScript ne va pas trouver une déclaration de monNom dans la portée de mettre_a_jour_nom2, donc il va monter d'un niveau, et il va trouver une déclaration de monNom dans la portée de mettre_a_jour_nom et il va utiliser celle-ci.</i></p> <pre>//"Sara" //"Sara"</pre>
<pre>var monNom = "Sara";  function mettre_a_jour_nom() { function mettre_a_jour_nom2() { console.log(monNom); }  mettre_a_jour_nom2() console.log(monNom); }  mettre_a_jour_nom();</pre>	<p><i>// Dans cette exemple, JavaScript recherche d'abord une déclaration de monNom à l'intérieur de la fonction mettre_a_jour_nom2, s'il ne la trouve pas, il va monter d'un niveau vers la fonction mettre_a_jour_nom, puis s'il ne trouve pas une déclaration de monNom, il va monter vers la portée global.</i></p> <pre>//"Sara" //"Sara"</pre>

## Récap :


- JavaScript a une portée globale et une portée locale.
- Les variables déclarées et initialisées en dehors de toute fonction deviennent des variables globales.
- Les variables déclarées et initialisées à l'intérieur de la fonction deviennent des variables locales de cette fonction.
- Les variables déclarées sans mot-clé var dans une fonction deviennent automatiquement des variables globales.
- Les variables globales peuvent être consultées et modifiées n'importe où dans le programme.
- Les variables locales ne sont pas accessibles en dehors de la déclaration de fonction.
- La variable globale et la variable locale peuvent avoir le même nom sans s'affecter mutuellement.

## 2) La différence de scope entre let, var et const

### 2.1) Hoisting (ou hissing en français) de variables

Le hoisting (ou hissing, en français) est un mécanisme JavaScript dans lequel les variables et les déclarations de fonction sont déplacées vers le haut de leur scope avant l'exécution du code, que leur portée soit globale ou locale.

#### Exemple de hoisting pour les variables:

 <code>console.log(x); // On utilise x</code> <code><b>var x;</b> // Mais on déclare x <u>après</u></code> <code>l'avoir utilisé</code>  <code>On obtient un undefined et pas une</code> <code>erreur « Uncaught ReferenceError: x</code> <code>is not defined » car le hoisting à</code> <code>hissé la var x ; au dessus de</code> <code>console.log</code>	<pre>&gt;&gt; console.log(x); var x;  undefined</pre>
---	---

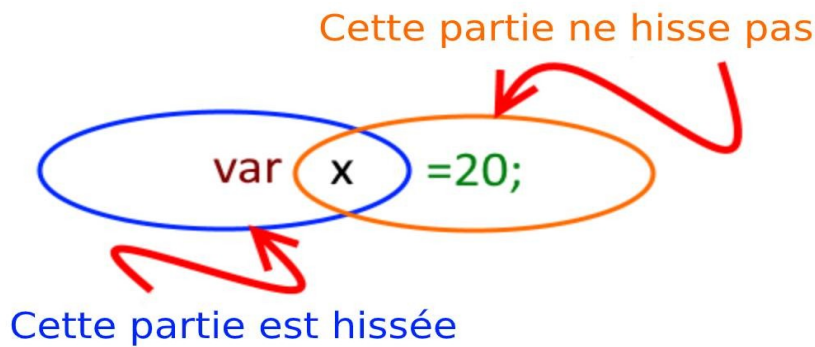
Note que le hoisting ne déplace que la déclaration. Les affectations sont laissées en place.

<code>console.log(x); // On utilise x</code> <code><b>var x = 5;</b> // Mais on déclare x</code> <code><u>après</u> l'avoir utilisé et on</code> <code>l'affecte le number 5</code>  <code>On obtient un undefined car le</code> <code>hoisting à hissé la déclaration de</code> <code>variable mais pas son affectation</code>	<pre>&gt;&gt; console.log(x); var x = 5;  undefined</pre>
--	---

**Un autre exemple :** ces deux exemples ne donnent pas le même résultat

<code><b>var x = 5;</b></code> <code><b>var y = 7;</b></code> <code>console.log(x) ;</code> <code>console.log(y) ;</code>	<code><b>var x = 5;</b></code> <code>console.log(x) ;</code> <code>console.log(y) ;</code> <code><b>var y = 7;</b> //hoisting de la</code> <code>déclaration mais pas de sa</code> <code>valeur affectée (donc, y sera</code> <code>donné comme undefined)</code>
<pre>&gt;&gt; var x = 5; var y = 7; console.log(x) ; console.log(y) ;  5 7</pre>	<pre>&gt;&gt; var x = 5; console.log(x) ; console.log(y) ; var y = 7;  5 undefined</pre>

Autrement dit :



Comme nous avons vu dans les exemples de scope, l'affectation d'une valeur à une variable **non déclarée (sans l'instruction var, let ou const)** la transforme en variable globale.

```
function  
mettre_a_jour_nom() {  
  monNom = "Sara";  
  //affectation de la valeur  
  "Sara" à une variable non-  
  déclarée (sans var, const  
  ou let)  
}  
mettre_a_jour_nom();  
console.log(window.monNom);  
//on appelle la variable  
globale mon.Nom et, à la  
place d'undefined, on  
obtient "Sara"
```

```
>> function mettre_a_jour_nom() {  
  monNom = "Sara";  
}  
mettre_a_jour_nom();  
console.log(window.monNom);
```

Sara

Cela signifie que:



**Toutes les variables non déclarées (sans const, let ou var)  
deviennent des variables globales.**

**Il est donc recommandé de toujours déclarer les variables**, qu'elles soient dans une fonction ou une portée globale.

```
function hoist () {  
  a = 20; // variable non-déclarée  
  var b = 100;  
}  
  
hoist();  
console.log(a);  
/* La variable a est accessible en tant que variable globale en  
dehors de la fonction hoist ()  
Sortie: 20  
*/  
  
console.log (b);  
/* La variable b est confinée dans la portée de la fonction  
hoist(). Nous ne pouvons pas l'appeler en dehors de ce scope.  
Sortie: Uncaught ReferenceError: b is not defined  
*/
```

## 2.3) ES6 – let, const et l'hosting

Contrairement à l'instruction « var », l'utilisation d'une variable **let ou const** avant qu'elle ne soit déclarée entraînera une ReferenceError.

Avec var (hissé) :

```
carName = "Volvo";  
var carName;
```

```
>> carName = "Volvo";  
    var carName;  
  
← "Volvo"
```

Avec let (non-hissé) :

```
carName = "Volvo";  
let carName;
```

```
>> carName = "Volvo";  
    let carName;
```

```
❗ ▶ Uncaught ReferenceError: can't access  
lexical declaration 'carName' before  
initialization  
    <anonymous> debugger eval code:1
```



Avec **const** (non-hissé et qui, en plus, doit impérativement avoir une valeur affectée dès sa déclaration) :

<pre>carName = "Volvo"; const carName;</pre>	<pre>&gt;&gt; carName = "Volvo"; const carName;</pre> <div>❗ Uncaught SyntaxError: missing = in const declaration [En savoir plus]</div>
--	--

## 2.4) Hoisting de fonction

Comme nous avons vu dans la **fiche 4** – suite, les fonctions JavaScript sont classées comme suit:

Déclarations de fonction :	<pre>function toto() {   console.log('Voici une déclaration   de fonction'); };</pre>
Expressions de fonction	<pre>var toto = function() {   console.log('Voici une expression de   fonction'); };</pre>

### Le hoisting dans les déclarations de fonction

Les déclarations de fonction sont hissées complètement vers le haut. Nous pouvons donc invoquer une fonction avant de la déclarer.

<pre>hoisted(); //fonction invoquée function hoisted() {   console.log('Cette   fonction a été hissée au   dessus de son invocation'); };</pre>	<pre>&gt;&gt; hoisted(); function hoisted() {   console.log('Cette fonction a été hissée au dessus   de son invocation'); };</pre> <div>Cette fonction a été hissée au debugger eval code:3:11 dessus de son invocation</div> <div>← undefined</div>
---	--

## Le hoisting dans les expressions de fonction

Les expressions de fonction ne sont pas hissées et l'interpréteur affichera une erreur `TypeError` car il voit l'expression de fonction comme une variable et non comme une fonction.

<pre>expression(); //fonction invquée var expression = function() {   console.log('Oups, une erreur...'); }</pre>	<pre>&gt;&gt; expression(); var expression = function() {   console.log('Oups, une erreur...'); }</pre> <div><div>! ▶ Uncaught TypeError: debugger</div><div>expression is not a function</div><div>&lt;anonymous&gt; ...er eval code:1</div><div><a href="#">[En savoir plus]</a></div></div>
---	--



### Voici l'ordre de priorités du hoisting :

- 1** L'affectation de variable a priorité sur la déclaration de fonction  
*(Pour rappel, une affectation de variable est une variable qui comporte une valeur)*
- 2** Les déclarations de fonction ont priorité sur les déclarations de variables  
*(Pour rappel, une déclaration de variable est une variable qui est déclarée mais qui ne comporte une valeur)*

### 1 Priorité de l'affectation de variable sur la déclaration de fonction :

<pre>function toto() {   console.log('Je suis une fonction') } var toto = 22; console.log(typeof toto);</pre>	<pre>&gt;&gt; function toto() {   console.log('Je suis une fonction') } var toto = 22; console.log(typeof toto);</pre> <div>number debugger</div>
---	---

## 2 Priorité de la déclaration de fonction sur la déclaration de variables

<pre>var toto; function toto() {   console.log('Je suis une fonction') }  console.log(typeof toto);</pre>	<pre>&gt;&gt; ▶ var toto;       function toto() {         console.log('Je suis une fonction')       }       ... function                                     debugger</pre>
---	---

### 2.5) Mode strict

D'après son nom, il s'agit d'une variante restreinte de JavaScript qui ne tolérera pas l'utilisation de variables avant qu'elles ne soient déclarées.

Le mode strict est déclaré en ajoutant "use strict"; au début d'un script. Déclaré au début d'un script, il a une portée globale (tout le code du script s'exécutera en mode strict).

Comme utiliser le « mode strict » :

<pre>'use strict';  console.log (hoist); // Sortie: Uncaught ReferenceError: hoist is not defined  hoist = 'Hissé';</pre>
---

### Exemple - Mode strict contre les variables non-déclarées :

Avec mode strict	Sans mode strict
<pre>'use strict'; hoist = 3.14; //variable non-déclarée</pre>	<pre>hoist = 3.14; //variable non-déclarée</pre>
<pre>&gt;&gt; "use strict";     hoist = 3.14;  ❗ ▶ Uncaught ReferenceError: assignment to    undeclared variable hoist    &lt;anonymous&gt; debugger eval code:2    [En savoir plus]</pre>	<pre>&gt;&gt; hoist = 3.14; ← 3.14</pre>

## Pourquoi utiliser le « mode strict » :

- Le mode strict facilite l'écriture d'un JavaScript «sécurisé».
- Il change la «mauvaise syntaxe» précédemment acceptée en erreurs réelles. Par exemple, en JavaScript normal, la saisie incorrecte d'un nom de variable crée une nouvelle variable globale. En mode strict, cela générera une erreur, ce qui rendra impossible la création accidentelle d'une variable globale.

## Non autorisé en mode strict :

L'utilisation d'une variable, sans la déclarer	"use strict"; x = 3.14;
Utiliser un objet, sans le déclarer	"use strict"; x = {p1:10, p2:20};
La suppression d'une variable (ou d'un objet)	"use strict"; var x = 3.14; delete x;
Supprimer une fonction	"use strict"; function x(p1, p2) {}; delete x;
Dupliquer un nom de paramètre	"use strict"; function x(p1, p1) {};
Le mot arguments ne peut pas être utilisé comme nom de variable	"use strict"; var arguments = 3.14;

...etc (nous allons revoir ce concept)