

# Fonctions, closures et espaces de nom (name spaces)

# Sommaire

Introduction.....	3
<b>I. Variables globales</b> .....	4
A. Visualiser les variables globales à la console JavaScript.....	4
<b>II. Closures</b> .....	5
A. Pourquoi faut-il utiliser <i>var</i> pour définir les variables locales?.....	7
B. Closure et gestionnaire d'évènements.....	8
C. Éviter les collisions de variables.....	9
D. Rendre son espace de nom hermétique à l'aide d'un objet anonyme.....	9
E. Pas facile à lire, non?.....	10

Crédits des illustrations:  
© DR.

## Les repères de lecture



Retour au chapitre



Définition



Objectif(s)



Espace Élèves



Vidéo /  
Audio



Point important /  
À retenir



Remarque(s)



Pour aller  
plus loin



Normes et lois



Quiz

# Introduction

En JavaScript, peut-être plus que tout autre langage, il est important de bien maîtriser l'accessibilité des variables. En effet, le langage JavaScript possède un mécanisme assez original dit de closure, très pratique pour partager facilement les variables entre différentes fonctions ou différents scripts, mais qui peut aussi générer des erreurs s'il n'est pas bien compris et maîtrisé.

Les gestionnaires d'événements utilisant des fonctions callback imbriquées, il est important de comprendre la passation de variables entre votre code appelant et les fonctions appelées.

Par ailleurs, il est très fréquent aujourd'hui d'utiliser plusieurs scripts dans vos pages HTML – des scripts que vous avez écrits vous-même ou des scripts écrits par d'autres. Dans ce contexte, il est important de veiller à éviter que les variables que vous utilisez dans votre script ne soient écrasées par d'autres scripts. De même, vous devez veiller à ne pas écraser sans le savoir les variables d'autres scripts.

## I. Variables globales

Une variable ou une fonction en JavaScript est un objet avec des méthodes et des propriétés; et chaque instruction fait partie d'un objet et il peut donc accéder aux méthodes et aux propriétés de cet objet.

Vous pouvez récupérer l'objet dans lequel s'exécute votre code à l'aide de la variable système **this**. Par exemple, si vous ouvrez la console JavaScript de votre navigateur et que vous tapez « **this** », vous récupérez l'objet **window initialisé par le navigateur**, c'est l'objet racine dans lequel tout script démarre son exécution. Vous pouvez le constater par vous-même en ouvrant la console JavaScript sur n'importe quelle page web :

```
> this
» Window {top: Window, window: Window, location: Location, external: Object, chrome: Object...}
> typeof this
"object"
> |
```

Fig. 1

Toute variable peut être vue comme une propriété de l'objet à partir duquel elle est peut être initialisée et modifiée.

Pour accéder à une propriété ou une variable, vous pouvez donc utiliser son nom ou la notation *objet.propriété*. Les instructions suivantes sont donc équivalentes :

```
var a="coucou";
this.a = "coucou";
windows.a = "coucou";
```

Fig.2

### A. Visualiser les variables globales à la console JavaScript

Voici un moyen très simple pour visualiser les variables globales, celles initialisées lors de la création du DOM par le navigateur et celles utilisées par les scripts.

Ouvrez la console du debugger JavaScript de Chrome et commencez à taper **window**.

L'autocomplétion vous affichera la liste des méthodes et propriétés globales courantes :

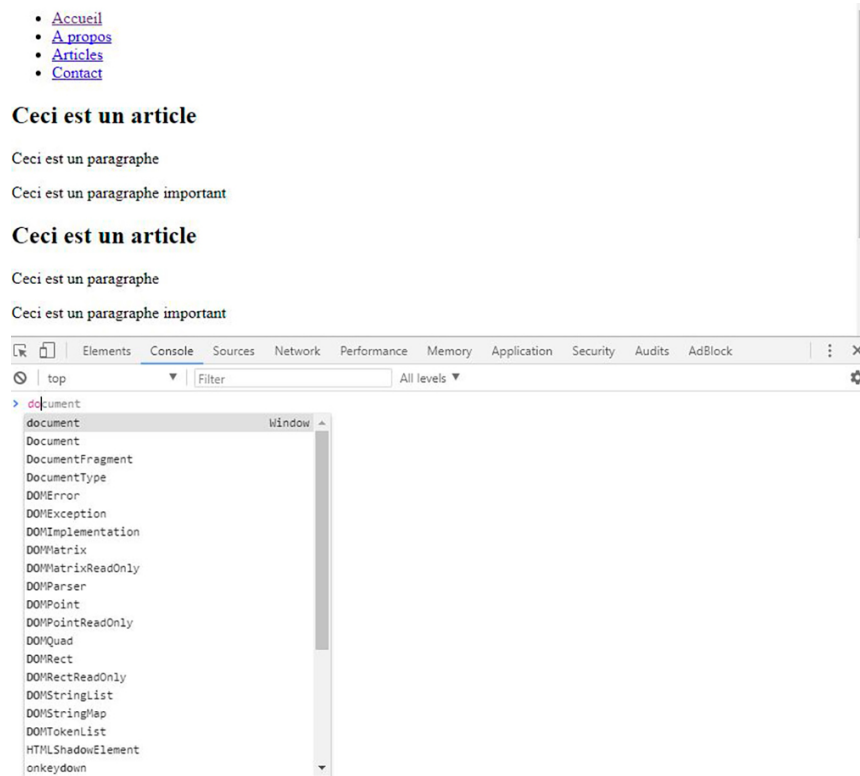


Fig.3

## II. Closures

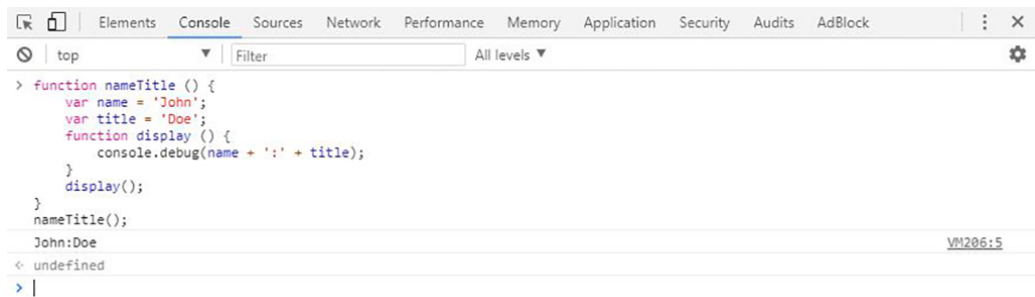
Le système de closure permet typiquement à une fonction fille de partager les variables définies dans une fonction parent.

*Exemple :*

```
1 function nameTitle () {  
2     var name = 'John';  
3     var title = 'Doe';  
4     function display () {  
5         console.debug(name + ':' + title);  
6     }  
7     display();  
8 }  
9 nameTitle();
```

Fig.4

Résultat affiché à la console JavaScript:



```
> function nameTitle () {  
  var name = 'John';  
  var title = 'Doe';  
  function display () {  
    console.debug(name + ':' + title);  
  }  
  display();  
}  
nameTitle();  
John:Doe  
undefined  
> |
```

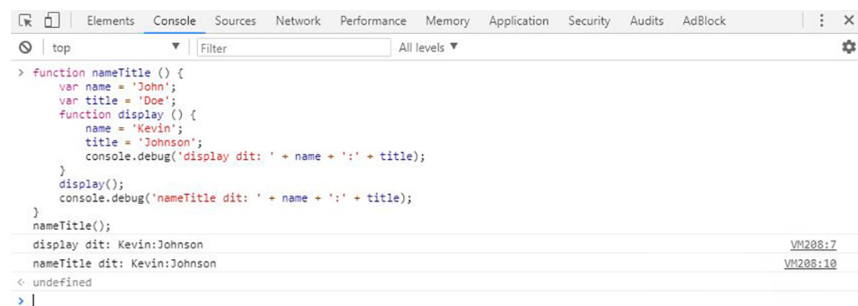
Fig.5

Comme vous pouvez le constater, la fonction imbriquée `display()` a accès aux variables `name` et `title` de la fonction parent. Nous aurions pu passer les variables `name` et `title` en argument de la fonction `display` pour parvenir au même résultat:

```
1 function nameTitle () {  
2   var name = 'John';  
3   var title = 'Doe';  
4   function display () {  
5     name = 'Kevin';  
6     title = 'Johnson';  
7     console.debug('display dit: ' + name + ':' + title);  
8   }  
9   display();  
10  console.debug('nameTitle dit: ' + name + ':' + title);  
11 }  
12 nameTitle();
```

Fig.6

Résultat affiché à la console JavaScript:



```
> function nameTitle () {  
  var name = 'John';  
  var title = 'Doe';  
  function display (name, title) {  
    name = 'Kevin';  
    title = 'Johnson';  
    console.debug('display dit: ' + name + ':' + title);  
  }  
  display();  
  console.debug('nameTitle dit: ' + name + ':' + title);  
}  
nameTitle();  
display dit: Kevin:Johnson  
nameTitle dit: Kevin:Johnson  
undefined  
> |
```

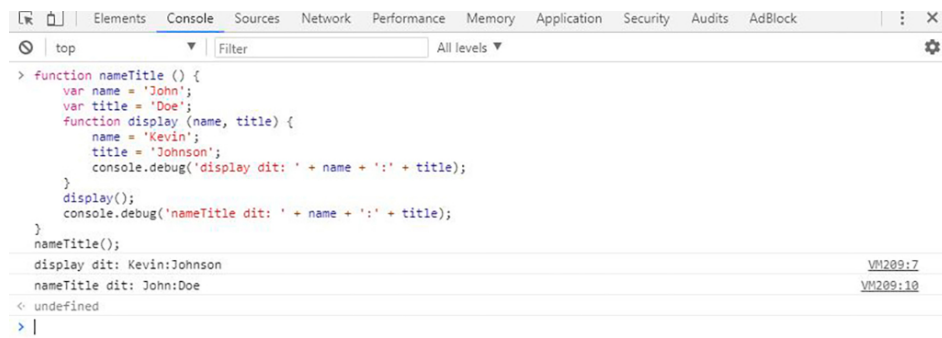
Fig.7

Mais lorsque l'on passe une variable en argument, celle-ci est clonée, ce n'est donc plus la même variable:

```
1 function nameTitle () {  
2   var name = 'John';  
3   var title = 'Doe';  
4   function display (name, title) {  
5     name = 'Kevin';  
6     title = 'Johnson';  
7     console.debug('display dit: ' + name + ':' + title);  
8   }  
9   display();  
10  console.debug('nameTitle dit: ' + name + ':' + title);  
11 }  
12 nameTitle();
```

Fig.8

Le résultat affiché à la console, prouve que les variables ne sont pas partagées :



```
> function nameTitle () {  
  var name = 'John';  
  var title = 'Doe';  
  function display (name, title) {  
    name = 'Kevin';  
    title = 'Johnson';  
    console.debug('display dit: ' + name + ':' + title);  
  }  
  display();  
  console.debug('nameTitle dit: ' + name + ':' + title);  
}  
nameTitle();  
display dit: Kevin:Johnson  
nameTitle dit: John:Doe  
< undefined  
> |
```

Fig.9

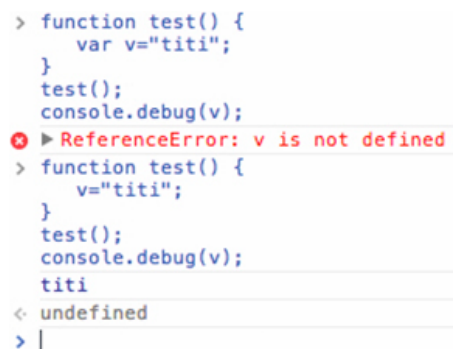
Dans une *closure*, les variables *name* et *title* sont comme des variables globales mais dont l'étendue d'accessibilité serait limitée par le haut à la fonction où elles sont définies. C'est ainsi qu'il faut comprendre le terme de *closure* (clôture en français).

## A. Pourquoi faut-il utiliser *var* pour définir les variables locales ?

Une variable définie à l'aide de **var**, définit une variable locale à la fonction existante, un début de *closure* (les fonctions ancêtres n'auront pas accès à cette variable).

Si, dans une fonction, vous omettez de déclarer vos variables locales (celles que vous savez n'être utilisées que dans la fonction), la variable sera définie dans la *closure* actuelle, les fonctions ancêtres de la *closure* auront donc accès à cette variable.

Vérifiez ce comportement à la console JavaScript :



```
> function test() {  
  var v="titi";  
}  
test();  
console.debug(v);  
✖ ▶ ReferenceError: v is not defined  
> function test() {  
  v="titi";  
}  
test();  
console.debug(v);  
titi  
< undefined  
> |
```

Fig. 10

**Particularité intéressante des *closures* : une fonction définissant une variable partage celle-ci avec toutes ses fonctions filles, quel que soit le niveau d'imbrication.**

Si vous oubliez de déclarer vos variables locales avec `var`, vous courrez donc le risque d'utiliser une variable utilisée par une fonction parent, c'est pourquoi, il faut prendre l'habitude de déclarer ces variables.

## B. Closure et gestionnaire d'évènements

Les gestionnaires d'évènement font appel à une fonction callback. La fonction callback partage donc les mêmes variables que la closure dont elle fait partie. Afin de le vérifier, modifiez ainsi votre exemple1.js :

```
window.addEventListener('load', function(event) {
  var element=document.getElementsByTagName('H1')[0];
  var color1="red";
  var color2="blue";
  element.addEventListener('mousedown', function(event) {
    this.style.color=color1;
  });
  element.addEventListener('mouseup', function(event) {
    this.style.color=color2;
  });
});
```

Fig. 11

Nous avons ici défini **2 variables color1 et color2**, ce sont ces variables que nous utilisons dans les fonctions callbacks qui appartiennent à la même closure.

Compliquons notre exemple par l'utilisation d'une fonction qui positionnera les couleurs :

```
window.addEventListener('load', function(event) {
  var color1="red";
  var color2="blue";
  function setColor1(element) {
    element.style.color = color1;
  }
  function setColor2(element) {
    element.style.color = color2;
  }
  var element=document.getElementsByTagName('H1')[0];
  element.addEventListener('mousedown', function(event) {
    setColor1(this);
  });
  element.addEventListener('mouseup', function(event) {
    setColor2(this);
  });
});
```

Fig. 12

Les fonctions `setColor1()` et `setColor2()` sont accessibles à partir des fonctions callback car elles font parties de la même closure. Les variables `color1` et `color2` sont pour la même raison accessibles à partir des fonctions `setColor1()` et `setColor2()` ;.



## C. Éviter les collisions de variables

Éviter les collisions de variables, c'est empêcher que 2 scripts utilisent les mêmes variables sans que cela ne soit souhaitable.

Pour éviter les collisions, vous devez donc créer le moins possible de variables, et, pour se faire, l'usage veut que vous utilisiez dans votre code des espaces de noms privés. Vous créez un espace de nom privé en créant un objet JavaScript en lui donnant un nom suffisamment différenciant pour éviter les collisions :

```
var exemple1 = {  
  init: function() {  
    window.addEventListener('load', function(event) {  
      var element=document.getElementsByTagName('H1')[0];  
      element.addEventListener('mousedown', function(event) {  
        this.style.color="red";  
      });  
      element.addEventListener('mouseup', function(event) {  
        this.style.color="blue";  
      });  
    });  
  }  
};  
exemple1.init();
```

Fig. 13

## D. Rendre son espace de nom hermétique à l'aide d'un objet anonyme

Dans l'exemple précédent, votre code n'est pas protégé complètement. En effet, n'importe quel script pourra avoir accès à exemple 1 et exemple 1.init et le modifier.

Si vous voulez rendre le code absolument inaccessible, la solution ultime, largement utilisée est d'englober le tout dans une fonction anonyme, c'est-à-dire sans nom et auto exécutable, sous la forme :

```
(function() { votre code }) () ;
```

Fig. 14

**(function() { votre code })** permet de définir une fonction anonyme ;

**(function() { votre code })()** l'exécute.

En l'appliquant à exemple1.js, cela donne :

```
(function() {  
  var exemple1 = {  
    init: function() {  
      window.addEventListener('load', function(event) {  
        var element=document.getElementsByTagName('H1')[0];  
        element.addEventListener('mousedown', function(event) {  
          this.style.color="red";  
        });  
        element.addEventListener('mouseup', function(event) {  
          this.style.color="blue";  
        });  
      });  
    }  
  };  
  exemple1.init();  
})();
```

Fig. 15

## E. Pas facile à lire, non ?

Mais pourtant largement utilisé – notamment par les « plugins » JavaScript – car ainsi les fonctions et propriétés définies dans le code ne peuvent plus être accessibles ou modifiées par un script externe, son espace de nom est hermétique.

Vous savez maintenant correctement initialiser des gestionnaires d'événement et éviter les collisions.