

Les fonctions

Que feriez-vous si vous devez exécuter plusieurs fois le même code ou un code similaire mais où les variables utilisées sont différentes ? Le copier-coller n'est pas une super bonne idée, c'est ennuyeux, ça provoque de la redondance, ça alourdit le code et le rend moins lisible et ça risque d'introduire des erreurs... D'où l'intérêt d'utiliser des fonctions !

Ici, oubliez le cours de math, une fonction c'est un bout de code qu'on a mis « à part » afin de pouvoir le réutiliser facilement !

Déclaration


Vous en avez vu dans les exemples précédents, une fonction se déclare comme ceci (les crochets indiquent des éléments facultatifs)

```
function [nomDeLaFonction]([argument1, argument2,...]) {  
    CODE  
}
```

La déclaration d'une fonction contient les éléments suivants :

- Le mot-clé *function*
- Un **nom** (facultatif, nous le verrons plus loin)
- Une liste d'**arguments**. Ils sont mis entre crochets ci-dessus car ils sont facultatifs. Ils permettent de donner des paramètres, des données en entrée à la fonction. Ils ne sont utilisables que dans le corps de la fonction.
- Un **corps** ; du code quoi ; qui peut éventuellement renvoyer une **valeur de retour**. La fonction peut se contenter de n'exécuter que du code, mais elle peut également retourner, renvoyer un résultat ! Voilà qui est pratique 😊

Par exemple, une fonction qui diviserait un nombre par 6 et renverrait le résultat :

```
 function divideBySix(n) {  
    return n / 6;  
}
```

On déclare généralement les fonctions en haut du bloc, où elle est utilisée car cela reflète le comportement par défaut du JavaScript, le « hoisting » : quand une fonction ou une variable est déclarée, sa déclaration est bougée en haut du *scope* où elle se trouve. Le *scope*, c'est comme une « bulle » où se trouve la fonction (ou la variable) déclarée.

Quand une fonction est nommée, ce nom est valide à travers le *scope* dans lequel cette fonction a été déclarée. Vous pouvez avoir plusieurs fonctions avec le même nom dans votre code, à condition que toutes ces fonctions soient restreintes au niveau de leur portée ; c'est-à-dire que chaque fonction soit limitée à sa propre « bulle ».

Portée des variables

On peut déclarer des variables dans le corps d'une fonction. De telles variables s'appellent des **variables locales**, car elles sont propres à la fonction, au contraire des variables globales, définies en dehors de toute fonction et disponible à travers tout le code – et donc, également, au sein des fonctions.

Une fois sorti du *scope* de la fonction, il n'est plus possible d'accéder à ses variables locales, si vous faites ça, votre code va bien planter ☹

Si vous déclarez et initialisez une variable locale du même nom qu'une variable globale, la valeur prise en compte dans la fonction sera celle de la variable locale. Une fois sorti de la fonction, la valeur globale reprendra le dessus. Veillez à utiliser des noms différents afin d'éviter toute confusion.

Si les variables sont propres à la fonction, mieux vaut les créer comme variables locales, c'est plus propre car il faut user des variables globales avec modération !



Créer des *span* sous forme de « tags » à partir d'un *array*, mais cette fois avec une fonction et utilisation d'une variable locale

<http://jsfiddle.net/cLLS2/>

Les variables sont dans le *scope* de la fonction dès leur déclaration jusqu'à la fin de la fonction, indépendamment de leur imbrication dans des blocs tels que des conditions ou des boucles.

JS

```
if(true){  
    var dummy = "Hello";  
}  
alert(dummy);
```

En JavaScript, le code ci-dessus est tout à fait valide ! Alors que dans des langages comme le C, le Java,... ce code ne marcherait pas et générerait une erreur. C'est parce que JavaScript est « *function scoped* » et non pas « *block scoped* ».

Appel

Une fonction s'appelle par son nom, et si des paramètres sont passés entre les parenthèses, ils seront assignés aux arguments spécifiés dans la déclaration de la fonction dans le même ordre.

Toujours par rapport à l'exemple précédent, la fonction est appelée de cette manière

JS

```
var allSpans = document.querySelectorAll(".span3");  
var result = hasClass(allSpans[0], "alert-pink")
```

Si un nombre différent d'arguments qu'il n'y a de paramètres est passé lors de l'appel, JavaScript gère la situation comme ceci:

- S'il y a plus d'arguments que de paramètres définis dans la déclaration, les arguments « excédentaires » ne sont tout simplement pas assignés
- S'il y a plus de paramètres que d'arguments, les paramètres qui n'ont pas d'arguments correspondant sont mis à *undefined*



N'oubliez pas les () lors de l'appel, c'est ce qui différencie l'appel d'une fonction de sa référence.

Les fonctions anonymes

Les fonctions anonymes sont des fonctions qui n'ont simplement pas de nom ! Mais à quoi ça sert alors ? On ne saura jamais les appeler ?

En fait les fonctions anonymes sont très pratiques pour effectuer du code lors d'une action, d'un événement, sans devoir déclarer une fonction qui ne servirait qu'à une seule chose bien précise.

Sachez déjà qu'il est possible d'assigner une fonction anonyme à une variable. Je vois que vous fronchez les sourcils... En fait il s'agit plutôt d'assigner une **référence** vers la fonction à une variable. Ça s'écrit comme ceci :

JS

```
//on assigne une référence vers une fonction anonyme
var changeBackground = function () {
    document.body.style.background =
    "url(\"http://subtlepatterns.com/patterns/escheresque_ste.png
    \") repeat scroll 0% 0% transparent";
};

//on appelle la fonction
changeBackground();
```



Testez ce code sur jsFiddle

<http://jsfiddle.net/fAUwy/>

Notez la présence d'un point-virgule à la fin de la déclaration de la fonction. Comme nous sommes dans le cas d'une assignation, il est nécessaire, alors qu'il ne l'est pas quand on déclare une fonction « normale ».

Vous verrez très souvent ce genre de syntaxe en JavaScript donc mieux vaut déjà s'y habituer !

Comme une fonction est une variable, elle peut donc – enfin, plutôt sa référence, être passée comme paramètres d'une autre fonction. C'est le système utilisé dans les gestionnaires d'événements et les *callbacks*.