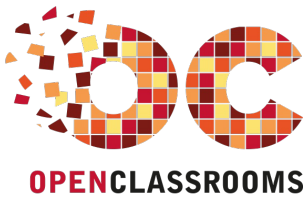
Les closures en Javascript

Par Darkodam



www.openclassrooms.com

Dernière mise à jour le 27/01/2010

Sommaire

Les closures en Javascript

Par  Darkodam

Mise à jour : 27/01/2010
Difficulté : Facile

Ce tutoriel nécessite une bonne connaissance de javascript, je vous conseille donc de lire le tutoriel Tout sur le Javascript ! écrit par JoSé2.

Bonjour à tous, et bienvenue dans mon premier mini-tutoriel.

Ici, vous allez apprendre le monde merveilleux des closures : Qu'est-ce que c'est ? A quoi ça sert ? Pourquoi les utilise-t-on ? Toutes ces questions trouveront leurs réponses dans les parties qui suivent. Détendez-vous, mettez-vous à l'aise, on va revoir ensemble les propriétés sur les fonctions et la portée des variables pour comprendre comment créer une closure. Une fois cette connaissance acquise, on verra quelques exemples de closures et les pièges récurrents dans lesquels vous tomberez sûrement (ou êtes déjà tombé) .

Suivez le guide .
Sommaire du tutoriel :

La closure ? C'est quoi ?

La closure (*fermeture* ou *clôture* en français) est une technique de programmation qui permet de séparer une variable de son contenu. Comme vous le savez sûrement, une variable possède deux composantes : un nom et une valeur (et accessoirement un type, mais ce n'est pas le sujet de ce tuto).

Nom de la variable	Valeur
positionSourisX	238
nombreDeColonnes	5
message	"Bonjour à toi"

Le principe d'une variable est que sa valeur peut varier d'un instant à l'autre, mais il arrive que l'on ait besoin de conserver cette valeur telle qu'elle est à un moment donné pour l'utiliser plus tard. Cette situation est assez fréquente lorsqu'on utilise le gestionnaire d'événements ou les fonctions de délais (`setTimeout` et `setInterval`). En utilisant une closure, on va conserver la valeur d'une variable de telle sorte qu'elle ne soit plus accessible en passant par la variable qui la contenait à l'origine.

Pour mieux comprendre, voici un petit schéma :

Admettons que l'on écrive une boucle allant de 1 à 4 et qu'à l'instant **t1** on capture la valeur du compteur.

*Dans ce tableau, **i** est le compteur*

Temps	Valeur de i	Valeur de la variable "cloturée"
t0	1	-
t1	2	2
t2	3	2
t3	4	2

Une fois la valeur de **i** "cloturée" à l'instant **t1**, elle devient indépendante. **i** continue à changer de valeur sans que cela influe sur la closure.

Pour réaliser ce tour de passe-passe  , il va falloir comprendre quelques petites choses à propos des fonctions et des variables.

Quelques rappels

Création d'une fonction

Une fonction n'est en fait qu'une variable un peu spéciale ; on peut lui réaffecter une valeur, la transmettre à une autre fonction, etc..

Code : JavaScript

```
// Une fonction peut être définie ainsi :
function UneFonction() {
 alert("Je suis une fonction");
}

UneFonction(); // affiche "Je suis une fonction"

// mais on peut aussi la définir ainsi :
// (On affecte en fait une fonction anonyme à une variable)
var UneAutreFonction = function() {
 alert("Je suis une autre fonction");
};

UneAutreFonction(); // affiche "Je suis une autre fonction"

// Une fonction étant une variable, on peut lui réaffecter une
valeur :
UneFonction = "Je ne suis plus une fonction :'(";

alert(UneFonction); // affiche "Je ne suis plus une fonction :'("

// On peut passer une fonction comme argument à une autre fonction
(comme n'importe quelle autre variable)
function UtiliserFonction(fct) {
 fct(); // ici, on exécute la fonction passée en paramètre
}

UtiliserFonction(UneAutreFonction); // affiche "Je suis une autre
fonction"
```

Portée des variables

Les variables définies en dehors d'une fonction sont accessibles dans la fonction. Elles peuvent être modifiées en dehors de la fonction aussi bien que par la fonction elle-même. On dit qu'elles ont une portée **globale**.

Code : JavaScript

```
var a = 10; // variable globale, cette variable sera accessible
dans toutes les fonctions
            // créées dans le script.

function MontrerGlobale() {
 // Comme aucune variable locale nommée a n'a été déclarée ici,
c'est la
 // variable globale qui sera affichée.
 alert(a);
}

function ModifierGlobale() {
 // même raisonnement que pour "MontrerGlobale" : On modifie la
variable globale
 a = a + 2;
}

MontrerGlobale();  // affiche "10"
ModifierGlobale(); // "a" vaut 12
MontrerGlobale();  // affiche "12"
a = 5;
MontrerGlobale(); // affiche "5"
```

Les variables définies dans une fonction, ainsi que les arguments d'une fonction ne sont disponibles qu'à l'intérieur de la fonction. On dit qu'elles ont une portée **locale**

Code : JavaScript

```
function CreerVariable() {
 var b = 10; // on crée une variable locale, uniquement accessible
à l'intérieur de
                    // cette fonction
}

CreerVariable();

alert(b); // Déclenche une erreur. La variable b n'ayant pas une
portée globale, elle n'est
          // pas accessible ici.
```

La variable **b** n'est pas accessible dans d'autres fonctions, uniquement dans celle où elle a été définie. Le code suivant aurait le même résultat que celui du dessus.

Code : JavaScript

```
function CreerVariable() {
 var b = 10; // on crée une variable locale, uniquement accessible
à l'intérieur de
                    // cette fonction
}

function MontrerVariable() {
 alert(b);
}

MontrerVariable(); // Déclenche une erreur. La variable b n'ayant
pas une portée globale,
                   // elle n'est pas accessible dans cette fonction.
```

Une variable définie dans une fonction (arguments compris) peut avoir le même nom qu'une variable définie globalement. Dans ce cas là, c'est la variable locale (c'est-à-dire celle définie dans la fonction) qui à priorité sur la globale **dans la fonction uniquement** :

Code : JavaScript

```
var c = "Je suis une variable globale";

function MaFonction() {
 var c = "Je suis une variable locale";
 // On vient de créer une variable locale nommée "c".
 // Tant que l'on restera dans la fonction, ce sera
 // cette variable qui sera prise en compte.
 // La variable globale "c" n'est donc plus disponible
 // dans cette fonction.
 alert(c);
}

// Une fois sortie de la fonction "MaFonction", la variable
// globale "c" reprend ses droits.

MaFonction(); // affiche "Je suis une variable locale"

alert(c); // affiche "Je suis une variable globale"
```

Passage de variable à une fonction

Lorsqu'une variable est passée en argument à une fonction, la valeur de la variable est copiée dans l'argument.

Code : JavaScript

```
function MaFonction( arg1 ) {
 alert( arg1 ); // affiche le contenu de la variable passée en
argument
 // on change la valeur de l'argument
 arg1 = "je suis la copie de la variable passée en argument";
 alert( arg1 ); // affiche "je suis la copie de la variable passée
en argument"
}

var maVariable = "Je suis une variable";

MaFonction( maVariable ); // affiche "Je suis une variable" puis
"je suis la copie de la variable passée en argument"

alert( maVariable ); // maVariable n'est pas modifiée, affiche "Je
suis une variable"
```

Le comportement est différent pour les objets et les tableaux (qui sont aussi des objets) car les variables qui permettent d'y accéder contiennent en fait une référence vers l'objet. C'est donc la référence qui est copiée dans l'argument et non l'objet, or Javascript fait automatiquement le lien entre une référence et l'objet sur lequel elle pointe. On accède donc au même objet.

Code : JavaScript

```
var MonObjet = {
 a : "je suis un membre de l'objet"
};

function ModifierObjet(arg1) {
 arg1.a = "J'ai été modifié =o";
}

alert(MonObjet.a); // affiche "je suis un membre de l'objet"

ModifierObjet(MonObjet);

alert(MonObjet.a); // affiche "J'ai été modifié =o"
```

Exécuter une fonction anonyme

Il arrive que l'on ait besoin d'exécuter un bout de code qui soit indépendant du reste du script et dont l'environnement est totalement contrôlé, en utilisant notamment des variables temporaires. La fonction se prête parfaitement à ce genre de cas, mais pourquoi créer une fonction dont le nom polluera l'espace global et que l'on utilisera qu'une fois dans tout le script ?

Pour ce genre de cas, il existe une syntaxe un peu spéciale qui permet d'exécuter une fonction anonyme : il faut englober la fonction anonyme dans des parenthèses. l'ensemble devient donc une fonction que l'on peut exécuter en ajoutant une paire de parenthèses à la fin. (**Note**: N'oubliez pas le point virgule final, son absence provoquant parfois des erreurs)

Code : JavaScript

```
( function() { /* Contenu de la fonction */ } ); // ici une
fonction anonyme est créée,
       // mais sans l'exécuter
( function() {
 var a = "Je suis une variable dans une fonction anonyme";
 alert(a);
} ) (); // en ajoutant une paire de parenthèses à l'ensemble, on
exécute la fonction

// On profite de tous les avantages des fonctions, on peut donc
passer des arguments
var b = 10;

( function(arg1) {
 arg1 += 2;
 alert("arg1 vaut : " + arg1); // affiche "arg1 vaut : 12"
} ) (b);

alert("b vaut : " + b); // affiche "b vaut : 10"
```

À l'assaut des closures !

Maintenant que l'on sait tout ça, on va pouvoir attaquer les closures proprement dites  .

Considérons le code suivant :

Code : JavaScript

```
function MaFonction(nombre) {
 function Ajouter(valeur) {
  // La variable "nombre" est accessible dans cette fonction, car
"nombre"
  // a été définie en dehors de la fonction Ajouter
  return nombre + valeur;
 }

 // Comme on l'a vu, Ajouter est une variable, j'ai donc le droit
de la rendre en
```

```
 // tant que résultat de la fonction
 return Ajouter;
}
```

Que se passe-t-il dans cette fonction ?
On définit un argument **nombre** qui prendra la valeur de la variable passée à la fonction.
On définit une fonction **Ajouter**. Dans cette fonction, la variable **nombre** est accessible car elle a été définie **en dehors** de la fonction **Ajouter**.
On retourne la fonction **Ajouter**.

Maintenant utilisons cette fonction :

Code : JavaScript - Exemple 8

```
var a = MaFonction(10);
// La variable "a" contient désormais la fonction "Ajouter". "a"
est désormais une
// fonction dans laquelle la variable "nombre" existe encore.

alert( a(2) ); // Affiche "12";
```

La closure nous a permis d'enfermer le nombre 10 dans une fonction. Et c'est là qu'est toute la magie des closures !

Pour mieux comprendre le principe, faisons une application de ce que l'on a appris :
On veut afficher avec une boucle les 3 premiers éléments d'un tableau de 5 éléments après un laps de temps d'une seconde.
Intuitivement voici ce que l'on pourrait écrire :

Code : JavaScript - Exemple 9

```
var a = ["elem1", "elem2", "elem3", "elem4", "elem5"];

for(var i = 0; i < 3; i++) {
 window.setTimeout(
  /* Argument 1 : Fonction à lancer après le délai */
  function() { alert(a[i]) },
  /* Argument 2 : Délai en millisecondes avant de lancer la
fonction */
  1000
 );
}
```

Que se passe-t-il au bout d'une seconde ?
Une boîte de dialogue apparaît 3 fois et affiche 3 fois "elem4". Ce qui n'est pas du tout ce qui était prévu  !

L'explication est simple : au bout d'une seconde, la fonction passée à `setTimeout` s'exécute. Dans cette fonction, la variable **a** est le tableau précédemment défini, et la variable **i** vaut... 3. Et oui, la variable **i** étant définie en dehors de la fonction anonyme, elle a été modifiée par la boucle **for** (à la fin de la boucle, **i** vaut 3). Et la valeur présente à l'indice 3 du tableau **a** est "elem4".

C'est là qu'interviennent les closures. Suivez bien parce qu'on va compliquer un peu les choses.

Code : JavaScript - Exemple 10

```
var a = ["elem1", "elem2", "elem3", "elem4", "elem5"];

for(var i = 0; i < 3; i++) {
 window.setTimeout(
  /* Argument 1 : Fonction à lancer après le délai */
  ( function(arg1) {
   // "arg1" prendra la valeur de "i" lors d'un tour de boucle.
   // Cette valeur devient indépendante de "i" et n'est plus
   // soumise au fonctionnement de la boucle

   // On retourne la fonction à exécuter à la fin du délai défini
```

```
        dans
           // "setTimeout"
           return function() {
            // Lorsque cette fonction s'exécutera, "arg1" contiendra
            // toujours la valeur qui lui a été transmise et n'aura pas
            // été modifié
            alert( a[arg1] );
           };
          } ) ( i ), // on passe en argument le compteur de la boucle
          /* Argument 2 : Délai en milisecondes avant de lancer la fonction
        */
          1000
         );
        }
```

Grâce à la closure, on a capturé la valeur de **i** lors d'un tour de boucle et on l'a enfermée dans une fonction que l'on a passée à `setTimeout` .
Et voilà, vous savez désormais le principe de fonctionnement d'une closure ainsi que comment la mettre en place. Les closures permettent de se sortir de situations autrement insolubles. Elles peuvent sembler difficiles à aborder au début, mais une fois qu'on a compris leur fonctionnement leur utilisation devient naturelle et salvatrice.

Amusez-vous bien !