Closures (Fermetures)

Une fermeture, ou *closure* en anglais, est une fonction qui fait utiliser des variables indépendantes (utilisées localement mais définies dans la portée englobante). Autrement dit, ces fonctions se « souviennent » de l'environnement dans lequel elles ont été créées (on dit aussi que la fonction capture son « environnement »).

Plus simplement elle enferme avec elle des variables qui lui sont externe provenant d’un scope parent.

Source: https://developer.mozilla.org

Portée lexicale

Avec l'exemple suivant :

```
1  function init() {
2      var nom = "Mozilla"; // nom est une variable locale de init
3      function afficheNom() { // afficheNom est une fonction interne de init
4          console.log(nom); // ici nom est une variable libre (définie dans la fonction parente)
5      }
6      afficheNom();
7  };
8  init();
```

La fonction `init` a une variable locale `nom` et une fonction interne `afficheNom`. La fonction interne est seulement visible de l'intérieur de `init`. Contrairement à `init`, `afficheNom` ne possède pas de variable locale propre, mais elle utilise la variable `nom` de la fonction parente (ceci dit `afficheNom` pourrait utiliser ses variables locales propres si elle en avait).

Closure :

```
function creerFonction() {
  var nom = "Mozilla";
  function afficheNom() {
    console.log(nom);
  }
  return afficheNom;
}

var maFonction = creerFonction();
maFonction();
```

Si on a l'exemple suivant :

et qu'on exécute ce code, on obtient exactement le même résultat qu'en exécutant l'appel de fonction `init()` étudié précédemment : le texte "Mozilla" est affiché dans la console. L'intérêt de ce code est qu'une fermeture contenant la fonction `afficheNom` est renvoyée par la fonction parente, avant d'être exécutée.

Le code continue à fonctionner, ce qui peut paraître contre-intuitif au regard de la syntaxe utilisée. Normalement, les variables locales d'une fonction n'existent que pendant l'exécution d'une fonction. Une fois que `creerFonction()` a fini son exécution, on peut penser que la variable `nom` n'est plus accessible. Cependant, le code fonctionne : la variable est donc accessible d'une certaine façon.

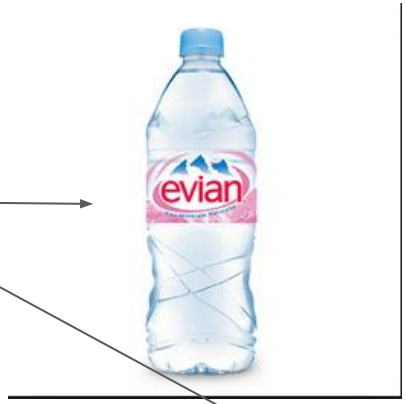
La solution est la suivante : `maFonction` est une fermeture. La fermeture combine la fonction `afficheNom` et son environnement. L'environnement est composé de toutes les variables locales de la portée présente lorsque la fermeture a été créée. Ici `maFonction` est une fermeture qui contient la fonction `afficheNom` et une référence à la variable `var nom = "Mozilla"` qui existait lorsque la fermeture a été créée.

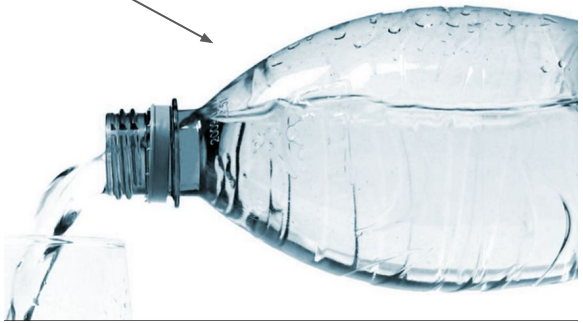
Voici un exemple supplémentaire : une fonction `faireAddition` :

C’est comme si notre fonction parente était une bouteille d’eau et que l’on rajoute des éléments dedans (autre fonction, variable..) et que l’on referme notre bouteille d’eau avec un bouchon “closure”, ce qui équivaut à mettre notre fonction dans une variable.

Cet variable devient donc notre nouveau contenant avec un environnement préservé que l’on peut réutiliser à un moment voulus dans son programme.

Bien évidemment on peut donc avoir différent contexte avec une base identique.





Les closures en pratique :

On a vu la théorie décrivant les fermetures. Est-ce qu'elles sont utiles pour autant ? Une fermeture permet d'associer des données (l'environnement) avec une fonction qui agit sur ces données. On peut faire un parallèle avec la programmation orientée objet car les objets permettent d'associer des données (les propriétés) avec des méthodes.

Ainsi, on peut utiliser une fermeture pour tout endroit où on utiliserait un objet et ce avec une seule méthode.

Beaucoup de code JavaScript utilisé sur le Web gère des événements : on définit un comportement, puis on l'attache à un événement déclenché par l'utilisateur (tel un clic ou une frappe clavier). Notre code est généralement une fonction de rappel (ou *callback*) exécutée en réponse à l'événement.

Voici un exemple concret : si on souhaite ajouter des boutons à une page afin d'ajuster la taille du texte, on pourrait définir la taille de police de l'élément `body` en pixels, et celles des autres éléments relativement à cette première taille grâce à l'unité `em` :

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```

Les boutons vont ensuite changer la taille de la police de l'élément `body`, ce changement étant répercuté aux autres éléments grâce aux unités relatives.

On a vu la théorie décrivant les fermetures. Est-ce qu'elles sont utiles pour autant ? Une fermeture permet d'associer des données (l'environnement) avec une fonction qui agit sur ces données. On peut faire un parallèle avec la programmation orientée objet car les objets permettent d'associer des données (les propriétés) avec des méthodes.

Ainsi, on peut utiliser une fermeture pour tout endroit où on utiliserait un objet et ce avec une seule méthode.

Beaucoup de code JavaScript utilisé sur le Web gère des événements : on définit un comportement, puis on l'attache à un événement déclenché par l'utilisateur (tel un clic ou une frappe clavier). Notre code est généralement une fonction de rappel (ou *callback*) exécutée en réponse à l'événement.

Voici un exemple concret : si on souhaite ajouter des boutons à une page afin d'ajuster la taille du texte, on pourrait définir la taille de police de l'élément `body` en pixels, et celles des autres éléments relativement à cette première taille grâce à l'unité `em` :

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```

Les boutons vont ensuite changer la taille de la police de l'élément `body`, ce changement étant répercuté aux autres éléments grâce aux unités relatives.

Émuler des méthodes privées avec des fermetures

Certains langages de programmation, comme Java, permettent d'avoir des méthodes privées, c'est-à-dire qu'on ne peut les utiliser qu'au sein de la même classe.
JavaScript ne permet pas de faire cela de façon native. En revanche, on peut émuler ce comportement grâce aux fermetures. Les méthodes privées ne sont pas seulement utiles en termes de restriction d'accès au code, elles permettent également de gérer un espace de nom (*namespace*) global qui isole les méthodes secondaires de l'interface publique du code ainsi rendu plus propre.
Voici comment définir une fonction publique accédant à des fonctions et des variables privées en utilisant des fermetures. Cette façon de procéder est également connue comme le patron de conception module :

```
var compteur = (function() {
  var compteurPrive = 0;
  function changeValeur(val) {
    compteurPrive += val;
  }
  return {
    increment: function() {
      changeValeur(1);
    },
    decrement: function() {
      changeValeur(-1);
    },
    valeur: function() {
      return compteurPrive;
    }
  };
})();

console.log(compteur.valeur()); /* Affiche 0 */
compteur.increment();
compteur.increment();
console.log(compteur.valeur()); /* Affiche 2 */
compteur.decrement();
console.log(compteur.valeur()); /* Affiche 1 */
```

Il y a beaucoup de différences par rapport aux exemples précédents. Au lieu de retourner une simple fonction, on retourne un objet anonyme qui contient 3 fonctions. Et ces 3 fonctions partagent le même environnement. L'objet retourné est affecté à la variable compteur, et les 3 fonctions sont alors accessibles sous les noms compteur.increment, compteur.decrement, et compteur.valeur.

L'environnement partagé vient du corps de la fonction anonyme qui est exécutée dès sa définition complète. L'environnement en question contient deux éléments privés : une variable compteurPrive et une fonction changeValeur. Aucun de ces deux éléments ne peut être utilisé en dehors de la fonction anonyme ; seules les trois fonctions renvoyées par la fonction anonyme sont publiques.

Ces trois fonctions publiques sont des fermetures qui partagent le même environnement. Grâce à la portée lexicale, chacune a accès à compteurPrive et à changeValeur.

On remarquera qu'on définit une fonction anonyme qui crée un compteur puis qu'on l'appelle immédiatement pour assigner le résultat à la variable compteur. On pourrait stocker cette fonction dans une variable puis l'appeler plusieurs fois afin de créer plusieurs compteurs.

```
var compteur = (function() {
  var compteurPrive = 0;
  function changeValeur(val) {
    compteurPrive += val;
  }
  return {
    increment: function() {
      changeValeur(1);
    },
    decrement: function() {
      changeValeur(-1);
    },
    valeur: function() {
      return compteurPrive;
    }
  };
})();

console.log(compteur.valeur()); /* Affiche 0 */
compteur.increment();
compteur.increment();
console.log(compteur.valeur()); /* Affiche 2 */
compteur.decrement();
console.log(compteur.valeur()); /* Affiche 1 */
```

Définition de 2 compteurs :

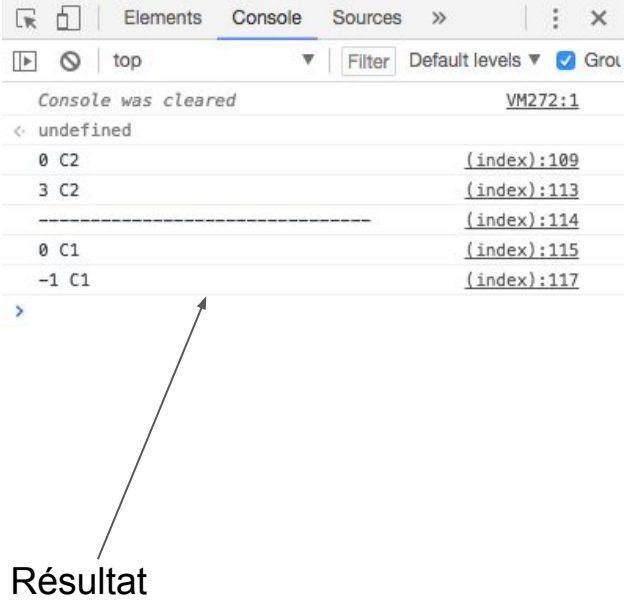
```
var compteurS = (function(){

      var compteurPrive = 0;

      function changeValeur(val){
      compteurPrive += val;
      }
      return {
         increment: function(){
          changeValeur(1);
         },
         decrement: function(){
          changeValeur(-1);
         },
      valeur: function(){
         return compteurPrive;
      }
      };
});

var compteur2 = compteurS();
var compteur1 = compteurS();

console.log(compteur2.valeur()+ " C2");
compteur2.increment();
compteur2.increment();
compteur2.increment();
console.log(compteur2.valeur()+ " C2");
console.log("--------------------------------")
console.log(compteur1.valeur()+ " C1");
compteur1.decrement();
console.log(compteur1.valeur()+ " C1");
```



Résultat

Créer un objet qui nous retourne dès fonction ou d'autre objet grâce au closure et aux IIFEs "Immediatly Invocte Function Expression".

index.html | module1.js × | module2.js | module3.js

```
const myClosure = (function(){

        var myPassword = "12345";

        function setPassword(newPassword){
            myPassword = newPassword;
        };

        function getPassword(){
            return myPassword;
        };

        return {
            getPassword : getPassword
        }


})();
```

La fonction IIFEs mis dans une closure retourne un objet avec un alias de la fonction a lequel on veut accéder.

index.html | module1.js | module2.js | module3.js ×

```
console.log(myClosure.getPassword());
```

On peut donc appeler la fonction getPassword dans un autre module en le préfixant du module dans lequel elle est.

Résultat