

Secteur Tertiaire Informatique
Filière « Etude et développement »

Séquence « Développer des pages Web »

Programmation orientée objet en JavaScript

Apprentissage

Mise en pratique

Evaluation

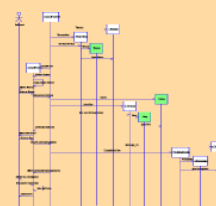
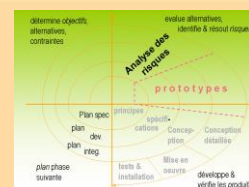
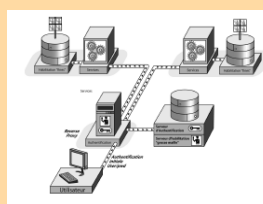


TABLE DES MATIERES

Table des matières	3
1. Les fondements du langage JavaScript.....	7
1.1 Rappels historiques.....	7
1.2 HTML et JavaScript.....	7
1.2.1 Construction classique.....	8
1.2.2 Construction moderne utilisant JQuery	10
1.3 Exercice	12
1.4 Type et portée des variables	12
1.5 Tableaux simples, tableaux associatifs et notation JSON.....	12
2. Les fonctions javascript	14
2.1 Fonction nommée	14
2.2 Fonction anonyme.....	14
2.3 Fonction autoexécutable	15
2.4 Passage de paramètres à une fonction	16
2.5 Notion de callback.....	16
2.6 Les closures JavaScript	18
2.7 Exercice	18
3. Les objets en JavaScript	18
3.1 JavaScript et le DOM	18
3.2 La gestion des événements en JavaScript	19
3.3 JavaScript, un « langage à prototypes »	20
3.3.1 Classes, instances et objets	20
3.3.2 Constructeur et propriétés	21
3.3.3 Méthodes et accesseurs	21
3.4 Les particularités de l'orienté objet 'prototype'	23
3.5 Exercice	24
3.6 comparaison des structures de données en JavaScript	25

Objectifs

Ce document a pour but de préciser certains **concepts et notations** couramment utilisés en programmation JavaScript côté client dans les applications Web, afin d'être en capacité d'aborder aussi bien le développement de ce qu'il est convenu d'appeler des « applis Web », basées sur le langage JavaScript, que le développement d'applications mobiles hybrides à base de pages HTML et de JavaScript comme ce qui est proposé avec le framework Cordova.

Du fait de l'histoire de JavaScript et de ses profondes évolutions successives, il existe de **nombreuses variantes de syntaxe assez déconcertantes** et pas toujours clairement explicitées dans les documentations de référence. Le premier objectif de ce document est de **comparer et clarifier ces différentes syntaxes**.

Les applications basées sur JavaScript utilisent généralement des frameworks complémentaires écrits eux-mêmes en **langage JavaScript orienté objet**. Mais **la dimension objet de JavaScript est elle aussi parfois déconcertante** pour des développeurs familiers de langages comme Java ou C#. Le deuxième objectif de ce document est de **clarifier cette dimension objet de JavaScript et d'explicitier les différentes syntaxes** en les rapprochant des concepts objet.

Enfin, il est important de faire le point sur les **traitements asynchrones et leurs nécessaires fonctions callback** largement utilisées, par exemple, par les applis Web utilisant Ajax et celles qui mettent en œuvre les bases de données embarquées dans les navigateurs Web.

La dernière partie apporte les informations nécessaires pour **utiliser en JavaScript une base de données WebSQL** intégrée aux navigateurs Firefox et Chrome (moteur WebKit).

Pré requis

Maîtriser la programmation de pages Web en langages HTML/CSS, maîtriser les bases de la programmation, être familiarisé avec l'usage de JavaScript en pages Web, et être initié à la logique de la programmation orientée objet.

Outils de développement

Aucun en particulier si ce n'est un éditeur de textes et un navigateur moderne compatible Chrome/Firefox pour reproduire les exemples et réaliser des exercices.

Méthodologie

Ce document rappelle les notions et variantes de notations importantes et renvoie à l'étude de ressources en ligne.

Le premier chapitre se propose d'illustrer l'évolution du langage à travers une même mini-application écrite de deux manières différentes. C'est l'occasion de préciser certaines notions et syntaxes qui seront explicitées dans les chapitres suivants.

Des exercices d'application seront proposés au fur et à mesure pour ancrer les apprentissages.

Programmation objet en JavaScript

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Le dernier chapitre se termine sur un exercice de synthèse.

Mode d'emploi

Symboles utilisés :



Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.



Propose des exercices ou des mises en situation pratiques.



Point important qui mérite d'être souligné !

Ressources

Ressources complémentaires :

LeDOMJavaScript.pptx

P-programmation-objet-javascript.pdf

Ressources en ligne sur Internet :

Pour apprendre ou approfondir le langage JavaScript :

<http://www.coursweb.ch/javascript>

<https://openclassrooms.com/courses/tout-sur-le-javascript>

<https://openclassrooms.com/courses/dynamisez-vos-sites-web-avec-javascript>

<http://openclassrooms.com/courses/le-javascript-moderne>

<http://openclassrooms.com/courses/bonnes-pratiques-javascript>

<http://openclassrooms.com/courses/les-closures-en-javascript>

<http://braincracking.org/2011/11/17/usage-avance-des-fonctions-javascript/>

<http://braincracking.org/2011/11/16/javascript-3-fondamentaux/>

Pour aborder la dimension objet de JavaScript :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Le_mod%C3%A8le_objet_JavaScript_en_d%C3%A9tails

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_objets

<http://blog.xebia.fr/2013/06/10/javascript-retour-aux-bases-constructeur-prototype-et-heritage/>

Pour comprendre Web SQL :

<http://www.debray-jerome.fr/articles/L-api-websql-en-html5.html>

Pour découvrir JQuery :

<http://openclassrooms.com/courses/un-site-web-dynamique-avec-jquery>

<http://learn.jquery.com/>

Programmation objet en JavaScript

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

<http://api.jquery.com/>

Lectures conseillées

1. LES FONDEMENTS DU LANGAGE JAVASCRIPT

1.1 RAPPELS HISTORIQUES

A l'origine, JavaScript a été inventé par un éditeur de logiciels (NetScape) et ajouté à son navigateur Web de manière à pouvoir **effectuer certains traitements côté client**, sur le poste de travail de l'internaute sans qu'une nouvelle requête soit adressée au serveur Web.

Les principales applications visées étaient essentiellement des **pré-traitements de formulaires** de saisie afin d'effectuer des **contrôles de vraisemblance**, ainsi que des **animations graphiques** de base.

JavaScript a tout donc d'abord apporté **l'interactivité** aux pages Web ; c'est pourquoi ce langage est particulièrement bien adapté aux **traitements événementiels des pages Web** et que sa forme la plus basique consiste en une imbrication un peu folle de code HTML et de code JavaScript.

Rapidement, des notions de **code HTML dynamique** ont été ajoutées aux navigateurs Web ; plus besoin d'allers-retours lents et coûteux entre poste utilisateur et serveur HTTP pour simplement personnaliser le contenu d'une page : masquage/affichage partiel, ajout de ligne à un tableau... selon les interactions de l'utilisateur sont devenues monnaie courante grâce à la **programmation événementielle en JavaScript**. Avec JavaScript, le contenu des pages HTML a pu devenir dynamique.

L'apparition de la technique AJAX a encore permis de développer cette tendance ; son aboutissement réside dans ce qu'on appelle aujourd'hui les **'applis Web'** ou **'applications Web mono-page'** dans lesquelles le dialogue et les affichages sont entièrement pilotés par JavaScript au sein d'une même 'page' (d'une même URL).

Une dernière variante voit le jour depuis peu avec **l'adaptation de ces techniques au développement d'applications à installer sur un terminal mobiles**, smartphone ou tablette (framework Cordova par exemple).

Enfin, il faut souligner que les dernières évolutions du langage JavaScript ont été **influencé par la domination de l'orienté objet en programmation** mais JavaScript, même s'il manipule des objets depuis toujours, a une **logique et une syntaxe assez différentes des langages orientés objet classiques** comme Java ou C#. Le but de ce document est donc d'aider à s'y retrouver...

1.2 HTML ET JAVASCRIPT

A l'origine, le code JavaScript était totalement entremêlé avec le code HTML ce qui ne facilitait pas la mise au point ni la maintenance, surtout à partir du moment où des personnes différentes ont été en charge du code HTML, du code JavaScript et du code CSS.

Aujourd'hui, on sépare totalement les différents scripts qui constituent une page Web (code HTML, scripts JavaScript et scripts CSS) mais il faut bien faire le lien entre eux et la syntaxe du langage a évolué en conséquence.

Pour illustrer cette évolution du langage, voici des extraits d'une mini-application réalisée de 2 manières différentes. Le but est de contrôler la saisie de données sur ce petit formulaire :

Petit formulaire

Pseudonyme

Mot de passe

Confirmation

E-mail

1.2.1 Construction classique

Le code HTML de présentation pourrait être celui-ci :

```
<body>
<form action=... ..>
  <h3>Petit formulaire</h3>
  <div>
    <div id="erreur">
      <p>Vous n'avez pas rempli correctement les champs du formulaire !</p>
    </div>

    <label for="pseudo">Pseudonyme</label>
    <input type="text" id="pseudo" class="champ" /><br /><br />
    <label for="mdp">Mot de passe</label>
    <input type="password" id="mdp" class="champ" /><br /><br />
    <label for="confirmation">Confirmation</label>
    <input type="password" id="confirmation" class="champ" /><br /><br />
    <label for="mail">E-mail</label>
    <input type="text" id="mail" class="champ" /><br /><br />
    <input type="submit" id="envoi" value="Envoyer" />
    <input type="reset" id="rafraichir" value="Rafraîchir" />
  </div>
</form>
<hr>
</body>
```

Cette div HTML est masquée par défaut
(style CSS display :none ;)

Reste à entremêler le code JavaScript. On a depuis toujours pris l'habitude d'isoler le code JavaScript dans des **fonctions** mais leur appel est fait à l'intérieur des balises HTML. Ainsi la partie <head> HTML devient :

```
<link type="text/css" rel="stylesheet" href="style.css">
```

```
<script language = "javascript">
    function verifier() {
        // indicateur d'erreur
        var erreur = false;
        // message d'erreur
        var msg = "";

        // contrôle pseudo obligatoire
        ... le reste du code ...
    }
</script>
```

Balise HTML pour déclarer les variables globales et les fonctions,

ou bien externalisation du code JavaScript :

```
<script language = "javascript" src="...js"></script>
```

Le lien avec la fonction pourrait être fait sur le bouton submit en entremêlant les 2 syntaxes :

```
<input type="submit" id="envoi" value="Envoyer" onclick="return verifier();" />
```

Attribut HTML définissant l'événement concerné

Syntaxe JavaScript dans une valeur d'attribut HTML

Dans la fonction JavaScript, le code des contrôles en série pourrait être basé sur ce modèle :

```
// contrôle pseudo obligatoire
if(window.document.getElementById("pseudo").value == "")
{
    erreur = true;
    msg += "\nPseudonyme obligatoire";
    window.document.getElementById("pseudo").style.borderColor = 'red';
};
```

Manipulation par JavaScript des éléments HTML

Manipulation par JavaScript des attributs CSS

Et la fin du script permet d'afficher un message global et de filtrer l'action du bouton submit :


```
// message erreur éventuel
if (erreur){
    window.document.getElementById("erreur").style.display = 'block';
    window.alert(msg);
};

// retour : filtrage ou non du submit
return !erreur;
```

Si la fonction retourne false, submit est neutralisé et l'utilisateur reste sur ce form, sinon l'action du form est lancée

1.2.2 Construction moderne utilisant JQuery

Actuellement, on construirait ce formulaire en séparant totalement le code JavaScript du code HTML et on utiliserait le framework JQuery pour simplifier l'écriture du code JavaScript.

 Au lieu de déclarer les ressources JavaScript internes ou externes en début de code HTML (section <head>), **on fait le lien avec le code externe en fin de page Web** de manière à ce que le navigateur ait déjà interprété le code HTML avant d'en faire usage par JavaScript :

```
<body>
... ..
<hr>


<!-- on inclut la bibliothèque -->
<script src="jquery-1.11.3.min.js"></script>
<script src="formTest.js"></script>
</body>
</html>
```

Balises HTML <script> en fin de <body> de la page HTML

Chaque élément HTML **identifié** peut être adressé très facilement par la fonction JQuery **\$()** qui accepte de nombreuses variante de sélecteurs d'objets HTML. Par exemple en début de script, on peut récupérer toutes les valeurs des champs de saisie du formulaire par :

```
// NB : les noms sont préfixés du car $ pour différencier ces var
// des var locales purement JavaScript
var $pseudo = $('#pseudo'),
    $mdp = $('#mdp'),
    $confirmation = $('#confirmation'),
    $mail = $('#mail'),
    $envoi = $('#envoi'),
    $reset = $('#rafraichir'),
    $erreur = $('#erreur'),
    $champs = $('.champ'); // retourne tous les éléments de classe champ
var $erreurSaisie = false; // indicateur d'erreur
```

Les fastidieuses méthodes JavaScript `.getElementById()`, `.getElementsByClassName()`... se résument à l'emploi de la fonction JQuery puissante **\$()** qui reste délicate dans sa syntaxe ;
ici, **#xxx** sélectionne sur une valeur d'attribut **id** HTML,
.yyy sélectionne sur une valeur d'attribut **class** HTML

 Le script JavaScript implémente lui-même les « handler d'événement » grâce à la méthode JavaScript **addEventListener()** ou ses équivalents JQuery pour chaque type d'événement, comme, ci-dessous, la méthode JQuery **.click()**. De plus on fait un usage massif des **fonctions anonymes** (voir chapitre 2). Par exemple, pour le bouton submit :

```
// fonction anonyme associée au bouton envoi
// lance la vérif de tous les champs
// annule éventuellement l'effet standard du bouton submit
$envoi.click(function(e) {
    $erreurSaisie = false;
```

La variable JavaScript qui stocke l'objet JQuery correspondant à l'élément HTML est dotée de nombreuses méthodes JQuery, notamment `.click()` qui implémente le handler d'événement correspondant

Programmat

Afpa © 2016 – Section Tertiaire Info

//on lance la fonction de vérification sur tous les champs :

```
verifier($pseudo);  
verifier($mdp);  
verifier($confirmation);  
verifier($mail);  
if($erreurSaisie){  
    e.preventDefault(); // annule l'action du bouton submit  
}
```



Une fonction événementielle reçoit en paramètre la référence à un objet Event doté de la méthode `.preventDefault()` qui permet de neutraliser le comportement HTML par défaut

```
});
```

Pour s'assurer que le navigateur a bien interprété toute la structure du document avant de lancer le script JavaScript, JQuery propose d'utiliser la méthode `.ready()` de l'objet JQuery correspondant à `window.document` pour implémenter la fonction de démarrage. Ainsi, le début du code de notre script pourrait être :

```
var $erreurSaisie = false; // indicateur d'erreur
```

```
$(document).ready(function() {
```

```
    // var globales
```

```
    // NB : leur nom est préfixé du car $ pour les différencier des var locales
```

```
    var $pseudo = $('#pseudo'),
```

```
        $mdp = $('#mdp'),
```

```
    ... .. la suite du code
```

```
    $champs = $('.champ'); // retourne tous les éléments de classe champ
```

```
    ... .. implémentation des fonctions événementielles
```

```
});
```

Avec cette construction, il n'y a plus d'imbrication des 2 langages et le code JavaScript est quasi-indépendant du code HTML car il lui suffit de connaître les valeurs des `id` des différents éléments HTML à manipuler.



NB : notez bien que la fonction de sélection JQuery `$()` retourne toujours un objet JQuery qui contient une ou plusieurs valeurs stockées dans un tableau, même si le sélecteur est très précis ; ainsi `$('#identifiant')` retourne un objet JQuery différent de `document.getElementById('identifiant')` ; en conséquence, on sélectionne par la fonction JQuery `$()` pour ensuite appliquer des méthodes JQuery sur l'objet ou en récupérer des propriétés JQuery ; il est toujours 'risqué' de mixer l'usage du JavaScript classique et de JQuery dans la manipulation des éléments du document HTML.



Pour aller plus loin dans l'apprentissage de JQuery, reportez-vous aux ressources citées en début de document.

1.3 EXERCICE



Mettez en pratique la construction moderne de code JavaScript/JQuery en réalisant l'exercice guidé 'Formulaire de saisie' du document 'P-programmation-objet-javascript.pdf'.

1.4 TYPE ET PORTEE DES VARIABLES

Les variables JavaScript n'ont pas besoin d'être déclarées. Leur type est auto-adaptable en fonction de leur contenu courant. **On prend l'habitude :**

- de les 'déclarer' grâce au mot-clé **var** (qui signifie 'type variant'),
- de leur assigner un nom correspondant à leur contenu,
- et de ne pas faire varier leur type dans un même script.



La **portée** (ou visibilité) des variables doit être soignée :

- une variable déclarée ou initialisée en dehors d'une fonction est dite globale et elle reste accessible par tout le code JavaScript ;
- par contre, une variable locale déclarée ou initialisée à l'intérieur d'une fonction n'est accessible que depuis cette fonction ; de même, les paramètres reçus par une fonction correspondent à des variables locales ;
- bien sûr, tous les éléments HTML du document restent de portée globale et sont accessibles depuis toutes les fonctions.

Dans notre exemple précédent, une variable globale (`$erreurSaisie`) permet de mémoriser si au moins une erreur a été détectée par une fonction.

Une variable définie dans une fonction (ainsi que les paramètres de la fonction) 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 a priorité sur la globale, pour la fonction uniquement.

1.5 TABLEAUX SIMPLES, TABLEAUX ASSOCIATIFS ET NOTATION JSON

Le langage JavaScript offre comme tous les langages la possibilité de manipuler des ensembles de variables sous forme de **tableaux**. L'intérêt principal des tableaux classiques réside dans l'indexation numérique des différentes valeurs du tableau ce qui permet une programmation efficace et généralisable par l'utilisation de boucles par exemple. En JavaScript, les tableaux n'ont **pas de dimension prédéfinie**, restent auto-adaptables à leur contenu, et peuvent contenir des données de types différents.

Un tableau peut se créer aussi bien :

- Par instanciation d'un objet de la classe prédéfinie `Array` :
`var jours = new Array('Lun', 'Mar', 'Mer', 'Jeu', 'Ven', 'Sam', 'Dim');`
- Par déclaration de ses éléments énumérés entre crochets :
`var jours = ['Lun', 'Mar', 'Mer', 'Jeu', 'Ven', 'Sam', 'Dim'];`

Attention aux variantes de notation !

L'accès à une donnée du tableau se fait par un indice classique (numéroté à partir de zéro) :

```
jours[0] = 'Lundi'; // la première valeur vaut maintenant 'Lundi'
alert(jours[6]); // affiche Dim
```

On rappelle qu'un tableau simple est doté d'une **propriété length** qui renseigne sur le nombre courant d'item du tableau. Enfin, la méthode `push()` permet d'ajouter un ou plusieurs items à un tableau ; la méthode `unshift()` fonctionne comme `push()`, excepté que les items sont ajoutés au début du tableau ; les méthodes `shift()` et `pop()` retirent respectivement le premier et le dernier élément du tableau.

Au-delà des tableaux classiques, JavaScript offre la notion de **tableaux associatifs** dans lesquels les différentes valeurs peuvent être **indexées par un libellé** en plus de leur indice, ce qui occasionne encore une variante de syntaxe.

```
var tableau = new Array(); // déclaration d'un tableau vide
tableau['nom'] = 'Moran';
tableau['prenom'] = 'Bob';
alert(tableau['nom']); // affiche Moran
```

Un tableau associatif peut aussi se définir en extension selon la **syntaxe JSON** pour les '**objets littéraux**' :

```
var tableau = {"nom": "Moran", "prenom": "Bob", "age": 26};
alert(tableau['nom']); // affiche Moran
```

NB : Attention qu'avec les notions **d'objet littéral** et la **notation JSON**, le délimiteur est la paire d'accolades `{}`.

Et comme un tableau associatif est très proche de la notion d'objet (voir chapitre 3), la **notation objet** est tout aussi possible pour les tableaux associatifs que pour les objets JavaScript :

```
alert(tableau.nom) ; // affiche Moran
```

Il n'est pas possible de parcourir un objet littéral ou un tableau associatif avec une boucle `for(var i=xx ; i<yy ; i++){}` classique, puisqu'une boucle de ce type est basée sur l'incrémentement d'un indice numérique.

Par contre, **sa variante, la boucle `for(var xx in yy){}`** est très simple et ne sert qu'à une seule chose : **parcourir un objet**. Attention à la syntaxe de cette boucle en JavaScript !

Nous pouvons mixer objets littéraux et tableaux.

```
const stagiaires = [ {nom : "Boisserie", prenom : "Julien"},
  {nom : "Cazal", prenom : "Fabrice"},
  {nom : "Cheney", prenom : "Jéréemie"},
  {nom : "Dobelin", prenom : "nicolas"}];
```



Voir aussi : <http://www.coursweb.ch/javascript/datastruct.html>

NB : Les objets littéraux sont souvent utilisés car ils peuvent se révéler très utiles ; par exemple, les fonctions, avec l'instruction **return**, ne savent retourner qu'une seule variable ; si on veut retourner plusieurs variables, il faut les placer dans un tableau/objet et retourner ce dernier. Et il est souvent plus commode d'utiliser un objet littéral.

2. LES FONCTIONS JAVASCRIPT

2.1 FONCTION NOMMÉE

A l'origine, toutes les fonctions JavaScript devaient être déclarées et définies dans la partie `<head>` de la page HTML selon la syntaxe :

```
function nom_fonction(liste_de_paramètres_reçus){instructions_à_exécuter} ;
```


Derrière le mot-clé `function` se cache aussi bien la définition d'une **procédure** (qui exécute ses traitements et ne retourne rien au final) que d'une **fonction** (qui exécute ses traitements et retourne une valeur finale grâce à l'instruction `return`).

Exemple de *fonction* :

```
function calculeSurface(largeur, hauteur) {  
    return largeur * hauteur; // calcul la surface et la retourne  
} ;
```

Exemple de *procédure* :

```
function coucou() {  
    alert("coucou"); // affiche un message  
} ;
```

 **L'exécution** d'une fonction est simplement provoquée par **l'appel de son nom suivi d'une paire de parenthèses** contenant les éventuels paramètres à passer à la fonction comme dans :

```
var s = calculeSurface(8, 4); // s prend 32 comme valeur  
coucou() ; // affiche coucou en boite de dialogue
```

Contrairement aux langages fortement typés comme Java ou C#, la *signature* d'une fonction JavaScript est assez 'souple' et n'impose pas le respect strict des paramètres attendus (ce qui permet de reproduire la notion de '*surcharge de méthode*' courante dans les langages orientés objet comme Java ou C#).



Voir <http://www.coursweb.ch/javascript/fonctions.html>

2.2 FONCTION ANONYME

Une **fonction anonyme** ne porte pas de nom ; elle est définie 'à la volée', ce qui surcharge considérablement le code au détriment de sa lisibilité. L'usage des fonctions anonymes est systématique dans les traitements asynchrones comme Ajax ou l'accès aux bases de données embarquées.

Exemple avec Ajax :

```
function controleIdentAjax()  
{  
    xhr = new XMLHttpRequest();  
    xhr.open("GET", "xxx.php?..... ", true);
```

Une fonction nommée classique...

```
xhr.send(null);

//on définit l'appel de la fonction au retour serveur (fonction callback).
xhr.onreadystatechange = function()
{
    if (xhr.readyState==4 && xhr.status == 200)
        .....suite du code à exécuter
};
}
```

...qui déclare une fonction anonyme associée à un événement asynchrone

2.3 FONCTION AUTOEXECUTABLE

Une nouvelle notation permet de **déclarer et exécuter immédiatement une fonction anonyme**. La syntaxe impose simplement de **faire suivre la définition de la fonction d'une paire de parenthèses afin de provoquer son exécution**.

Exemple :

```
var test = function() {
    console.log('hello world');
}();
```

Pour provoquer l'exécution immédiate d'une fonction, on peut encore **l'englober dans une autre paire de parenthèses** sans oublier de **la faire suivre par sa paire de parenthèses** (Sainte Axe, priez pour nous !)

Exemple :

```
(function() {
    console.log('hello world');
})();
```

Ou encore :

```
(function() {
    console.log('hello world');
})();
```

Cette notation a de plus l'avantage de **créer un espace de travail 'privé', isolé de l'environnement du script**, dans lequel on peut déclarer et utiliser des variables et fonctions invisibles de l'extérieur, ce qui peut être très utile au démarrage d'une application JavaScript.

Ainsi, la structure d'un projet Cordova créé par Visual Studio contient un script par défaut de type :

```
function () {
    "use strict";

    document.addEventListener( 'deviceready', onDeviceReady.bind( this ), false );

    function onDeviceReady() {
        // Gérer les événements de suspension et de reprise Cordova
        document.addEventListener( 'pause', onPause.bind( this ), false );
        document.addEventListener( 'resume', onResume.bind( this ), false );

        // TODO: Cordova a été chargé. Effectuez l'initialisation qui nécessite Cordova ici.
    }
}
```



```

};

function onPause() {
    // TODO: cette application a été suspendue. Enregistrez l'état de l'application ici.
};

function onResume() {
    // TODO: cette application a été réactivée. Restaurez l'état de l'application ici.
};
}();

```


Tout le code spécifique peut maintenant être intégré dans l'espace privé défini par ces parenthèses et il sera isolé des 'effets de bord' potentiellement générés par les nombreux autres scripts composant l'application mobile.

2.4 PASSAGE DE PARAMETRES A UNE FONCTION


Lors de l'appel à l'exécution d'une fonction, **il suffit de placer des noms de variables ou des valeurs entre les parenthèses pour lui passer des paramètres**, comme dans une fonction `alert()` classique (exemple : `var msg = 'coucou' ; alert(msg) ;`) ou comme dans l'exemple donné plus haut (`calculeSurface(8, 4) ;`)

Le comportement est différent pour la passation d'objets ou tableaux (qui sont aussi des objets) car les variables qui permettent d'y accéder contiennent en fait une référence vers l'objet ; **dans ce cas, c'est toujours la référence vers l'objet qui est passée à la fonction**, et non une valeur.

2.5 NOTION DE CALLBACK

 **Une callback est une fonction de retour, nommée ou anonyme, placée en paramètre d'une autre fonction pour être exécutée à un moment donné**, à la suite de la fonction principale ou à un moment indéterminé en cas de fonctionnement asynchrone.

Par exemple, la fonction standard JavaScript `setTimeout()` contient en 1° paramètre le nom de la fonction à appeler à l'issue du délai donné en 2° paramètre : `setTimeout(hello, 1000) ;`

 **Attention qu'il s'agit bien de désigner la fonction à exécuter, sans demander son exécution immédiate ; c'est pourquoi, le nom de la fonction ne doit pas être suivi des habituelles parenthèses !**

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

Pour passer des paramètres à une fonction callback, il va falloir utiliser une technique qui consiste à englober l'appel de la fonction de rappel dans une fonction anonyme. Cette fonction anonyme correspond bien à une référence à une fonction et non à une demande d'exécution (Sainte Axe, priez pour nos neurones !).

Par exemple pour afficher un compteur de secondes perpétuel, l'exemple précédent deviendrait :

```

function hello(msg) {
    alert(msg) ;
};

```



```

var i = 0;
setInterval(function() {
    i++;
    hello("je compte " + i);
}, 1000);

```

Ici, à chaque seconde la fonction anonyme est déclenchée et elle appelle la fonction nommée `hello()` en lui passant une nouvelle valeur du texte à afficher en boîte de dialogue.

Bien souvent, les développeurs se passent de fonction nommée et **imbriquent la définition et l'appel de la fonction paramétrée à l'intérieur de la première fonction anonyme**. Cette construction aboutirait donc à (Sainte Axe, ayez pitié de nous !) :

```


var i = 0;
setInterval(function() {
    i++;
    (function(texte) {
        alert(texte); }("je compte " + i));
}, 1000);

```

Attention d'indenter au mieux le code de façon à s'y retrouver dans les imbrications et leurs nécessaires délimiteurs (,),{,} et autre ;


La fonction anonyme interne est appelée avec passation d'un paramètre issu de la fonction englobante

La forme peut sembler déroutante mais elle s'explique par les points techniques abordés précédemment...

 Attention aux pièges de syntaxe que constituent ces imbrications de fonctions au niveau des accolades, parenthèses et autres virgules ! Un faux-pas, et plus rien ne fonctionne...

Tous les traitements asynchrone comme Ajax ou l'accès aux bases de données embarquées reposent sur l'usage de fonctions callback.

2.6 LES CLOSURES JAVASCRIPT

 **En JavaScript il est fréquent de définir une fonction à l'intérieur d'une fonction, en particulier pour les traitements asynchrones ; comme cette fonction est englobée dans la fonction qui la contient, elle n'est pas accessible en dehors de sa fonction englobante.**

Une closure est créée lorsqu'une fonction est définie dans le corps d'une autre fonction et qu'elle fait référence à des paramètres ou des variables locales à la fonction dans laquelle elle est définie.

Une closure est un espace mémoire créé automatiquement quand une fonction B est définie dans une fonction A, et que B accède à des variables définies dans A.


La syntaxe des *closures* devient assez redoutable mais ce mécanisme est bien souvent nécessaire dans les traitements asynchrones. En effet, en JavaScript, comme on utilise massivement le mécanisme des fonctions callbacks, il faut trouver un moyen de passer l'état du programme aux fonctions callbacks, et généralement on le fait via des closures.

Le dernier exemple ci-dessus inclut une closure ; la fonction anonyme interne utilise une variable (*texte*) dont la valeur reste 'figée' de temps de son exécution.

Voir les ressources OpenClassRooms :

<http://openclassrooms.com/courses/les-closures-en-javascript>


2.7 EXERCICE

 Mettez en pratique des fonctions nommées et anonymes, avec ou sans passation de paramètres en réalisant l'exercice 'Tableaux et objets' du document 'P-programmation-objet-javascript.pdf'.

3. LES OBJETS EN JAVASCRIPT

3.1 JAVASCRIPT ET LE DOM

Le **Document Object Model** (ou **DOM**) est une interface de programmation (ou **API**) pour les documents XML et HTML ; c'est donc un ensemble d'outils qui permettent de faire communiquer entre eux, dans le cas présent, les langages HTML et JavaScript.

 **L'interpréteur JavaScript représente les éléments HTML de la page Web selon une structure arborescente d'objets** ; à la racine, l'objet `window` représente l'instance du navigateur ; il référence l'objet `location` qui symbolise la barre d'adresse et l'objet `document` qui représente la page Web elle-même référençant tous ses éléments.


Historiquement on pouvait accéder par JavaScript aux éléments de la page à travers un système de noms à tiroirs représentant l'arborescence de la page HTML. Par exemple : `window.document.forms[0].elements[0].value` retourne la valeur du premier éléments HTML permettant la saisie dans le premier formulaire de la page.

Maintenant, la méthode `.getElementById()` permet d'adresser directement tout élément HTML doté d'un **attribut id**. Les méthodes `.getElementsByClassName()` et `.getElementsByTagName()` permettent d'adresser un tableau d'éléments selon leur **attribut class** ou leur **type d'élément**. Enfin, les deux méthodes `.querySelector()` et `.querySelectorAll()` permettent de grandement simplifier la sélection d'éléments dans l'arbre DOM ; ces deux méthodes prennent pour paramètre un seul argument : une chaîne de caractères qui reprend la syntaxe des **sélecteurs CSS**.

Programmation objet en JavaScript

Afpa © 2016 – Section Tertiaire Informatique – Filière « Etude et développement »

Le framework **jQuery** simplifie encore les choses grâce à sa fonction **\$()** dont le paramètre est aussi un sélecteur CSS (car cette fonction JQuery n'est qu'un habillage -une *encapsulation*- des méthodes JavaScript standards).

 Pour bien comprendre la correspondance entre les éléments HTML et les objets JavaScript, déroulez l'animation **LeDOMJavaScript.pptx**.

Chaque objet correspondant à un élément HTML de la page est automatiquement doté de propriétés correspondant principalement aux attributs HTML, et de méthodes permettant leur manipulation par programmation.


Par exemple : `window.location.href` retourne l'URL courante de la page et `window.location.replace(xxx)` permet de la modifier à la place de l'utilisateur.

3.2 LA GESTION DES EVENEMENTS EN JAVASCRIPT

Historiquement, un événement était déclaré à l'intérieur du code HTML (voir chapitre 1).

Le standard DOM-0 a apporté des propriétés JavaScript supplémentaires correspondant aux différents événements, comme dans l'exemple :

```
document.getElementById('envoi').onclick = function() {...};
```

 Le **standard DOM-2 actuel** permet le rattachement de plusieurs fonctions événementielles pour un même événement et ajoute **l'objet Event** qui fourmille d'informations complémentaires sur l'événement déclenché et qui permet de mieux contrôler le déroulement, par exemple grâce à ses méthodes **.preventDefault()** et **stopPropagation()**.

L'exemple précédent devient en version DOM-2 :

```
document.getElementById('envoi').addEventListener('click', function() {...}, 'false');
```

Le dernier argument est un booléen pour spécifier si l'on souhaite utiliser la **phase de capture** ou bien celle de **bouillonnement**.

Ces deux phases sont deux étapes distinctes de l'exécution d'un événement. La première, la *capture* (*capture* en anglais), s'exécute *avant le déclenchement de l'événement*, tandis que la deuxième, le *bouillonnement* (*bubbling* en anglais), s'exécute *après que l'événement a été déclenché*. Toutes deux permettent de définir le sens de propagation des événements, car, en raison de l'imbrication des éléments, un événement *click* survenant sur un objet `select` par exemple peut aussi bien être détecté par cet objet `select` que par l'objet `form` qui le contient ou encore par l'objet `document` lui-même s'ils sont aussi associés à l'événement *click*.

 Voir : <http://www.coursweb.ch/javascript/handle-events.html>

Et <https://openclassrooms.com/courses/dynamisez-vos-sites-web-avec-javascript/les-evenements-24>

3.3 JAVASCRIPT, UN « LANGAGE A PROTOTYPES »

En JavaScript on crée des objets d'une manière un peu différente des langages orientés objets plus classiques comme Java ou C#. On dit que **JavaScript est un langage à prototypes**.

Il s'agit donc maintenant d'explicitier les différentes syntaxes en relation avec les concepts objet.

3.3.1 Classes, instances et objets

Une **classe** en JavaScript se définit par une **fonction nommée** selon l'écriture :

```
function nom_de_la_classe() {membres_de_la_classe};
```

Un **objet**, instance de cette classe sera créé grâce à l'opérateur **new** :

```
var nom_de_l_objet = new nom_de_la_classe();
```

Comme une classe se définit en JavaScript comme une fonction, on peut aussi définir une sorte de 'classe anonyme' grâce à une variable pointant vers une fonction anonyme :

```
var nom_de_pointeur_de_fonction = function() {membres_de_la_classe} ;
```

Il sera alors aussi nécessaire de procéder à l'instanciation de l'objet grâce à l'opérateur **new** :

```
var nom_de_l_objet = new nom_de_pointeur_de_fonction();
```

La notation **JSON** permet de **définir un objet littéral par l'énumération de ses membres** :

```
var nom_de_l_objet = {  
  prop1 :valeur1,  
  prop2 :valeur2,...  
  methode1 : function(...) {...},  
  methode2 , function(...) {...}, ...  
}
```

Attention à la syntaxe particulière qui fait bien souvent perdre beaucoup de temps en mise au point du code...

Ces différentes syntaxes sont couramment utilisées mais pas pour le même usage :

- **On définit une classe quand le script aura besoin de manipuler plusieurs instances** (objets Métier comme des clients, des produits...) ; la logique objet classique conduit à définir des classes *nommées* mais on rencontre souvent des 'classes anonymes' dans le monde JavaScript ;
- **On définit souvent un objet en notation JSON quand une seule instance est nécessaire** ; dans ce cas, l'objet n'ayant plus de méthode 'constructeur' (voir ci-dessous), il n'est plus paramétrable.

NB : on utilise souvent un mix des deux notations, par exemple une fonction de classe dont une propriété est définie sous forme d'un objet littéral.

3.3.2 Constructeur et propriétés

La partie déclarative principale de la classe, `function()`, qu'elle soit nommée ou anonyme, correspond à la méthode **constructeur de la classe** et peut à ce titre recevoir des paramètres pour l'initialisation de ses propriétés.

Les **propriétés** (exposées) d'un objet sont définies à l'intérieur de la fonction par le préfixe `this.` alors que les **attributs** (variables privées) se définissent classiquement par le mot-clé `var`.

Exemple :

```
function Employe (unnom, unebranche, unsalaire) {  
    this.nom = unnom ; // propriété exposée  
    this.branche = unebranche ; // propriété exposée  
    var salaire = unsalaire ; // attribut privé  
}
```

Ici, la classe `Employe` contient 2 propriétés et un attribut privé qui sont initialisés par le constructeur.

3.3.3 Méthodes et accesseurs

Les méthodes des classes ou des objets **sont des fonctions** décrites dans la fonction principale générale selon la syntaxe :

```
this.nom_de_la_fonction = function(paramètres){corps_de_la_fonction} ;
```

ou, en syntaxe JSON :

```
this.nom_de_la_fonction : function(paramètres){corps_de_la_fonction},
```

Exemple : fonction exposée permettant de calculer le salaire annuel :

```
function Employe (unnom, unebranche, unsalaire) {  
    ... ..  
    // le salaire est un attribut privé : pas de préfixe this.  
    this.salaireAnnuel = function(){return salaire * 12 ;}  
}
```

Exemple : fonction exposée (procédure, setter) de modification du salaire (qui ne peut qu'augmenter) :

```
function Employe (unnom, unebranche, unsalaire) {  
    ... ..  
    this.setSalaire= function(nouveauSalaire) {  
        if(salaire < nouveauSalaire)  
            {salaire = nouveauSalaire;};  
    }  
}
```

Exemple : accesseur en lecture à l'attribut salaire :

```
function Employe (unnom, unebranche, unsalaire) {  
    ... ..  
    this.getSalaire = function(){return salaire ;}  
}
```

Rien de bien particulier ici car tout cela reste conforme aux principes de l'orienté objet classique.

Le langage JavaScript supporte maintenant les structures classiques de levées d'erreur (*try/catch/throw/classe Error*) communes dans les langages modernes orientés objet ; ainsi la fonction *setter* précédente pourrait très bien lever une erreur si le nouveau salaire proposé est inférieur au salaire courant :

```
this.setSalaire= function(nouveauSalaire) {  
    if(salaire < nouveauSalaire)  
        {salaire = nouveauSalaire;;}  
    else  
        {throw new Error("Erreur : le salaire ne peut diminuer") ;}  
}
```

Dès lors, le script principal doit protéger l'appel de ce *setter* par une structure *try/catch* :

```
// instantiation d'1 objet  
var Pierre = new Employe('Pierre', 'Développement', 2000) ;  
try { // attention, danger...  
    Pierre.setSalaire(1500) ; // provoque levée d'erreur  
}  
catch (e) { // e est une instance d'Exception  
    alert(e.message) ;  
}
```

3.4 LES PARTICULARITES DE L'ORIENTE OBJET 'PROTOTYPE'

Dans la logique orientée objet classique, chaque objet est une instance d'une classe ; cette classe, qui peut être vue comme un *'moule à objet'* (sauf pour les classes *'static'*). Tous les objets *'sortis du même moule'* ont la même structure définie par la liste de leurs membres, privés ou exposés décrits dans la classe.

En JavaScript, chaque instance d'une classe dispose de la structure définie dans la classe mais il peut avoir ses propres variantes de structure.

Tout objet JavaScript instancié peut à tout moment voir sa structure modifiée par ajout ou suppression de membres, ces modifications ne s'appliquant qu'à une instance précise.

En JavaScript, tout objet dispose d'une propriété *prototype* qui correspond en quelque sorte à structure d'une classe dans la programmation objet classique ; on peut y définir des propriétés et méthodes qui s'appliqueront à toutes les instances, déjà existantes ou futures.

Ces mécanismes permettent en quelque sorte de reproduire la notion de *spécialisation* d'objets comme dans l'exemple ci-dessous.

Voici une illustration de ces principes avec la classe précédente des employés :

```
function Employe(unnom, unebranche, unsalaire) {
    this.nom = unnom ; // propriété exposée
    this.branche = unebranche ; // propriété exposée
    var salaire = unsalaire; // attribut privé

    // méthodes
    this.salaireAnnuel = function() {
        return salaire * 12 ;
    } ;
    this.modifieSalaire = function(nouveauSalaire) {
        if(salaire < nouveauSalaire)
            {salaire = nouveauSalaire;};
    };
    this.getSalaire = function(){return salaire ;}
} ;

// instanciation de 2 objets
var Pierre = new Employe('Pierre', 'Développement', 2000) ;
var Paul = new Employe('Paul', 'Production', 1800) ;

// manipulation d'un objet
```

```
Pierre.modifieSalaire(2500); // affecte l'attribut privé
alert(Pierre.salaireAnnuel()); // Ok
alert(Pierre.nom) ; // Ok
//alert(Pierre.salaire) ; // pas accessible de l'extérieur

// enrichissement du prototype
Employe.prototype.sonChef = null; // ajout propriété à la classe
Pierre.sonChef = Paul ; // affectation de la propriété pour cet objet
alert(Pierre.sonChef.nom) ; // Ok

// propriété particulière d'un objet
Pierre.chef = Paul ;
alert(Pierre.chef.nom) ; // Ok
//alert(Paul.chef.nom) ; // erreur, cet objet ne dispose pas de la propriété
```



La notion de `prototype` permet aussi de reproduire les concepts d'héritage/spécialisation ; pour aller plus loin voir :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Le_mod%C3%A8le_objet_JavaScript_en_d%C3%A9tails

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_objets

<http://blog.xebia.fr/2013/06/10/javascript-retour-aux-bases-constructeur-prototype-et-heritage/>

3.5 EXERCICE



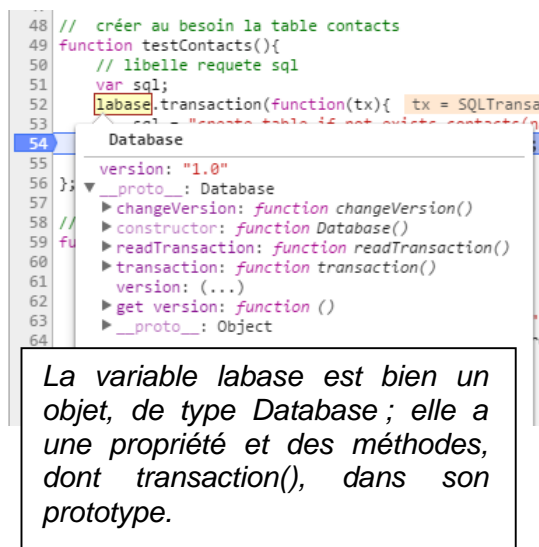
Mettez en pratique la mise en œuvre d'objets et classes JavaScript en réalisant l'exercice guidé 'Classe, Objet et Prototype' du document 'P-programmation-objet-javascript.pdf'

3.6 COMPARAISON DES STRUCTURES DE DONNEES EN JAVASCRIPT

Vous venez de réaliser des variantes de la même application ; ces variantes reposent essentiellement sur la structure interne des données manipulées par vos scripts.

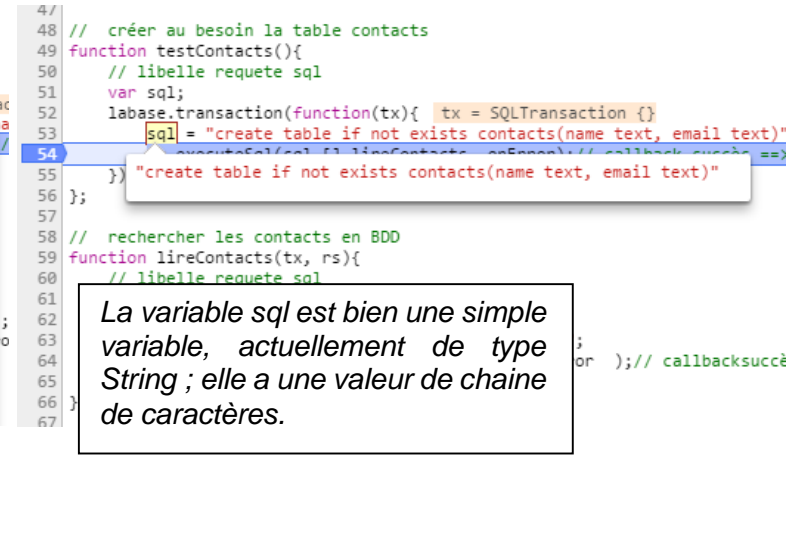
Pour y voir plus clair, maintenant que tout cela fonctionne, relancez chacune des variantes et placez des points d'arrêt de manière à explorer dans le débogueur les structures de données internes à JavaScript. Vous devriez pouvoir faire des observations proches de ce qui suit.

Version 'classique' :



```
48 // créer au besoin la table contacts
49 function testContacts(){
50 // libelle requete sql
51 var sql;
52 labase.transaction(function(tx){ tx = SQLTransaction {}
53 // sql = "create table if not exists contacts(name text, email text)"
54
55 Database
56 version: "1.0"
57 __proto__: Database
58 changeVersion: function changeVersion()
59 constructor: function Database()
60 readTransaction: function readTransaction()
61 transaction: function transaction()
62 version: (...)
63 get version: function ()
64 __proto__: Object
```

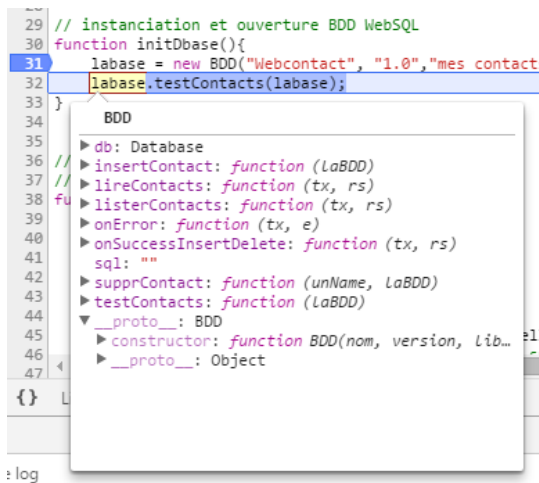
La variable labase est bien un objet, de type Database ; elle a une propriété et des méthodes, dont transaction(), dans son prototype.



```
47 // créer au besoin la table contacts
48 function testContacts(){
49 // libelle requete sql
50 var sql;
51 labase.transaction(function(tx){ tx = SQLTransaction {}
52 sql = "create table if not exists contacts(name text, email text)"
53 // sql = "create table if not exists contacts(name text, email text)"
54
55 }
56 };
57
58 // rechercher les contacts en BDD
59 function lireContacts(tx, rs){
60 // libelle requete sql
61
62
63
64
65
66 }
67
```

La variable sql est bien une simple variable, actuellement de type String ; elle a une valeur de chaîne de caractères.

Version objet instance d'une classe :

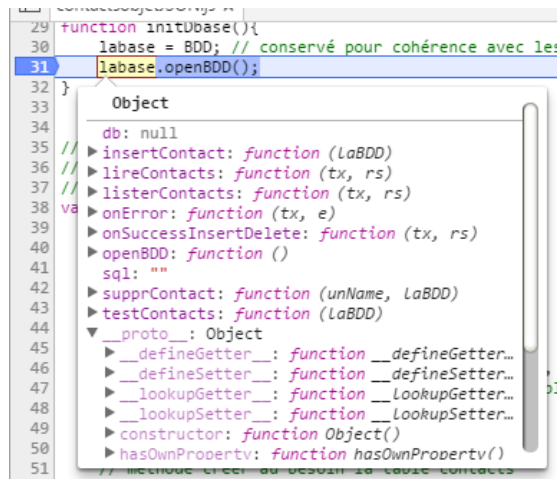


```
29 // instantiation et ouverture BDD WebSQL
30 function initDbase(){
31 labase = new BDD("Webcontact", "1.0", "mes contacts")
32 labase.testContacts(labase);
33
34 }
35
36 BDD
37 db: Database
38 insertContact: function (LaBDD)
39 lireContacts: function (tx, rs)
40 listerContacts: function (tx, rs)
41 onError: function (tx, e)
42 onSuccessInsertDelete: function (tx, rs)
43 sql: ""
44 supprContact: function (unName, LaBDD)
45 testContacts: function (LaBDD)
46 __proto__: BDD
47 constructor: function BDD(nom, version, lib...
48 __proto__: Object
```

La variable labase est bien un objet, de type BDD ; elle les propriétés et méthodes décrites dans sa classe, dont la chaîne sql, plus celles de son prototype de base (bien pauvre).

Même structure quand la 'classe' est définie par une 'function'.

Version objet JSON :



La variable labase est bien un objet, de type Object ; elle les propriétés et méthodes décrites dans sa structure, dont la chaine sql, plus celles de son prototype de base (bien chargé !).

CREDITS

ŒUVRE COLLECTIVE DE L'AFPA

Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)

B. Hézard – Formateur

S. Thomy - Formateur

Ch. Perrachon – Ingénieure de formation

Date de mise à jour : 27/4/16