

Secteur Tertiaire Informatique

Filière « Etude et développement »

Séquence « Développer des pages Web »

**Découverte du langage JavaScript**



**Apprentissage**

Mise en situation

Evaluation

Table des matières

[Table des matières 3](#_Toc449435201)

[1. Introduction à JavaScript 7](#_Toc449435202)

[1.1 JavaScript c’est quoi ? 7](#_Toc449435203)

[1.2 Historique 8](#_Toc449435204)

[1.3 JavaScript dans le développement moderne 9](#_Toc449435205)

[2. Les bases du langage JavaScript 9](#_Toc449435206)

[2.1 La balise <script> 9](#_Toc449435207)

[2.2 La syntaxe de JavaScript 10](#_Toc449435208)

[2.3 Où inclure le code en JavaScript ? 11](#_Toc449435209)

[2.4 **Exercice: Fichier externe .JS 11](#_Toc449435210)

[2.5 Les variables 12](#_Toc449435211)

[2.6 Les opérateurs 14](#_Toc449435212)

[2.7 Les conditions 15](#_Toc449435213)

[2.8 Les répétitions 16](#_Toc449435214)

[3. Les outils de développement 17](#_Toc449435215)

[3.1 Le débogage 17](#_Toc449435216)

[3.2 Les outils de Google Chrome 19](#_Toc449435217)

[3.3 Les outils de Microsoft Internet Explorer et Edge 20](#_Toc449435218)

[3.4 Les outils de Mozilla FireFox 21](#_Toc449435219)

[3.5 **Exercice : testez l’exemple fourni « mes-variables.html » 22](#_Toc449435220)

[4. Les fonctions 23](#_Toc449435221)

[4.1 Les instructions de fonctions 24](#_Toc449435222)

[4.2 La portée des variables 24](#_Toc449435223)

[4.3 **Exercice : La portee 25](#_Toc449435224)

[4.4 Les paramètres des fonctions 26](#_Toc449435225)

[4.5 **Exercice : L’objet arguments 26](#_Toc449435226)

[4.6 Les expressions de fonctions 27](#_Toc449435227)

[4.7 Les fonctions anonymes 28](#_Toc449435228)

[4.8 Les fonctions « Callback » 29](#_Toc449435229)

[4.9 Les fonctions auto-exécutables 30](#_Toc449435230)

[5. L’interactivite avec l’HTML 31](#_Toc449435231)

[5.1 Le DOM 31](#_Toc449435232)

[5.2 JavaScript et les propriétés des éléments 33](#_Toc449435233)

[5.3 JavaScript et les éléments de formulaires HTML 34](#_Toc449435234)

[5.4 **Exercice : Manipulons le dom 35](#_Toc449435235)

[6. L’interactivité avec les événements 36](#_Toc449435236)

[6.1 Généralités 36](#_Toc449435237)

[6.2 Les types d’événements 36](#_Toc449435238)

[6.3 Mettre en place des événements 37](#_Toc449435239)

[6.4 L’objet event 38](#_Toc449435240)

[6.5 Supprimer un événement 38](#_Toc449435241)

[6.6 **Exercice : Ordre des événements 39](#_Toc449435242)

[6.7 ** Exercice : Manipulons les événements 39](#_Toc449435243)

[6.8 ** Exercice : Le post-it 39](#_Toc449435244)

[6.9 ** Exercice : Le chrono 39](#_Toc449435245)

[7. Les formulaires 40](#_Toc449435246)

[7.1 Généralités 40](#_Toc449435247)

[7.2 Déclaration d'un formulaire 40](#_Toc449435248)

[7.3 Accès au formulaire et à ses éléments 40](#_Toc449435249)

[7.4 Contrôles de saisie et soumission des formulaires 41](#_Toc449435250)

[7.5 ** Exercice : Le contrôle de champ 43](#_Toc449435251)

[8. Pour aller plus loin : zoom sur des objets natifs 44](#_Toc449435252)

[8.1 Objets window, history, location, screen 44](#_Toc449435253)

[8.2 ** Exercice : Le passage d’information 45](#_Toc449435254)

[8.3 L’objet Navigator 45](#_Toc449435255)

[8.4 **Exercice : L’objet navigator 45](#_Toc449435256)

[8.5 L’objet Date 46](#_Toc449435257)

[8.6 L’objet Array 47](#_Toc449435258)

[9. L’opérateur double négation 48](#_Toc449435259)

[10. Les expressions régulières 49](#_Toc449435260)

[10.1 L'objet RegExp 49](#_Toc449435261)

[10.2 Syntaxe des expressions régulières 50](#_Toc449435262)

[10.3 Récapitulatif des caractères spéciaux utilisés dans les expressions régulières : 53](#_Toc449435263)

[10.4 **Exercice : Contrôle d’adresse email 53](#_Toc449435264)

Objectifs

Ce support de formation présente les bases du langage JavaScript et propose une progression d’exercices traitant des fondamentaux du développement d’une application Web en utilisant la technologie JavaScript.

Pré requis

Les prés requis nécessaires afin de suivre avec profit cette formation sont les notions de base du développement Web : les concepts élémentaires du Web, la syntaxe HTML et les propriétés CSS, la gestion des formulaires HTML.

De bonnes bases en algorithmique sont également nécessaires.

Outils de développement

On peut aussi bien utiliser l’IDE Eclipse, l’IDE NetBeans, ou des éditeurs spécialisés développement Web comme Sublime Text, Brackets ou autre.

Nous visualiserons les résultats sur les différents navigateurs Chrome, FireFox, Internet Explorer…

Notez que toutes les ressources nécessaires sont gratuites au téléchargement.

Méthodologie

Ce support présente l’ensemble des connaissance et compétences de base à acquérir pour l’apprentissage du langage JavaScript pour le développement d’applications Web modernes.

En fin de chapitres, vous trouverez les liens vers des travaux pratiques présents dans un second support « **P-decouverte-javascript.pdf** ». Il est conseillé de les réaliser au fur et à mesure de manière à bien ancrer les apprentissages.

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

* Documentation en ligne :  
  <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference>
* CoursWeb.ch :  
  <http://www.coursweb.ch/javascript>
* Expressions régulières :  
  <http://www.commentcamarche.net/contents/javascript/jsregexp.php3>

# Introduction à JavaScript

## JavaScript c’est quoi ?

Le *langage de programmation JavaScript* ***côté client*** est une extension au langage de description Html qui définit toute page Web. Le langage JavaScript peut rendre dynamique les pages Web notamment en accédant directement aux éléments de la page Html, en les manipulant et en interagissant avec l’utilisateur. Les *scripts JavaScript*, *qui s'ajoutent ici aux balises Html*, peuvent être comparés aux macros d'un traitement de texte en ce sens qu’ils permettent d’automatiser un certain nombre de tâches et de réaliser des contrôles de saisie.

JavaScript côté client peut faire beaucoup de choses :

* Ajouter, supprimer et modifier le contenu Html dynamiquement ;
* Interagir sur le code CSS ;
* Animer et ajouter des effets sur les textes, les images…
* Intercepter les événements (souris, clavier, *timer*…) ;
* Contrôler les saisies effectuées par l’utilisateur dans les formulaires Html ;
* Générer des menus dynamiques ;
* Détecter le type de navigateur de manière à adapter le rendu graphique ou l’ergonomie aux fonctionnalités spécifiques.

*Les scripts JavaScript sont téléchargés avec la page* ; ils peuvent être inclus directement dans le code source Html ou stockés dans des fichiers distincts reliés à la page Web. Après téléchargement depuis le serveur Web, ils vont être gérés et exécutés par un *interpréteur JavaScript intégré dans le navigateur*. Les instructions JavaScript seront donc traitées en direct et surtout sans retard par le navigateur (pas besoin de recharger la page ni d’allers-retours entre navigateur et serveur Web pour récupérer des ressources).

JavaScript côté client, utilisé seul, possède certaines limites qui font parties de ses avantages :

* Il ne peut pas accéder à des fichiers stockés sur le disque dur du poste utilisateur ;
* *Il est limité à la manipulation du navigateur, du contenu de la page Web en cours ainsi que de la barre d’adresse du navigateur et des éventuelles informations passées en paramètres dans l’url de la page ;*
* Il ne peut pas accéder directement aux bases de données du serveur Web ;
* Il n’est pas multitâche ;

Mais l’arrivée du Html5 et ses API corrigent certains « manques » (balise Html <canvas>, base de données locale…).

La bibliothèque JavaScript ***Ajax*** permet d’accéder aux bases de données du serveur Web via des requêtes HTTP secondaires sans quitter la page Web en cours.

JavaScript ***côté serveur*** se développe beaucoup ces dernières années. L’outil ***Node.JS*** en est un très bon exemple et devient une alternative aux autres technologies côté serveur comme les Servlet en JAVA ou les technoligies JSP[[1]](#footnote-1), l’ASP[[2]](#footnote-2), PHP[[3]](#footnote-3), Python, Ruby …

Aujourd’hui, JavaScript est un langage utilisé également pour créer des applications mobiles (JQuery Mobile, PhoneGAP …), des applications Windows avec ***WinJS***, manipuler des fichiers PDF et en modifiant leurs apparences et ajouter des validations de formulaires…

## Historique

JavaScript a été initialement développé par Netscape et s'appelait alors LiveScript. Adopté à la fin de l'année 1995 par la firme Sun (qui a aussi développé Java), il prit alors son nom de JavaScript.

Microsoft a implémenté le langage JScript qui est semblable à JavaScript dans son navigateur Web Internet Explorer. Microsoft encourageait plutôt l’utilisation de VBScript côté client ; son navigateur possède les deux interpréteurs mais *le standard universellement reconnu reste le langage JavaScript, aujourd’hui relativement homogène d’un navigateur à l’autre*.

Nous en sommes à la version 1.8, la version 2.0 ayant été abandonnée. Ce qui n'est pas sans poser certains problèmes de compatibilité des pages comportant du code JavaScript, selon le navigateur utilisé.

Le noyau de JavaScript est défini par le standard ECMA[[4]](#footnote-4)-262, approuvé par l’ISO[[5]](#footnote-5)-16262. Depuis 2009, JavaScript est basée sur le standard ECMA-262 version 5.

Tout d’abord un peu oublié pendant la bataille entre le HTML et le XHTML, avec l’arrivée du HTML 5, l'avenir de JavaScript est bien relancé.

Aujourd’hui la majorité des navigateurs sont compatibles ECMA Edition 5.

Cependant, les navigateurs continuent à ajouter des fonctionnalités non standard.

*Html 5 est bien parti pour normaliser le tout…*

## JavaScript dans le développement moderne

Dans le développement Web moderne, on attend à ce qu’un site fonctionne correctement quel que soit l’équipement utilisé (PC, Tablette, Téléphone, système d’exploitation, navigateur…) Même sur des anciens navigateurs et même si JavaScript est désactivé.

*Il est fortement recommandé d’externaliser le code JavaScript dans des fichiers source distincts des pages Web ;* le code JavaScript devient alors *non intrusif*  car il est séparé du code Html.

On utilise l’*enrichissement progressif*, en séparant les couches :

1. La sémantique ou structure (Html)
2. La présentation (CSS)
3. Le comportement (JavaScript)

Ainsi, la page est toujours fonctionnelle même si JavaScript et CSS sont désactivés par l’utilisateur.

# Les bases du langage JavaScript

## La balise <script>

Dans la logique du langage Html, il faut signaler au navigateur par une balise, que ce qui suit est du code JavaScript (et non du code Html ou VBScript par exemple). C'est le rôle de la balise double ***<*script*>***.

Depuis la version HTML 4.01 et le XHTML[[6]](#footnote-6), cette balise prend un attribut, le type MIME[[7]](#footnote-7), pour indiquer le format du code qu’elle contient :

<script type="text/JavaScript">

Auparavant, les spécifications utilisaient l’attribut "language" :

<script language="JavaScript">

En utilisant du code HTML 5, on n’est plus obligé de spécifier le type MIME.

<script> …code JavaScript … </script>

## La syntaxe de JavaScript

*JavaScript est sensible à la casse*. Ainsi pour afficher une boîte de dialogue d’alerte, il faudra écrire **alert**() et non **Alert**().

Pour l'écriture des instructions JavaScript, on utilisera l'alphabet ASCII classique (à 128 caractères) comme en Html. Les caractères accentués comme *é* ou *à* ne peuvent être employés que dans les chaînes de caractères.

Pour déclarer une chaîne de caractères, les guillemets " et l'apostrophe ' peuvent être utilisés à condition de ne pas les mélanger. Si vous souhaitez utiliser des guillemets dans vos chaînes de caractères, tapez \" ou \' pour les différencier.

JavaScript ignore les espaces, les tabulations et les sauts de lignes.

Les commentaires en JavaScript suivent les conventions utilisées en C et C++ :

**//** Commentaire sur une seule ligne

**/\*** Commentaire

sur

plusieurs   
 lignes **\*/**

Les points-virgules terminent les instructions. Cependant, les interpréteurs JavaScript acceptent les instructions isolées sans terminaison en point-virgule.

*Pensez toujours à finir vos instructions par un point-virgule afin d’éviter les erreurs d’exécution !*

*Si vous utilisez des pages en HTML 4 ou XHTML et que vous voudriez valider votre page dans le validateur du W3C[[8]](#footnote-8), vous devrez encapsuler votre code JavaScript entre des balises de commentaires Html. Autrement, si votre code comprend des marqueurs <, >, &, le validateur les prendra pour des balises et ne validera pas votre page :*

<script type="text/JavaScript" language="JavaScript">

<!-- Masquer le script pour les anciens navigateurs

code JavaScript

// Cesser de masquer le script -->

</script>

## Où inclure le code en JavaScript ?

Il existe plusieurs moyens d’ajouter du code JavaScript à votre page :

* Entre les balises <script> … </script> ; elles-mêmes positionnées n’importe où entre les balises <head>…</head> et <body>…</body>.
* Directement dans les balises d’éléments Html via les attributs de gestion des événements :

<input type="button" **onClick="alert('Hello');"** /> <!—- Intrusif -->

* En appelant un fichier JavaScript externe :

<script **src="maBibli.js"**> // Pas de code JavaScript ici ! </script>

*Une bonne pratique aujourd’hui consiste à ce que le comportement de JavaScript soit « non intrusif ».*

*Pour la réutilisabilité du code, il est conseillé d’implémenter votre code JavaScript dans des fichiers externes et de les appeler ensuite* dans les pages Web grâce à la balise Html <script src=…>.

De plus, *il est préférable d’appeler votre code JavaScript à la toute fin de votre code Html, juste avant la balise fermante </body>,* pour 2 raisons :

* Le navigateur traite votre page Html de haut en bas (y compris vos ajouts en JavaScript). Si le code JavaScript est lourd à charger, votre page risque d’être longue à charger également.
* JavaScript est très utilisé pour modifier les éléments du code Html (le DOM). Votre code JavaScript ne pourra atteindre les éléments Html de la page qu’une fois ces derniers chargés. Par conséquent, il ne doit être interprété qu’à la fin du chargement de la page.

## **Exercice: Fichier externe .JS

Réalisez maintenant l’exercice ‘Fichier externe JS’ de manière à construire une page Web simple associée à un script JavaScript externe.

## Les variables

Les variables peuvent se déclarer n’importe où dans le code et de deux façons :

1. Soit de façon explicite avec le mot clé ***var*** *(pour variant)*.   
   Par exemple :

**var** numAdherent = 1; // Déclaration et initialisation d’une variable  
// Déclaration de plusieurs variableset initialisation des 2 premières

**var** nomAdherent = "Darme",   
 prenomAdherent = "Jean",  
 age;

1. soit de façon implicite. On écrit directement le nom de la variable suivi de la valeur qu'on lui attribue et JavaScript s'en accommode. Par exemple :

// Déclaration et initialisation de 2 variables   
numAdherent = 2;   
prenomAdherent = "Luc";  
age; // va provoquer une erreur de déclaration de variable

Attention ! Malgré cette apparente facilité, la façon dont on déclare la variable aura une grande importance pour la "visibilité" (la "portée") de la variable dans le programme JavaScript.

*Les variables sont typées dynamiquement***.** Selon la valeur qu’on lui affecte, la variable prendra le type correspondant.

JavaScript utilise 5 types de données : les nombres, les chaînes de caractères, les booléens, les objets et le mot clé *undefined* pour les variables non initialisée.

Exemple :

var maVariable; // son type est **undefined**maVariable = 324; // son type devient **number (base 10)**

maVariable = 0324; // son type reste **number (base 8)**maVariable = 0x324; // son type reste **number (base 16)**  
maVariable = "Bonjour"; // son type devient **string**

maVariable = true; // son type devient **boolean**

maVariable = new Array(); // son type devient **Object ou Array**

Les tableaux de variables Array peuvent posséder une ou plusieurs dimensions et leur taille peut s’auto adapter à leur contenu courant (voir les compléments en fin de document) :

Exemple :

// crée un tableau à 1 dimension pouvant contenir 10 valeurs

tabScore = new Array(10) ;

// affecte le 2° poste de la valeur 15

tabScore[1] = 15 ;

A noter qu’une donnée de type string est automatiquement dotée de ‘*méthodes’* simplifiant ses manipulations :

|  |  |
| --- | --- |
| Méthodes | Descriptions |
| .charAt() | Retourne le caractère selon son indice (base zéro) |
| .indexOf() , lastIndexOf() | Retourne l’indice d’un caractère à partir du début ou de la fin de la chaîne |
| .trim() | Supprime les espaces inutiles en début et fin de chaîne |
| .toUpperCase(), .toLowerCase() | Convertit en MAJUSCULES, en minuscules |
| .substr(), .substring() | Extrait une sous-chaîne de caractères |
| .big(), .italics() | Transforme en plus grand, en italique (comme le fait HTML) |

Le nom d’une variable (ou d’une fonction) se nomme *identificateur*.

Voici la liste des mots clés réservés à ne pas utiliser pour nommer vos variables JavaScript : « *break, case, catch, class, const, continue, debugger, default, delete, do, else, enum, export, extends, false, finally, for, function, of, import, in, instanceof, new, null, return, switch, super, this, throw, true, try, typeof, var, void, while, with* ».

On peut vérifier le type en cours d’une variable avec la fonction *typeof****()*** ou l’attribut*constructor* (voir exercice 3.5).

***Pour la clarté de votre script, on ne peut que vous conseiller :***

* ***de déclarer toutes les variables d’un bloc au début de bloc,.***
* ***d'utiliser à chaque fois le mot-clé var pour déclarer une variable,***
* ***de lui assigner un nom correspondant à son contenu,***
* ***et de ne pas faire varier son type dans un même script.***

## Les opérateurs

Voici la liste des opérateurs les plus courants mis à disposition par JavaScript :

|  |  |
| --- | --- |
| Opérateurs | Descriptions |
| +, -, \*, /, %, = | Opérateurs arithmétiques de base |
| ==, ===, <, >, <=, >=, !=, !== | Opérateurs de comparaison |
| +=, -=, \*=, /=, %= | Opérateurs associatifs |
| &&, ||, ! | Opérateurs logiques (AND, OR et NOT) |
| X++, x-- | Opérateurs d’incrémentation et de décrémentation |
| + | Concaténation de chaînes de caractères |

L’opérateur d’identité === contrôle également le même typage des 2 valeurs. Exemple :

console.log(42 == "42"); // retourne true

console.log(0 == false); // retourne true

console.log(42 === "42"); // retourne false

console.log(0 === false); // retourne false

*Pour être plus précis dans vos comparaisons, préférez l’opérateur d’identité ===.*

## Les conditions

A un moment ou à un autre de la programmation, on aura besoin de tester une condition. Ce qui permettra d'exécuter ou non une série d'instructions.

**« Si maman si » ou l’expression IF :**

**if (**condition vraie**)** une seule instruction**;**

**if (**condition vraie**)  {**

une;

ou plusieurs instructions;

**}**

**if (**condition vraie**) {**

instructions1;

**} else if {**

instructions2;

**} else {**

Instructions3;

**}**

Moins lisibles, *les expressions ternaires retournent une valeur***.**

**(**test condition**) ?** valeur si vrai **:** valeur si faux**;**

Exemple :

var genre = "f";

alert((genre == "h")? "Monsieur" : "Madame");

Et si ma condition de test propose plusieurs sorties… je ***switch*** :

var animal = "oiseau";

**switch(**animal**) {**

case "chien": …

case "oiseau" : …

case "poisson": …

**case** "vache" **:**

console.log("C'est un vertébré");

**break;**

case "mouche" :…

**default :**

console.log("C'est un invertébré");

**}**

*Pour tous les tests conditionnels, ordonnez les tests du plus probable au moins probable.*

## Les répétitions

**« Je t’ai répété 100 fois … », la boucle FOR :**

**for (**var i = 0 **;** i < 100 **;** i++ **) {**

console.log("Préfère la boucle FOR si tu connais le nombre !");

**}**

**« Tant que tu ne comprends pas, je recommencerai… », les boucles WHILE :**

var i;

**while (**!(i)**) {**

i = confirm("As-tu compris ?");

**}**

**do {** // l’instruction suivante sera exécutée au moins 1 fois !

i = prompt("laisse vide ou annule");

**} while (**i**);**

L'instruction ***break*** permet d'interrompre prématurément une boucle *for* ou *while (mais elle ne devrait jamais être utilisée en bonne programmation structurée)*.

L'instruction ***continue*** permet de sauter une instruction dans une boucle *for* ou *while* et de passer à l’itération suivante de la boucle (*sans sortir de celle-ci comme le fait break, mais elle ne devrait jamais être utilisée elle aussi*).

Exemple :

var compt=0;

while (compt<10) {

compt++;

if (compt == 3) continue;

if (compt == 6) break;

console.log("ligne : " + compt);

}

# Les outils de développement

Pour apprendre et exploiter le langage JavaScript, il vous faut au minimum :

* Un navigateur qui reconnaît et interprète le langage JavaScript,
* Et un éditeur de texte (NotePad++, SublimeText …).

Pour faciliter le débogage, vous pouvez aussi utiliser des outils plus spécialisés, comme :

* Une interface de développement intégrée (Eclipse, NetBeans …),
* Ou des outils de mise au point intégrés au(x) navigateur(s).

## Le débogage

Au chargement du script par le navigateur, JavaScript passe en revue les différentes erreurs de syntaxe qui pourraient empêcher le bon déroulement du script. En cas d’anomalie, l’interpréteur ne va pas plus loin et le navigateur se comporte comme si la page Html ne contenait pas de script JavaScript.

*Si JavaScript décèle une erreur de syntaxe, c’est tout le chargement du bloc qui est annulé !*

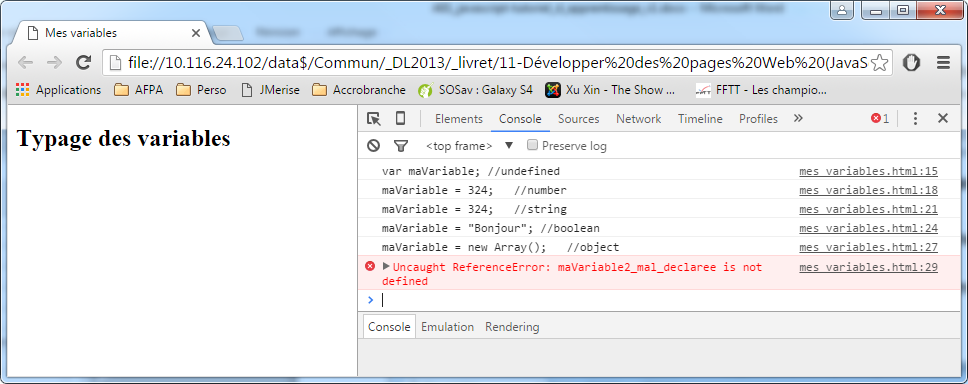
L’utilisation de la fonction *alert()* aide très souvent le développeur à déceler une erreur en affichant des boites de dialogue supplémentaires permettant de tracer le déroulement d’un script.

Un autre bon moyen de contrôler un ensemble de valeurs est de les afficher dans la console du navigateur au moyen de la fonction *console.log()*.

La console du navigateur est accessible en lançant l’outil de développement du navigateur.

*C’est également dans cette console que l’on pourra récupérer les informations sur les erreurs de syntaxe.*

Exemple sous Chrome :



Tous les navigateurs modernes possèdent des outils permettant d’explorer la structure de la page, les styles CSS et d’afficher des informations sur le code JavaScript, les valeurs courantes des variables, ainsi que les éventuelles erreurs de syntaxe.

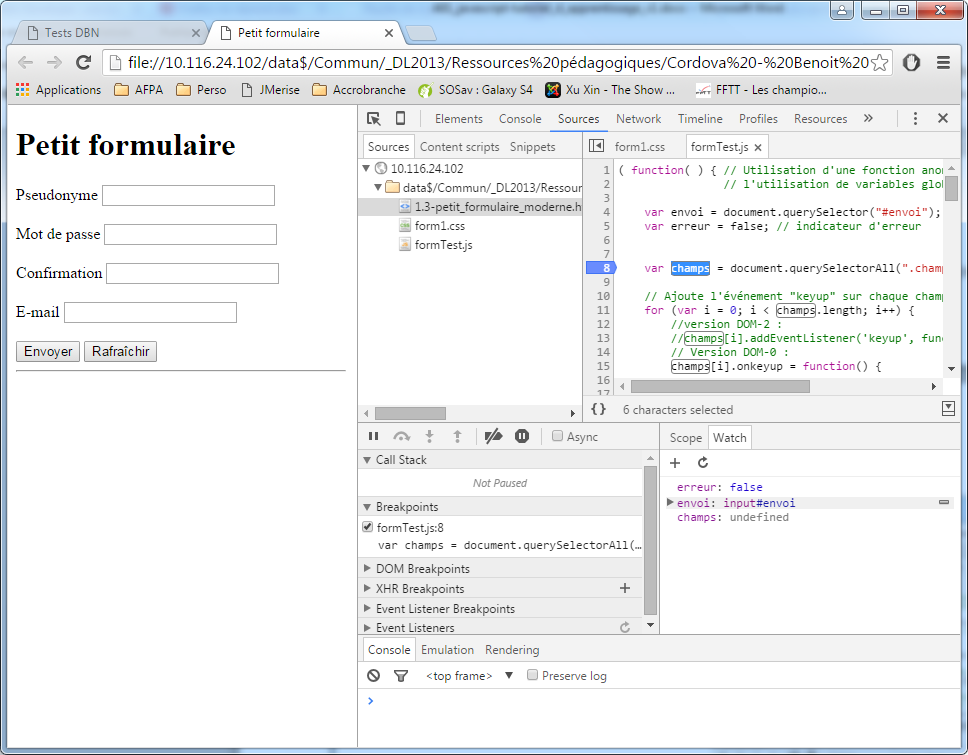
Chaque navigateur utilise des outils propriétaires. Leurs utilisations et les informations retournées sont différentes d’un navigateur à l’autre.

Ces outils permettent notamment :

* D’inspecter les éléments HTML et CSS et de modifier « à la volée » leurs valeurs.
* De déboguer le code JavaScript avec des *points d’arrêt* et des *exécutions pas à pas*.
* De contrôler les requêtes HTTP, notamment pour AJAX.

## Les outils de Google Chrome

On peut appeler la barre de développement de Chrome avec le raccourci ***F12*** ou le raccourci ***CTRL+MAJ+I*** ou via le menu « ***paramètres/plus d’outils/outils de développement*** »



« clavier » pour contrôle déroulement

Fenêtre du débogueur placée sur la droite

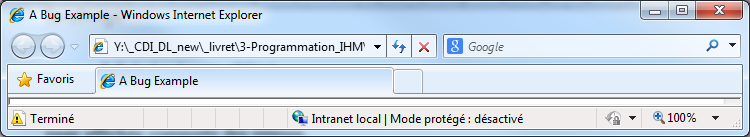
Exploration des variables JavaScript

Exploration du code JavaScript et point d’arrêt

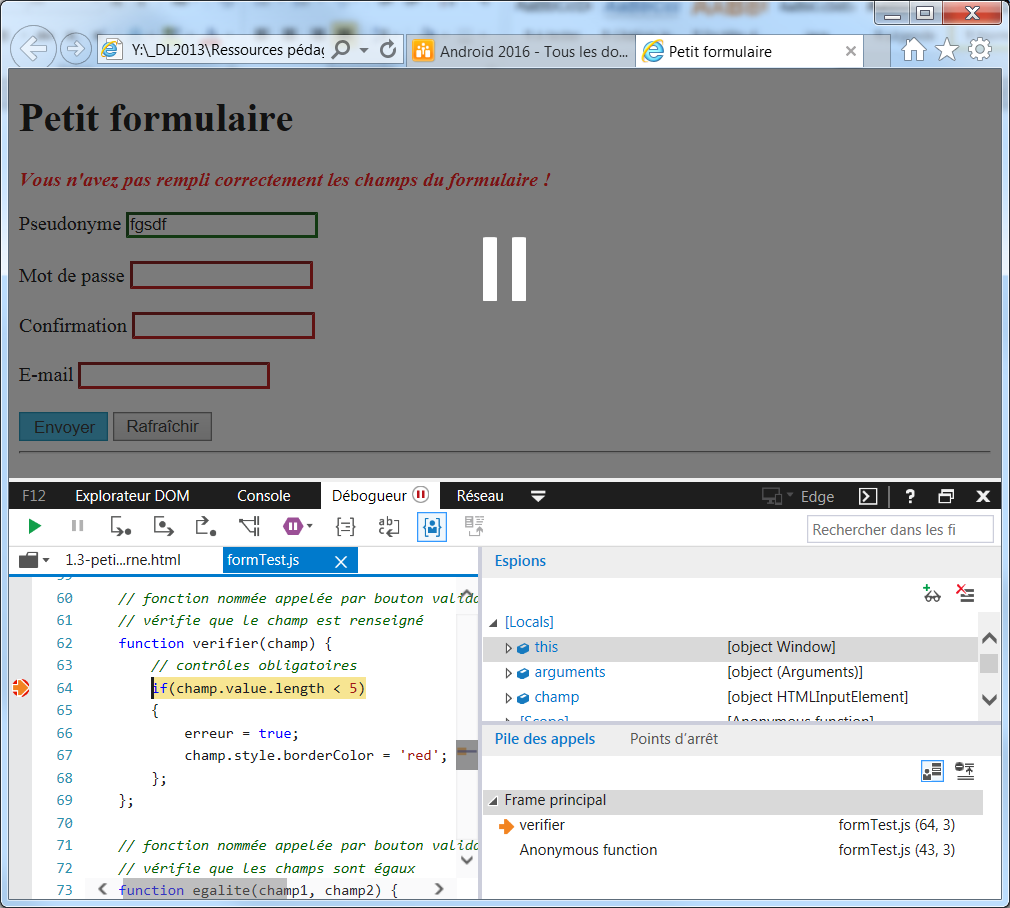
Fenêtre principale d’affichage de la page Web

Pour avancer pas à pas après une pause sur un point d’arrêt, utiliser les touches de fonction F10 et F11.

## Les outils de Microsoft Internet Explorer et Edge

Les anciennes versions d’Internet Explorer affichaient une barre d’état dans laquelle apparaissait un icône d’alerte si la page comportait des erreurs.

On peut appeler la barre de développement d’Internet Explorer avec le raccourci ***F12*** ou via le menu « ***paramètres/outils de développement*** »



« clavier » pour contrôle déroulement

Fenêtre du débogueur placée en partie basse

Fenêtre principale d’affichage de la page Web

Exploration de variables JavaScript

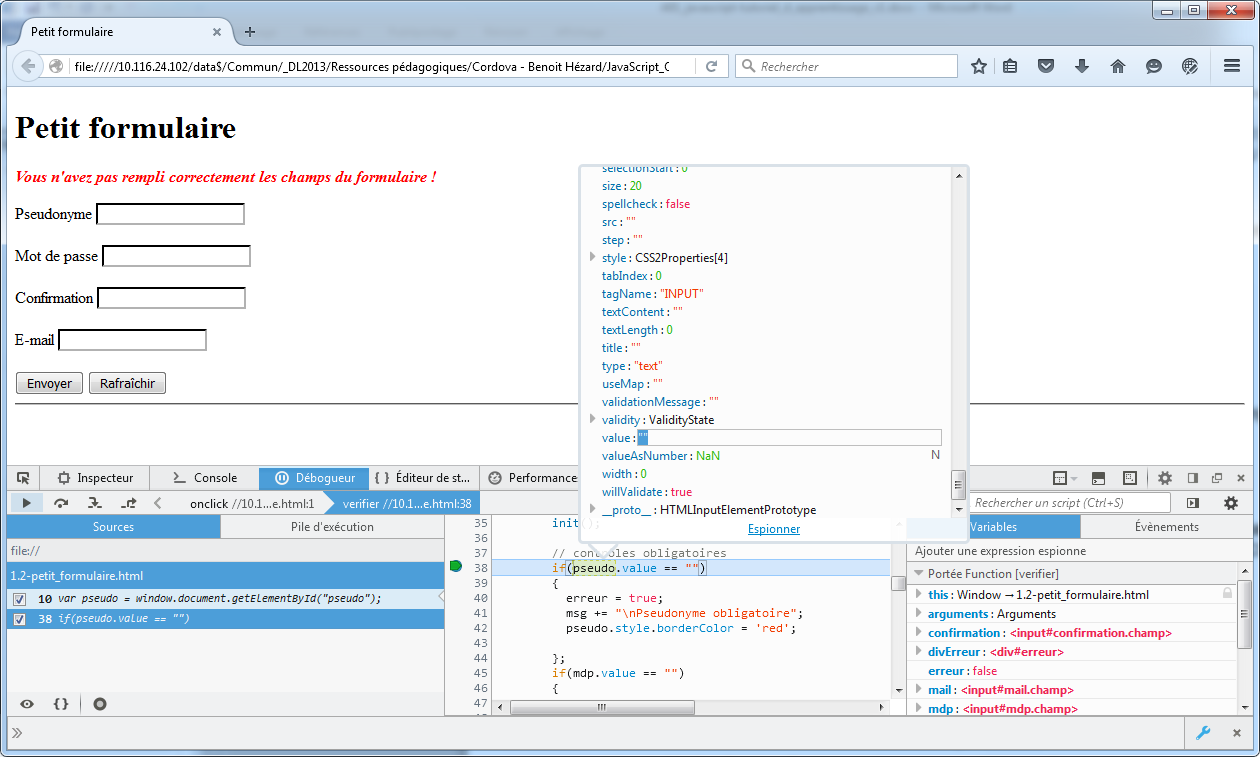
Exploration du code JavaScript et point d’arrêt

Pour avancer pas à pas après une pause sur un point d’arrêt, utiliser les touches de fonction ***F10*** et ***F11*** aussi bien avec Internet Explorer, Edge ou Chrome

Référence : <https://msdn.microsoft.com/fr-fr/library/bg182326(v=vs.85).aspx>

## Les outils de Mozilla FireFox

On peut appeler la barre de développement de Mozilla FireFox avec le raccourci ***F12*** ou via le menu « ***paramètres/Développement/outils de développement*** »



Les outils de développement intégrés dans les dernières versions du navigateur Mozilla FireFox sont très performants.

Le navigateur Mozilla FireFox propose aussi des plug-ins très intéressants comme :

* ***Web Developer***. Il ajoute une barre d’outils qui affiche en direct la validité du code HTML/CSS et JavaScript. Il peut désactiver « à la volée » le code CSS, le JavaScript, les images, gérer les cookies, les formulaires, lancer la console d’erreur, et bien plus…
* ***Firebug***. Extension très utilisée avant l’arrivée du dernier module intégré de développement de Mozilla.



## **Exercice : testez l’exemple fourni « mes-variables.html »

Chargez dans vos différents navigateurs le fichier fourni ‘**Mes-variables.html’** et observez à travers les outils de débogage l’état des variables, les traces générées dans la console et les messages d’erreurs.

# Les fonctions

Une *fonction* a pour but principal de définir un *bloc d’instructions* à un seul endroit du script, *réutilisable* et exécutable autant que nécessaire par simples appels depuis le script principal (ou depuis une autre fonction). Les notions sous-jacentes sont celles de ‘sous-programme’ et de ‘modularisation du code’.

Les *procédures* n’existent pas en tant que telles en JavaScript ; une fonction qui ne retourne rien est donc une procédure. JavaScript utilise l’instruction **return** pour retourner une valeur et redonner la main au programme appelant.

Si l’instruction **return** n’est pas spécifiée, la fonction retournera **undefined**.

Le langage JavaScript permet de définir ses propres fonctions (mot-clé **function** ) et possède des fonctions natives très utiles.

Exemple :

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| eval(string) | Méthode qui évalue la chaîne passée en argument comme un script JavaScript. Exemple :  eval("x=10;y=20;console.log(x\*y);"); |
| isNaN(x) | Méthode qui retourne ***true*** si le paramètre *x* n’est pas un nombre. |
| parseFloat(string) | Méthode qui convertit la chaîne en nombre à virgule flottante. |
| parseInt(string) | Méthode qui convertit la chaîne en entier. |

*Attention : Les fonctions parseFloat() et parseInt() acceptent une chaîne de caractères. Si la chaîne commence par un nombre, la fonction le renverra. Si le premier caractère n'est pas un nombre, la fonction renverra "NaN" (Not A Number).*

Orientation objet oblige, de nombreuses fonctions standards sont livrées sous formes de ‘*méthodes d’instances’* ou ‘*méthodes statiques’*.

Exemples :

Math.sqrt() est une *méthode statique* de l’objet prédéfini Math ; elle calcule la racine carrée du nombre fourni en paramètre.

"Centre Afpa".indexOf("C") retourne la valeur 0, soit l’indice du caractère ‘C’ dans la chaîne qui est implicitement un objet string doté de nombreuses méthodes.

Il existe plusieurs sortent de fonctions :

* Les instructions de fonctions (les plus courantes),
* Les expressions de fonctions,
* Les fonctions anonymes (qui servent à isoler une partie du code),
* Les fonctions « Callback »,
* Les fonctions auto-exécutables,
* Les fonctions issues d’un objet (les méthodes et les constructeurs).

## Les instructions de fonctions

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

function nomFonction(liste\_paramètres\_reçus){instructions\_à\_exécuter};

Exemples :

// calcul la surface et la retourne

function calculeSurface (largeur, hauteur) {  
 return largeur \* hauteur;    
};

// affiche un message constant

function coucou() { alert("coucou !"); }

L'appel d'une fonction se fait le plus simplement du monde par le nom de la fonction suivi des parenthèses qui incluent les éventuels paramètres à passer à la fonction :

Var s = calculeSurface(8, 4); // s est affecté par la valeur 32

coucou(); // affiche "coucou !" dans une boîte de dialogue

*En JavaScript les instructions de fonctions sont automatiquement remontées en haut du bloc de script. Il est donc possible d’appeler ces fonctions avant de les avoir déclarées !*

## La portée des variables

*Avec les fonctions, le bon usage des variables locales et globales prend toute son importance.*

Une variable déclarée dans une fonction par le mot clé **var** aura une portée limitée à cette seule fonction. C’est une *variable locale* accessible uniquement par cette fonction.

Les *paramètres éventuels* de la fonction constituent aussi des *variables locales*.

En revanche, toute variable déclarée sans le mot clé ***var*** aura une *portée globale*. Elle sera une propriété de l’objet prédéfini **window**.

Les variables déclarées à l’extérieur de la fonction ou globales sont bien entendu visibles elles aussi dans la fonction.

Exemple :

var nomExterne = "Hein "; // var ‘locale’ pour le script, donc globale

function portee(nom) {

var prenom = "Terieur "; // var locale

nomGlobale = "Halle "; // var globale

console.log(window.nomGlobale + nom + prenom);

console.log(nomGlobale + nomExterne + prenom);

}

portee("Ex ");

console.log(prenom);// provoque une erreur

## **Exercice : La portee

Réalisez maintenant l’exercice ‘**La portée’** de manière à expérimenter les déclarations et accès aux variables locales et globales.

## Les paramètres des fonctions

Contrairement aux langages fortement typés comme Java ou C#, la ***signature*** *ou* ***liste des paramètres*** 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#).

La *fonction constructeur de l’objet standard* ***Date*** en est un parfait exemple : Date() ; retourne la date et heure du jour alors que Date("December 17, 2015 03:24:00") ; ou encore Date(2015,11,17) ; retournent une date (et une heure) spécifiées. Dans ces différents cas, le nombre et le type des arguments sont très variables.

A chaque appel d’une fonction, l’objet **arguments** stocke tous les paramètres envoyés lors de l’appel de la fonction. Ainsi, en JavaScript, *le développeur n’écrit qu’une seule définition de fonction en cas de variantes/surcharges mais il se doit de tester les paramètres reçus*.

Exemple : la fonction ci-dessous accepte de 0 à n paramètres en fonction de la forme géométrique dont on veut calculer le périmètre

function perimetre(largeur, longueur) {

var resultat = 0;

// test si au moins un paramètre reçu

if (!largeur) resultat = 0;

else if (!longueur) resultat = 4\*largeur; // 1 param recu : carré

else if (arguments.length == 2)

resultat = (largeur + longueur)\*2; // 2 param : rectangle

else {

for (i in arguments) resultat += arguments[i]; // polygone

}

console.log(resultat);

}

perimetre(); // affiche 0

perimetre(5); // affiche 20

perimetre(9,6); // affiche 30

perimetre(3,7,20,8); // affiche 38

## **Exercice : L’objet arguments

Réalisez maintenant l’exercice « **l’objet argument** » de manière à enrichir cette fonction périmètre.

## Les expressions de fonctions

Les expressions de fonctions passent par la création d’une variable affectée par la définition d’une fonction :

var getCalculeSurface = function calculeSurface(largeur, hauteur) {  
 return largeur \* hauteur;    
}; // Attention à ne pas oublier le point-virgule de l’instruction !

var s = getCalculSurface(8, 4);

Dans ce cas, la fonction elle-même n’a plus besoin de nom. On dit qu’elle est *anonyme* :

var getCalculeSurface = function (largeur, hauteur) {  
 return largeur \* hauteur;  
};

var s = getCalculSurface(8, 4);

On peut aussi affecter une fonction déjà créée à une variable :

var getCoucou = coucou;

function coucou() { alert("coucou !"); }

getCoucou(); // affiche "coucou !" dans une boîte de dialogue

*Attention qu’il s’agit bien de* ***demander l’exécution immédiate de*** *la fonction ; c’est pourquoi, le nom de la variable doit être suivi des habituelles parenthèses !*

Les expressions de fonctions diffèrent des instructions de fonctions :

* *Elles peuvent être déclarées n’importe où dans le code (dans un if() par exemple),*
* Elles ne sont pas remontées automatiquement en haut du bloc de script (elles ne peuvent donc pas être appelées avant d’avoir été déclarées).
* Si l’expression de fonctions utilise une fonction anonyme, elle ne pourra pas être récursive.

## Les fonctions anonymes

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.

Les fonctions anonymes sont très utilisées dans le langage JavaScript, notamment dans la gestion des évènements, les objets, les closures, et les callback …

En voici déjà un aperçu :

JavaScript propose 4 fonctions dîtes temporelles :

* var id = **setTimeout**(fct1, temps) : crée un *timer* qui appelle la fonction *fct1()* après le *temps* écoulé (en milliseconde).
* var id = **setInterval**(fct2, temps) : crée un *timer* qui répète l’appel de la fonction *fct2()* à toutes les intervalles de *temps* (en milliseconde).
* ***clearTimeout(****id****)*** qui arrête le *timer* avant l’expiration du délai fixé.
* ***clearInterval(****id****)*** qui arrête le *timer* avant le prochain appel de la fonction.

*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 !*

Les fonctions appelées fct1() et fct2() peuvent même être des fonctions anonymes déclarées à l’intérieur de l’appel de la fonction de *timer*.

Exemple d’un minuteur :

var i = 9;

var decompte = **setInterval**(function() {

console.log(i--); // décompte de 10 à 1

}, 1000); // se lance toutes les secondes

var minuteur = **setTimeout**(function() {

var d = new Date();

var date = d.getHours() + ":" + d.getMinutes();

alert("Après 10 secondes, il est " + date);

**clearInterval**(decompte); // stoppe le décompte

}, 10000); // se lance après 10 secondes

## Les fonctions « Callback »

*Un callback est une fonction de retour, nommée ou anonyme, placée en paramètre d’une autre fonction qui n’est pas exécutée aussitôt mais à un moment donné*, à la suite d’un lapse de temps défini, ou d’un événement précis en cas de fonctionnement asynchrone.

💣

*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 !).

Reprenons l’exemple du minuteur précédent en ajoutant un paramètre pour définir la durée de départ :

var temps = 10;

**setInterval**(function() {

(function(duree) {

console.log(duree);}(temps--));

}, 1000);

Ici, à chaque seconde la fonction anonyme est déclenchée et elle appelle une fonction anonyme en lui passant en paramètre temps--.

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 asynchrones comme Ajax ou l’accès aux bases de données embarquées reposent sur l’usage de fonctions callback.

## Les fonctions auto-exécutables

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 !)

💣

Ou encore :

**(function()** {

console.log('hello world');

**})()**;

Exemple :

**(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.

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.

(function() {

console.log('hello world');

function fct1() { … };

function fct2() { … };

…

})();

*Toutes ces notions avancées sur les fonctions JavaScript font l’objet d’une étude plus approfondie lors d’une autre séance.*

# L’interactivite avec l’HTML

## Le DOM

Le **DOM ou Document Object Model** est une interface de programmation (ou **API**[[9]](#footnote-9)) 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.

Le W3C[[10]](#footnote-10) a défini un DOM standard qui permet d’accéder à tous les éléments du document. Ce standard a évolué dans le temps, la version actuelle étant DOM-2. Il réside quelques différences entre les navigateurs en termes d’implémentation du DOM.

💣

*Le DOM convertit la description Html en une structure arborescente d’objets JavaScript*  
*(plus rapide et plus facile à traiter).*

A la racine, l’objet **window** représente l’instance du navigateur ; il référence principalement l’objet **location** qui symbolise la barre d’adresse, l’objet **history** qui représente l’historique des pages visitées par l’utilisateur, et l’objet **document** qui représente la page Web en cours, le contenu du <body> Html, lui-même référençant tous ses éléments dont le tableau **images** qui référence tous les éléments <img> de la page.

Voici le modèle objet du navigateur :

window

document

frames[]

location

history

screen

navigator

forms[]

anchors[]

links[]

images[]

applets[]

embeds[]

elements[]

Button  
Checkbox  
FileUpload  
Hidden  
Password  
Radio  
Reset  
Select  
Submit  
Text  
Textarea

options[]

***anchors****[] : tableau des ancres (balises <a>)*

***links****[] : tableau des liens (attributs href dans les balises <a> et <area>)*

***embeds****[] : tableau des objets multimedia (balises <embed>)*

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, l’instruction suivante retourne la valeur du premier élément HTML permettant la saisie dans le premier formulaire de la page.

window.document.forms[0].elements[0].value

Depuis l’arrivée du standard DOM-2, la méthode ***.*getElementByID()** de l’objet document permet d’adresser directement tout élément HTML doté d’un **attributid*.*** 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 des éléments dans l'arbre DOM ; ces deux méthodes prennent pour paramètre un ou plusieurs **sélecteurs CSS** séparés par des virgules.

Exemple :

var allLIAndOl = document.querySelectorAll("li,ol") ;

Pour bien comprendre la correspondance entre les éléments HTML et les objets JavaScript, déroulez l’animation « **leDOMJavaScript.ppsx »** puis parcourez le code source du fichier « **leDOMJavaScript.html** » et testez le en plaçant des points d’arrêt et en observant la console d’un débogueur JavaScript.





*Pour en savoir plus :* [*http://www.alsacreations.com/article/lire/1445-dom-queryselector-queryselectorall-selectors-api.html*](http://www.alsacreations.com/article/lire/1445-dom-queryselector-queryselectorall-selectors-api.html)

## JavaScript et les propriétés des éléments

Pour récupérer, modifier ou ajouter des attributs aux éléments HTML, JavaScript possède les 2 fonctions **getAttribute()** et **setAttribute()**. Exemple :

var elem = document.getElementById("div1");

var nomDiv1 = elem.getAttribute('name'); // récupère l’attribut 'name'

elem.setAttribute('class', "maClasse"); // ajoute l’attribut "class"

*Chaque objet JavaScript correspondant à un élément Html de la page est automatiquement doté de propriétés correspondant aux attributs Html* (en minuscule), et de méthodes permettant leur manipulation par programmation JavaScript.

Exemples :

window.location.href // retourne l’URL courante de la page

window.location.replace(xxx) ; // modifie l’url courante du navigateur

var monImage = document.querySelector("#img1");

monImage.src = ‘xxx.jpg’ ; // modifie sur la source de l’image

// Récupère le contenu de l’élément dont l’id = "msg"

var msg = document.querySelector("#msg").value;

var message = document.getElementById(‘msg’).value;

// change la couleur de fond de page

document.bgColor = ‘blue’ ; // js reprend le système de couleurs html

Si un nom d’attribut Html est composé de plusieurs mots, la propriété correspondante en JavaScript attache les mots et force en majuscule la 1ère lettre de chaque mot (hormis le 1er mot).

Exemple :

document.querySelector("#nom").**readOnly** = true;

Certains attributs Html sont des mots-clés réservés en JavaScript. Exemples :

* L’attribut ***for*** devient la propriété ***htmlFor***,
* L’attribut ***class*** devient la propriété ***className ou classList***

JavaScript donne ainsi *accès pour consultation ou modification aux* *classes de styles CSS* des éléments Html ; il permet en outre *d’accéder à tous les attributs CSS* grâce à la propriété **style** qui elle-même dispose de nombreuses propriétés correspondant aux différents attributs CSS.

Exemple :

document.getElementById("div1").style.border = “3px solid blue”;

Il existe 3 propriétés qui permettent de récupérer le code présent dans un élément du DOM :

* La propriété innerHTML : récupère le code Html inclus dans un élément,
* Les propriétés textContent et innerText : récupèrent le code brut sans les balises HTML.
  + innerText est reconnue par IE et Chrome mais pas Firefox.
  + textContent est reconnue par Firefox et Chrome mais pas IE.

## JavaScript et les éléments de formulaires HTML

En ce qui concerne les formulaires Html, il est très fréquent de contrôler en JavaScript les saisies effectuées par l’utilisateur.

Le contenu d’un élément de formulaire est bien entendu disponible par la propriété JavaScript **value** qui correspond à l’attribut Html value des éléments de saisie ; la propriété value des boutons de commande correspond en Html au libellé affiché sur le bouton ; elle n’est en général exploitée par JavaScript que pour changer dynamiquement le libellé d’un bouton (ex : Start 🡸🡺 Stop).

La propriété name reprend la valeur de l’attribut Html name.

Les attributs Html disabled et readonly peuvent être consultés ou modifiés par JavaScript (propriétés booléennes disabled et readOnly) de manière à activer/désactiver des zones de saisie selon le contexte.

Tous les composants affichés d’un formulaire Html disposent de méthodes JavaScript focus() et blur() pour respectivement prendre ou rejeter le curseur de saisie. De plus, les zones de saisie de texte peuvent automatiquement sélectionner leur contenu affiché grâce à la méthode select().

Les ***listes déroulantes***, éléments Html <select>, disposent en JavaScript d’un tableau options correspondant aux différents éléments Html <option> contenus. Bien entendu la propriété value de l’objet select prend la valeur de la propriété value de l’objet option choisi par l’utilisateur.

Pour les ***boutons radio*** (qui portent la même valeur d’attribut name en Html) JavaScript permet d’accéder à chacun des boutons pour vérifier sa propriété booléenne checked et sa value éventuelle.

De même pour les ***cases à cocher*** Html, JavaScript donne accès aux propriétés name, value et checked.

A noter que même si Html préconise d’écrire une valeur textuelle pour les attributs de pré-sélection par l’utilisateur (checked=’checked’ ou selected=’selected’), JavaScript considère les propriétés checked et selected comme étant de type boolean (valeur true ou false).

Enfin, un ***formulaire*** dispose des propriétés action et method, correspondant aux attributs Html similaires, et d’une méthode submit() qui permet de reproduire par programmation la soumission d’un formulaire telle que la réalise l’utilisateur en cliquant un bouton Html type="submit" (le chapitre 7 détaille la gestion des formulaires par JavaScript).

Pour aller plus loin dans l’exploration des propriétés et méthodes des objets JavaScript, consultez le document complémentaire « **Résumé des objets JavaScript.pdf** » ainsi que la documentation de référence sur



<https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement>

## **Exercice : Manipulons le dom

Réalisez maintenant l’exercice « **Manipulons le DOM** ».

Cet exercice vous permet de manipuler simplement des éléments Html de saisie sans prendre en compte les fonctionnalités de formulaire.

# L’interactivité avec les événements

## Généralités

Avec les événements et surtout leur gestion par JavaScript, nous abordons le côté "*magique*" de JavaScript.

En Html classique, il y a des événements que vous connaissez bien : ce sont le clic de la souris sur un lien pour vous transporter sur une autre page Web et le clic d’un bouton de formulaire ; ce sont hélas les seuls événements que gère Html. Heureusement, JavaScript va permettre de gérer tout ce qui peut se passer pendant la vie d’une page Web dans un navigateur, pour votre plus grand plaisir.

Les événements JavaScript, associés aux fonctions, aux méthodes et aux formulaires, ouvrent grand la porte pour une réelle *interactivité* de vos pages.

JavaScript peut générer toutes sortes d’événements sur les objets du DOM Html comme le clic sur une image, le survol de la souris sur une div, le changement de valeur ou de *focus* d’un champ de saisie, l’appui sur une touche du clavier, ou encore la soumission d’un formulaire par l’utilisateur, …

*Chaque événement JavaScript sera associé à une fonction dite asynchrone*. La fonction ne s’exécutera que lorsque l’événement sera déclenché !

## Les types d’événements

Passons en revue différents événements implémentés en JavaScript.

|  |  |
| --- | --- |
| **Description** | **Attribut HTML** |
| Cliquer sur un bouton, un lien ou tout autre élément du DOM. | onclick, ondblclick, onmousedown, onmouseup |
| Lorsqu’un élément prend ou perd le focus ou change de valeur. | onfocus, onblur, onchange |
| Déplacer le pointeur de la souris sur un lien ou tout autre élément. | onmouseover, onmouseout, onmousemove |
| Taper au clavier. | onkeyup, onkeydown, onkeypress |
| Charger ou quitter une page. | onload, onunload |
| Sélectionner un champ dans un élément de formulaire. | onselect |
| Envoyer un formulaire. | onsubmit |
| Modifier la taille de la fenêtre | onresize |
| En cas d’erreur (comme le chargement d’une image qui échoue). | onerror |

## Mettre en place des événements

Vous avez 3 méthodes à votre disposition pour ajouter des événements sur des objets de votre page Html :

* Directement dans les balises Html en ajoutant l’attribut d’événement correspondant :

<input type="button" id="btn1" **onclick**="maFunctionJS();" />

* En JavaScript via le DOM-0 en passant par les propriétés événementielles des objets :

var btn = document.getElementById("btn1");

btn.**onclick** = function(e) { … };

* En JavaScript via le DOM-2, en ajoutant un *handler* d’événement à l’objet :

var btn = document.getElementById("btn1");

btn.**addEventListener**("**click**", function(e) { … }, false);

Notez que l’argument baptisé ici ‘e’, de type event, correspond à la transmission à la fonction de la référence à l’événement déclencheur (voir plus loin).

*La 1ère méthode est aujourd’hui dépréciée et à proscrire* pour 2 raisons :

* Tout d’abord, l’appel à la fonction JavaScript est faite depuis le code Html, elle est donc *intrusive*. Elle ne respecte pas l’enrichissement progressif et la séparation des couches.
* De plus, le DOM-2 apporte beaucoup d’information via l’objet **event**. Hors, cet objet ne peut être créé via l’attribut HTML.

Avec la deuxième méthode, toute association d’une fonction à un événement est écrasée lorsque l’on associe de nouveau une fonction à ce même événement. *Si on utilise des bibliothèques externes, on risque d’écraser un événement avec le DOM-0 !*

Avec la dernière méthode, le DOM-2 permet d’associer à un même élément plusieurs fonctions liées au même type d’événement ; il suffit d’appeler successivement plusieurs fois la méthode addEventListener().

Le DOM-2 permet également de contrôler à quelle phase du cycle l’événement sera appelé. (via le 3ème paramètre booléen de la méthode addEventListener()).

Imaginez que vous ayez une image dans une div et que ces 2 éléments soient associés à des fonctions différentes pour le même évènement « onmouseover ». Si vous passez votre souris sur l’image, quel événement sera déclenché en 1er ?

Quand un type d’événement est déclenché, il va se propager dans l’arbre du DOM depuis l’élément document jusqu’à l’élément le plus bas dans la hiérarchie du DOM (phase descendante ou dite de ***capture***) puis il va remonter dans l’arbre jusqu’à l’élément document (phase montante ou dite de ***bouillonnement***). Le dernier paramètre de la méthode addEventListener() permet de contrôler cela.

Voir <https://www.w3.org/TR/DOM-Level-3-Events/#event-flow>

*Remarque : La phase descendante est très peu utilisée. Les attributs Html, le DOM-0 et les anciennes versions des navigateurs ne gèrent que la phase montante (valeur "false").*

## L’objet event

Avec la méthode addEventListener() DOM-2, on peut mentionner un paramètre pour la fonction appelée par l’événement. Ce paramètre sera un objet **event** qui contiendra différentes informations comme :

* Le type d’événement déclencheur,
* L’élément source,
* Si c’est la souris, ses coordonnées,
* Si c’est le clavier, la touche frappée …

L’objet **event** permet également d’annuler les opérations liées à l’événement et d’empêcher sa propagation avec les méthodes **preventDefault()** et**stopPropagation()**.

## Supprimer un événement

On peut supprimer une capture d’événement avec la méthode **removeEventListener()** :

var btn = document.getElementById("btn1");

btn.**addEventListener**("click", maFunctionJS, false);

btn.**removeEventListener**("click", maFunctionJS, false);

## **Exercice : Ordre des événements

Testez l’exemple fourni « **ordre des événements.html** » : étudiez le code et testez dans votre navigateur en observant la console et en plaçant des points d’arrêt pour exécuter le script pas à pas.

## ** Exercice : Manipulons les événements

Reprenez l’exercice « Manipulons de DOM » pour implémenter par JavaScript le *handler* d’événement sur le bouton Choix à l’aide d’une fonction JavaScript (interne ou externe) en auto-exécution dès le chargement de la page. Testez que le comportement reste bien identique.

## ** Exercice : Le post-it

Réalisez maintenant l’exercice « **le Post-it** » de manière à mettre en œuvre la manipulation des éléments de la page lors d’événements divers.

## ** Exercice : Le chrono

Réalisez maintenant l’exercice « **le Chrono** » de manière à mettre en œuvre et gérer un *timer* JavaScript (abordé en 4.7).

# Les formulaires

## Généralités

Avec JavaScript, les formulaires Html prennent une toute autre dimension. N'oublions pas qu'en JavaScript, on peut accéder à chaque élément d'un formulaire pour y aller lire ou écrire une valeur, y associer un gestionnaire d'événement... Tous ces éléments renforceront grandement les capacités interactives de vos pages.

## Déclaration d'un formulaire

Un formulaire est l'élément Html déclaré par les balises <form></form>.

Il faut noter qu'en JavaScript, l’attribut id="id\_du\_formulaire"a toute son importance pour désigner le chemin d’accès global aux éléments. En outre, les attributs action et methodsont déterminants sur le comportement du navigateur lors de la soumission du formulaire mais ils restent facultatifs tant que vous ne faites pas appel au serveur Web.

## Accès au formulaire et à ses éléments

En JavaScript, on peut accéder à un objet de la page via l’arborescence du DOM du navigateur et des tableaux indicés et associatifs ; ainsi on peut accéder au *formulaire lui-même* :

document.forms[0]…

document.forms['monFormulaire']…

On peut également accéder directement à tout élément via les deux méthodes suivantes qui reprennent les attributs name et id des *éléments* du formulaire :

// retourne l’élément unique indicé par id = 'idElt'

document.getElementById('idElt');

// retourne la liste des éléments ayant comme attribut HTML

// name : 'nameElt' – par exemple des boutons radio

document.getElementsByName('nameElt');

## Contrôles de saisie et soumission des formulaires

JavaScript est très utile et très apprécié pour valider les formulaires Html côté client car le processus est plus rapide et efficace pour l’utilisateur que la validation côté serveur Web. Toutefois, il reste primordial de contrôler également les formulaires côté serveur dans le cas où l’utilisateur désactiverait JavaScript et pour détecter des requêtes au serveur malintentionnées.

Ces contrôles peuvent être faits en JavaScript avec les gestionnaires d’événements comme onClick, onBlur, onSubmit, onReset… mais aussi en utilisant les *méthodes de l’objet formulaire* qui simulent une action Html comme submit() et reset().

* La méthode submit() reproduit l’événement lié au bouton Html de type *"submit"* en validant le formulaire.
* La méthode reset()reproduit l’événement lié au bouton Html de type *"reset"* en réinitialisant le formulaire.

Avec HTML5, JavaScript a perdu de son intérêt pour les contrôles de saisie de base comme les champs obligatoires (required= ‘required’), le contrôle de valeurs par rapport à un modèle (pattern= ‘expression\_régulière’), comme pour une adresse email, un numéro de téléphone ou un code postal par exemple. Toutefois, ces contrôles standards Html5 génèrent des messages d’erreur standards et un algorithme JavaScript peut se révéler nécessaire quand on souhaite affiner les messages à l’utilisateur.

Si une action est mentionnée dans la balise Html <form>, elle sera exécutée automatiquement par le navigateur lorsque l’utilisateur déclenche le bouton *submit* afin d’envoyer les données saisie au serveur Web. En cas d’erreur détectée par le script, JavaScript permet d’annuler cette action grâce à une variante de l’instruction return. En effet, *cette instruction return permet aussi de contrôler le déroulement des événements standards Html*. Pour un contrôle de saisie dans un formulaire, la fonction de contrôle doit retourner un booléen qui est lui-même retourné par le script associé à l’événement de soumission comme dans l’exemple ci-dessous :

<html>

<head>

<script>

function checkForm(f) {

alert("Contrôle champ " + f.elements['chp'].value);

return false; // n’envoie pas le formulaire

}

</script>

</head>

<body>

<form action= "" id="form1" onSubmit="**return** checkForm(this);">

<input type="text" name="chp" value="essai" />

<input type= "submit" value="Valider" />

</form>

</body>

</html>

Dans l’exemple ci-dessus, JavaScript intercepte l’envoi du formulaire avec l’événement onSubmit. Cet événement appelle la méthode checkForm attend une réponse booléenne pour envoyer ou non le formulaire.

L’argument this est la référence de l’objet en cours, ici le formulaire.

La fonction checkForm retourne toujours false ; le formulaire ne sera jamais envoyé.

On peut également intercepter l’envoi du formulaire avec l’événement onClick :

<head>

<script>

function checkForm2(f) {

alert("Contrôle champ " + f.elements['chp'].value);

f.submit(); // envoie explicitement le formulaire

}

</script>

</head>

<body>

<form action="" id="form1">

<input type="text" name="chp" value="essai" />

<input type= "submit" value="Valider"

onClick="**return** checkForm(this.form)" />

<input type= "button" value="btnValider" onClick="checkForm2(this.form)" />

</form>

</body>

Dans cet exemple, le bouton *submit* appelle la fonction précédente checkForm() en lui passant l’argument this.form.

*L’argument this est la référence de l’objet en cours*, ici le bouton *submit*; this.form désigne le formulaire parent du bouton *submit* car tout élément de formulaire possède une référence à son *form*.

L’instruction return appelée dans l’événement onClick permet de soumettre ou non le formulaire suivant la valeur de retour de la fonction checkForm().

L’événement onClick du second bouton appelle la fonction checkForm2() qui appelle à son tour la méthode JavaScript submit() du formulaire passé en paramètre ; le formulaire sera ici envoyé après l’affichage d’une boite d’alerte.

Un élément de type *button* peut donc simuler la soumission du formulaire quand le script déclenche la méthode submit() du formulaire, comme dans l’exemple ci-dessus. Toutefois, *on préférera contrôler la soumission d’un formulaire en interceptant l’événement onSubmit du formulaire ou l’événement onClick du bouton submit* car ces deux événements sont aussi bien déclenchés par la frappe au clavier de la touche *Entrée*. Cette construction est utile pour améliorer l’ergonomie des applications Web nécessitant des saisies massives (car l’utilisateur ne manipule plus la souris mais utilise les raccourcis clavier *Tab*, *Entrée*, *Espace…*) ; on retrouve aussi couramment cette construction dans les petits formulaires de recherche qui ne contiennent bien souvent pas de bouton d’envoi, la recherche étant déclenchée par la touche *Entrée*.

Pour aller plus loin dans l’exploration des propriétés, événements et méthodes des objets JavaScript, consultez le document complémentaire « **Résumé des objets JavaScript.pdf** » ainsi que la documentation de référence sur



<https://developer.mozilla.org/en-US/docs/Web/API/Document>

## ** Exercice : Le contrôle de champ

Réalisez maintenant l’exercice « **Contrôle de champ** » de manière à contrôler de différentes manières le contenu saisi dans un champs *text* et à filtrer la soumission du formulaire.

# Pour aller plus loin : zoom sur des objets natifs

## Objets window, history, location, screen

Les objets sont des composants essentiels de JavaScript, langage de programmation orienté objet. Ils sont dotés de propriétés (variables) et de méthodes (fonctions).

Le langage JavaScript côté client Web créé automatiquement et met à disposition du développeur un certain nombre d’objets globaux en plus de ceux correspondant aux éléments Html de la page Web.

L’objet global **window** *représente l’instance du navigateur*.

Parmi les fonctionnalités natives, on a déjà vu dans nos exemples des fonctions qui permettent d’interagir avec l’utilisateur : **alert()**, **confirm()**et **prompt()**. Ce sont en fait des **méthodes** de l’objet **window**.

De même, toute variable dite *globale* sera en réalité une **propriété** de l’objet **window**.

Si aucun nom d’objet n’est spécifié, c’est l’objet **window** qui est implicite.

Certains objets sont très simples d’utilisation comme l’objet **Math** qui fournit des propriétés et des méthodes de calculs. Exemple :

var aireCercle = 2 \* rayon \* Math.PI;

var lanceDeDe = Math.ceil(Math.random()\*6); // lance un dé à 6 faces

L’objet **History** permet dans récupérer l’historique des pages visitées par l’utilisateur et de naviguer parmi ces pages avec ses méthodes back(), forward() et go().

history.back(); // revient à la page précédemment visitée

history.go(-1); // revient aussi à la page précédemment visitée

L’objet **Location** fournit des informations sur l’url de la page en cours.

location.href // obtient ou modifie l’url complète de la page en cours

location.reload(); // recharge la page courante

location.replace(url); /\* charge une nouvelle page définie par l’url  
 sans alimenter l’historique de navigation \*/

L’objet **Screen** fournit les informations sur l’écran avec ses propriétés availHeight, availWidth, colorDepth, height et width. Cet objet permet d’adapter les affichages par JavaScript en fonction des caractéristiques de l’écran tout comme on le fait couramment maintenant avec CSS et les techniques de Responsive Design.

## ** Exercice : Le passage d’information

Réalisez l’exercice « **le passage d’information** » de manière à exprimer par JavaScript une requête HTTP contenant une URL de type *get* avec des paramètres.

## L’objet Navigator

On préférera toujours utiliser les instructions JavaScript standards mais certaines applications (surtout pour un intranet homogène) peuvent nécessiter des fonctionnalités particulières. Vous avez aussi sans doute déjà rencontré une page ou un site Web qui annonce poliment que votre navigateur est trop ancien, ou pas conforme aux exigences du site, et qui vous propose un lien pour télécharger un autre navigateur.

Ce type de contrôle peut être l’affaire de JavaScript. En effet, JavaScript met à disposition du développeur un objet qui représente le navigateur de l’utilisateur. C’est par cet objet que le développeur peut savoir le type de browser utilisé de manière à adapter le code JavaScript aux fonctionnalités spécifiques du navigateur lorsque cela est nécessaire.

L’objet JavaScript navigator reste assez simple mais son exploitation est parfois périlleuse. Pourtant il peut renseigner à la fois sur le navigateur qui joue la page mais aussi sur le système d’exploitation du poste utilisateur. Le problème est que tout cela est un peu ‘en vrac’ au sein de 4 propriétés.

Au-delà de l’exploitation de ces propriétés de l’objet navigator, il existe de nombreux ‘trucs et astuces’ permettant de détecter le type de navigateur par l’intermédiaire de tests des fonctionnalités supportées. En effet, si un interpréteur JavaScript enrichit le modèle objet standard en ajoutant une propriété truc à l’objet document, la simple instruction if(document.truc) répondra true quand le navigateur supporte cette propriété, et false sinon.

Chaque éditeur de navigateur propose des fonctionnalités spécifiques pour chaque version de logiciel publié ; les sites techniques et forums spécialisés regorgent donc de ‘trucs et astuces’ actualisés pour réaliser ces tests.

## **Exercice : L’objet navigator

Réalisez l’exercice « l’objet navigator » de manière à explorer les 4 propriétés de l’objet JavaScript navigator puis d’explorer comment tester le support de fonctionnalités spécifiques.

## L’objet Date

*Un objet Date permet de représenter une date calendaire et d’effectuer des calculs entre dates* comme la détermination d’une date future (pour une échéance par exemple) ou le nombre de jours écoulés entre deux dates (pour un calcul de retard par exemple).

La création d’un objet Date peut se faire de différentes façons :

var myDate = new Date(); // date du jour

var myDate = Date.now(); // date du jour en nombre de millisecondes

var myDate = new Date(millisecondes); // depuis le 1er janvier 1970

var myDate = new Date(annee,mois,jour); // mois compris entre 0 et 11

var myDate = new Date(annee, mois, jour, heure, minute, seconde);

var myDate = new Date(chaîne de caractère représentant une date);

Après avoir créé un objet Date, on peut appeler ses méthodes :

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| getFullYear() | Méthode qui retourne l’année sur 4 chiffres. |
| getMonth() | Méthode qui retourne le mois compris entre 0 et 11 ! |
| getDate() | Méthode qui retourne le jour du mois compris entre 1 et 31. |
| getDay() | Méthode qui retourne le jour de la semaine compris entre 0 (dimanche) et 6. |
| getHours() | Méthode qui retourne les heures comprises entre 0 et 23. |
| getMinutes() | Méthode qui retourne les minutes comprises entre 0 et 59. |
| getSeconds() | Méthode qui retourne les secondes comprises entre 0 et 59. |
| getTime() | Méthode qui retourne l’heure courante sous forme d’un entier représentant le nombre de millisecondes écoulées depuis le 1er janvier 1970 00:00:00. |
| toLocaleString() | Méthode qui retourne la date sous forme de chaîne de caractères. |

L’objet Date possède l’équivalent des méthodes getXXX() en méthodes setXXX(x) pour assigner une nouvelle valeur à une partie de la date. Exemple : myDate.setMonth(10) ;.

## L’objet Array

L'objet natif Array (ou tableau) est une liste d'éléments indexés dans lesquels on pourra ranger des données. En JavaScript, les tableaux n’ont *pas de dimension prédéfinie*, ils restent auto-adaptables à leur contenu, et peuvent contenir des données de types différents.

*L’indexation des éléments commence à 0*.

Il existe plusieurs façons de créer un tableau simple :

var tab = new Array (x); // où x est le nombre d'éléments du tableau.

var tab1 = new Array(); // on ne précise pas la taille du tableau.

tab1[2] = "Philippe"; // le tableau tab1 créé contient 3 éléments.

var tab2 = new Array (10, 5, 4, 20); // tableaux créés et

var tab3 = [10, 5, 4, 20]; // initialisés en 1 seule étape.

Après avoir créé un tableau, on peut appeler ses méthodes :

|  |  |
| --- | --- |
| **Instruction** | **Description** |
| length | Donnée membre qui retourne le nombre d'éléments du tableau. |
| join() join(string) | Méthode qui regroupe tous les éléments du tableau dans une seule chaîne. Les différents éléments sont séparés par un caractère séparateur spécifié en argument (la virgule par défaut). |
| reverse() | Méthode qui inverse l'ordre des éléments (ne les trie pas). |
| sort() | Méthode qui trie les éléments par ordre alphabétique (à condition qu'ils soient de même nature) |
| push() | Méthode qui ajoute un nouvel élément à la fin du tableau. |
| pop() | Méthode qui supprime le dernier élément du tableau. |
| unshift() | Méthode qui ajoute un nouvel élément au début du tableau. |
| shift() | Méthode qui supprime le premier élément du tableau. |
| slice(deb, fin) | Méthode qui retourne la partie du tableau commençant à l’indice *deb* et se terminant à l’indice *fin-1*. |

Avec JavaScript, on peut également créer des *tableaux associatifs* sous forme de ***clé=valeur*** dont la clé n’est plus un indice mais une chaîne de caractères :

tab2['titi'] = "Grominet";

Ces tableaux deviennent des « Objets Littéraux » que l’on peut créer directement ainsi :

var tContacts = {"epaleur":"Thor",

"truand":"Tony",

"abierre":"Kale"};

Cette syntaxe, appelée *notation JSON* est très couramment utilisée en JavaScript moderne.

Les valeurs des tableaux associatifs ou objets littéraux peuvent être récupérées de 2 façons :

console.log(tContacts['truand']);

console.log(tContacts.truand);

La boucle **for** classique est très souvent utilisée pour parcourir un tableau simple :

for (var i = 0, taille = tab2.length; i < taille; i++){

console.log(i + ':' + tab2[i]);

}

Sa variante, la boucle **for..in** permet de récupérer *tous* les éléments, mais on perd la notion d’indice :

for (var elt in tab2) {

console.log(elt + ':' + tab2[elt]);

}

*Pour aller plus loin :*

Les tableaux possèdent une méthode **forEach()** qui appelle une fonction (callback) qui prend en paramètre chaque élément du tableau, son indice et le tableau :

tab2.forEach(function(elt, index, array) {

console.log(index + "/" + array.length + ":" + elt);}

);

# L’opérateur double négation

Dans certains cas, il est intéressant de pouvoir transformer une valeur « *falsey* » en booléen pour tester si une propriété est reconnue par le navigateur ou si un tableau possède tel indice ou telle clé :

Undefined, NaN, null, 0, "" // liste des valeurs « falsey »

Exemple :

var tab2 = new Array (10, 5, 4, 20); // indice de 0 à 3

console.log(tab2[5]); // retourne "undefined"

console.log(!!tab2[5]) ; // retourne "false"

# Les expressions régulières

Les expressions régulières permettent de manipuler les chaînes de caractères de façon très poussée avec un minimum de programmation (mais au prix d’une syntaxe obscure). On retrouve cette technique des expressions régulières dans tous bon nombre de langages modernes (Java, C#, PHP …) avec des variantes minimes d’un langage à l’autre ; l’investissement réalisé pour comprendre cette syntaxe est donc capitalisable pour d’autres langages.

*La syntaxe des expressions régulières est extrêmement complète (et complexe)* et nous nous contenterons ici d'introduire le sujet. *Il existe des livres de plus de 400 pages sur ce sujet...*

Pour JavaScript, les expressions régulières sont souvent utilisées pour la vérification des données saisies dans les formulaires (présence du **@** et absence d'espace dans une adresse email par exemple).

*Une expression régulière est basée sur la définition d’un masque et le filtrage d’une donnée par rapport à ce masque.*

Exemples de masques simples :

"abc" : chaîne contenant la chaîne "abc"

"abc+" : chaîne qui contient "ab" suivie de un ou plusieurs "c" ("abc", "abcc" ...)

"abc\*" : chaîne qui contient "ab" suivie de zéro ou plusieurs "c" ("ab", "abc","abcc"  ...)

Nous aurions pu faire un pari la dessus : et oui, dans JavaScript, les expressions régulières sont aussi représentées par des objets. Ici, il s'agit de l'objet prédéfini **RegExp**.

## L'objet RegExp

Il faut tout d’abord créer un objet RegExp. La syntaxe d’instanciation est la suivante :

var monReg = new RegExp(expression, options);

* **monReg** symbolise un identificateur correct (une chaine, un booléen) ;
* **expression** représente le masque d’expression régulière ;
* **options**, les options de cette expression. Pour l'instant, ça ne nous dit pas grand-chose. Mais ce n'est qu'un début.

À noter qu'une expression régulière peut également être créée de manière littérale, comme par exemple ci-dessous :

var at = /@/;

*Non! Vous ne rêvez pas! La ligne précédente est tout à fait correcte. Il ne s'agit pas d'une chaîne de caractères, vu qu'il n'y a ni guillemets, ni apostrophes, et ce n'est pas non plus une variable : c'est une expression régulière. Mais tout ceci deviendra plus clair dans un instant.*

## Syntaxe des expressions régulières

Tout d'abord, *les options*. Il y en a deux :

* '**g**' qui permet de réaliser une recherche globale, c'est à dire sur l'ensemble de la chaîne de caractères. Par exemple, si l'expression peut s'appliquer à deux endroits dans la chaîne, les deux occurrences seront prises en compte, ce qui n'est pas le cas si g est absent, où seule la première occurrence est prise en compte.

En voici un exemple, avec la méthode replace() de l'objet string pour remplacer tous les 'a' en 'A' :

var monReg = /a/g; // ou bien var monReg = new RegExp("a", "g");

var chaine = "abracadabra";

chaine = chaine.replace(monReg, "A");

console.log(chaine); // Sortie -> "AbrAcAdAbrA"

* '**i**'permet de rendre insensible à la casse l'expression. Ces options sont cumulables. On peut donc écrire :

var monReg = /abr/gi;

var chaine = "aBracadAbRa";

chaine = chaine.replace(monReg, "ABR");

console.log(chaine); // Sortie -> "ABRacadABRa"

Maintenant, voyons d'un peu plus près la syntaxe de ces expressions.

Les *crochets [] permettent de spécifier des alternatives entre plusieurs caractères*. Par exemple, si on veut rechercher les mots "moi" ou "toi", mais pas le mot "roi" :

var monReg = /[tm]oi/i;

with(console){

log(monReg.test("moi"));// true

log(monReg.test("toi"));// true

log(monReg.test("roi"));// false

}

Au lieu d'énumérer les lettres, on peut préférer *spécifier un intervalle à l’aide d’un tiret "****-****"* :

var monReg = /[a-m]ou/i; // entre 'a' et 'm' suivi de 'ou'

with(console){

log(monReg.test("cou"));// true

log(monReg.test("pou"));// false

}

On peut aussi *spécifier des alternatives, séparées par le caractère "****|****" (pipe)* :

var monReg = /fr|com/; // 'fr' ou 'com'

with(console){

log(monReg.test("robert@truc.fr"));// true

log(monReg.test("webmaster@cyber.com"));// true

log(monReg.test("marcel@proust.eu"));// false

}

*L'étoile "****\*****" indique que le caractère qui la précède peut intervenir 0 ou plusieurs fois dans la chaîne :*

var monReg = /a\*tchoum/; // 'a' 0 à N fois

var chaine1 = "aaaaaaaaaaaaatchoum!";

var chaine2 = "tchoum";

chaine1 = chaine1.replace(monReg, "atchoum");

chaine2 = chaine2.replace(monReg, "atchoum");

console.log(chaine1);// Sortie -> atchoum!

Console.log(chaine2);// Sortie -> atchoum

*Par contre, si on veut que le caractère 'a' intervienne au moins* ***une fois****, il nous faut substituer l'étoile par un plus "****+****" :*

var monReg = /a+tchoum/;

var chaine1 = "aaaaaaaaaaaaatchoum!";

var chaine2 = "tchoum";

chaine1 = chaine1.replace(monReg, "atchoum");

chaine2 = chaine2.replace(monReg, "atchoum");

console.log(chaine1);// Sortie -> atchoum!

Console.log(chaine2);// Sortie -> tchoum (rien n'est modifié)

On peut aussi utiliser le *point d'interrogation "****?****" pour préciser que le caractère précèdent est optionnel* :

var monReg = /bienvenue?/; // Avec ou sans e ?

with(console){

log(monReg.test("bienvenue"));// true

log(monReg.test("bienvenu"));// true

}

*Si on veut qu'un caractère intervienne un nombre précis de fois, on utilise les accolades* ***{*** *et* ***}****.*

Voici comment détecter des codes de la forme **XXX-XXX-XX.XX**, ou **X** est un nombre :

var monReg = /[0-9]{3}-[0-9]{3}-[0-9]{2}\.[0-9]{2}/;

with(console){

log(monReg.test("123-456-78.90"));// true

log(monReg.test("4567-76-322.1"));// false

}

*On peut aussi spécifier un nombre minimal et un nombre maximal de fois en utilisant les accolades sous la forme* ***{min, max}****.*

*Au lieu de spécifier un unique caractère, on peut spécifier un mot en le plaçant entre parenthèses :*

var monReg = /Raph(ael)?/;

document.write(monReg.test("Raphael")); // true

document.write(monReg.test("Raph")); // true

## Récapitulatif des caractères spéciaux utilisés dans les expressions régulières :

**Caractère Utilité**

[] Les crochets définissent une liste de caractères autorisés.

() Les parenthèses définissent un élément composé de l'expression régulière qu'elle contient.

{} Les accolades indiquent le nombre de fois que l'élément précédant peut se reproduire.

- Le tiret représente un intervalle.

. Le point représente n’importe quel caractère.

\* L’astérisque indique zéro, une ou plusieurs occurrences de l’élément précédant.

+ Le plus indique une ou plusieurs occurrences de l'élément précédant.

? Le point d'interrogation indique la présence éventuelle de l'élément précédant

| Le pipe indique un OU entre l’élément qui le précède et celui qui le suit.

^ Le chapeau, placé en début d'expression, signifie "chaîne commençant par ... "

Utilisé à l'intérieur d'une liste [], il signifie "ne contenant pas les caractères suivants...ex : [^abc]

$ Le dollars, placé en fin d'expression, signifie "chaîne finissant par … "

Pour en savoir plus :   
<http://www.commentcamarche.net/contents/javascript/jsregexp.php3>

## **Exercice : Contrôle d’adresse email

Vous pouvez réaliser les variantes de l’exercice « le contrôle de champ » de manière à vous entraîner à la manipulation des expressions régulières afin de vérifier le bonne forme d’une adresse email saisie dans une zone de texte.

**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)**

Damien Bin- formateur

Benoit Hézard - formateur

Chantal Perrachon – Ingénieure de formation>

**Date de mise à jour** : 27/04/16

**Reproduction interdite**

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l’adaptation ou la reproduction par un art ou un procédé quelconque. »

1. JSP : JavaServer Pages [↑](#footnote-ref-1)
2. ASP : Active Server Pages (C#) [↑](#footnote-ref-2)
3. PHP : Hypertext Preprocessor Language [↑](#footnote-ref-3)
4. ECMA : European Computer Manufacturer’s Association ([www.ecma-international.org](http://www.ecma-international.org)) [↑](#footnote-ref-4)
5. ISO : International Organization for Standardization [↑](#footnote-ref-5)
6. XHTML : Extensible HyperText Markup Language [↑](#footnote-ref-6)
7. MIME : Multipurpose Internet Mail Extension : Il permet de définir un format de données. [↑](#footnote-ref-7)
8. W3C : World Wide Web Consortium [↑](#footnote-ref-8)
9. API : **Application Programmable Interface**, traduisez « interface de programmation » ou « interface pour l'accès programmé aux applications » ou ensemble de fonctions permettant d’accéder aux services d’une application. [↑](#footnote-ref-9)
10. W3C : World Wide Web Consortium. Organisme pour standardiser les différentes technologies du Web. [↑](#footnote-ref-10)