

Sommaire

Sommaire	1
Informations sur le tutoriel	3
Dynamisez vos sites web avec Javascript !	5
Informations sur le tutoriel	5
Partie 1 : Les bases du Javascript	6
Introduction au Javascript	6
Qu'est-ce que le Javascript ?	6
Définition	7
Javascript, le langage de scripts	8
Javascript, pas que le Web	8
Petit historique du langage	10
Premiers pas en Javascript	11
Afficher une boîte de dialogue	12
Le Hello World!	12
Les nouveautés	12
La boîte de dialogue alert	13
La syntaxe Javascript	14
Les instructions	14
Les espaces	14
Les commentaires	16
Les fonctions	17
Où placer le code dans la page	18
Le Javascript "dans la page"	18
Le Javascript externe	19
Positionner l'élément <script>	19
Quelques aides	20
Les variables	22
Qu'est-ce qu'une variable ?	22
Déclarer une variable	23
Les types de variables	24
Tester l'existence de variables avec "typeof"	25
Les opérateurs arithmétiques	26
Quelques calculs simples	26
Simplifier encore plus vos calculs	27
Initiation à la concaténation et à la conversion des types	28
La concaténation	28
Interagir avec l'utilisateur	28
Convertir une chaîne de caractères en nombre	29
Convertir un nombre en chaîne de caractères	30
Les conditions	31
La base de toute condition : les booléens	32
Les opérateurs de comparaison	32
Les opérateurs logiques	33
Combiner les opérateurs	34
La condition "if - else"	35
La structure "if" pour dire "si"	36
Petit intermède : la fonction confirm	37
La structure "else" pour dire "sinon"	37
La structure "else if" pour dire "sinon si"	38
La condition "switch"	39
Les ternaires	41
Les conditions sur les variables	43
Tester l'existence de contenu d'une variable	44
Le cas de l'opérateur OU	44
Un petit exercice pour la forme !	45
Présentation de l'exercice	45
Correction	45
Les boucles	46
L'incréméntation	47
Le fonctionnement	47
L'ordre des opérateurs	47
La boucle while	49
Répéter tant que...	49
Exemple pratique	50
Quelques améliorations	50
La boucle do while	51

La boucle for	52
for, la boucle pour l'incrémentation	53
Reprenons notre exemple	53
Les fonctions	55
Concevoir des fonctions	56
Créer sa première fonction	56
Un exemple concret	57
La portée des variables	58
La portée des variables	59
Les variables globales	59
Les variables globales ? Avec modération !	60
Les arguments et les valeurs de retour	62
Les arguments	62
Les valeurs de retour	66
Les fonctions anonymes	68
Les fonctions anonymes : les bases	68
Retour sur l'utilisation des points-virgules	68
Les fonctions anonymes : Isoler son code	69
Les objets et les tableaux	72
Introduction aux objets	73
Que contiennent les objets ?	73
Exemple d'utilisation	74
Objets natifs déjà rencontrés	74
Les tableaux	75
Les indices	76
Déclarer un tableau	76
Récupérer et modifier des valeurs	77
Opérations sur les tableaux	77
Ajouter et supprimer des items	78
Chaînes de caractères et tableaux	78
Parcourir un tableau	79
Parcourir avec for	80
Attention à la condition	80
Les objets littéraux	81
La syntaxe d'un objet	82
Accès aux items	83
Ajouter des items	83
Parcourir un objet avec for in	83
Utilisation des objets littéraux	84
Exercice récapitulatif	85
Énoncé	85
Correction	85
TP : Convertir un nombre en toutes lettres	86
Présentation de l'exercice	86
La marche à suivre	87
L'orthographe des nombres	87
Tester et convertir les nombres	87
Il est temps de se lancer !	89
Correction	90
Le corrigé complet	90
Les explications	91
Conclusion	98
Partie 2 : Modeler vos pages web	98
Manipuler le code HTML (Partie 1/2)	99
Le Document Object Model	100
Petit historique	100
L'objet window	100
Le document	101
Naviguer dans le document	102
La structure DOM	102
Accéder aux éléments	103
L'héritage des propriétés et des méthodes	104
Editer les éléments HTML	105
Les attributs	106
Le contenu : innerHTML	108
innerText et textContent	109
innerText	110
textContent	110
Manipuler le code HTML (Partie 2/2)	112
Naviguer entre les nœuds	113
La propriété parentNode	113
nodeType et nodeName	113
Lister et parcourir des nœuds enfants	114
Attention aux nœuds vides	117

Créer et insérer des éléments	119
Ajouter des éléments HTML	119
Ajouter des nœuds textuels	121
Notions sur les références	124
Les références	125
Cloner, remplacer, supprimer...	126
Cloner un élément	127
Remplacer un élément par un autre	127
Supprimer un élément	128
Autres actions	128
Vérifier la présence d'éléments enfants	129
Insérer à la bonne place : insertBefore()	129
Une bonne astuce : insertAfter()	129
Minis-TD : recréer une structure DOM	130
Premier exercice	131
Deuxième exercice	133
Troisième exercice	135
Quatrième exercice	137
Conclusion des TD	139
Les événements	139
Que sont les événements ?	141
La théorie	141
La pratique	142
Les événements au travers du DOM	145
Le DOM-0	145
Le DOM-2	145
Les phases de capture et de bouillonnement	148
L'objet Event	151
Généralités sur l'objet Event	151
Les fonctionnalités de l'objet Event	151
Déclencher soi-même les événements	158
La procédure standard	158
La procédure selon Internet Explorer < 9	161
Les formulaires	162
Les attributs	162
Un attribut classique : value	163
Les booléens avec : disabled, checked et readonly	163
Les listes déroulantes avec : selectedIndex et options	164
Les méthodes et un retour sur quelques événements	166
Les méthodes spécifiques à l'élément <form>	166
La gestion du focus et de la sélection	166
Explications sur l'événement change	167
Manipuler le CSS	167
Éditer les propriétés CSS	168
Quelques rappels sur le CSS	169
Éditer les styles CSS d'un élément	169
Récupérer les propriétés CSS	171
La fonction getComputedStyle()	172
Les attributs de type "offset"	173
Votre premier script interactif !	177
Présentation de l'exercice	178
Correction	179
TP : Un formulaire interactif	182
Présentation de l'exercice	183
Correction	185
Le corrigé au grand complet : HTML, CSS et Javascript	185
Les explications	191
Partie 3 : Les objets	196
Les objets	197
Petite problématique	198
Objet constructeur	198
Ajouter des méthodes	202
Ajouter une méthode	202
Ajouter des méthodes aux objets natifs	204
Ajout de méthodes	204
Remplacer des méthodes	205
Limitations	205
Les namespaces	207
Définir un namespace	207
Un style de code	207
L'emploi de this	208
Vérifier l'unicité du namespace	209
Le cas des constructeurs	209
Les chaînes de caractères	211

Les types primitifs	211
L'objet String	213
Propriétés	214
Méthodes	214
La casse et les caractères	215
toLowerCase() et toUpperCase()	216
Accéder aux caractères	216
Supprimer les espaces blancs avec trim()	217
Rechercher, couper et extraire	219
Connaître la position avec indexOf() et lastIndexOf()	219
Extraire une chaîne avec substring(), substr() et slice()	219
Couper une chaîne en un tableau avec split()	220
Tester l'existence d'une chaîne de caractères	221
Les expressions régulières [Partie 1]	222
Les regex en Javascript	222
Utilisation	223
Recherches de mots	224
Début et fin de chaîne	226
Les caractères et leurs classes	226
Les intervalles de caractères	227
Trouver un caractère quelconque	227
Les quantificateurs	228
Les 3 symboles quantificateurs	228
Les accolades	228
Les métacaractères	230
Les métacaractères au sein des classes	230
Types génériques et assertions	232
Les types génériques	232
Les assertions	232
Les expressions régulières [Partie 2]	232
Construire une regex	234
L'objet RegExp	236
Méthodes	236
Propriétés	236
Les parenthèses capturantes	236
Les recherches non-greedy	238
Rechercher et remplacer	241
L'option g	241
Rechercher et capturer	242
Utiliser une fonction pour le remplacement	243
Autres recherches	245
Rechercher la position d'une occurrence	245
Récupérer toutes les occurrences	245
Couper avec une regex	245
Les données numériques	246
L'objet Number	247
L'objet Math	249
Les attributs	249
Les méthodes	249
Les inclassables	253
Les fonctions de conversion	253
Les fonctions de contrôle	253
La gestion du temps	255
Le système de datation	255
Introduction aux systèmes de datation	256
L'objet Date	256
Mise en pratique : Calculer le temps d'exécution d'un script	258
Les fonctions temporelles	258
Utiliser setTimeout() et setInterval()	259
Annuler une action temporelle	260
Mise en pratique : Les animations !	261
Partie 4 : Annexe	262
Déboguer votre code	263
Le débogage : qu'est-ce que c'est ?	264
Les consoles d'erreurs	264
Mozilla Firefox	265
Internet Explorer	265
Opera	266
Google Chrome	266
Safari	267
Les bugs les plus courants	268
Les kits de développement	270

Dynamisez vos sites web avec Javascript !

Bienvenue à tous,

Vous voici sur la page d'accueil du cours traitant du langage web **Javascript** ! Au cours de la lecture de ce cours vous apprendrez comment dynamiser vos pages web et les rendre beaucoup plus attrayantes auprès de vos visiteurs. Ce cours traitera de nombreux sujets, en partant des bases vous apprendrez à réaliser des animations, des applications complexes et à utiliser ce langage conjointement avec l'HTML 5, la nouvelle version du fameux langage de balisage du W3C !

Ce cours va principalement aborder l'usage du Javascript conjointement avec l'HTML, il est donc de rigueur que vous sachiez coder à la fois en HTML et en CSS. Le PHP peut-être un plus mais vous n'en aurez réellement besoin que lorsque nous aborderons la partie [Ajax](#) qui traite des communications entre le Javascript et un serveur.

Afin d'approfondir vos connaissances vous découvrirez aussi l'utilisation d'un framework Javascript, un genre de boîte à outils vous permettant de coder plus rapidement et plus efficacement. Le framework abordé au cours de ce tuto sera [le fameux Mootools](#) !

Dès le début de votre apprentissage, vous serez capable de faire des interactions basiques avec l'utilisateur (image de gauche), et une fois votre apprentissage complété vous pourrez même créer des applications complètes (image de droite) !




Voilà, nous espérons vous avoir convaincu de vous lancer dans l'apprentissage de ce fabuleux langage qu'est le Javascript !

Sur ce, bonne lecture !

Informations sur le tutoriel

Auteurs : [Nesquik69](#) et [Thunderseb](#)

Difficulté : 

Licence :



Partie 1 : Les bases du Javascript

Comme tout langage de programmation, le Javascript possède quelques particularités qui lui sont propres : sa syntaxe, les fonctions anonymes, son modèle d'objet, etc... En clair, tout ce qui fait la particularité d'un langage. D'ailleurs, vous découvrirez rapidement que le Javascript est un langage assez hors-pair car il a des particularités assez spéciales. Bref, cette partie est indispensable pour tout débutant en programmation et même pour ceux qui connaissent déjà un langage de programmation car les différences avec les autres langages sont nombreuses.

Introduction au Javascript

Avant de commencer directement dans le vif du sujet, ce chapitre va vous apprendre ce qu'est le Javascript, ce qu'il permet de faire, et quand il peut ou doit être utilisé. Et comme l'histoire intéresse quand même pas mal de monde, vous avez même le droit à une partie consacrée à l'histoire de ce langage !

Qu'est-ce que le Javascript ?

Définition

Citation : Définition

Javascript est un langage de programmation de scripts orienté objet.

Dans cette définition un peu barbare se trouvent plusieurs éléments que nous allons décortiquer.

Un langage de programmation

Tout d'abord, un *langage de programmation* est un langage qui permet aux développeurs d'écrire du *code source* qui sera analysé par l'ordinateur.

Un *développeur*, ou un programmeur, est une personne qui développe des programmes. Ça peut être un professionnel (un ingénieur, un informaticien ou un analyste programmeur) ou bien un amateur.

Le *code source* est écrit par le développeur. C'est un ensemble d'actions, appelée *instructions*, qui vont permettre de donner des ordres à l'ordinateur et ainsi de faire fonctionner le programme. Le code source est quelque chose de caché, un peu comme un moteur dans une voiture : le moteur est caché, mais il est bien là, et c'est lui qui fait en sorte que la voiture puisse être propulsée. Dans le cas d'un programme, c'est pareil, c'est le code source qui régit le fonctionnement du programme.

En fonction du code source, l'ordinateur exécute différentes actions, comme ouvrir un menu, démarrer une application, effectuer une recherche, enfin bref, tout ce que l'ordinateur est capable de faire. Il existe énormément de langages de programmation, [la plupart étant listés sur cette page](#).

Programmer des scripts

Javascript permet de programmer des *scripts*. Comme dit plus haut, un langage de programmation permet d'écrire du code source qui sera analysé par l'ordinateur. Il existe 3 manières d'utiliser du code source.

Langage compilé

Le code source est donné à un programme appelé compilateur qui va lire le code source et le convertir dans un langage que l'ordinateur sera capable d'interpréter : c'est le langage binaire, fait de 0 et de 1. Les langages comme le C ou le C++ sont des langages dits compilés.

Langage pré-compilé

Ici, le code source est compilé partiellement, généralement dans un code plus simple à lire pour l'ordinateur, mais qui n'est pas encore du binaire. Ce code intermédiaire devra être lu par ce que l'on appelle une machine virtuelle, qui exécutera ce code. Les langages comme le C# ou le Java sont dits pré-compilés.

Langage interprété

Dans ce cas, il n'y a pas de compilation. Le code source reste tel quel, et si on veut exécuter ce code, on doit le fournir à un interpréteur qui se chargera de le lire et de réaliser les actions demandées.

Les scripts sont majoritairement interprétés. Ainsi donc, quand on dit que Javascript est un langage de script, cela signifie qu'il s'agit d'un langage interprété ! Il est donc nécessaire de posséder un interpréteur pour faire fonctionner du code Javascript, et un interpréteur, vous en utilisez un fréquemment : il est inclus dans votre navigateur Web !

Chaque navigateur possède un interpréteur Javascript, qui diffère selon le navigateur. Si vous utilisez Internet Explorer, son interpréteur Javascript s'appelle *JScript* (l'interpréteur de la version 9 s'appelle *Chakra*), alors que celui de Mozilla Firefox se nomme *SpiderMonkey* tandis que celui de Google Chrome est *V8*.

Langage orienté objet

Il reste un dernier fragment à analyser : *orienté objet*. Ce concept est assez compliqué à définir maintenant et sera approfondi par la suite notamment au niveau de la partie II. Sachez toutefois qu'un langage de programmation orienté objet est un langage qui contient des éléments, appelés *objets*, et que ces différents objets possèdent des caractéristiques spécifiques ainsi que des manières différentes de les utiliser. Le langage fournit des objets de base comme les images, les dates, les chaînes de caractères... mais il est également possible de créer nous-même des objets pour nous faciliter la vie et obtenir un code source plus clair (facile à lire) et une manière de programmer beaucoup plus intuitive (logique).

Il est bien probable que vous n'ayez rien compris à ce passage si vous n'avez jamais fait de programmation, mais ne vous en faites pas : vous comprendrez bien assez vite comment tout cela fonctionne 😊.

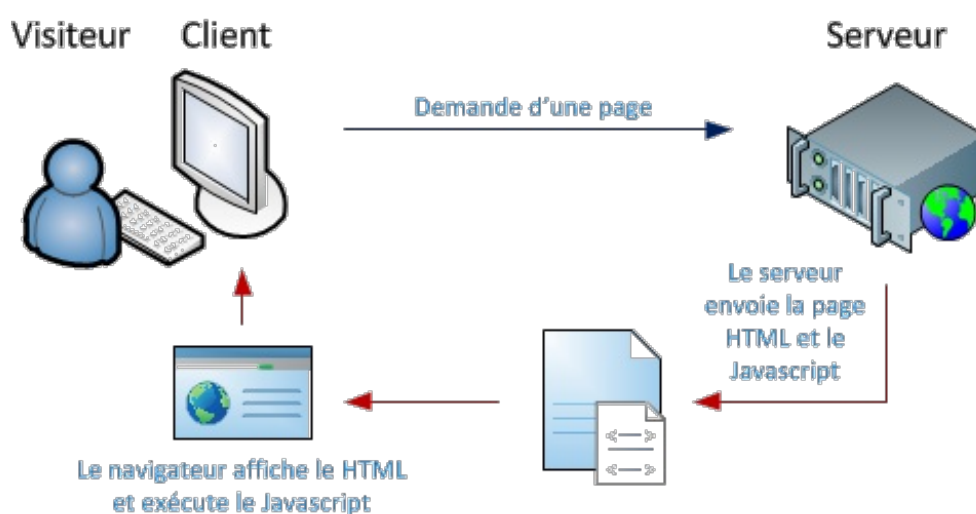
Javascript, le langage de scripts

Le Javascript est à ce jour utilisé majoritairement sur Internet, conjointement aux pages Web (HTML ou XHTML). Le Javascript s'inclut directement dans la page Web (ou dans un fichier externe) et permet de *dynamiser* une page HTML, en ajoutant des interactions avec l'utilisateur, des animations, de l'aide à la navigation, comme par exemple :

- afficher/masquer du texte
- faire défiler des images
- créer un diaporama avec un aperçu "en grand" des images
- créer des infobulles
- ...

Javascript est un langage dit *client-side*, c'est-à-dire que les scripts sont exécutés par le navigateur chez l'internaute (le *client*). Cela diffère des langages de scripts dits *server-side* qui sont exécutés par le serveur Web. C'est le cas des langages tel que [PHP](#).

C'est important, car la finalité des scripts client-side et server-side n'est pas la même. Un script server-side va s'occuper de "créer" la page Web qui sera envoyée au navigateur. Ce dernier va alors afficher la page, et ensuite, exécuter les scripts client-side tels que Javascript. Voici un schéma reprenant ce fonctionnement :



Javascript, pas que le Web

Si Javascript a été conçu pour être utilisé conjointement avec le HTML, le langage a depuis évolué vers d'autres destinées. Javascript est régulièrement utilisé pour réaliser des extensions pour différents programmes, un peu comme les scripts codés en

Lua ou en [Python](#).

Javascript peut aussi être utilisé pour réaliser des applications. Mozilla Firefox est l'exemple le plus connu : l'interface du navigateur est créée avec une sorte de HTML appelé XUL et c'est Javascript qui est utilisé pour "motoriser" l'interface. D'autres logiciels reposent aussi sur cette technologie, comme TomTom HOME qui sert à gérer votre GPS TomTom via votre PC.



A gauche, le navigateur Firefox 4 et à droite, le gestionnaire TomTom HOME

Petit historique du langage

En 1995, Brendan Eich travaille chez Netscape Communication Corporation, la société qui éditait le célèbre navigateur Netscape Navigator, alors principal concurrent d'Internet Explorer.

Brendan développe LiveScript, un langage de script qui s'inspire du langage Java, et qui est destiné à être installé sur les serveurs développés par Netscape. Netscape se met à développer une version client de LiveScript, qui sera renommée JavaScript en hommage au langage Java créée par la société Sun Microsystems. En effet, à cette époque, le langage Java était de plus en plus populaire, et appeler LiveScript JavaScript était une manière de faire de la publicité, et à Java, et à JavaScript lui-même. Mais attention, au final, ces deux langages sont radicalement différents ! N'allez pas confondre Java et Javascript car ces deux langages n'ont clairement pas le même fonctionnement.



Brendan Eich



La graphie de base est *JavaScript*, avec un *S* majuscule. Il est cependant courant de lire *Javascript*, comme ce sera le cas dans ce tutoriel.

Javascript sort en décembre 1995 et est embarqué dans le navigateur Netscape 2. De par la popularité du langage, Javascript est un succès, aussi bien que Microsoft développe une version semblable, appelée JScript, qu'il embarque dans Internet Explorer 3, en 1996.

Netscape décide d'envoyer sa version de Javascript à l'Ecma International pour que le langage soit standardisé, c'est-à-dire pour qu'une référence du langage soit créée et que le langage puisse ainsi être utilisé par d'autres personnes et embarqué dans d'autres logiciels. L'ECMA standardise le langage sous le nom d'*ECMAScript*.

Depuis, les versions de l'ECMAScript ont évolué. La version la plus connue et mondialement utilisée est la version ECMAScript 1.5, parue en 1999.

L'ECMAScript et ses dérivés

L'ECMAScript est la référence de base. De cette référence découlent des *implémentations*. On peut évidemment citer Javascript qui est implémenté dans la plupart des navigateurs, mais encore :

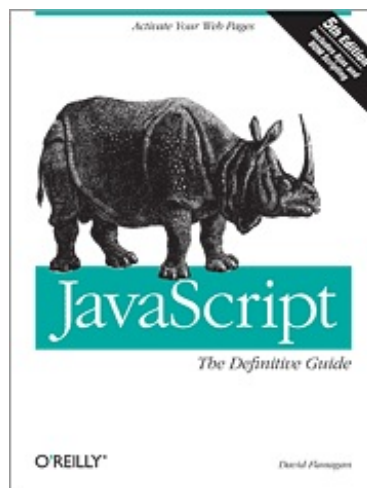
- **JScript** : qui est l'implémentation embarquée dans Internet Explorer. C'est aussi le nom de l'interpréteur d'Internet Explorer ;
- **JScript.NET** : qui est embarqué dans le framework .NET de Microsoft ;
- **ActionScript** : qui est l'implémentation faite par Adobe au sein de Flash ;
- **EX4** : qui est l'implémentation de la gestion du XML d'ECMAScript au sein de SpiderMonkey, l'interpréteur Javascript de Firefox.

Les versions de Javascript

Javascript 1.5 est pris en charge par la totalité des navigateurs actuels. Les versions 1.6 et 1.7 existent et apportent de nouvelles fonctionnalités, mais les navigateurs tardent à implémenter ces nouveautés. Seul Firefox tire son épingle du jeu et implémente correctement les dernières versions.

Ce cours porte essentiellement sur Javascript 1.5. Quand des fonctionnalités plus récentes seront évoquées, il en sera fait mention, pour éviter les incompatibilités.

Un logo bien étrange



Il n'a pas de logo officiel pour représenter le Javascript. L'éditeur O'Reilly a publié un livre intitulé **JavaScript: The Definitive Guide** et qui est utilisé comme référence par la plupart des développeurs Javascript. Sur ce livre figure un rhinocéros qui est devenu l'emblème du Javascript.

Premiers pas en Javascript

Comme spécifié dans l'introduction de ce cours, le Javascript est un langage essentiellement utilisé avec le HTML, vous allez donc apprendre dans ce chapitre comment intégrer ce langage à vos pages web, découvrir sa syntaxe de base et afficher un message sur l'écran de l'utilisateur.

Afin de ne pas vous laisser dans le vague, vous découvrirez aussi à la fin de ce chapitre quelques liens qui pourront probablement vous être utiles au cours de la lecture de ce cours.

Concernant l'éditeur de texte à utiliser (dans lequel vous allez écrire vos codes Javascript), celui que vous avez l'habitude d'utiliser avec le HTML supporte très probablement le Javascript aussi. Dans le cas contraire, nous vous conseillons [Notepad++](#) pour Windows.

Afficher une boîte de dialogue

Le Hello World!

Ne dérogeons pas à la règle traditionnelle qui veut que tous les tutoriels de programmation commencent par afficher le texte **Hello World!** (*Bonjour le monde !*) à l'utilisateur. Voici un code HTML simple contenant une instruction Javascript, placée au sein d'un élément `<script>` :

Code : HTML - Hello World!

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>

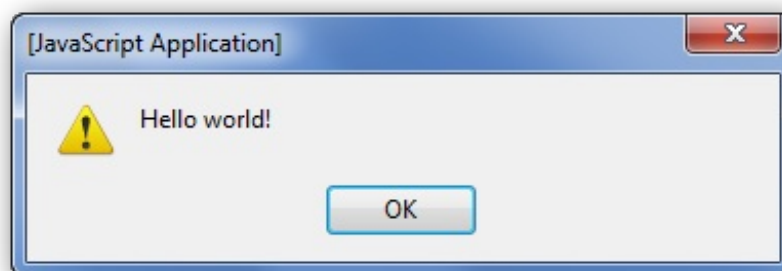
  <body>

    <script type="text/javascript">
      alert('Hello world!');
    </script>

  </body>
</html>
```

Essayer !

Écrivez ce code dans un fichier HTML, et ouvrez ce dernier avec notre navigateur habituel. Une boîte de dialogue s'ouvre, vous présentant le texte *Hello World!* :



Il s'agit ici de Firefox. L'apparence de la boîte de dialogue varie en fonction du navigateur utilisé

Vous remarquerez que ci-dessus je vous ai fourni un lien nommé "**Essayer !**" afin que vous puissiez tester le code. Vous constaterez rapidement que ce ne sera pas toujours le cas car mettre en ligne tous les codes n'est pas forcément nécessaire surtout quand il s'agit d'afficher une simple phrase comme ci-dessus.



Bref, nous, les auteurs, avons décidé de vous fournir des liens d'exemple quand le code nécessitera une interaction de la part de l'utilisateur. Ainsi, les codes avec, par exemple, un simple calcul ne demandant aucune action de la part de l'utilisateur ne seront pas mis en ligne. En revanche, à défaut de mettre certains codes en ligne, le résultat de chaque code sera toujours écrit dans les commentaires.

Les nouveautés

Dans le code HTML donné précédemment, on remarque quelques nouveautés.

Tout d'abord, un élément `<script>` est présent : c'est lui qui contient du code Javascript que voici :

Code : JavaScript

```
alert('Hello world!');
```

Il s'agit d'une instruction, c'est-à-dire une commande, un ordre, ou plutôt une action que l'ordinateur va devoir réaliser. Les langages de programmation sont constitués d'une suite d'instructions qui, mises bout à bout, permettent d'obtenir un programme ou un script complet.

Dans l'exemple ci-dessus, il n'y a qu'une instruction : l'appel de la fonction **alert()**.

La boîte de dialogue **alert**

alert est une instruction simple, appelée une *fonction*, qui permet d'afficher une boîte de dialogue contenant un message. Ce message est placé entre apostrophes, elles-mêmes placées entre les parenthèses de la fonction **alert**.



Ne vous en faites pas pour le vocabulaire. Cette notion de fonction sera vue en détail par la suite. Pour l'instant, reprenez que l'instruction **alert** sert juste à afficher une boîte de dialogue.

La syntaxe Javascript

La syntaxe du Javascript n'est pas compliquée.

Les instructions

De manière générale, les instructions doivent être séparées par un point-virgule que l'on place à la fin de chaque instruction :

Code : JavaScript

```
instruction_1;  
instruction_2;  
instruction_3;
```

En réalité le point-virgule n'est pas obligatoire si l'instruction qui suit se trouve sur la ligne suivante, comme ci-dessus. En revanche, si vous écrivez plusieurs instructions sur une même ligne, comme ci-dessous, le point-virgule est obligatoire. Si le point-virgule n'est pas mis, l'interpréteur ne va pas comprendre qu'il s'agit d'une autre instruction et risque de retourner une erreur.

Code : JavaScript

```
Instruction_1; Instruction_2  
Instruction_3
```



Mais attention ! Ne pas mettre les points-virgules est considéré comme une **mauvaise pratique**, c'est quelque chose que les développeurs Javascript évitent de faire, même si le langage le permet. Ainsi, dans ce tutoriel, toutes les instructions seront terminées par un point-virgule.

La compression des scripts

Certains scripts sont disponibles sous une forme dite compressée, c'est-à-dire que tout le code est écrit à la suite, sans retours à la ligne. Cela permet d'alléger considérablement le poids d'un script et ainsi de faire en sorte que la page soit chargée plus rapidement. Des programmes existent pour "compresser" un code Javascript. Mais si vous avez oublié un seul point-virgule, votre code compressé ne fonctionnera plus, puisque les instructions ne seront pas correctement séparées.

C'est aussi une des raisons qui fait qu'il faut TOUJOURS mettre les points-virgules en fin d'instruction.

Les espaces

Javascript n'est pas sensible aux espaces. Cela veut dire que vous pouvez aligner des instructions comme vous le voulez, sans que cela ne gêne en rien l'exécution du script. Par exemple, ceci est correct :

Code : JavaScript

```
instruction_1;  
    instruction_1_1;  
    instruction_1_2;  
instruction_2;      instruction_3;
```

Indentation et présentation

L'indentation, en informatique, est une façon de structurer du code pour le rendre plus lisible. Les informations sont hiérarchisées en plusieurs niveaux et on utilise des espaces ou des tabulations pour décaler les instructions vers la droite et ainsi créer une hiérarchie. Voici un exemple de code *indenté* :

Code : JavaScript

```
function toggle(elemID) {
    var elem = document.getElementById(elemID);

    if (elem.style.display == 'block') {
        elem.style.display = 'none';
    } else {
        elem.style.display = 'block';
    }
}
```

Le code ci-dessus est indenté en 4 espaces, c'est-à-dire que le décalage est chaque fois un multiple de 4. Un décalage de 4 espaces est courant, tout comme un décalage de 2. Il est possible d'utiliser des tabulations pour indenter du code. Les tabulations présentent l'avantage d'être affichées différemment suivant l'éditeur utilisé, et de cette façon, si vous donnez votre code à quelqu'un, l'indentation qu'il verra dépendra de son éditeur et il ne sera pas "perturbé" par une indentation qu'il n'apprécie pas (par exemple, je n'aime pas les indentations de 2, je préfère celles de 4).

Voici le même code que ci-dessus, mais non-indenté, pour vous montrer que l'indentation est une aide à la lecture :

Code : JavaScript

```
function toggle(elemID) {
var elem = document.getElementById(elemID);

if (elem.style.display == 'block') {
elem.style.display = 'none';
} else {
elem.style.display = 'block';
}
}
```

La présentation des codes est importante aussi, un peu comme si vous rédigez une lettre : ça ne se fait pas n'importe comment. Il n'y a pas de règles prédéfinies comme pour l'écriture des lettres, donc il faudra vous arranger pour organiser votre code de façon claire. Dans le code indenté donné ci-dessus, vous pouvez voir qu'il y a des espaces un peu partout pour aérer le code, qu'il y a une seule instruction par ligne (à l'exception des **if else**, mais nous verrons cela dans deux chapitres). Certains développeurs écrivent leur code comme ça :

Code : JavaScript

```
function toggle(elemID){
    var elem=document.getElementById(elemID);
    if(elem.style.display=='block'){
        elem.style.display='none';
    }else{elem.style.display='block';}
}
```

Vous conviendrez comme moi que c'est tout de suite moins lisible non ? Gardez à l'esprit que votre code doit être propre, même si vous êtes le seul à y toucher : vous pouvez laisser le code de côté quelques temps et le reprendre par la suite, et là, bon amusement pour vous y retrouver.

Les commentaires

Les commentaires sont des annotations faites par le développeur pour expliquer le fonctionnement d'un script, d'une instruction ou même d'un groupe d'instructions. Les commentaires ne gênent pas l'exécution d'un script.

Il existe deux types de commentaires : les commentaires de fin de ligne, et les commentaires multilignes.

Commentaires de fin de ligne

Ils servent à commenter une instruction. Un tel commentaire commence par deux slashes, suivis du commentaire :

Code : JavaScript

```
instruction_1; // Ceci est ma première instruction
instruction_2;
// La troisième instruction ci-dessous :
instruction_3;
```

Le texte placé dans un commentaire est ignoré lors de l'exécution du script, ce qui veut dire que vous pouvez mettre ce que bon vous semble en commentaire, même une instruction (qui ne sera évidemment pas exécutée) :

Code : JavaScript

```
instruction_1; // Ceci est ma première instruction
instruction_2;
// La troisième instruction ci-dessous pose problème, je l'annule
temporairement
// instruction_3;
```

Commentaires multilignes

Ce type de commentaire permet d'écrire beaucoup plus de texte, tout en permettant les retours à la ligne. Un commentaire multilignes commence par /* et se termine par */ :

Code : JavaScript

```
/* Ce script comporte 3 instructions :
- Instruction 1 qui fait telle chose
- Instruction 2 qui fait une telle autre chose
- Instruction 3 qui termine le script
*/
instruction_1;
instruction_2;
instruction_3; // Fin du script
```

Remarquez qu'un commentaire multilignes peut aussi être affiché sur une ligne :

Code : JavaScript

```
instruction_1; /* Ceci est ma première instruction */  
instruction_2;
```

Les fonctions

Dans l'exemple du Hello world, on utilise la fonction `alert`. Nous reviendrons en détail sur le fonctionnement des fonctions, mais pour les chapitres suivants, il sera nécessaire de connaître sommairement leur syntaxe.

Une fonction se compose de deux choses : son nom, suivi d'un couple de parenthèses (une ouvrante, et une fermante) :

Code : JavaScript

```
myFunction(); // "function" veut dire "fonction" en anglais
```

Entre les parenthèses se trouvent les *arguments*, que l'on appelle aussi *paramètres*. Ceux-ci contiennent des valeurs qui sont transmises à la fonction. Dans le cas du Hello world, ce sont les mots `Hello world!` qui sont passés en paramètre :

Code : JavaScript

```
alert('Hello world!');
```

Où placer le code dans la page

Les codes Javascript sont insérés au moyen de l'élément `<script>`. Cet élément possède un attribut **type** qui sert à indiquer le type de langage que l'on va utiliser. Dans notre cas, il s'agit de Javascript, mais ça pourrait être autre chose, comme du **VBScript**, bien que ce soit extrêmement rare.



En HTML 4 et XHTML 1.x, l'attribut **type** est obligatoire. En revanche, en HTML 5, il ne l'est pas. Cet attribut sera toutefois toujours mentionné au fil de ce cours, bien que les exemples soient donnés en HTML 5.

L'attribut **type** prend comme valeur **text/javascript**, qui est en fait le type MIME d'un code Javascript.



Le type **MIME** est un identifiant qui décrit un format de données. Ici, dans **text/javascript**, il s'agit de données textuelles et c'est du Javascript.

Le Javascript "dans la page"

Pour placer du code Javascript directement dans votre page Web, rien de plus simple, on fait comme dans l'exemple du Hello world : on place le code au sein de l'élément `<script>` :

Code : HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <script type="text/javascript">
      alert('Hello world!');
    </script>
  </body>
</html>
```

L'encadrement des caractères réservés

Si vous utilisez les normes HTML 4.01 et XHTML 1.x, il est souvent nécessaire d'utiliser des *commentaires d'encadrement* pour que votre page soit conforme à ces normes. Si par contre, comme dans ce cours, vous utilisez la norme HTML 5, les commentaires d'encadrement sont inutiles.

Les commentaires d'encadrement servent à isoler le code Javascript pour que le [validateur du W3C](#) ne l'interprète pas.. Si par exemple votre code Javascript contient des chevrons `<` et `>`, le validateur va croire qu'il s'agit de balises HTML mal fermées, et donc va invalider la page. Ce n'est pas grave en soi, mais une page sans erreurs, c'est toujours mieux !

Les commentaires d'encadrement ressemblent à des commentaires HTML et se placent comme ceci :

Code : HTML

```
<body>
  <script type="text/javascript">
```

```
<!--  
    valeur_1 > valeur_2;  
//-->  
</script>  
</body>
```

Le Javascript externe

Il est possible, et même conseillé, d'écrire le code Javascript dans un fichier externe, portant l'extension `.js`. Ce fichier est ensuite appelé depuis la page Web au moyen de l'élément `<script>` et de son attribut `src` qui contient l'url du fichier `.js`. Voici tout de suite un petit exemple :

Code : JavaScript - Contenu du fichier hello.js

```
alert('Hello world!');
```

Code : HTML - Page Web

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Hello World!</title>  
  </head>  
  
  <body>  
    <script type="text/javascript" src="hello.js"></script>  
  
  </body>  
</html>
```

On suppose ci-dessus que le fichier `hello.js` se trouve dans le même répertoire que la page Web.



Il vaut mieux privilégier un fichier externe plutôt que d'inclure le code Javascript directement dans la page, pour la simple et bonne raison que le fichier externe est mis en cache par le navigateur, et n'est donc pas rechargé à chaque chargement de page, ce qui accélère l'affichage de la page.

Positionner l'élément `<script>`

La plupart des cours de Javascript, et des exemples donnés un peu partout, montrent qu'il faut placer l'élément `<script>` au sein de l'élément `<head>` quand on l'utilise pour charger un fichier Javascript. C'est correct, oui, mais il y a mieux !

Une page Web est lue par le navigateur de façon linéaire, c'est-à-dire qu'il lit d'abord le `<head>`, puis les éléments de `<body>` les uns à la suite des autres. Si vous appelez un fichier Javascript dès le début du chargement de la page, le navigateur va donc charger ce fichier, et si ce dernier est volumineux, le chargement de la page s'en trouvera ralenti. C'est normal puisque le navigateur va charger le fichier avant de commencer à afficher le contenu de la page.

Pour pallier ce problème, on conseille de placer les éléments `<script>` liant un fichier `.js` juste avant la fermeture de

l'élément `<body>` , comme ceci :

Code : HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>

    <p>
      <!--
Contenu de la page Web
...
-->
    </p>

    <script type="text/javascript" src="hello.js"></script>

  </body>
</html>
```



Il est à noter que certains navigateurs modernes chargent automatiquement les fichiers Javascript en dernier, mais ce n'est pas toujours le cas. C'est pour cela qu'il faut s'en tenir à cette méthode.

Quelques aides

Les documentations

Pendant la lecture de ce cours, il se peut que vous ayez besoin de plus de renseignements sur diverses choses abordées, normalement toutes les informations nécessaires sont fournies mais si vous le souhaitez vous pouvez consulter un de ces deux sites web :

- [Mozilla Developer Center - Référence de Javascript 1.5](#)
- [SelfHTML - Référence Javascript](#)

Ces deux sites web sont des documentations, dans le jargon informatique il s'agit de documents listant tout ce qui constitue un langage de programmation (instructions, fonctions, etc...). Généralement, tout est trié par catégorie et quelques exemples sont fournis, mais gardez bien à l'esprit que les documentations n'ont aucun but pédagogique, elles remplissent leur travail : lister tout ce qui fait un langage sans trop s'étendre sur les explications. Donc si vous recherchez comment utiliser une certaine fonction (comme **alert**) c'est très bien, mais ne vous attendez pas à apprendre les bases du Javascript grâce à ces sites, c'est possible mais suicidaire si vous débutez en programmation 😊.

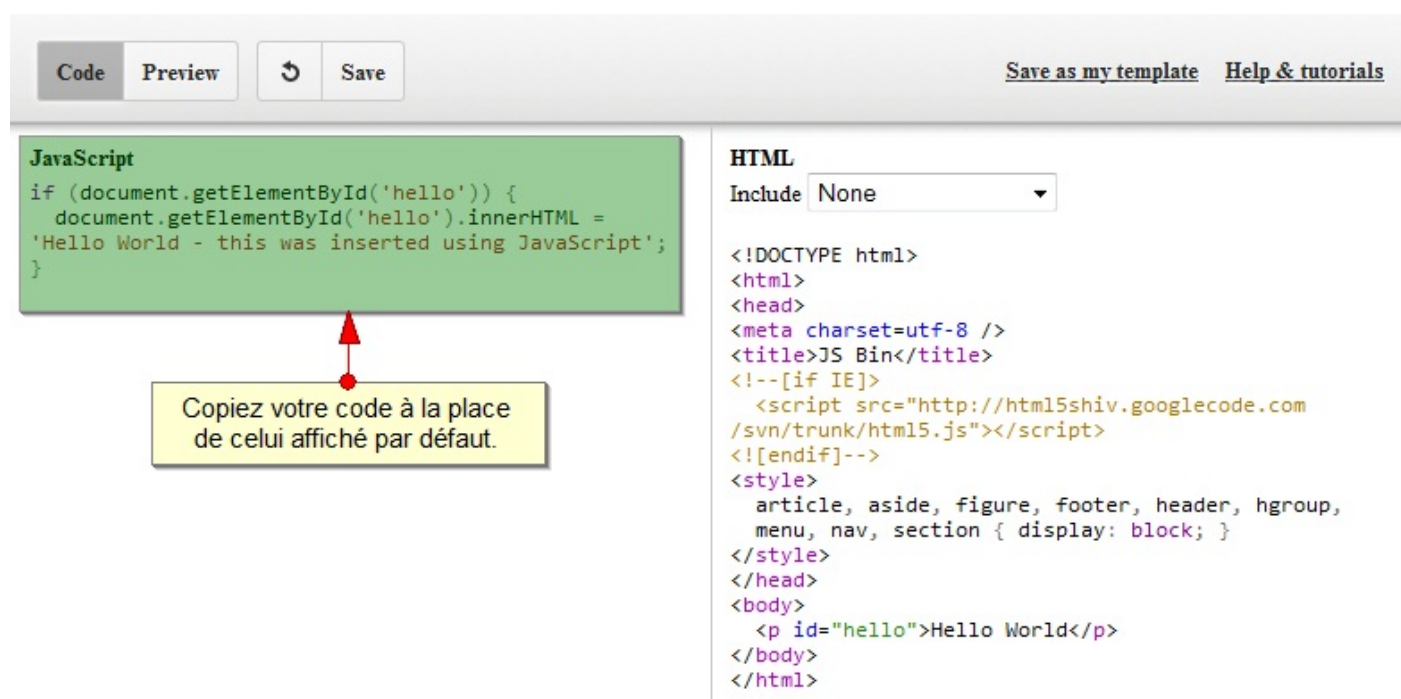
Tester rapidement certains codes

Au cours de votre lecture, vous trouverez de nombreux exemples de codes, certains d'entre eux sont mis en ligne sur le Site du Zéro mais pas tous (il n'est pas possible de tout mettre en ligne, il y a trop d'exemples). Généralement, les exemples mis en ligne sont ceux qui requièrent une action de la part de l'utilisateur, toutefois si vous souhaitez en tester d'autres je vous conseille alors l'utilisation du site suivant :

JS BIN

Ce site est très utile car il vous permet de tester des codes en passant directement par votre navigateur web, ainsi vous n'avez pas besoin de créer de fichier sur votre PC pour tester un malheureux code de quelques lignes.

Pour l'utiliser, rien de plus simple, vous copiez le code que vous souhaitez puis vous le collez comme indiqué dans l'image ci-dessous :



The screenshot shows the JS Bin website interface. At the top, there are buttons for 'Code', 'Preview', a refresh icon, and 'Save'. On the right, there are links for 'Save as my template' and 'Help & tutorials'. The main area is split into two columns: 'JavaScript' and 'HTML'. The JavaScript column contains the following code:

```
if (document.getElementById('hello')) {
  document.getElementById('hello').innerHTML =
  'Hello World - this was inserted using JavaScript';
}
```

The HTML column contains the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>JS Bin</title>
<!--[if IE]>
  <script src="http://html5shiv.googlecode.com
/svn/trunk/html5.js"></script>
<![endif]-->
<style>
  article, aside, figure, footer, header, hgroup,
  menu, nav, section { display: block; }
</style>
</head>
<body>
  <p id="hello">Hello World</p>
</body>
</html>
```

A red arrow points from a yellow box below the JavaScript code to the code editor area. The yellow box contains the text: 'Copiez votre code à la place de celui affiché par défaut.'

Une fois que vous avez copié le texte, il ne vous reste plus qu'à cliquer sur le bouton "Preview" en haut à gauche et votre code sera exécuté immédiatement. Essayez donc avec ce code pour voir :

Code : JavaScript

```
alert('Bien, vous savez maintenant utiliser le site JS BIN.');
```

Voilà tout pour les liens, n'oubliez pas de vous en servir lorsque vous en avez besoin, ils peuvent vous être très utiles !

Les variables

Nous abordons enfin le premier chapitre technique de ce cours !

Tout au long de sa lecture vous allez découvrir l'utilisation des variables, les différents types principaux qu'elles peuvent contenir et surtout comment faire vos premiers calculs. Vous serez aussi initiés à la concaténation et à la conversion des types. Et enfin, un élément important de ce chapitre : vous allez apprendre l'utilisation d'une nouvelle fonction vous permettant d'interagir avec l'utilisateur !

Qu'est-ce qu'une variable ?

Pour faire simple, une variable est un espace de stockage permettant d'enregistrer tout type de donnée que ce soit une chaîne de caractères, une valeur numérique ou bien des structures un peu plus particulières.

Déclarer une variable

Tout d'abord, qu'est-ce que "déclarer une variable" veut dire ? Il s'agit tout simplement de lui réserver un espace de stockage en mémoire, rien de plus. Une fois la variable déclarée, vous pouvez commencer à y stocker des données sans problème.

Pour déclarer une variable, il vous faut d'abord lui trouver un nom. Il est important de préciser que le nom d'une variable ne peut contenir que des caractères **alphanumériques**, autrement dit, les lettres de A à Z et les chiffres de 0 à 9, l'underscore (_) et le dollar (\$) sont aussi acceptés. Autre chose : le nom de la variable ne peut pas commencer par un chiffre et ne peut pas être constitué uniquement de mots-clés utilisés par le Javascript. Par exemple, vous ne pouvez pas créer une variable nommée **var** car vous allez constater que ce mot-clé est déjà utilisé, en revanche vous pouvez créer une variable nommée **var_**.



Concernant les mots-clés utilisés par le Javascript, on peut les appeler "les mots réservés", tout simplement parce que vous n'avez le droit d'en faire usage en tant que nom de variable. Vous trouverez sur cette page tous les mots réservés par Javascript.

Pour déclarer une variable, il vous suffit d'écrire la ligne suivante :

Code : JavaScript

```
var myVariable;
```

Le mot-clé **var** est présent pour indiquer que vous déclarez une variable. Une fois déclarée, il ne vous est plus nécessaire d'utiliser ce mot-clé pour cette variable et vous pouvez y stocker ce que vous souhaitez :

Code : JavaScript

```
var myVariable;  
myVariable = 2;
```

Le signe égal "=" sert à attribuer une valeur à votre variable, ici nous lui avons attribué le chiffre 2. Quand on donne une valeur à une variable, on dit que l'on fait une affectation, car on affecte une valeur à la variable.

Il est possible de simplifier ce code en une seule ligne :

Code : JavaScript

```
var myVariable = 5.5; // Comme vous pouvez le constater, les nombres  
à virgule s'écrivent avec un point.
```

De même, vous pouvez déclarer et assigner des variables sur une seule et même ligne :

Code : JavaScript

```
var myVariable1, myVariable2 = 4, myVariable3;
```

Ici, nous avons déclaré 3 variables en une ligne mais seulement la deuxième s'est vu attribuer une valeur.



Une petite précision ici s'impose. Quand vous utilisez une seule fois l'instruction **var** pour déclarer plusieurs variables, vous devez placer une virgule après chaque variable (et son éventuelle attribution de valeur) et vous ne devez utiliser le point-virgule (qui termine une instruction) qu'à la fin de la déclaration de toutes les variables.

Et enfin une dernière chose qui pourra vous être utile de temps en temps :

Code : JavaScript

```
var myVariable1, myVariable2;  
myVariable1 = myVariable2 = 2;
```

Les deux variables contiennent maintenant le même nombre : 2 ! Vous pouvez faire la même chose avec autant de variables que vous le souhaitez.

Les types de variables

Contrairement à de nombreux langages, le Javascript est un langage typé dynamiquement. Cela veut dire, généralement, que toute déclaration de variable se fait avec le mot-clé **var** sans distinction du contenu. Tandis que dans d'autres langages, comme le C, il est nécessaire de préciser quel type de contenu la variable va devoir contenir.

En Javascript, nos variables sont typées dynamiquement, ce qui veut dire que l'on peut y mettre du texte en premier lieu puis l'effacer et y mettre un nombre quel qu'il soit, et ce, sans contraintes.

Commençons tout d'abord par voir quels sont les trois types principaux en Javascript :

- **Le type numérique (alias "number")** : Il représente tous les nombres, que ce soit un entier, un négatif, un nombre scientifique, etc... Bref, c'est le type pour les nombres.
Pour assigner un type numérique à une variable, il vous suffit juste d'écrire le nombre seul : **var** number = 5;
Tout comme de nombreux langages, le Javascript reconnaît plusieurs écritures pour les nombres, comme les nombres décimaux **var** number = 5.5; ou l'écriture scientifique **var** number = 3.65e+5; ou encore l'écriture hexadécimale **var** number = 0x391;
Bref, il existe pas mal de façons pour écrire les valeurs numériques !
- **Les chaînes de caractères (alias "string")** : Ce type représente n'importe quel texte. On peut l'assigner de deux façons différentes :

Code : JavaScript

```
var text1 = "Mon premier texte"; // Avec des guillemets.  
var text2 = 'Mon deuxième texte'; // Avec des apostrophes.
```

Il est important de préciser que si vous écrivez **var** myVariable = '2'; alors le type de cette variable est une chaîne de caractères et non pas un type numérique.

Une autre précision importante, si vous utilisez les apostrophes pour "encadrer" votre texte et que vous souhaitez utiliser des apostrophes dans ce même texte, il vous faudra alors "échapper" vos apostrophes de cette façon :

Code : JavaScript


```
var text = 'Ça c\'est quelque chose !';
```

Pourquoi ? Car si vous n'échappez pas votre apostrophe, le Javascript croira que votre texte s'arrête à l'apostrophe contenu dans le mot "c'est". À noter que ce problème est identique pour les guillemets.

En ce qui me concerne, j'utilise généralement les apostrophes mais quand mon texte en contient trop je mets des guillemets, c'est plus simple. Mais libre à vous de faire comme vous le souhaitez !

- **Les booléens (alias "boolean")** : Les booléens sont un type bien particulier que vous n'étudierez réellement qu'au chapitre suivant. Dans l'immédiat, pour faire simple, un booléen est un type à deux états qui sont les suivants : **vrai** ou **faux**. Ces deux états s'écrivent de la façon suivante :

Code : JavaScript

```
var isTrue = true;  
var isFalse = false;
```

Voilà pour les trois principaux types. Il en existe d'autres, mais nous ne les verrons que quand ce sera nécessaire.

Tester l'existence de variables avec "typeof"

Il se peut que vous ayez un jour où l'autre besoin de tester l'existence d'une variable ou d'en vérifier son type. Dans ce genre de situations, l'instruction **typeof** est très utile, voici comment l'utiliser :

Code : JavaScript

```
var number = 2;  
alert(typeof number); // Affiche "number".  
  
var text = 'Mon texte';  
alert(typeof text); // Affiche "string".  
  
var aBoolean = false;  
alert(typeof aBoolean); // Affiche "boolean".
```

Simple non ? Et maintenant voici comment tester l'existence d'une variable :

Code : JavaScript

```
alert(typeof nothing); // Affiche "undefined".
```

Voilà un type de variable très important ! Si le **typeof** vous renvoie "undefined" c'est soit que votre variable est inexistante, soit qu'elle est déclarée mais ne contient rien.

Les opérateurs arithmétiques

Maintenant que vous savez déclarer une variable et lui attribuer une valeur, nous pouvons entamer la partie concernant les opérateurs arithmétiques. Vous verrez plus tard qu'il existe plusieurs sortes d'opérateurs mais dans l'immédiat nous voulons faire des calculs, nous allons donc nous intéresser exclusivement aux opérateurs arithmétiques. Ces derniers sont à la base de tout calcul et sont au nombre de cinq :

Opérateur	Signe assigné
addition	+
soustraction	-
multiplication	*
division	/
modulo	%

Concernant ce dernier opérateur, le modulo est tout simplement le reste d'une division. Par exemple, si vous divisez 3 par 2 alors il vous reste 1, c'est le modulo !

Quelques calculs simples

Faire des calculs en programmation est quasiment tout aussi simple que sur une calculatrice, exemple :

Code : JavaScript

```
var result = 3 + 2;
alert(result); // Affiche "5".
```

Alors vous savez faire des calculs avec deux chiffres c'est bien, mais avec deux variables contenant elles-mêmes des chiffres c'est mieux :

Code : JavaScript

```
var number1 = 3, number2 = 2, result;
result = number1 * number2;
alert(result); // Affiche "6".
```

On peut aller encore plus loin comme ça en écrivant des calculs impliquant plusieurs opérateurs ainsi que des variables :

Code : JavaScript

```
var divisor = 3, result1, result2, result3;

result1 = (16 + 8) / 2 - 2 ; // 10
result2 = result1 / divisor;
result3 = result1 % divisor;

alert(result2); // Résultat de la division : 3,33
alert(result3); // Reste de la division : 1
```

Vous remarquerez que j'ai utilisé des parenthèses pour le calcul de la variable **result1**. Elles s'utilisent comme en maths : grâce à elles le navigateur calcule d'abord $16 + 8$ puis divise le résultat par 2.

Simplifier encore plus vos calculs

Par moment, dans vos codes, vous aurez besoin d'écrire des codes de ce genre :

Code : JavaScript

```
var number = 3;
number = number + 5;
alert(number); // Affiche "8";
```

Ce n'est pas spécialement long ou compliqué à faire, mais ça peut devenir très vite rébarbatif, il existe donc une solution plus simple pour ajouter un chiffre à une variable :

Code : JavaScript

```
var number = 3;
number += 5;
alert(number); // Affiche "8";
```

Ce code a exactement le même effet que le précédent mais est plus rapide à écrire.

À noter que ceci ne s'applique pas uniquement aux additions mais fonctionne avec tous les autres opérateurs arithmétiques :

Code : Autre

```
+=
-=
*=
/=
%=
```

Initiation à la concaténation et à la conversion des types

Certains opérateurs ont des particularités cachées. Prenons l'opérateur (+), en plus de faire des additions, il permet de faire ce que l'on appelle des concaténations entre des chaînes de caractères.

La concaténation

Une concaténation consiste à ajouter une chaîne de caractères à la fin d'une autre, prenons un exemple :

Code : JavaScript

```
var hi = 'Bonjour', name = 'toi', result;
result = hi + name;
alert(result); // Affiche "Bonjourtoi".
```

Cet exemple va vous afficher la phrase "Bonjourtoi". Vous remarquerez qu'il n'y a pas d'espace entre les deux mots, en effet, la concaténation respecte ce que vous avez écrit dans les variables à la lettre près. Si vous voulez un espace, il vous faut ajouter un espace à une des variables, comme ceci : `var hi = 'Bonjour ';`

Autre chose, vous vous souvenez toujours de l'addition suivante ?

Code : JavaScript

```
var number = 3;
number += 5;
```

Eh bien vous pouvez faire la même chose avec les chaînes de caractères :

Code : JavaScript

```
var text = 'Bonjour ';
text += 'toi';
alert(text); // Affiche "Bonjour toi".
```

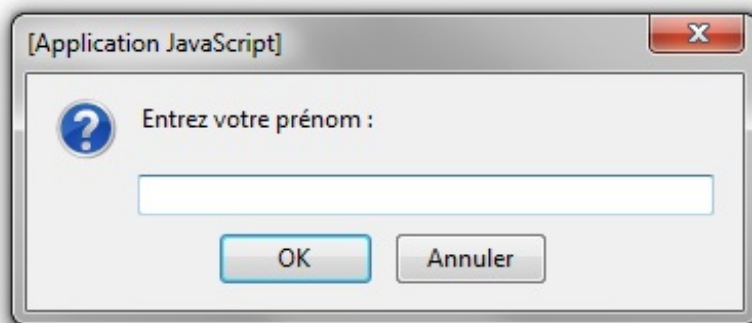
Interagir avec l'utilisateur

La concaténation est le bon moment pour introduire votre toute première interaction avec l'utilisateur grâce à la fonction **prompt**, voici comment l'utiliser :

Code : JavaScript

```
var userName = prompt('Entrez votre prénom :');
alert(userName); // Affiche le prénom entré par l'utilisateur.
```

Essayer !



Un aperçu de la fonction `prompt`

La fonction **prompt** s'utilise comme **alert** mais elle a une petite particularité : elle renvoie ce que l'utilisateur a écrit, et pour récupérer le texte il faut le stocker dans une variable, voilà pourquoi on écrit de cette manière :

Code : JavaScript

```
var text = prompt('Tapez quelque chose :');
```

Ainsi, le texte tapé par l'utilisateur se retrouvera directement stocké dans la variable **text**.

Maintenant nous pouvons essayer de dire bonjour à nos visiteurs :

Code : JavaScript

```
var start = 'Bonjour ', name, end = ' !', result;

name = prompt('Quel est votre prénom ?');
result = start + name + end;
alert(result);
```

[Essayer !](#)

À noter que dans notre cas de figure actuel, nous concaténons des chaînes de caractères entre elles, mais sachez que vous pouvez très bien concaténer une chaîne de caractères et un nombre de la même manière :

Code : JavaScript

```
var text = 'Voici un nombre : ', number = 42, result;

result = text + number;
alert(result); // Affiche "Voici un nombre : 42".
```

Convertir une chaîne de caractères en nombre

Essayons maintenant de faire une addition avec des chiffres fournis par l'utilisateur :

Code : JavaScript

```
var first, second, result;

first = prompt('Entrez le premier chiffre :');
second = prompt('Entrez le second chiffre :');
result = first + second;

alert(result);
```

Essayer !

Si vous avez essayé ce code, vous avez sûrement remarqué qu'il y a un problème. Admettons que vous ayez tapé deux fois le chiffre 1, le résultat sera 11... Pourquoi ? Et bien la raison a déjà été écrite quelques lignes plus haut :

Citation

Une fois que l'utilisateur a écrit puis appuyé sur le bouton OK, vous récupérez alors **sous forme de chaînes de caractères** ce qu'il a écrit dans le champ de texte, ceci est stocké dans votre variable.

Voilà le problème, tout ce qui est écrit dans le champ de texte de **prompt** est récupéré sous forme d'une chaîne de caractères, que ce soit un chiffre ou non. Du coup, si vous utilisez l'opérateur (+), vous ne ferez pas une addition mais une concaténation !

C'est là que la conversion des types intervient, le concept est simple : il suffit de convertir la chaîne de caractères en nombres. Pour cela, vous allez avoir besoin de la fonction **parseInt** qui s'utilise de cette manière :

Code : JavaScript

```
var text = '1337', number;

number = parseInt(text);
alert(typeof number); // Affiche "number".
alert(number); // Affiche "1337".
```

Maintenant que vous savez comment vous en servir, on va pouvoir l'adapter à notre code :

Code : JavaScript

```
var first, second, result;

first = prompt('Entrez le premier chiffre :');
second = prompt('Entrez le second chiffre :');
result = parseInt(first) + parseInt(second);

alert(result);
```

Essayer !

Maintenant, si vous écrivez deux fois le chiffre 1, vous obtiendrez bien 2 comme résultat.

Convertir un nombre en chaîne de caractères

Pour clore ce chapitre, nous allons voir comment convertir un nombre en chaîne de caractères. Il est déjà possible de concaténer un nombre et une chaîne sans conversion, mais pas deux nombres, car ceux-ci s'ajouteraient à cause de l'emploi du

'+'. D'où le besoin de convertir un nombre en chaîne. Voici comment faire :

Code : JavaScript

```
var text, number1 = 4, number2 = 2;  
text = number1 + ' ' + number2;  
alert(text); // Affiche "42";
```

Qu'avons-nous fait ? Nous avons juste ajouté une chaîne de caractères vide entre les deux nombres, ce qui aura eu pour effet de les convertir en chaîne de caractères.

Il existe une solution un peu moins archaïque que de rajouter une chaîne vide mais vous la découvrirez plus tard.

Les conditions

Dans le chapitre précédent vous avez appris comment créer et modifier des variables, c'est déjà bien mais malgré tout on se sent encore un peu limité dans nos codes. Dans ce chapitre, vous allez donc découvrir les conditions de tout type et surtout vous rendre compte que les possibilités pour votre code sont déjà bien plus ouvertes car vos conditions vont influencer directement sur la façon dont va réagir votre code à certains critères.

En plus des conditions, vous allez aussi pouvoir approfondir vos connaissances sur un fameux type de variable : le booléen !

La base de toute condition : les booléens

Dans ce chapitre, nous allons aborder les conditions, mais pour cela il nous faut tout d'abord revenir sur un type de variable dont je vous avais parlé au chapitre précédent : les booléens.

À quoi vont-ils nous servir ? À obtenir un résultat comme **true** (vrai) ou **false** (faux) lors du test d'une condition.

Pour ceux qui se posent la question, une condition est une sorte de "test" afin de vérifier qu'une variable contient bien une certaine valeur ou est bien d'un certain type. Bien sûr les comparaisons ne se limitent pas aux variables seules, mais pour le moment nous allons nous contenter de ça, ce sera largement suffisant pour commencer.

Tout d'abord, de quoi sont constituées les conditions ? De valeurs à tester et deux types d'opérateurs : un logique et un de comparaison.

Les opérateurs de comparaison

Comme leur nom l'indique, ces opérateurs vont vous permettre de comparer diverses valeurs entre elles. En tout, ils sont au nombre de 8, les voici :

Opérateur	Signification
==	égal à
!=	différent de
===	contenu et type égal à
!==	contenu ou type différent de
>	supérieur à
>=	supérieur ou égal à
<	inférieur à
<=	inférieur ou égal à

Je ne vais pas vous faire un exemple pour chacun d'entre eux mais je vais au moins vous montrer comment les utiliser afin que vous puissiez essayer les autres :

Code : JavaScript

```
var number1 = 2, number2 = 2, number3 = 4, result;

result = number1 == number2; // Au lieu d'une seule valeur, on en
écrit deux avec l'opérateur de comparaison entre elles.
alert(result); // Affiche "true", la condition est donc vérifiée car
les deux variables contiennent bien les mêmes valeurs.

result = number1 == number3;
alert(result); // Affiche "false", la condition n'est pas vérifiée
car 2 est différent de 4.

result = number1 < number3;
alert(result); // Affiche "true", la condition est vérifiée car 2 est
bien inférieur à 4.
```

Comme vous voyez le concept n'est pas bien compliqué, il suffit d'écrire deux valeurs avec l'opérateur de comparaison

souhaité entre les deux et un booléen est retourné, si celui-ci est **true** alors la condition est vérifiée, si c'est le **false** alors elle ne l'est pas.



Lorsque qu'une condition renvoie **true** on dit qu'elle est vérifiée.

Sur ces huit opérateurs, deux d'entre eux peuvent être difficiles à comprendre pour un débutant il s'agit de (**===**) et (**!==**), afin que vous ne soyez pas perdus voyons leur fonctionnement avec quelques exemples :

Code : JavaScript

```
var number = 4, text = '4', result;

result = number == text;
alert(result); // Affiche "true" alors que "number" est un chiffre
et "text" est une chaîne de caractères.

result = number === text;
alert(result); // Affiche "false" car cet opérateur compare aussi
les types des variables en plus de leurs valeurs.
```

Vous comprenez leur principe maintenant ? Les conditions "normales" font des conversions de type pour vérifier les égalités, ce qui fait que si vous voulez différencier un chiffre 4 d'une chaîne de caractères contenant le chiffre 4 il vous faudra alors utiliser le triple égal (**===**).

Voilà tout pour les opérateurs de comparaison, vous avez tous les outils dont vous avez besoin pour faire quelques expérimentations, passons maintenant à la suite.

Les opérateurs logiques

Pourquoi ces opérateurs sont-ils nommés comme étant "logiques" ? Car ils fonctionnent sur le même principe que la [Table de Vérité](#) en électronique. Avant de décrire leur fonctionnement, il nous faut d'abord les lister, ils sont au nombres de trois :

Opérateur	Type de logique	Utilisation
&&	ET	valeur1 && valeur2
	OU	valeur1 valeur2
!	NON	!valeur

L'opérateur ET

Cet opérateur vérifie la condition lorsque toutes les valeurs qui lui sont passées valent **true** . Si une seule d'entre elles vaut **false** alors la condition ne sera pas vérifiée, exemple :

Code : JavaScript

```
var result = true && true;
alert(result); // Affiche "true".

result = true && false;
alert(result); // Affiche "false".
```

L'opérateur OU

Cet opérateur est plus "souple" car il renvoie **true** si une des valeurs qui lui est soumise contient **true**, qu'importe les autres valeurs, exemple :

Code : JavaScript

```
var result = true || false;
alert(result); // Affiche "true".

result = false || false;
alert(result); // Affiche "false".
```

L'opérateur NON

Cet opérateur se différencie des deux autres car il ne prend qu'une seule valeur à la fois. Si il se nomme "NON" c'est parce que sa fonction est d'inverser la valeur qui lui est passée, ainsi **true** deviendra **false** et inversement, exemple :

Code : JavaScript

```
var result = false;

result = !result; // On stocke dans "result" l'inverse de "result",
                  // c'est parfaitement possible.
alert(result); // Affiche "true" car on voulait l'inverse de
               // "false".

result = !result;
alert(result); // Affiche "false" car on a inversé de nouveau
               // "result", on est donc passé de "true" à "false".
```

Combiner les opérateurs

Bien, nous sommes presque au bout de la partie concernant les booléens, rassurez-vous, ce sera plus simple sur la suite de ce chapitre. Toutefois, avant de passer à la suite, je voudrais m'assurer que vous avez bien compris que tous les opérateurs que nous venons de découvrir peuvent se combiner entre eux.

Tout d'abord un petit résumé (lisez attentivement) : les opérateurs de comparaison acceptent chacun deux valeurs en entrée et renvoient un booléen, tandis que les opérateurs logiques acceptent plusieurs booléens en entrée et renvoient un booléen. Si vous avez bien lu, vous comprendrez que nous pouvons donc coupler les valeurs de sortie des opérateurs de comparaison avec les valeurs d'entrée des opérateurs logiques, exemple :

Code : JavaScript

```
var condition1, condition2, result;

condition1 = 2 > 8; // false
condition2 = 8 > 2; // true

result = condition1 && condition2;
alert(result); // Affiche "false";
```

Il est bien entendu possible de raccourcir le code en combinant tout ça sur une seule ligne, dorénavant toutes les conditions seront sur une seule ligne dans ce tuto :

Code : JavaScript

```
var result = 2 > 8 && 8 > 2;  
alert(result); // Affiche "false";
```

Voilà tout pour les booléens et les opérateurs conditionnels, nous allons enfin pouvoir commencer à utiliser les conditions comme il se doit.

La condition "if - else"

Enfin nous abordons les conditions ! Ou, plus exactement, **les structures conditionnelles**, mais j'écrirai dorénavant le mot "condition" qui sera quand même plus rapide à écrire et à lire.

Avant toute chose, je précise qu'il existe trois types de conditions, nous allons commencer par la condition **if else** qui est la plus utilisée.

La structure "if" pour dire "si"



Mais à quoi sert une condition ? On a pas déjà vu les opérateurs conditionnels juste avant qui permettent déjà d'obtenir un résultat ?

Effectivement, nous arrivons à obtenir un résultat sous forme de booléen, mais c'est tout. Maintenant, il serait bien que ce résultat puisse influencer sur l'exécution de votre code. Nous allons tout de suite rentrer dans le vif du sujet avec un exemple très simple :

Code : JavaScript

```
if (true) {
  alert("Ce message s'est bien affiché.");
}

if (false) {
  alert("Pas la peine d'insister, ce message ne s'affichera pas.");
}
```

Tout d'abord, voyons de quoi est constitué une condition :

- La structure conditionnelle **if**.
- De parenthèses qui contiennent la condition à analyser, ou plus précisément le booléen retourné par les opérateurs conditionnels.
- D'accolades qui permettent de définir la portion de code qui sera exécutée si la condition se vérifie. À noter que moi je place la première accolade à la fin de la première ligne de condition, mais vous pouvez très bien les placer comme vous le souhaitez.

Comme vous pouvez le constater, le code d'une condition est exécuté si le booléen reçu est **true** alors que **false** empêche l'exécution du code.

Et vu que nos opérateurs conditionnels renvoient des booléens, nous allons donc pouvoir les utiliser directement dans nos conditions :

Code : JavaScript

```
if (2 < 8 && 8 >= 4) { // Cette condition renvoie "true", le code
  est donc exécuté.
  alert('La condition est bien vérifiée.');
```

```
}

if (2 > 8 || 8 <= 4) { // Cette condition renvoie "false", le code
  n'est donc pas exécuté.
  alert("La condition n'est pas vérifiée mais vous ne le saurez pas
  vu que ce code ne s'exécute pas.");
}
```

Comme vous pouvez le constater, avant je décomposais toutes les étapes d'une condition dans plusieurs variables, dorénavant

je vous conseille de tout mettre sur une seule et même ligne car ce sera plus rapide à écrire pour vous et plus facile à lire pour tout le monde.

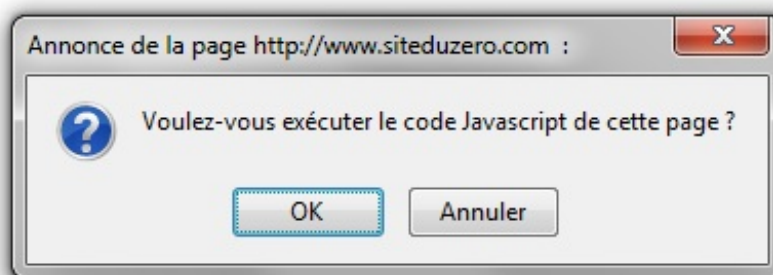
Petit intermède : la fonction confirm

Afin d'aller un petit peu plus loin dans le cours, je vais vous apprendre l'utilisation d'une fonction bien pratique : **confirm** ! Son utilisation est simple : on lui passe en paramètre une chaîne de caractères qui sera affichée à l'écran et elle retourne un booléen en fonction de l'action de l'utilisateur, vous allez comprendre en essayant :

Code : JavaScript

```
if (confirm('Voulez-vous exécuter le code Javascript de cette page ?')) {  
    alert('Le code a bien été exécuté !');  
}
```

Essayer !



Un aperçu de la fonction confirm

Comme vous pouvez le constater, le code s'exécute lorsque vous cliquez sur le bouton **OK** et ne s'exécute pas lorsque vous cliquez sur **Annuler**, en clair : dans le premier cas la fonction renvoie **true** et dans le deuxième cas elle renvoie **false**. Ce qui en fait une fonction très pratique à utiliser avec les conditions.

Après ce petit intermède nous pouvons revenir à nos conditions.

La structure "else" pour dire "sinon"

Admettons maintenant que vous souhaitiez exécuter un code suite à la vérification d'une condition et exécuter un autre code si elle n'est pas vérifiée, il est possible de le faire avec deux conditions **if** mais il existe une solution beaucoup plus simple, la structure **else** :

Code : JavaScript

```
if (confirm('Pour accéder à ce site vous devez avoir 18 ans ou plus,  
cliquez sur "OK" si c\'est le cas.')) {  
    alert('Vous allez être redirigé vers le site.');}  
  
else {  
    alert("Désolé, vous n'avez pas accès à ce site.");  
}
```

[Essayer !](#)

Comme vous pouvez le constater, la structure **else** permet d'exécuter un certain code si la condition n'a pas été vérifiée, et vous allez rapidement vous rendre compte qu'elle vous sera très utile à de nombreuses occasions.

Concernant la façon d'indenter vos structures **if else**, je vous conseille de procéder de la façon suivante :

Code : JavaScript

```
if ( /* condition */ ) {  
    // Le code.  
} else {  
    // Le code.  
}
```

Ainsi la structure **else** suit directement l'accolade de fermeture de la structure **if**, pas de risque de se tromper quant au fait de savoir quelle structure **else** appartient à quelle structure **if**. Et puis c'est, à mon goût, un peu plus "propre" à voir. Enfin vous n'êtes pas obligés de faire de cette façon, il s'agit juste d'un conseil.

La structure "else if" pour dire "sinon si"

Bien, vous savez exécuter du code si une condition se vérifie et si elle ne se vérifie pas, mais il serait bien de fonctionner de la façon suivante :

- Une première condition est à tester.
- Une deuxième condition est présente et sera testée si la première échoue.
- Et si aucune condition ne se vérifie, la structure **else** fait alors son travail.

Cet espèce de cheminement est bien pratique pour tester plusieurs conditions à la fois et exécuter leur code correspondant. La structure **else if** permet cela, exemple :

Code : JavaScript

```
var floor = parseInt(prompt("Entrez l'étage où l'ascenseur doit se  
rendre (de -2 à 30) :"));  
  
if (floor == 0) {  
    alert('Vous vous trouvez déjà au rez-de-chaussée.');} else if (-2 <= floor && floor <= 30) {  
    alert("Direction l'étage n°" + floor + ' !');} else {  
    alert("L'étage spécifié n'existe pas.");  
}
```

[Essayer !](#)

À noter que la structure **else if** peut-être utilisée plusieurs fois de suite, la seule chose qui lui est nécessaire pour pouvoir fonctionner est d'avoir écrit une condition avec la structure **if** juste avant.

La condition "switch"

Ci-dessus nous avons étudié le fonctionnement de la condition **if else** qui est très utile dans de nombreux cas, toutefois elle n'est pas très pratique pour faire du "cas par cas" ; c'est là qu'intervient **switch** !

Prenons un exemple : nous avons un meuble avec 4 tiroirs contenant chacun des objets différents, et il faut que l'utilisateur puisse connaître le contenu du tiroir dont il entre le chiffre. Si nous voulions le faire avec **if else** ce serait assez long et fastidieux :

Code : JavaScript

```
var drawer = parseInt(prompt('Choisissez le tiroir à ouvrir (1 à 4) :'));

if (drawer == 1) {

    alert('Contient divers outils pour dessiner : du papier, des crayons, etc...');

} else if (drawer == 2) {

    alert('Contient du matériel informatique : des câbles, des composants, etc...');

} else if (drawer == 3) {

    alert('Ah ? Ce tiroir est fermé à clé ! Dommage !');

} else if (drawer == 4) {

    alert('Contient des vêtements : des chemises, des pantalons, etc...');

} else {

    alert("Info du jour : le meuble ne contient que 4 tiroirs et, jusqu'à preuve du contraire, les tiroirs négatifs n'existent pas.");

}
```

C'est long non ? Et en plus ce n'est pas très adapté à ce que l'on souhaite faire. Le plus gros problème c'est de devoir réécrire constamment la condition, mais avec **switch** c'est un peu plus facile :

Code : JavaScript

```
var drawer = parseInt(prompt('Choisissez le tiroir à ouvrir (1 à 4) :'));

switch (drawer) {

    case 1:

        alert('Contient divers outils pour dessiner : du papier, des crayons, etc...');

        break;

    case 2:

        alert('Contient du matériel informatique : des câbles, des composants, etc...');

        break;

    case 3:

        alert('Ah ? Ce tiroir est fermé à clé ! Dommage !');

        break;

}
```



```
    case 4:
        alert('Contient des vêtements : des chemises, des pantalons,
etc...');
        break;

    default:
        alert("Info du jour : le meuble ne contient que 4 tiroirs et,
jusqu'à preuve du contraire, les tiroirs négatifs n'existent pas.");
}
```

Essayer !

Comme vous pouvez le constater, le code n'est pas spécialement plus court mais il est déjà mieux organisé et donc plus compréhensible. Détaillons maintenant son fonctionnement :

- On écrit le mot-clé **switch** suivi de la variable à analyser entre parenthèses et d'une paire d'accolades.
- Dans les accolades se trouvent tous les cas de figure pour notre variable, définis par les mots-clés **case** suivis de la valeur qu'ils doivent prendre en compte (cela peut-être un nombre mais aussi du texte) et de deux points.
- Tout ce qui suit les deux points d'un **case** sera exécuté si la variable analysée par le **switch** contient la valeur du **case**.
- À chaque fin d'un **case** on écrit l'instruction **break** pour "casser" le **switch** et ainsi éviter d'exécuter le reste du code qu'il contient.
- Et enfin on écrit le mot-clé **default** suivi de deux points. Le code qui suit cette instruction sera exécuté si aucun des cas précédents n'a été exécuté. Attention, cette partie est optionnelle, vous n'êtes pas obligés de l'intégrer à votre code.

Je pense que dans l'ensemble vous n'aurez pas de mal à comprendre le fonctionnement du **switch**, en revanche l'instruction **break** vous posera peut-être problème, je vous invite donc à essayer le code sans cette instruction à aucun endroit du code.

Vous commencez à comprendre le problème ? Sans l'instruction **break** vous exécutez tout le code contenu dans le **switch** à partir du **case** que vous avez choisi, ainsi, si vous choisissez le tiroir n°2 c'est comme si vous exécutiez ce code :

Code : JavaScript

```
alert('Contient du matériel informatique : des câbles, des
composants, etc...');
alert('Ah ? Ce tiroir est fermé à clé ! Dommage !');
alert('Contient des vêtements : des chemises, des pantalons,
etc...');
alert("Info du jour : le meuble ne contient que 4 tiroirs et,
jusqu'à preuve du contraire, les tiroirs négatifs n'existent pas.");
```

Dans certains cas, ce système peut être pratique mais c'est très rare.

Les ternaires

Et voici enfin le dernier type de condition, *les ternaires*. Vous allez voir que ce type est très particulier tout d'abord parce qu'il est très rapide à écrire (mais peu lisible) et surtout parce qu'il renvoie une valeur.

Pour que vous puissiez bien comprendre dans quel cas de figure vous pouvez utiliser les ternaires, je vais commencer par un petit exemple avec la condition **if else** :

Code : JavaScript

```
var startMessage = 'Votre catégorie : ',
    endMessage,
    adult = confirm('Êtes-vous majeur ?');

if (adult) { // la variable "adult" contient un Booléen, on peut
    donc directement la soumettre à la structure if sans opérateur
    conditionnel.
    endMessage = '18+';
}

else {
    endMessage = '-18';
}

alert(startMessage + endMessage);
```

Essayer !

Comme vous pouvez le constater, le code est plutôt long pour un résultat assez moindre. Avec les ternaires vous pouvez vous permettre de simplifier votre code de façon substantielle :

Code : JavaScript

```
var startMessage = 'Votre catégorie : ',
    endMessage,
    adult = confirm('Êtes-vous majeur ?');

endMessage = adult ? '18+' : '-18';

alert(startMessage + endMessage);
```

Alors comment fonctionnent les ternaires ? Pour le comprendre il faut regarder la ligne 5 du code ci-dessus : `endMessage = adult ? '18+' : '-18';`

Si l'on décompose cette ligne on peut voir :

- La variable **endMessage** qui va accueillir le résultat de la ternaire.
- La variable **adult** qui va être analysée par la ternaire.
- Un point d'interrogation suivi d'une valeur (un nombre, du texte, etc...).
- Deux points suivis d'une deuxième valeur et enfin le point-virgule marquant la fin de la ligne d'instructions.

Le fonctionnement est simple : si la variable **adult** vaut **true** alors la valeur retournée par la ternaire sera celle écrite juste après le point d'interrogation, si elle vaut **false** alors la valeur retournée sera celle après les deux points.

Pas très compliqué n'est-ce pas ? Les ternaires sont des conditions très simples et rapides à écrire, mais elles ont la mauvaise réputation d'être assez peu lisibles (on ne les remarque pas facilement dans un code de plusieurs lignes). Je vois beaucoup de personnes qui en déconseillent l'utilisation, pour ma part je vous conseille plutôt de vous en servir car elles sont très utiles. Si

vous épurez bien votre code les ternaires seront facilement visibles, ce qu'il vous faut éviter ce sont des codes de ce style :

Code : JavaScript

```
alert('Votre catégorie : ' + (confirm('Êtes-vous majeur ?') ? '18+'  
: '-18'));
```

Impressionnant n'est-ce pas ? Notre code initial faisait 13 lignes et maintenant tout est condensé en une seule ligne. Toutefois, il faut reconnaître que c'est très peu lisible. Les ternaires sont très utiles pour raccourcir des codes mais il ne faut pas pousser leurs capacités à leur paroxysme ou bien vous vous retrouverez avec un code que vous ne saurez plus lire vous-même.

Bref, les ternaires c'est bon, mangez-en ! Mais pas jusqu'à l'indigestion !

Les conditions sur les variables

Le Javascript est un langage assez particulier dans sa manière de coder, vous vous en rendrez compte par la suite si vous connaissez déjà un autre langage déjà plus "conventionnel". Le cas particulier que nous allons étudier ici concerne le test des variables : il est possible de tester si une variable possède une valeur sans même utiliser l'instruction **typeof**!

Tester l'existence de contenu d'une variable

Pour tester l'existence de contenu d'une variable, il faut tout d'abord savoir que tout se joue au niveau de la conversion des types. Vous savez que les variables possèdent plusieurs types : les nombres, les chaînes de caractères, etc... Eh bien ici nous allons découvrir que le type d'une variable (quel qu'il soit) peut être converti en booléen même si à la base on possède un nombre ou une chaîne de caractères.

Voici un exemple simple :

Code : JavaScript

```
var conditionTest = 'Fonctionnera ? Fonctionnera pas ?';

if (conditionTest) {
  alert('Fonctionne !');
} else {
  alert('Ne fonctionne pas !');
}
```

[Essayer !](#)

Le code nous affiche le texte "Fonctionne !". Pourquoi ? Tout simplement parce que la variable **conditionTest** a été convertie en booléen et que son contenu est évalué comme étant vrai (**true**).

Qu'est-ce qu'un contenu vrai ou faux ? Eh bien, il suffit simplement de lister les contenus faux pour le savoir : un nombre qui vaut zéro ou bien une chaîne de caractères vide. C'est tout, ces deux cas sont les seuls à être évalués comme étant à **false**. Bon, après il est possible d'évaluer des attributs, des méthodes, des objets, etc... Seulement, vous verrez cela plus tard.

Le cas de l'opérateur OU

Encore un cas à part : l'opérateur **OU** ! Celui-ci, en plus de sa fonction principale, permet de renvoyer la première variable possédant une valeur évaluée à **true** ! Exemple :

Code : JavaScript

```
var conditionTest1 = '', conditionTest2 = 'Une chaîne de caractères';

alert(conditionTest1 || conditionTest2);
```

[Essayer !](#)

Au final, ce code nous retourne la valeur "Une chaîne de caractères". Pourquoi ? Eh bien parce que l'opérateur **OU** va se charger de retourner la valeur de la première variable dont le contenu est évalué à **true**.

Un petit exercice pour la forme !

Bien, maintenant que vous avez appris à vous servir des conditions, il serait intéressant de faire un petit exercice pour que vous puissiez vous entraîner.

Présentation de l'exercice

Qu'est-ce que l'on va essayer de faire ? Quelque chose de tout simple : fournir un commentaire selon l'âge de la personne. Vous devez fournir un commentaire sur 4 tranches d'âge différentes qui sont les suivantes :

Tranche d'âge	Exemple de commentaire
1 à 17 ans	"Vous n'êtes pas encore majeur."
18 à 49 ans	"Vous êtes majeur mais pas encore senior."
50 à 59 ans	"Vous êtes senior mais pas encore retraité."
60 à 120 ans	"Vous êtes retraité, profitez de votre temps libre !"

Le déroulement du code sera le suivant :

- L'utilisateur charge la page web.
- Il est ensuite invité à taper son âge dans une fenêtre d'interaction.
- Une fois l'âge fourni l'utilisateur obtient un petit commentaire.

L'intérêt de cet exercice n'est pas spécialement le fait de sortir un commentaire pour chaque tranche d'âge, mais surtout que vous cherchiez à utiliser la structure conditionnelle la plus adaptée et que vous puissiez préparer votre code à toutes les éventualités.

Correction

Et voici la correction :

Secret (cliquez pour afficher)

Code : JavaScript

```
var age = parseInt(prompt('Quel est votre âge ?')); // Ne pas oublier : Il faut "parser" (cela consiste à analyser) la valeur renvoyée par prompt pour avoir un nombre !

if (age <= 0) { // Il faut bien penser au fait que l'utilisateur peut rentrer un âge négatif.
    alert("Oh vraiment ? Vous avez moins d'un an ? C'est pas très crédible =p");
} else if (1 <= age && age < 18) {
    alert("Vous n'êtes pas encore majeur.");
} else if (18 <= age && age < 50) {
    alert('Vous êtes majeur mais pas encore senior.');
```

```
    } else if (50 <= age && age < 60) {  
        alert('Vous êtes senior mais pas encore retraité.');
```

```
    } else if (60 <= age && age <= 120) {  
        alert('Vous êtes retraité, profitez de votre temps libre !');
```

```
    } else if (age > 120) { // Ne pas oublier les plus de 120 ans, ils  
    existent probablement pas mais on le met dans le doute ^^ .  
        alert("Plus de 120 ans ?!! C'est possible ça ?!");
```

```
    } else { // Si prompt contient autre chose que les intervalles de  
    nombre ci-dessus alors l'utilisateur a écrit n'importe quoi.  
        alert("Vous n'avez pas entré d'âge !");  
    }
```

Essayer !

Alors, est-ce que vous aviez bien pensé à toutes les éventualités ? J'ai un doute pour la condition de la structure **else** 🤔 ! En effet, l'utilisateur peut choisir de ne pas rentrer un nombre mais un mot ou une phrase quelconque, dans ce cas la fonction **parseInt** ne va pas réussir à trouver de nombre et va donc renvoyer la valeur **NaN** qui signifie "Not a Number", nos différentes conditions ne se vérifieront donc pas et la structure **else** sera donc finalement exécutée, avertissant ainsi l'utilisateur qu'il n'a pas entré de nombre.

Pour ceux qui ont choisi d'utiliser les ternaires ou les **switch**, je vous conseille de relire un peu le tuto car ils ne sont clairement pas adaptés à ce type d'utilisation.

Les boucles

Les programmeurs sont réputés pour être des gens fainéants ce qui n'est pas totalement faux, puisque le but de la programmation est de faire exécuter des choses à un ordinateur, pour ne pas les faire nous même. Ce chapitre va mettre en lumière ce comportement intéressant : nous allons en effet voir comment répéter des actions, pour ne pas écrire plusieurs fois les mêmes instructions. Mais avant ça, on va parler de l'incrément.

L'incréméntation

Considérons le calcul suivant :

Code : JavaScript

```
var number = 0;
number = number + 1;
```

La variable **number** contient donc la valeur 1. Seulement, l'instruction pour ajouter "1" est assez lourde à écrire et souvenez-vous, nous sommes des fainéants. Javascript, comme d'autres langages de programmation, permet ce que l'on appelle l'*incréméntation*, ainsi que son contraire, la *décréméntation*.

Le fonctionnement

L'incréméntation permet d'ajouter une unité à un nombre au moyen d'une syntaxe courte. A l'inverse, la décréméntation permet de soustraire une unité.

Code : JavaScript

```
var number = 0;

number++;
alert(number); // Affiche 1

number--;
alert(number); // Affiche 0
```

Il s'agit donc d'une méthode assez rapide pour ajouter ou soustraire une unité à une variable (on dit *incréménter* et *décréménter*), et cela nous sera particulièrement utile tout au long de ce chapitre.

L'ordre des opérateurs

Il existe deux manières d'utiliser l'incréméntation en fonction de la position de l'opérateur ++ (ou --). On a vu qu'il pouvait se placer après la variable, mais il peut aussi se placer avant. Petit exemple :

Code : JavaScript

```
var number_1 = 0;
var number_2 = 0;

number_1++;
++number_2;

alert(number_1); // Affiche 1
alert(number_2); // Affiche 1
```



number_1 et **number_2** ont tous deux été incrémentés. Quelle est donc la différence entre les deux procédés ?

La différence réside en fait dans la priorité de l'opération et ça a de l'importance si vous voulez récupérer le résultat de l'incrément. Dans l'exemple suivant, `++number` retourne la valeur de **number** incrémentée, c'est-à-dire **1**.

Code : JavaScript

```
var number = 0;
var output = ++number;

alert(number); // Affiche 1
alert(output); // Affiche 1
```

Maintenant, si on place l'opérateur après la variable à incrémenter, l'opération retourne la valeur de **number** avant qu'elle ne soit incrémentée :

Code : JavaScript

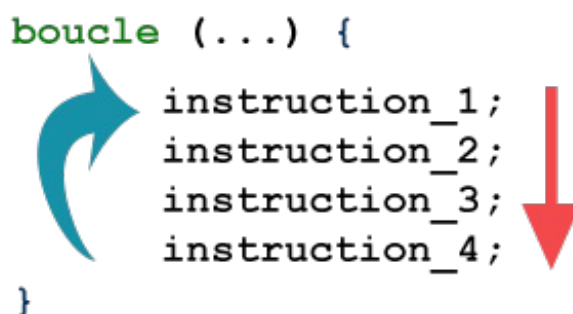
```
var number = 0;
var output = number++;

alert(number); // Affiche 1
alert(output); // Affiche 0
```

Ici donc, l'opération `number++` a retourné la valeur de **number** non incrémentée.

La boucle while

Une boucle est une structure analogue aux structures conditionnelles vues dans le chapitre précédent sauf qu'ici il s'agit de répéter une série d'instructions. La répétition se fait jusqu'à ce qu'on dise à la boucle de s'arrêter. A chaque fois que la boucle se répète on parle d'*itération* (qui est en fait un synonyme de *répétition*).



Pour faire fonctionner une boucle, il est nécessaire de définir une condition. Tant que la condition est vraie (**true**), la boucle se répète. Dès que la condition est fausse (**false**), la boucle s'arrête.

Voici un exemple de la syntaxe d'une boucle **while** :

Code : JavaScript

```
while (condition) {  
  instruction_1;  
  instruction_2;  
  instruction_3;  
}
```

Répéter tant que...

La boucle **while** se répète tant que la condition est validée. Cela veut donc dire qu'il faut s'arranger, à un moment, pour que la condition ne soit plus vraie, sinon la boucle se répéterait à l'infini, ce qui serait fâcheux.

En guise d'exemple, on va incrémenter un nombre, qui vaut 1, jusqu'à ce qu'il vaille 10 :

Code : JavaScript

```
var number = 1;  
  
while (number < 10) {  
  number++;  
}  
  
alert(number); // Affiche 10
```

Au départ, **number** vaut 1. Arrive ensuite la boucle, qui va demander si **number** est strictement plus petit que 10. Comme c'est vrai, la boucle est exécutée, et **number** est incrémenté. A chaque fois que les instructions présentes dans la boucle sont exécutées, la condition de la boucle est ré-évaluée pour savoir s'il faut ré-exécuter la boucle ou non. Dans l'exemple ci-dessus, la boucle se répète jusqu'à ce que **number** soit égal à 10. Si **number** vaut 10, la condition `number < 10` est fausse, et la boucle s'arrête. Quand la boucle s'arrête, les instructions qui suivent la boucle (l'instruction **alert** dans notre exemple) sont exécutées normalement.

Exemple pratique

Imaginons un petit script qui va demander à l'internaute son prénom, ainsi que les prénoms de ses frères et ses sœurs. Ce n'est pas compliqué à faire me direz-vous, puisqu'il s'agit d'afficher une boîte de dialogue **prompt** pour chaque prénom. Seulement, comment savoir à l'avance le nombre de frères et sœurs ?

Nous allons utiliser une boucle **while**, qui va demander, à chaque passage dans la boucle, un prénom supplémentaire. La boucle ne s'arrêtera que quand l'utilisateur choisira de ne plus entrer de prénom.

Code : JavaScript

```
var nicks = '', nick,
    proceed = true;

while (proceed) {
    nick = prompt('Entrez un prénom :');

    if (nick) {
        nicks += nick + ' '; // Ajoute le nouveau prénom ainsi qu'un
        // espace juste après
    } else {
        proceed = false; // Aucun prénom n'a été rentré, donc on fait en
        // sorte d'invalider la condition
    }
}

alert(nicks); // Affiche les prénoms à la suite
```

[Essayer !](#)

La variable **proceed** est ce qu'on appelle une *variable témoin*, ou bien une *variable de boucle*. C'est une variable qui n'intervient pas directement dans les instructions de la boucle mais qui sert juste pour tester la condition. J'ai choisi de la nommer **proceed**, qui veut dire *poursuivre* en anglais.

A chaque passage dans la boucle, un prénom est demandé, et sauvé temporairement dans la variable **nick**. On effectue alors un test sur **nick** pour savoir si elle contient quelque chose, et dans ce cas, on ajoute le prénom à la variable **nicks**. Remarquez que j'ajoute aussi un simple espace, pour séparer les prénoms. Si par contre **nick** ne contient rien, ce qui veut dire que l'utilisateur n'a pas rentré de prénom ou a cliqué sur **Annuler**, on change la valeur de **proceed** en **false**, ce qui invalidera la condition, et cela empêchera la boucle de refaire une itération.

Quelques améliorations

Utilisation de **break**

Dans l'exemple des prénoms, j'utilise une variable de boucle pour pouvoir arrêter la boucle. Cependant, il existe un mot-clé pour arrêter la boucle d'un seul coup. Ce mot-clé est **break**, et il s'utilise exactement comme dans la structure conditionnelle **switch**, vue au chapitre précédent. Si l'on reprend l'exemple, voici ce que ça donne avec un **break** :

Code : JavaScript

```
var nicks = '', nick;

while (true) {
    nick = prompt('Entrez un prénom :');
    if (!nick) break;
    nicks += nick + ' ';
}
```

```
    if (nick) {  
        nicks += nick + ' '; // Ajoute le nouveau prénom ainsi qu'un  
        espace juste après  
    } else {  
        break; // On quitte la boucle  
    }  
}  
  
alert(nicks); // Affiche les prénoms à la suite
```

Essayer !

Utilisation de *continue*

Cette instruction est plus rare, car les opportunités de l'utiliser ne sont pas toujours fréquentes. **continue**, un peu comme **break**, permet de mettre fin à une itération, mais attention, elle ne provoque pas la fin de la boucle : l'itération en cours est stoppée, et la boucle passe à l'itération suivante.

La boucle do while

La boucle **do while** ressemble très fortement à la boucle **while**, sauf que dans ce cas la boucle est toujours exécutée au moins une fois. Dans le cas d'une boucle **while**, si la condition n'est pas valide, la boucle n'est pas exécutée. Avec **do while**, la boucle est exécutée une première fois, puis la condition est testée pour savoir si la boucle doit continuer.

Voici la syntaxe d'une boucle **do while** :

Code : JavaScript

```
do {  
  instruction_1;  
  instruction_2;  
  instruction_3;  
} while (condition);
```

On note donc une différence fondamentale dans l'écriture par rapport à la boucle **while**, ce qui permet de bien faire la différence entre les deux. Cela dit, l'utilisation des boucles **do while** n'est pas très fréquent, et il est fort possible que vous n'en ayez jamais l'utilité car généralement les programmeurs utilisent une boucle **while** normale, avec une condition qui fait que celle-ci est toujours exécutée une fois.



Attention à la syntaxe de la boucle **do while** : il y a un point-virgule après la parenthèse fermante du **while** !

La boucle for

La boucle **for** ressemble dans son fonctionnement à la boucle **while**, mais son architecture paraît compliquée au premier abord. La boucle **for** est en réalité une boucle qui fonctionne assez simplement, mais qui semble très complexe pour les débutants en raison de sa syntaxe. Une fois que cette boucle est maîtrisée, il est fort à parier que c'est celle-là que vous utiliserez le plus souvent.

Le schéma d'une boucle **for** est le suivant :

Code : JavaScript

```
for (initialisation; condition; incrémentation) {  
  instruction_1;  
  instruction_2;  
  instruction_3;  
}
```

Dans les parenthèses de la boucle ne se trouve plus juste la condition, mais 3 blocs : *initialisation*, *condition*, et *incrémentement*. Ces 3 blocs sont séparés par un **point-virgule** ; c'est un peu comme si les parenthèses contenaient 3 instructions distinctes.

for, la boucle pour l'incrémentement

La boucle **for** possède donc 3 blocs qui la définissent. Le troisième est le bloc d'*incrémentement* qu'on va utiliser pour incrémenter une variable à chaque itération de la boucle. De ce fait, la boucle **for** est très pratique pour compter ainsi que pour répéter la boucle un nombre défini de fois.

Dans l'exemple suivant, on va afficher 5 fois une boîte de dialogue **alert**, qui affichera le numéro de chaque itération :

Code : JavaScript

```
for (var iter = 0; iter < 5; iter++) {  
  alert('Itération n°' + iter);  
}
```

Dans le premier bloc, l'initialisation, on initialise une variable appelée **iter** qui vaut 0 ; le mot-clé **var** est requis, comme pour toute initialisation. On définit dans la condition que la boucle continue tant qu'**iter** est strictement inférieur à 5. Enfin, dans le bloc *incrémentement*, et indique qu'**iter** sera incrémentée.



Mais, il ne m'affiche que "Itération n°4" à la fin, il n'y a pas d'itération n°5 ?

C'est tout à fait normal pour deux raisons : le premier tour de boucle porte l'indice 0, donc si on compte de 0 à 4, il y a bien 5 tours : 0, 1, 2, 3 et 4. Ensuite, l'incrémentement n'a pas lieu avant chaque itération, mais à la fin de chaque itération. Donc, le tout premier tour de boucle est fait avec **iter** qui vaut 0, avant d'être incrémenté.

Reprenons notre exemple

Avec les quelques points de théorie que nous venons de voir, on peut réécrire notre exemple des prénoms, tout en montrant qu'une boucle **for** peut-être utilisée sans le comptage :

Code : JavaScript

```
for (var nicks = '', nick; true;) {
  nick = prompt('Entrez un prénom :');

  if (nick) {
    nicks += nick + ' ';
  } else {
    break;
  }
}

alert(nicks);
```

Essayer !

Dans le bloc d'*initialisation* (le premier), on commence par initialiser nos deux variables. Vient alors le bloc avec la *condition* (le deuxième), qui vaut simplement `true`. On termine par le bloc d'*incréméntation* et... il n'y en a pas besoin ici, puisqu'il n'y a pas besoin d'incrémenter. On le fera pour un autre exemple juste après. Ce troisième bloc est vide, mais existe. C'est pour cela qu'on doit quand même mettre le point-virgule après le deuxième bloc (la *condition*).

Maintenant, modifions la boucle de manière à compter combien de prénoms ont été enregistrés. Pour ce faire, on va créer une variable de boucle, nommée `i`, qui sera incrémentée à chaque passage de boucle.



On nomme généralement les variables de boucles **for i**. Si une boucle se trouve dans une autre boucle, la variable de cette boucle sera nommée **j**, puis **k** et ainsi de suite. C'est une sorte de convention implicite, que l'on retrouve dans la majorité des langages de programmation.

Code : JavaScript

```
for (var i = 0, nicks = '', nick; true; i++) {
  nick = prompt('Entrez un prénom :');

  if (nick) {
    nicks += nick + ' ';
  } else {
    break;
  }
}

alert('Il y a ' + i + ' prénoms :\n\n' + nicks);
```

Essayer !

La variable de boucle a été ajoutée dans le bloc d'*initialisation*. Le bloc d'*incréméntation* a lui aussi été modifié : on indique qu'il faut incrémenter la variable de boucle `i`. Ainsi, à chaque passage dans la boucle, `i` est incrémentée, ce qui va nous permettre de compter assez facilement le nombre de prénoms ajoutés.



Juste à titre d'information, les deux caractères `"\n"` sont là pour faire des sauts de ligne. Un `"\n"` permet de faire un saut de ligne, donc dans le code ci-dessus nous faisons deux sauts de ligne.

Portée des variables de boucle

En Javascript, il est déconseillé de déclarer des variables au sein d'une boucle (entre les accolades), pour des soucis de

performance (vitesse d'exécution) et de logique : il n'y a en effet pas besoin de déclarer une même variable à chaque passage dans la boucle ! Il est conseillé de déclarer les variables directement dans le bloc d'*initialisation*, comme montré dans les exemples de ce cours. Mais attention : une fois que la boucle est exécutée, la variable existe toujours, ce qui explique que dans l'exemple précédent on puisse récupérer la valeur de `i` une fois la boucle terminée. Ce comportement est différent de celui du populaire langage C, dans lequel une variable déclarée dans une boucle est "détruite" une fois la boucle exécutée.

Priorité d'exécution

Les 3 blocs qui constituent la boucle `for` ne sont pas exécutés en même temps :

- *initialisation* : juste avant que la boucle ne démarre. C'est comme si les instructions d'initialisation avaient été écrites juste avant la boucle, un peu comme pour une boucle `while` ;
- *condition* : avant chaque passage de boucle, exactement comme la condition d'une boucle `while` ;
- *incréméntation* : après chaque passage de boucle. Cela veut dire que, si vous faites un `break` dans une boucle `for`, le passage dans la boucle lors du `break` ne sera pas comptabilisé.

La boucle `for` est très utilisée en Javascript, bien plus que la boucle `while`, contrairement à d'autres langages de programmation. Comme nous le verrons par la suite, le fonctionnement même de Javascript fait que la boucle `for` est nécessaire dans la majorité des cas comme la manipulation des tableaux ainsi que des objets. Ce sera vu plus tard. Nous verrons aussi une variante de la boucle `for`, appelée `for in`, mais que l'on ne peut aborder maintenant car elle ne s'utilise que dans certains cas spécifiques.

Les fonctions

Les fonctions, voici un chapitre très important, tant pour sa longueur que pour les connaissances qu'il permet d'acquérir !

Au cours de ce chapitre vous apprendrez donc à découvrir ce que sont exactement les fonctions et comment en créer vous-même. Tout y passera, vous saurez gérer vos variables dans les fonctions, utiliser à fond le système des arguments, retourner des valeurs, créer des fonctions dites "anonymes", bref, tout ce qu'il vous faut pour faire des fonctions utiles !

Sur ce, on va tout de suite commencer parce que y'a du boulot !

Concevoir des fonctions

Dans les chapitres précédents vous avez découvert quatre fonctions : **alert**, **prompt**, **confirm** et **parseInt**. En les utilisant, vous avez pu constater que chacune de ces fonctions avait pour but de mener à bien une action précise, reconnaissable par un nom explicite.

Pour faire simple, si l'on devait associer une fonction à un objet de la vie de tous les jours, ce serait le moteur d'une voiture : vous tournez juste la clé pour démarrer le moteur et celui-ci fait déplacer tout son mécanisme pour renvoyer sa force motrice vers les roues. C'est pareil avec une fonction : vous l'appellez en lui passant éventuellement quelques paramètres, elle va ensuite exécuter le code qu'elle contient puis va renvoyer un résultat en sortie.

Le plus gros avantage d'une fonction est que vous pouvez exécuter un code assez long et complexe juste en appelant la fonction le contenant. Cela réduit considérablement votre code et le simplifie d'autant plus ! Seulement, vous êtes bien limités en utilisant seulement les fonctions natives du Javascript. C'est pourquoi il vous est possible de créer vos propres fonctions, c'est ce que nous allons étudier tout au long de ce chapitre.



Quand on parle de fonction ou variable native, il s'agit d'un élément déjà pré-intégré au langage que vous utilisez. Ainsi, l'utilisation des fonctions `alert/prompt/confirm` est permise car elles existent déjà de façon native.

Créer sa première fonction

On ne va pas y aller par quatre chemins, voici comment écrire une fonction :

Code : JavaScript

```
function myFunction(arguments) {  
    // Le code que la fonction va devoir exécuter.  
}
```

Décortiquons un peu tout ça et analysons un peu ce que l'on peut lire dans ce code :

- Le mot-clé **function** est présent à chaque déclaration de fonction. C'est lui qui permet de dire "Voilà, j'écris ici une fonction !".
- Vient ensuite le nom de votre fonction, ici "myFunction".
- S'ensuit un couple de parenthèses contenant ce que l'on appelle des arguments. Ces arguments servent à fournir des informations à la fonction lors de son exécution. Par exemple, avec la fonction **alert** quand vous lui passez en argument ce que vous voulez afficher à l'écran.
- Et vient enfin un couple d'accolades contenant le code que votre fonction devra exécuter.

Il est important de préciser que tout code écrit dans une fonction ne s'exécutera que si vous "appelez" cette dernière ("appeler" une fonction signifie l'"exécuter"). Sans ça, le code qu'elle contient ne s'exécutera jamais.



Bien entendu, tout comme les variables, les noms de fonctions sont limités aux caractères alphanumériques, aux chiffres et aux deux caractères suivants : `_` et `$`.

Bien, maintenant que vous connaissez un peu le principe d'une fonction, voici un petit exemple :

Code : JavaScript

```
function showMsg() {  
    alert('Et une première fonction, une !');  
}  
  
showMsg(); // On exécute ici le code contenu dans la fonction.
```


Essayer !

Dans ce code nous pouvons voir la déclaration d'une fonction nommée **showMsg** qui exécute elle-même une autre fonction qui n'est autre que **alert** avec un message prédéfini.

Bien sûr, tout code écrit dans une fonction ne s'exécute pas immédiatement, sinon l'intérêt serait nul. C'est pourquoi à la fin du code on appelle la fonction afin de l'exécuter, ce qui nous affiche le message souhaité.

Un exemple concret

Comme je le disais plus haut, l'intérêt d'une fonction réside notamment dans le fait de ne pas avoir à réécrire plusieurs fois le même code. Nous allons ici étudier un cas intéressant qui fait que l'utilisation d'une fonction va se révéler utile. Voici l'exemple :

Code : JavaScript

```
var result;

result = parseInt(prompt('Donnez le nombre à multiplier par 2 :'));
alert(result * 2);

result = parseInt(prompt('Donnez le nombre à multiplier par 2 :'));
alert(result * 2);
```

Comme vous pouvez le constater, on a écrit ici exactement deux fois le même code, ce qui nous donne un résultat peu efficace. On peut envisager d'utiliser une boucle mais si on veut afficher un texte entre les deux opérations comme ceci alors la boucle devient inutilisable :

Code : JavaScript

```
var result;

result = parseInt(prompt('Donnez le nombre à multiplier par 2 :'));
alert(result * 2);

alert('Vous en êtes à la moitié !');

result = parseInt(prompt('Donnez le nombre à multiplier par 2 :'));
alert(result * 2);
```

Notre solution, ici, est donc de faire appel au système des fonctions de la façon suivante :

Code : JavaScript

```
function byTwo() {
    var result = parseInt(prompt('Donnez le nombre à multiplier par 2 :'));
    alert(result * 2);
}

byTwo();

alert('Vous en êtes à la moitié !');

byTwo();
```

Essayer !

Concrètement, qu'est-ce qui a changé ? Eh bien, tout d'abord, nous avons créé une fonction qui contient le code à exécuter 2 fois (ou autant de fois qu'on le souhaite). Ensuite, nous faisons la déclaration de notre variable **result** directement dans notre fonction (oui, c'est possible, vous allez obtenir de plus amples explications juste ci-dessous) et surtout nous appelons deux fois notre fonction plutôt que de réécrire le code qu'elle contient.

Voilà l'utilité basique d'une fonction : éviter la répétition d'un code. Mais leur utilisation peut être largement plus poussée, continuons donc sur notre lancée !

La portée des variables

Bien, vous savez créer une fonction basique mais pour le moment vous ne pouvez rien faire de bien transcendant. Pour commencer à faire des fonctions vraiment utiles il vous faut apprendre à utiliser les arguments et les valeurs de retour mais on va tout d'abord devoir passer par une étude barbant des fonctions :

La portée des variables

Derrière ce titre se cache un concept assez simple à comprendre mais pas forcément simple à mettre en pratique car on peut facilement être induit en erreur si on ne fait pas attention. Tout d'abord, nous allons commencer par faire un constat assez flagrant à l'aide de deux exemples :

Code : JavaScript

```
var ohai = 'Hello world !';

function sayHello() {
  alert(ohai);
}

sayHello();
```

[Essayer !](#)

Ici, pas de problème, on déclare une variable dans laquelle on stocke du texte puis on crée une fonction qui se charge de l'afficher à l'écran et enfin on exécute cette dernière. Maintenant, on va légèrement modifier l'ordre des instructions mais l'effet doit normalement rester le même :

Code : JavaScript

```
function sayHello() {
  var ohai = 'Hello world !';
}

sayHello();

alert(ohai);
```

[Essayer !](#)

Alors ? Aucun résultat ? Ce n'est pas surprenant ! Il s'est produit ce que l'on appelle une erreur : En clair, le code s'est arrêté car il n'est pas capable d'exécuter ce que vous lui avez demandé. L'erreur en question nous indique que la variable **ohai** n'existe pas au moment de l'afficher avec la fonction **alert** alors que pourtant on a bien déclaré cette variable dans la fonction **sayHello**.



Et si je déclare la variable **ohai** en-dehors de la fonction ?

Là, ça fonctionnera ! Voilà tout le concept de la portée des variables : toute variable déclarée dans une fonction n'est utilisable que dans cette même fonction ! Ces variables spécifiques à une seule fonction ont un nom : les variables locales.

Les variables globales

À l'inverse des variables locales, celles déclarées en-dehors d'une fonction sont nommées les variables globales car elles sont accessibles partout dans votre code, y compris à l'intérieur de vos fonctions.

À ce propos, qu'est-ce qui se produirait si je créais une variable globale nommée **message** et une variable locale du même nom ? Essayons !

Code : JavaScript

```
var message = 'Ici la variable globale !';

function showMsg() {
  var message = 'Ici la variable locale !';
  alert(message);
}

showMsg();

alert(message);
```

Essayer !

Comme vous pouvez le constater, quand on exécute la fonction, la variable locale prend le dessus sur la variable globale de même nom pendant tout le temps de l'exécution de la fonction. Mais une fois la fonction terminée (et donc, la variable locale détruite) c'est la variable globale qui reprend ses droits.

Il existe une solution pour utiliser la variable globale dans une fonction malgré la création d'une variable locale de même nom, mais nous étudierons cela bien plus tard car ce n'est actuellement pas de notre niveau.

À noter que, dans l'ensemble, il est plutôt déconseillé de créer des variables globales et locales de même nom, cela est souvent source de confusion.

Les variables globales ? Avec modération !

Maintenant que vous savez faire la différence entre les variables globales et locales, il vous faut savoir quand est-ce qu'il est bon d'utiliser l'une ou l'autre. Car malgré le sens pratique des variables globales (vu qu'elles sont accessibles partout) elle sont parfois à proscrire car elles peuvent rapidement vous perdre dans votre code (et engendrer des problèmes si vous souhaitez distribuer une partie de votre code, mais vous découvrirez cela par vous-même). Voici un exemple de ce qu'il ne faut pas faire :

Code : JavaScript

```
var var1 = 2, var2 = 3;

function calculate() {
  alert(var1 * var2); // Affiche "6". (Sans blague ?)
}

calculate();
```

Dans ce code, vous pouvez voir que les variables **var1** et **var2** ne sont utilisées que pour la fonction **calculate** et pour rien d'autre, or ce sont ici des variables globales. Par principe, cette façon de faire est stupide, vu que ces variables ne servent qu'à la fonction **calculate** autant les déclarer dans la fonction de la façon suivante :

Code : JavaScript

```
function calculate() {
  var var1 = 2, var2 = 3;
```

```
    alert(var1 * var2); // Affiche "6". (Encore ?!)
  }

  calculate();
```

Ainsi, ces variables n'iront pas interférer avec d'autres fonctions qui peuvent utiliser des variables de même nom. Et surtout, ça reste quand même plus logique !



Juste un petit avertissement : je vois beaucoup de personnes râler sous prétexte que certains codes contiennent des variables globales. Les variables globales ne sont pas un mal, elles peuvent être utiles dans certains cas, il suffit juste de savoir s'en servir à bon escient. Et pour que vous arriviez à vous en servir correctement, il vous faut pratiquer 😊.

Bien, vous avez terminé la partie concernant la portée des variables. Faites bien attention ! Cela peut vous paraître simple comme ça mais il est facile de se faire piéger, je vous conseille de faire tous les tests qui vous passent par la tête afin de bien explorer toutes les possibilités et les éventuels pièges.

Les arguments et les valeurs de retour

Maintenant que vous connaissez le concept de la portée des variables, nous allons pouvoir aborder les arguments et les valeurs de retour. Ils permettent de faire communiquer vos fonctions avec le reste de votre code. Ainsi, les arguments permettent d'envoyer des informations à votre fonction tandis que les valeurs de retour représentent tout ce qui est retourné par votre fonction une fois que celle-ci a fini de travailler.

Les arguments

Créer et utiliser un argument

Comme dit ci-dessus, les arguments sont des informations envoyées à une fonction. Ces informations peuvent servir à beaucoup de choses, libre à vous de les utiliser comme vous le souhaitez. D'ailleurs, il vous est déjà arrivé d'envoyer des arguments à certaines fonctions, par exemple avec la fonction **alert** :

Code : JavaScript

```
// Voici la fonction alert sans argument, elle n'affiche rien.  
alert();  
  
// Et avec un argument, elle affiche ce que vous lui demandez.  
alert('Mon message à afficher');
```

Selon les fonctions, vous n'aurez parfois pas besoin de spécifier d'arguments, parfois il vous faudra en spécifier un, voir même plusieurs. Il existe aussi des arguments facultatifs que vous n'êtes pas obligé de spécifier.

Pour créer une fonction avec un argument, il vous suffit d'écrire de la façon suivante :

Code : JavaScript

```
function myFunction (arg) { // À noter que vous pouvez mettre un  
  espace entre le nom de la fonction et la parenthèse ouvrante si vous  
  le souhaitez  
  // Votre code.  
}
```

Ainsi, si vous passez un argument à cette même fonction, vous retrouverez dans la variable **arg** ce qui a été passé en argument, exemple :

Code : JavaScript

```
function myFunction(arg) { // Notre argument est la variable "arg".  
  // Une fois que l'argument a été passé à la fonction, vous allez  
  le retrouver dans la variable "arg".  
  alert('Votre argument : ' + arg);  
}  
  
myFunction('En voilà un beau test !');
```

Essayer !

Encore mieux ! Puisqu'un argument n'est qu'une simple variable, vous pouvez très bien lui passer ce que vous souhaitez, tel

que le texte écrit par un utilisateur :

Code : JavaScript

```
function myFunction(arg) { // Notre argument est la variable "arg".
    // Une fois que l'argument a été passé à la fonction, vous allez
    // le retrouver dans la variable "arg".
    alert('Votre argument : ' + arg);
}

myFunction(prompt('Que souhaitez vous passer en argument à la
fonction ?'));
```

Essayer !

Certains d'entre vous seront peut-être étonnés de voir la fonction **prompt** s'exécuter avant la fonction **myFunction**. Ceci est parfaitement normal, faisons un récapitulatif de l'ordre d'exécution de ce code :



- La fonction **myFunction** est déclarée, son code est donc enregistré en mémoire mais ne s'exécute pas tant qu'on ne l'appelle pas.
- À la dernière ligne, nous faisons appel à la fonction **myFunction** mais en lui passant un argument, la fonction va donc attendre de recevoir tous les arguments avant de s'exécuter.
- La fonction **prompt** s'exécute puis renvoie la valeur entrée par l'utilisateur, ce n'est qu'une fois cette valeur renvoyée que la fonction **myFunction** va pouvoir s'exécuter car les arguments auront enfin tous été reçus.
- Enfin la fonction **myFunction** s'exécute !

La portée des arguments

Si nous avons étudié dans la partie précédente la portée des variables ce n'est pas pour rien : cette portée s'applique aussi aux arguments. Ainsi, lorsqu'une fonction reçoit un argument, celui-ci est stocké dans une variable dont vous avez choisi le nom lors de la déclaration de la fonction. Voici, en gros, ce qui se passe quand un argument est reçu dans la fonction :

Code : JavaScript

```
function scope(arg) {
    // Au début de la fonction, la variable "arg" est créée avec le
    // contenu de l'argument qui a été passé à la fonction.

    alert(arg); // On peut maintenant utiliser l'argument comme on le
    // souhaite : l'afficher, le modifier, etc...

    // Une fois l'exécution de la fonction terminée, toutes les
    // variables contenant les arguments sont détruites.
}
```

Ce fonctionnement est exactement le même que lorsque vous créez vous-même une variable dans la fonction : elle ne sera accessible que dans cette fonction et nulle part ailleurs. Les arguments sont propres à leur fonction, ils ne serviront à aucune autre fonction.

Les arguments multiples

Si votre fonction a besoin de plusieurs arguments pour fonctionner il faudra les écrire de la façon suivante :

Code : JavaScript

```
function moar(first, second) {  
    // On peut maintenant utiliser les variables "first" et "second"  
    // comme on le souhaite :  
    alert('Votre premier argument : ' + first);  
    alert('Votre deuxième argument : ' + second);  
}
```

Comme vous pouvez le constater, les différents arguments sont séparés par une virgule, comme lorsque vous voulez déclarer plusieurs variables avec un seul mot-clé **var** ! Maintenant, pour exécuter notre fonction, il ne nous reste plus qu'à passer les arguments souhaités à notre fonction de cette manière :

Code : JavaScript

```
moar('Un !', 'Deux !');
```

Bien sûr, on peut toujours faire interagir l'utilisateur sans problème :

Code : JavaScript

```
moar(prompt('Entrez votre premier argument :'), prompt('Entrez votre  
deuxième argument :'));
```

[Essayer le code complet !](#)

Vous remarquerez d'ailleurs que la lisibilité de la ligne de code ci-dessus n'est pas très bonne, je vous conseille de modifier la présentation quand le besoin s'en fait ressentir. Pour ma part, j'aurais plutôt tendance à écrire cette ligne de cette manière :

Code : JavaScript

```
moar(  
    prompt('Entrez votre premier argument :'),  
    prompt('Entrez votre deuxième argument :')  
);
```

C'est plus propre, non ?

Les arguments facultatifs

Maintenant, admettons que l'on crée une fonction basique pouvant accueillir un argument mais que l'on ne le spécifie pas à l'appel de la fonction, que se passera-t-il ? Essayons ça tout de suite :

Code : JavaScript

```
function optional(arg) {  
    alert(arg); // On affiche l'argument non spécifié pour voir ce  
    // qu'il contient.  
}
```



```
optional();
```

Essayer !

undefined, voilà ce que l'on obtient, et c'est parfaitement normal ! La variable **arg** a été déclarée par la fonction mais pas initialisée car vous ne lui avez pas passé d'argument. Le contenu de cette variable est donc indéfini.



Mais, dans le fond, à quoi peut bien servir un argument facultatif ?

Prenons un exemple concret : imaginez que l'on décide de créer une fonction qui affiche à l'écran une fenêtre demandant à inscrire quelque chose (comme la fonction **prompt**). La fonction possède deux arguments : le premier doit contenir le texte à afficher dans la fenêtre, et le deuxième (qui est un booléen) autorise ou non l'utilisateur à quitter la fenêtre sans entrer de texte. Voici la base de la fonction :

Code : JavaScript

```
function prompt2(text, allowCancel) {  
    // Le code... Que l'on ne créera pas =p  
}
```

L'argument **text** est évidemment obligatoire vu qu'il existe une multitude de possibilités. En revanche, l'argument **allowCancel** est un booléen, il n'y a donc que deux possibilités :

- À **true**, l'utilisateur peut fermer la fenêtre sans entrer de texte.
- À **false**, l'utilisateur est obligé d'écrire quelque chose avant de pouvoir fermer la fenêtre.

Comme la plupart des développeurs souhaitent généralement que l'utilisateur entre une valeur, on peut considérer que la valeur la plus utilisée sera **false**.

Et c'est là que l'argument facultatif entre en scène ! Un argument facultatif est évidemment facultatif (sans blague ?!) mais doit généralement posséder une valeur par défaut dans le cas où l'argument n'a pas été rempli, dans notre cas ce sera **false**. Ainsi, on peut donc améliorer notre fonction de la façon suivante :

Code : JavaScript

```
function prompt2(text, allowCancel) {  
    if (typeof allowCancel === 'undefined') { // Souvenez-vous de  
        typeof, pour vérifier le type d'une variable  
        allowCancel = false;  
    }  
    // Le code... Que l'on ne créera pas =p  
}  
  
prompt2('Entrez quelque chose :'); // On exécute la fonction  
seulement avec le premier argument, pas besoin du deuxième.
```

De cette façon, si l'argument n'a pas été spécifié pour la variable **allowCancel** (comme ci-dessus) on attribue alors la valeur **false** à cette dernière.

Bien sûr, les arguments facultatifs ne possèdent pas obligatoirement une valeur par défaut, mais au moins vous saurez comment faire si vous en avez besoin.

Petit piège à éviter : inversons le positionnement des arguments de notre fonction : le second argument passe en premier et vice-versa. On se retrouve ainsi avec l'argument facultatif en premier et celui obligatoire en second, la première ligne de notre code est donc modifiée de cette façon :

Code : JavaScript

```
function prompt2(allowCancel, text) {
```

Imaginons maintenant que l'utilisateur de votre fonction souhaite ne remplir que l'argument obligatoire, il va donc écrire ceci :

Code : JavaScript

```
prompt2('Le texte');
```

Oui, mais le problème c'est qu'au final son texte va se retrouver dans la variable **allowCancel** au lieu de la variable **text** !

Alors quelle solution existe-t-il donc pour résoudre ce problème ? Aucune ! Vous devez impérativement mettre les arguments facultatifs de votre fonction en dernière position, vous n'avez pas le choix.

Les valeurs de retour

Comme leur nom l'indique, nous allons parler ici des valeurs que l'on peut retourner avec une fonction. Souvenez-vous pour les fonctions **prompt**, **confirm** et **parseInt**, chacune d'entre elles renvoyait une valeur que l'on stockait généralement dans une variable, nous allons donc apprendre à faire exactement la même chose ici mais pour nos propres fonctions.



Il est tout d'abord important de préciser que les fonctions ne peuvent retourner qu'une seule et unique valeur chacune, pas plus ! Il est possible de contourner légèrement le problème en renvoyant un *tableau* ou un *objet* mais vous étudierez le fonctionnement de ces deux éléments au chapitre suivant, on ne va donc pas s'y attarder dans l'immédiat.

Pour faire retourner une valeur à notre fonction, rien de plus simple, il suffit d'utiliser l'instruction **return** suivie de la valeur à retourner, exemple :

Code : JavaScript

```
function sayHello() {  
    return 'Bonjour !'; // L'instruction "return" suivie d'une valeur,  
    // cette dernière est donc renvoyée par la fonction.  
}  
  
alert(sayHello()); // Ici on affiche la valeur retournée par la  
// fonction "sayHello".
```

Maintenant essayons d'ajouter une ligne de code après la ligne contenant notre **return** :

Code : JavaScript

```
function sayHello() {
```

```
    return 'Bonjour !'; // L'instruction "return" suivie d'une valeur,
    // cette dernière est donc renvoyée par la fonction.
    alert('Attention ! Le texte arrive !');
}

alert(sayHello()); // Ici on affiche la valeur retournée par la
// fonction "sayHello".
```

Essayer !

Comme vous pouvez le constater, notre premier **alert** ne s'est pas affiché ! Cela s'explique par la présence du **return** : cette instruction met fin à la fonction, puis retourne la valeur. Pour ceux qui n'ont pas compris, la fin d'une fonction est tout simplement l'arrêt de la fonction à un point donné (dans notre cas, à la ligne du **return**) avec, éventuellement, le renvoi d'une valeur.

Ce fonctionnement explique d'ailleurs pourquoi on ne peut pas faire plusieurs renvois de valeurs pour une même fonction : si on écrit deux **return** à la suite, seul le premier sera exécuté puisque le premier **return** aura déjà mis un terme à l'exécution de la fonction.

Voilà tout pour les valeurs de retour. Leur utilisation est bien plus simple que pour les arguments mais reste vaste quand même, je vous conseille de vous entraîner à vous en servir car elles sont très utiles !

Les fonctions anonymes

Après les fonctions, voici les fonctions anonymes ! Ces fonctions particulières sont extrêmement importantes en Javascript ! Elles vous serviront pour énormément de chose : les objets, les évènements, les variables statiques, les closures, etc... Bref, des trucs que vous apprendrez plus tard 🤖. Non, vous n'allez pas en avoir l'utilité immédiatement, il vous faudra lire encore quelques chapitres supplémentaires pour commencer à vous en servir réellement. Toujours est-il qu'il vaut mieux commencer à les apprendre tout de suite, alors ne tardons pas et commençons dès maintenant !

Les fonctions anonymes : les bases

Comme leur nom l'indique, ces fonctions spéciales sont anonymes car elles ne possèdent pas de nom ! Voilà la seule et unique différence avec une fonction traditionnelle, ni plus, ni moins. Pour déclarer une fonction anonyme, il vous suffit d'écrire comme pour une fonction classique mais sans indiquer le nom :

Code : JavaScript

```
function (arguments) {  
  // Le code de votre fonction anonyme.  
}
```

Et là, normalement, une question devrait immédiatement vous venir à l'esprit : Comment fait-on pour exécuter cette fonction si elle ne possède pas de nom ?

Eh bien il existe de très nombreuses façons de faire ! Cependant, dans l'état actuel de vos connaissances, nous devons nous limiter à une seule solution : assigner notre fonction à une variable. Nous verrons les autres solutions au fil des chapitres suivants (je vous avais bien dit que vous ne sauriez pas encore exploiter tout le potentiel de ces fonctions).

Pour assigner une fonction anonyme à une variable, rien de plus simple :

Code : JavaScript

```
var sayHello = function () {  
  alert('Bonjour !');  
};
```

Ainsi, il ne nous reste plus qu'à appeler notre fonction par le biais du nom de la variable à laquelle nous l'avons affectée :

Code : JavaScript

```
sayHello(); // Affiche "Bonjour !".
```

[Essayer le code complet !](#)

On peut dire, en quelque sorte, que la variable **sayHello** est devenue une fonction ! En réalité, ce n'est pas le cas, nous devrions plutôt parler de *référence*, mais nous nous pencherons sur ce concept plus tard.

Retour sur l'utilisation des points-virgules

Certains d'entre vous auront sûrement noté le point-virgule après l'accolade fermante de la fonction dans le deuxième code, pourtant il s'agit d'une fonction, on ne devrait normalement pas en avoir besoin ! Et bien si !

En Javascript, il faut savoir distinguer dans son code les structures et les instructions. Ainsi, les fonctions, les conditions, les boucles, etc... sont des *structures*, tandis que tout le reste (assignation de variable, exécution de fonction, etc...) sont des *instructions*.

Bref, si j'écris :

Code : JavaScript

```
function structure() {  
    // Le code.  
}
```

Il s'agit d'une structure seule, pas besoin de point-virgule. Tandis que si j'écris :

Code : JavaScript

```
var instruction = 1234;
```

Il s'agit d'une instruction permettant d'assigner une valeur à une variable, le point-virgule est nécessaire. Maintenant, si j'écris de cette manière :

Code : JavaScript

```
var instruction = function() {  
    // Le code.  
};
```

Il s'agit alors d'une instruction assignant une structure à une variable, le point virgule est donc toujours nécessaire car, malgré la présence d'une structure, l'action globale reste bien une instruction.

Les fonctions anonymes : Isoler son code

Une utilisation intéressante des fonctions anonymes concerne l'isolation d'une partie de votre code, le but étant d'éviter qu'une partie de votre code n'affecte tout le reste.

Ce principe peut s'apparenter [au système de sandbox](#) mais en beaucoup moins poussé. Ainsi, il est possible de créer une zone de code isolée permettant la création de variables sans aucune influence sur le reste du code. L'accès au code en-dehors de la zone isolée reste toujours partiellement possible (ce qui fait donc que l'on ne peut pas réellement parler de "sandbox") mais, même si cet accès reste possible, ce système peut s'avérer très utile. Découvrons tout cela au travers de quelques exemples !

Commençons donc par découvrir comment créer une première zone isolée :

Code : JavaScript

```
// Code externe.  
  
(function() {  
    // Code isolé.  
})();
```

```
// Code externe.
```

Houlà, une syntaxe bizarre ! Il est vrai que ce code peut dérouter un petit peu au premier abord, je vais donc vous expliquer ça pas à pas.

Tout d'abord, on distingue une fonction anonyme :

Code : JavaScript

```
function() {  
  // Code isolé.  
}
```

Viens ensuite deux paires de parenthèses, une première paire encadrant la fonction et une deuxième paire suivant la première :

Code : JavaScript

```
(function() {  
  // Code isolé.  
})()
```

Pourquoi ces parenthèses ? Eh bien pour une raison simple : une fonction, lorsqu'elle est déclarée, n'exécute pas immédiatement le code qu'elle contient, elle attend d'être appelée. Or, nous, nous souhaitons exécuter ce code immédiatement ! La solution est donc d'utiliser ce couple de parenthèses.

Pour expliquer simplement, prenons l'exemple d'une fonction non-anonyme :

Code : JavaScript

```
function test() {  
  // Code.  
}  
  
test();
```

Comme vous pouvez le constater, pour exécuter la fonction **test** immédiatement après sa déclaration, nous avons dû l'appeler par la suite, mais il est possible de supprimer cette étape en utilisant le même couple de parenthèses que pour les fonctions anonymes :

Code : JavaScript

```
(function test() {  
  // Code.  
})();
```

Le premier couple de parenthèses va "isoler" la fonction pour que l'on puisse ensuite indiquer, avec le deuxième couple de parenthèses, que cette fonction doit être exécutée. Le code évolue donc de cette manière :

Code : JavaScript

```
// Ce code :  
  
    (function test() {  
  
    }) ();  
  
// Devient :  
  
    (test) ();  
  
// Qui devient :  
  
    test ();
```

Alors, pour une fonction non-anonyme, la solution sans ces deux couples de parenthèses est plus propre, mais pour une fonction anonyme il n'y a pas le choix : on ne peut plus appeler une fonction anonyme une fois déclarée (sauf si elle a été assignée à une variable), c'est pourquoi on doit utiliser ces parenthèses.

Une fois les parenthèses ajoutées, la fonction (qui est une structure) est exécutée, ce qui fait que l'on obtient une instruction, il faut donc ajouter un point-virgule :

Code : JavaScript

```
(function() {  
    // Code isolé.  
}) ();
```

Et voilà enfin notre code isolé !



Je vois juste une fonction anonyme immédiatement exécutée pour ma part... En quoi mon code est isolé ?

Notre fonction anonyme fonctionne exactement comme une fonction classique, sauf qu'elle ne possède pas de nom et qu'elle est exécutée immédiatement, ce sont les deux seules différences. Ainsi donc, la règle de la portée des variables s'applique aussi à cette fonction anonyme.

Bref, l'intérêt de cet "isolement de code" concerne la portée des variables : vous pouvez créer autant de variables que vous le souhaitez dans cette fonction avec les noms que vous souhaitez, tout sera détruit une fois que votre fonction aura fini de s'exécuter. Exemple (lisez bien les commentaires) :

Code : JavaScript

```
var test = 'noir'; // On crée une variable "test" contenant le mot  
"noir".  
  
(function() { // Début de la zone isolée.  
  
    var test = 'blanc'; // On crée la même variable avec le contenu  
"blanc" dans la zone isolée.  
  
    alert('Dans la zone isolée, la couleur est : ' + test);  
  
}) (); // Fin de la zone isolée. Les variables créées dans cette zone  
sont détruites.
```

```
alert('Dans la zone non-isolée, la couleur est : ' + test); // Le
texte final contient bien le mot "noir" vu que la "zone isolée" n'a
aucune influence sur le reste du code.
```

Essayer !



Une information un peu tardive, mais je ne voulais pas vous embrouiller dans toutes les explications ci-dessus : le premier couple de parenthèses n'est pas obligatoire pour exécuter la fonction, je vous conseille toutefois de les utiliser pour une question de lisibilité !

Allez, une dernière chose avant de finir ce chapitre !

Les zones isolées sont pratiques mais parfois on aimerait bien enregistrer dans le code global une ou deux valeurs générées dans une zone isolée. Pour cela il vous suffit de procéder de la même façon qu'avec une fonction classique, c'est-à-dire, comme ceci :

Code : JavaScript

```
var sayHello = (function() {
    return 'Yop !';
})();
alert(sayHello); // Affiche "Yop !".
```

Et voilà tout ! Le chapitre des fonctions est ENFIN terminé ! Rassurez-vous si vous avez eu du mal à le lire, moi je peux vous assurer que j'ai eu du mal à le rédiger 😊 !

Bref, ce chapitre est très conséquent, je vous conseille de le relire un autre jour si vous n'avez pas encore tout compris. Et pensez bien à vous exercer entre temps !

Les objets et les tableaux

Les objets sont une notion fondamentale en Javascript. Il s'agit ici du dernier gros chapitre de la partie I de ce cours, accrochez-vous !

Dans ce chapitre, nous verrons comment utiliser des objets, ce qui va nous permettre d'introduire l'utilisation des tableaux, un type d'objet bien particulier et très courant en Javascript. Nous verrons comment créer des objets simples, les *objets littéraux*, qui vont se révéler indispensables.

Introduction aux objets

Il a été dit dans l'introduction du cours que le Javascript est un langage *orienté objet*. Cela veut dire que le langage dispose d'**objets**.

Un objet est un concept, une idée ou une chose. Un objet possède une structure qui lui permet de pouvoir fonctionner et d'interagir avec d'autres objets. Javascript met à notre disposition des objets natifs, c'est-à-dire des objets directement utilisables. Vous avez déjà manipulés de tels objets sans le savoir : un nombre, une chaîne de caractères ou même un booléen.



Ce ne sont pas des variables ?

Si, mais en fait, une variable contient un objet. Par exemple, si je crée une chaîne de caractères, comme ceci :

Code : JavaScript

```
var myString = 'Ceci est une chaîne de caractères';
```

La variable **myString** contient un objet, et cet objet représente une chaîne de caractères. C'est la raison pour laquelle on dit que Javascript n'est pas un langage typé, car les variables contiennent toujours la même chose : un objet. Mais cet objet peut être de nature différente (un nombre, un booléen...).

Outre les objets natifs, Javascript nous permet de fabriquer nos propres objets. Ceci fera toutefois partie d'un chapitre à part, car la création d'objets est plus compliquée que l'utilisation des objets natifs.



Toutefois, attention, le Javascript n'est pas un langage orienté objet du même style que le C++, le C# ou le Java. Javascript est un langage *orienté objet par prototype*. Si vous avez déjà des notions de programmation orientée objet, vous verrez quelques différences, mais les principales viendront par la suite, lors de la création d'objets.

Que contiennent les objets ?

Les objets contiennent 3 choses distinctes :

- Un constructeur
- Des propriétés
- Des méthodes

Le constructeur

Le constructeur est un code qui est exécuté quand on utilise un nouvel objet. Le constructeur permet d'effectuer des actions comme définir diverses variables au sein même de l'objet (comme le nombre de caractères d'une chaîne de caractères). Tout cela est fait automatiquement pour les objets natifs, et on en reparlera quand nous aborderons l'orienté objet, dans lequel nous créerons nous-même ce genre de structures.

Les propriétés

Ce nombre de caractères va être placé dans une variable au sein de l'objet : c'est ce que l'on appelle une **propriété**. Une propriété est une variable contenue dans l'objet, et qui contient des informations nécessaires au fonctionnement de l'objet.

Les méthodes

Enfin, il est possible de modifier l'objet. Cela se fait par l'intermédiaire des **méthodes**. Les méthodes sont des fonctions contenues dans l'objet, et qui permettent de réaliser des opérations sur le contenu de l'objet. Par exemple, dans le cas d'une chaîne de caractères, il existe une méthode qui permet de mettre la chaîne de caractères en majuscules.

Exemple d'utilisation

On va créer une chaîne de caractères, pour ensuite afficher le nombre de caractères, et transformer la chaîne de majuscules. Soit la mise en pratique de la partie théorique que nous venons de voir.

Code : JavaScript

```
var myString = 'Ceci est une chaîne de caractères'; // On crée un
objet String

alert(myString.length); // On affiche le nombre de caractères, au
moyen de la propriété length

alert(myString.toUpperCase()); // On récupère la chaîne en
majuscules, avec la méthode toUpperCase()
```

Essayer !

On remarque quelque chose de nouveau dans ce code : la présence d'un point. Le point permet d'accéder aux propriétés et aux méthodes d'un objet. Ainsi, quand je fais `myString.length`, je demande à Javascript de me donner le nombre de caractères contenus dans `myString`. La propriété **length** contient ce nombre, qui a été défini quand j'ai créé l'objet. Ce nombre est également mis à jour quand on modifie la chaîne de caractères :

Code : JavaScript

```
var myString = 'Test';
alert(myString.length); // Affiche 4

myString = 'Test 2';
alert(myString.length); // Affiche 6 (l'espace est considéré comme
un caractère)
```

Essayer !

C'est pareil pour les méthodes : avec `myString.toUpperCase()`, je demande à Javascript de changer la *casse* de la chaîne, ici, tout mettre en majuscules. A l'inverse, la méthode `toLowerCase()` permet de tout mettre en minuscules.

Objets natifs déjà rencontrés

Nous en avons déjà rencontré 3 :

- **Number** : l'objet qui gère les nombres
- **Boolean** : l'objet qui gère les booléens
- **String** : l'objet qui gère les chaînes de caractères

Nous allons maintenant découvrir **Array**, l'objet qui gère les tableaux !

Les tableaux

Souvenez-vous : dans le chapitre sur les boucles, il était question de demander à l'utilisateur les prénoms de ses frères et sœurs. Les prénoms étaient concaténés dans une chaîne de caractères, puis affichés. A cause de cette méthode de stockage, à part réafficher les prénoms tels quels, on ne sait pas faire grand-chose.

C'est dans un tel cas que les **tableaux** entrent en jeu. Un tableau, ou plutôt un *array* en anglais, est une variable qui contient plusieurs valeurs, appelées **items**. Chaque item est accessible au moyen d'un *indice* (**index** en anglais) et dont la numérotation commence à partir de zéro. Voici un schéma représentant un tableau, qui stocke 5 items :

Indice	0	1	2	3	4
Donnée	Valeur 1	Valeur 2	Valeur 3	Valeur 4	Valeur 5

Les indices

Comme vous le voyez dans le tableau ci-dessus, la numérotation des items commence à 0 ! C'est très important, car il y aura toujours un décalage d'une unité : l'item numéro 1 porte l'indice 0, et donc le cinquième item porte l'indice 4.

Vous devrez donc faire très attention à ne pas vous emmêler les pincesaux, et à toujours garder ceci en tête, sinon ça vous posera problème.

Déclarer un tableau

On utilise bien évidemment **var** pour déclarer un tableau, mais la syntaxe pour définir les valeurs est spécifique :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline', 'Guillaume'];
```

Le contenu du tableau se définit entre crochets, et chaque valeur est séparée par une virgule. Les valeurs sont introduites comme pour des variables simples, c'est-à-dire qu'il faut des guillemets ou des apostrophes pour définir les chaînes de caractères :

Code : JavaScript

```
var myArray_a = [42, 12, 6, 3];  
var myArray_b = [42, 'Sébastien', 12, 'Laurence'];
```

On peut schématiser le contenu du tableau **myArray** ainsi :

Indice	0	1	2	3	4
Donnée	Sébastien	Laurence	Ludovic	Pauline	Guillaume

L'index numéro **0** contient **Sébastien**, tandis que le **2** contient **Ludovic**.

La déclaration par le biais de crochets est la syntaxe courte. Il se peut que vous rencontriez un jour une syntaxe plus longue qui est vouée à disparaître. Voici à quoi ressemble cette syntaxe :

Code : JavaScript



```
var myArray = new Array('Sébastien', 'Laurence', 'Ludovic',  
    'Pauline', 'Guillaume');
```

Le mot-clé **new** de cette syntaxe demande à Javascript de définir un nouvel array dont le contenu se trouve en paramètre (un peu comme une fonction). Vous verrez l'utilisation de ce mot-clé plus tard. En attendant il faut que vous sachiez que la syntaxe ci-dessus est dépréciée et qu'il est conseillé d'utiliser celle avec les crochets.

Récupérer et modifier des valeurs

Comment faire pour récupérer la valeur de l'index 1 de mon tableau ? Rien de plus simple, il suffit de spécifier l'index voulu, entre crochets, comme ceci :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline',  
    'Guillaume'];  
  
alert(myArray[1]); // Affiche Laurence
```

Sachant cela, il est facile de modifier le contenu d'un item du tableau :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline',  
    'Guillaume'];  
  
myArray[1] = 'Clarisse';  
  
alert(myArray[1]); // Affiche Clarisse
```

[Essayer !](#)

Opérations sur les tableaux

Ajouter et supprimer des items

La méthode **push()** permet d'ajouter un ou plusieurs items à un tableau :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence'];

myArray.push('Ludovic'); // Ajoute Ludovic à la fin du tableau
myArray.push('Pauline', 'Guillaume'); // Ajoute Pauline et Guillaume
à la fin du tableau
```

Comme dit dans la partie théorique sur les objets, les méthodes sont des fonctions, et donc, peuvent recevoir des paramètres. Ici, **push()** peut recevoir un nombre illimité de paramètres, et chaque paramètre représente un item à ajouter à la fin du tableau.

La méthode **unshift()** fonctionne comme **push()**, excepté que les items sont ajoutés au début du tableau. Cette méthode n'est pas très fréquente mais peut être utile.

Les méthodes **shift()** et **pop()** retirent respectivement le premier et le dernier élément du tableau.

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline',
              'Guillaume'];

myArray.shift(); // Retire Sébastien
myArray.pop(); // Retire Guillaume

alert(myArray); // Affiche Laurence, Ludovic, Pauline
```

[Essayer !](#)

Chaînes de caractères et tableaux

Les chaînes de caractères possèdent une méthode **split()** qui permet de les découper en un tableau, en fonction d'un séparateur. Prenons l'exemple suivant :

Code : JavaScript

```
var cousinsString = 'Pauline Guillaume Clarisse',
    cousinsArray = cousinsString.split(' ');

alert(cousinsString);
alert(cousinsArray);
```

La méthode **split()** va couper la chaîne de caractères à chaque fois qu'elle va rencontrer un espace. Les portions ainsi découpées sont placées dans un tableau, ici, **cousinsArray**.



Remarquez que quand vous affichez un array via **alert()** les éléments sont séparés par des virgules et il n'y a pas



d'apostrophes ou de guillemets. C'est dû à **alert()** qui pour afficher un objet (un tableau, un booléen, un nombre...) le transforme en une chaîne de caractères.

L'inverse de **split()**, c'est-à-dire créer une chaîne de caractères depuis un tableau, se nomme **join()** :

Code : JavaScript

```
var cousinsString_2 = cousinsArray.join('-');  
alert(cousinsString_2);
```

[Essayer le code complet !](#)

Ici, une chaîne de caractères va être créée, et les valeurs de chaque item seront séparées par un tiret. Si vous ne spécifiez rien comme séparateur, les "bouts" seront collés les uns aux autres.



Comme vous pouvez le constater, une méthode peut très bien retourner une valeur, tout comme le ferait une fonction indépendante d'un objet. D'ailleurs, on constate que **split** et **join** retournent toutes les deux le résultat de leur exécution, elles ne l'applique pas directement à l'objet.

Parcourir un tableau

Soyez attentifs, il s'agit ici d'un gros morceau ! Parcourir un tableau est quelque chose que vous allez faire très très fréquemment en Javascript, surtout plus tard, quand on verra comment interagir avec les éléments HTML.



Parcourir un tableau signifie passer en revue chaque item du tableau, soit pour afficher les items un à un, soit pour leur faire subir des modifications.

Nous avons vu dans le chapitre sur les boucles la boucle **for**. Elle va nous servir à parcourir les tableaux. La boucle **while** peut aussi être utilisée, mais **for** est la plus adaptée pour ça. Nous allons voir aussi une variante de la boucle **for** : la boucle **for in**.

Parcourir avec for

Reprenons le tableau avec les prénoms :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline', 'Guillaume'];
```

Le principe pour parcourir est simple : il faut faire autant d'itérations qu'il y a d'items. Le nombre d'items d'un tableau se récupère avec la propriété **length**, exactement comme pour le nombre de caractères d'une chaîne de caractères. A chaque itération, on va avancer d'un item dans le tableau, en utilisant la variable de boucle **i** : à chaque itération, elle s'incrémente, ce qui permet d'avancer dans le tableau item par item. Voici un exemple :

Code : JavaScript

```
for (var i = 0; i < myArray.length; i++) {  
    alert(myArray[i]);  
}
```

[Essayer le code complet !](#)

On commence par définir la *variable de boucle* **i**. Ensuite, on règle la condition pour que la boucle s'exécute tant qu'on n'a pas atteint le nombre d'items. `myArray.length` représente le nombre d'items du tableau, c'est-à-dire **5**.



Le nombre d'items est différent des indices. S'il y a 5 items, comme ici, les indices vont de 0 à 4. Autrement dit, le dernier item possède l'indice 4, et non 5. C'est très important pour la condition, car on pourrait croire à tort que le nombre d'items est égal à l'indice du dernier item.

Attention à la condition

J'ai volontairement mal rédigé le code précédent. En effet, dans le chapitre sur les boucles, j'ai dit que le second bloc d'une boucle **for**, le bloc de condition, était exécuté à chaque itération. Ici ça veut donc dire que `myArray.length` est calculé à chaque itération, ce qui, à part ralentir la boucle, n'a que peu d'intérêt puisque le nombre d'items du tableau ne change pas.

L'astuce est de définir une seconde variable, dans le bloc d'initialisation, qui contiendra `myArray.length`. On utilisera cette

variable pour la condition :

Code : JavaScript

```
for (var i = 0, c = myArray.length; i < c; i++) {  
    alert(myArray[i]);  
}
```



Utilise *c* comme nom de variable, qui signifie *count* (compter), mais vous pouvez utiliser ce que vous voulez.

Les objets littéraux

S'il est possible d'accéder aux items d'un tableau via leur indice, il peut être pratique d'y accéder au moyen d'un identifiant. Par exemple, dans le tableau des prénoms, l'item appelé **soeur** pourrait retourner la valeur **Laurence**.

Pour ce faire, on va créer nous-mêmes un array sous la forme d'un objet littéral. Voici un exemple :

Code : JavaScript

```
var family = {
  self:    'Sébastien',
  sister:  'Laurence',
  brother: 'Ludovic',
  cousin_1: 'Pauline',
  cousin_2: 'Guillaume'
};
```

Cette déclaration va créer un objet analogue à un tableau, excepté le fait que chaque item sera accessible au moyen d'un identifiant, ce qui donne, en schéma, ceci :

Identifiant	self	sister	brother	cousin_1	cousin_2
Donnée	Sébastien	Laurence	Ludovic	Pauline	Guillaume

La syntaxe d'un objet

Quelques petites explications s'imposent sur les objets, et tout particulièrement sur leur syntaxe. Précédemment dans ce chapitre vous avez vu que pour créer un array vide il suffisait d'écrire :

Code : JavaScript

```
var myArray = [];
```

Pour les objets c'est à peu près similaire sauf que l'on met des accolades à la place des crochets :

Code : JavaScript

```
var myObject = {};
```

Pour définir dès l'initialisation les items à ajouter à l'objet, il suffit d'écrire :

Code : JavaScript

```
var myObject = {
  item1 : 'Texte 1',
  item2 : 'Texte 2'
};
```

Comme écrit sur le code ci-dessus, il suffit de taper l'identifiant souhaité suivi de deux points et de la valeur à lui attribuer. La séparation des items se fait comme pour un tableau : avec une virgule.

Accès aux items

Revenons à notre objet littéral : Ce que nous avons créé est un objet, et les identifiants sont en réalité des **propriétés**, exactement comme la propriété **length** d'un tableau ou d'une chaîne de caractère. Donc, pour récupérer le nom de ma sœur, il suffit de faire :

Code : JavaScript

```
family.sister;
```

Il existe une autre manière, semblable à celle pour accéder aux items d'un tableau tout en connaissant l'indice, sauf qu'ici, on va simplement spécifier le nom de la propriété :

Code : JavaScript

```
family['sister'];
```

Ça va nous être particulièrement utile si l'identifiant est contenu dans une variable, comme ce sera le cas avec la boucle que nous allons voir par après. Exemple :

Code : JavaScript

```
var id = 'sister';  
alert(family[id]); // Affiche Laurence
```



Cette façon de faire convient également aux propriétés d'une chaîne de caractères ou d'un tableau. Ainsi, si mon tableau se nomme **myArray**, je peux faire `myArray['length']` pour récupérer le nombre d'items.

Ajouter des items

Ici, pas de méthode **push()**. En revanche, il est possible d'ajouter un item en spécifiant un identifiant qui n'est pas encore présent. Par exemple, si je veux ajouter mon oncle dans le tableau :

Code : JavaScript

```
family['uncle'] = 'Didier'; // Didier est ajouté, et est accessible  
via l'identifiant 'uncle'
```

Parcourir un objet avec **for in**

Il n'est pas possible de parcourir un objet littéral avec une boucle **for**. Normal, puisqu'une boucle **for** est surtout capable d'incrémenter une variable numérique, ce qui dans le cas d'un objet littéral ne nous est d'aucune utilité puisque nous devons posséder l'identifiant. En revanche, la boucle **for in** se révèle très intéressante !

La boucle **for in** est l'équivalent de la boucle `foreach` de PHP : elle est très simple et ne sert qu'à une seule chose : parcourir un objet.

Le fonctionnement est quasiment le même que pour un tableau, excepté qu'ici il suffit de fournir une "variable clé" qui reçoit un identifiant (au lieu d'un index) et de spécifier l'objet à parcourir :

Code : JavaScript

```
for (var id in family) { // On stocke l'identifiant dans "id" pour
    parcourir l'objet "family".

    alert(family[id]);
}
```

[Essayer le code complet !](#)



Pourquoi ne pas appliquer le **for in** sur les tableaux avec index ?

Parce que les tableaux se voient souvent attribuer des méthodes supplémentaires par certains navigateurs ou certains frameworks, ce qui fait que la boucle **for in** va vous les énumérer en même temps que les items du tableau.

Il y a aussi un autre facteur important à prendre en compte : la boucle **for in** est plus gourmande qu'une boucle **for** classique (source : [dev.opera.com/...](http://dev.opera.com/)).

Utilisation des objets littéraux

Les objets littéraux ne sont pas souvent utilisés, mais peuvent se révéler très utiles pour ordonner un code. On les utilise aussi dans les fonctions : les fonctions, avec **return**, ne savent retourner qu'une seule variable. Si on veut retourner plusieurs variables, il faut les placer dans un tableau et le retourner. Mais il est plus commode d'utiliser un objet littéral.

L'exemple classique est la fonction qui calcule des coordonnées d'un élément HTML sur une page Web. Il faut ici retourner les coordonnées x et y.

Code : JavaScript

```
function getCoords() {
    /* Script incomplet, juste pour l'exemple */

    return { x: 12, y: 21 };
}

var coords = getCoords();

alert(coords.x); // 12
alert(coords.y); // 21
```

La valeur de retour de la fonction **getCoords()** est mise dans la variable **coords**, et l'accès à **x** et **y** en est simplifié.

Exercice récapitulatif

Le chapitre suivant contient un TP, c'est-à-dire un Travail Pratique. Cela dit, avant de le débiter, je vous propose un petit exercice qui reprend de façon simple ce que nous avons vu dans ce chapitre.

Énoncé

Dans le chapitre sur les boucles, on a utilisé un script pour demander à l'utilisateur les prénoms de ses frères et sœurs. Les prénoms étaient alors stockés dans une chaîne de caractères. Pour rappel, voici ce code :

Code : JavaScript - Rappel

```
var nicks = '', nick;

while (true) {
  nick = prompt('Entrez un prénom :');

  if (nick) {
    nicks += nick + ' '; // Ajoute le nouveau prénom ainsi qu'un
    espace juste après
  } else {
    break; // On quitte la boucle
  }
}

alert(nicks); // Affiche les prénoms à la suite
```

Ce que je vous demande ici, c'est de stocker les prénoms dans un tableau. Pensez à la méthode **push()**. A la fin, il faudra afficher le contenu du tableau, avec une **alert()**, seulement si le tableau contient des prénoms ; en effet, ça ne sert à rien de l'afficher s'il ne contient rien. Pour l'affichage, séparez chaque prénom par un espace. Si le tableau ne contient rien, faites-le savoir à l'utilisateur, toujours avec une **alert()**.

Correction

Secret (cliquez pour afficher)

Code : JavaScript

```
var nicks = [], // Création du tableau vide
    nick;

while (nick = prompt('Entrez un prénom :')) { // Si la valeur
  assignée à la variable "nick" est valide (différente de "null")
  alors la boucle s'exécute
  nicks.push(nick); // Ajoute le nouveau prénom au tableau
}

if (nicks.length > 0) { // On regarde le nombre d'items
  alert(nicks.join(' ')); // Affiche les prénoms à la suite
} else {
  alert('Il n\'y a aucun prénom en mémoire !');
}
```

Essayer !

J'ai donc repris le code donné dans l'énoncé, et je l'ai modifié pour y faire intervenir un tableau : **nicks**. A la fin, je vérifie si le tableau contient des items, avec la condition `nicks.length > 0`. Le contenu du tableau est affiché avec la méthode **join()**, qui me permet de spécifier le séparateur. Car en effet, si j'avais fait `alert(nicks)`, les prénoms seraient séparés par une virgule.

TP : Convertir un nombre en toutes lettres

Nous arrivons enfin au premier TP de ce cours ! Ce dernier a pour but de vous faire réviser et mettre en pratique l'essentiel de ce que vous avez appris dans les chapitres précédents, je vous conseille donc fortement de le lire et d'essayer de faire l'exercice proposé.

Dans les grandes lignes, ce TP vous permettra de réviser l'utilisation des variables, conditions, boucles, fonctions et tableaux. Il vous permettra aussi d'approfondir vos connaissances sur l'identification et la conversion des nombres.



Ce TP est assez long et complexe (mais faisable, évidemment) ! C'est pourquoi vous risquez d'avoir quelques problèmes si vous ne savez pas comment détecter et corriger les erreurs dans vos codes source. Je vous conseille donc de lire le chapitre concernant le débogage dans la partie "Annexe" de ce cours avant de vous lancer.

Présentation de l'exercice

Ce TP sera consacré à un exercice bien particulier : la conversion d'un nombre en toutes lettres. Ainsi, si l'utilisateur entre le chiffre **41**, le script devra retourner ce nombre en toutes lettres : "quarante-et-un". Ne vous inquiétez pas : vous en êtes parfaitement capable, et je vais même vous aider un peu avant de vous donner le corrigé !

La marche à suivre

Pour mener à bien votre exercice, voici quelles sont les étapes que votre script devra suivre (vous n'êtes pas obligé de faire comme ça, mais c'est conseillé) :

- L'utilisateur est invité à entrer un nombre entre 0 et 999.
- Ce nombre doit être envoyé à une fonction qui se charge de le convertir en toutes lettres.
- Cette même fonction doit contenir un système permettant de séparer les centaines, les dizaines et les unités. Ainsi, la fonction doit être capable de voir que dans le nombre 365 il y a 3 centaines, 6 dizaines et 5 unités. Pour obtenir ce résultat, pensez bien à utiliser le modulo. Exemple : $365 \% 10 = 5$
- Une fois le nombre découpé en 3 chiffres, il ne reste plus qu'à les convertir en toutes lettres.
- Lorsque la fonction a fini de s'exécuter, elle renvoie le nombre en toutes lettres.
- Une fois le résultat de la fonction obtenu, il est affiché à l'utilisateur.
- Lorsque l'affichage du nombre en toutes lettres est terminé, on redemande un nouveau nombre à l'utilisateur.

L'orthographe des nombres

Dans le cas de notre exercice, je vais employer l'écriture des nombres "à la Française", c'est-à-dire la version la plus compliquée... Je vous conseille de faire comme moi, cela vous entraînera d'autant plus qu'en utilisant l'écriture Belge ou Suisse. D'ailleurs, puisque l'écriture des nombres en Français est assez tordue, [je vous conseille d'aller faire un tour ici](#) afin de vous remémorer les bases.

Pour information, je vais employer les règles orthographiques de 1990 (oui, vous lisez bien un cours de programmation !) donc j'écrirai les nombres de la manière suivante : *cinq-cent-cinquante-cinq*

Et non pas comme ça : *cinq cent cinquante-cinq*



Vu que j'ai déjà reçu pas mal de reproches sur cette façon d'écrire, je préfère vous prévenir que continuer d'être la cible de nombreuses plaintes.

À noter que vous n'êtes pas obligés de respecter toutes ces règles orthographiques, ce qui compte c'est que votre code puisse afficher les nombres en toutes lettres, les fautes orthographiques sont secondaires.

Tester et convertir les nombres

Afin que vous puissiez avancer sans trop de problèmes dans la lecture de votre code, il va falloir étudier l'utilisation des fonctions `parseInt` et `isNaN`.

Retour sur la fonction `parseInt`

Alors, concernant `parseInt`, il s'agit juste d'un approfondissement de son utilisation vu que vous savez déjà vous en servir. Cette fonction possède en réalité non pas un mais deux arguments, le deuxième est très utile dans certains cas, comme celui-ci par exemple :

Code : JavaScript

```
alert(parseInt('010')); // Affiche "8".
```

Et là vous constatez que le chiffre affiché n'est pas 10 comme souhaité mais 8 ! Pourquoi ? Tout simplement parce que la fonction **parseInt** supporte plusieurs bases de calcul, ainsi on peut lui dire que le premier argument est en binaire, la fonction nous retournera alors le nombre en base 10 (notre propre base de calcul, le système décimal) après avoir fait la conversion **base 2 (binaire) -> base 10 (décimal)**. Donc, si j'écris :

Code : JavaScript

```
alert(parseInt('100', 2)); // Affiche "4".
```

J'obtiens bien le chiffre 4 en binaire, c'est bon !



Mais tout à l'heure le deuxième argument n'était pas spécifié et pourtant on a eu une conversion aberrante, pourquoi ?

Tout simplement parce que si le deuxième argument n'est pas spécifié, la fonction **parseInt** va tenter de trouver elle-même la base arithmétique du nombre que vous avez passé en premier argument. Ce comportement est stupide vu que la fonction se trompe très facilement, la preuve : notre premier exemple sans le deuxième argument a interprété notre chiffre comme étant en base 8 (octale) simplement parce que la chaîne de caractères commence par un 0.

Bref, pour convertir correctement notre premier nombre, il nous faut indiquer à **parseInt** que ce dernier est en base 10, comme ceci :

Code : JavaScript

```
alert(parseInt('010', 10)); // Affiche "10".
```

Maintenant nous obtenons bien le chiffre 10 ! Rappelez-vous bien de ce deuxième argument, il vous servira pour le TP et, à n'en pas douter, dans une multitude de problèmes futurs !

La fonction *isNaN*

Jusqu'à présent, pour tester si une variable était un nombre, vous utilisiez l'instruction **typeof** , sauf qu'elle pose problème :

Code : JavaScript

```
var test = parseInt('test'); // Contient au final la valeur "NaN".  
alert(typeof test); // Affiche "number".
```

Voilà le problème : notre variable contient la valeur **NaN** qui signifie **Not a Number** et pourtant l'instruction **typeof** nous renvoie "number" en guise de type, c'est assez contradictoire non ? En fait, **typeof** est limité pour tester les nombres, à la place mieux vaut utiliser la fonction **isNaN** (**is Not a Number**) :

Code : JavaScript

```
var test = parseInt('test'); // Contient au final la valeur "NaN".  
alert(isNaN(test)); // Affiche "true".
```


Pourquoi **true** ? Tout simplement parce que **isNaN** renvoie **true** quand la variable n'est pas un nombre, et **false** dans le cas contraire.

Il est temps de se lancer !

Vous voilà maintenant prêts à vous lancer dans l'écriture de votre code, je précise de nouveau que les nombres à convertir vont de **0** à **999**, ni plus, ni moins. Après, rien ne vous empêche de faire plus si le cœur vous en dit, mais évitez de faire moins, vous manqueriez une belle occasion de vous entraîner correctement 😊.

Bon courage !

Correction

Allez hop ! C'est fini ! J'espère que vous avez réussi à faire quelque chose d'intéressant, normalement vous en êtes parfaitement capable ! Le sujet est, certes, un peu tordu mais vous avez largement assez de connaissances pour pouvoir le réaliser.

Toutefois, si vous avez bloqué alors que vous connaissiez très bien ce que l'on a appris dans les chapitres précédents, ne vous inquiétez pas : la programmation est un domaine où la logique règne en maître (bon, d'accord, pas tout le temps !), il faut donc de l'expérience pour en arriver à développer de façon logique. Ce que je veux dire, c'est que si vous n'avez pas réussi à coder l'exercice aujourd'hui, ce n'est pas un drame ! Faites une pause, essayez de faire des exercices plus simples et revenez ensuite sur cet exercice.

De toute manière, c'est bien simple, si vous arrivez à lire et comprendre le corrigé que je vous fournis ci-dessous alors vous êtes capables de réaliser cet exercice tout aussi bien que moi, voir même mieux !

Le corrigé complet

Voici donc la correction. Je précise que ce code n'est pas universel ! Il existe de nombreuses autres façons de coder cet exercice, et ma version n'est pas forcément la meilleure. Donc, si vous cherchez à recoder cet exercice après avoir lu le corrigé : ne refaites pas exactement comme moi ! Inventez votre propre solution, innovez selon vos idées ! Après tout, on dit qu'il existe autant d'algorithmes que de personnes dans le monde, car chacun possède sa propre façon de penser, vous devriez donc être capables de réaliser une version de ce code en réfléchissant par vous-même !

Code : JavaScript

```
function num2Letters(number) {

    if (isNaN(number) || number < 0 || 999 < number) {
        return 'Veuillez entrer un nombre entier compris entre 0 et 999.';
    }

    var units2Letters = ['', 'un', 'deux', 'trois', 'quatre', 'cinq', 'six', 'sept', 'huit', 'neuf', 'dix', 'onze', 'douze', 'treize', 'quatorze', 'quinze', 'seize', 'dix-sept', 'dix-huit', 'dix-neuf'],
        tens2Letters = ['', 'dix', 'vingt', 'trente', 'quarante', 'cinquante', 'soixante', 'soixante', 'quatre-vingt', 'quatre-vingt'];

    var units = number % 10,
        tens = (number % 100 - units) / 10,
        hundreds = (number % 1000 - number % 100) / 100;

    var unitsOut, tensOut, hundredsOut;

    if (number === 0) {
        return 'zéro';
    } else {

        // Traitement des unités

        unitsOut = (units === 1 && tens > 0 && tens !== 8 ? 'et-' : '') + units2Letters[units];

        // Traitement des dizaines

        if (tens === 1 && units > 0) {

            tensOut = units2Letters[10 + units];
            unitsOut = '';

        } else if (tens === 7 || tens === 9) {
```

```

        tensOut = tens2Letters[tens] + '-' + (tens === 7 && units ===
1 ? 'et-' : '') + units2Letters[10 + units];
        unitsOut = '';

    } else {

        tensOut = tens2Letters[tens];

    }

    tensOut += (units === 0 && tens === 8 ? 's' : '');

    // Traitement des centaines

    hundredsOut = (hundreds > 1 ? units2Letters[hundreds] + '-' :
'' ) + (hundreds > 0 ? 'cent' : '') + (hundreds > 1 && tens == 0 &&
units == 0 ? 's' : '');

    // Retour du total

    return hundredsOut + (hundredsOut && tensOut ? '-': '') +
tensOut + (hundredsOut && unitsOut || tensOut && unitsOut ? '-': '')
+ unitsOut;
    }

}

var userEntry;

while (userEntry = prompt('Indiquez le nombre à écrire en toutes
lettres (entre 0 et 999) :')) {

    alert(num2Letters(parseInt(userEntry, 10)));

}

```

[Essayer !](#)

Les explications

Vous avez le code, bien ! Mais maintenant il vous faut le comprendre, et on commence tout de suite !



Lisez bien les commentaires des codes ci-dessous, j'y explique énormément de choses concernant certains passages difficiles.

Le "squelette" du code

Un code doit toujours posséder ce que j'appelle un "squelette" autour duquel il peut s'articuler, c'est-à-dire un code de base contenant les principales structures de votre script (comme un objet, une boucle, une fonction, etc...) que l'on va pouvoir étoffer au fur et à mesure de l'avancée du code. Dans notre cas, nous avons besoin d'une fonction qui fera la conversion des nombres, ainsi que d'une boucle pour demander à l'utilisateur d'entrer un nouveau nombre sans jamais s'arrêter (sauf si l'utilisateur le demande), voici ce que ça donne :

Code : JavaScript

```
function num2Letters(number) { // "num2Letters" se lit "number to
```

```

letters", le "2" est une abréviation souvent utilisée en
programmation.
// Notre fonction qui s'occupera de la conversion du nombre. Elle
possède un argument lui permettant de recevoir le nombres en
question.
}

var userEntry; // Contient le texte entré par l'utilisateur.

while (userEntry = prompt('Indiquez le nombre à écrire en toutes
lettres (entre 0 et 999) :')) {
  /*
  Dans la condition de la boucle, on stocke le texte de l'utilisateur
dans la variable "UserEntry". Ce qui fait que si l'utilisateur
n'a rien entré (valeur nulle, donc équivalente à false) la condition
ne sera pas validée donc la boucle while ne s'exécutera pas et dans
le cas contraire la boucle s'exécutera.
  */
}

```

L'appel de la fonction de conversion

Notre boucle fonctionne ainsi : on demande un nombre à l'utilisateur qu'on envoie à la fonction de conversion. Voici comment procéder :

Code : JavaScript

```

while (userEntry = prompt('Indiquez le nombre à écrire en toutes
lettres (entre 0 et 999) :')) {

  /*
  On "parse" (en base 10) la chaîne de caractères de l'utilisateur
pour l'envoyer ensuite
à notre fonction de conversion qui renverra le résultat à la
fonction "alert".
  */

  alert(num2Letters(parseInt(userEntry, 10)));
}

```

Initialisation de la fonction de conversion

Le squelette de notre code est prêt et la boucle est complète, il ne nous reste plus qu'à faire la fonction, le plus gros du travail en fait ! Pour vous expliquer son fonctionnement, je vais la découper en plusieurs parties, ici nous allons voir l'initialisation qui concerne la vérification de l'argument **number** et la déclaration des variables nécessaires au bon fonctionnement de notre fonction.

Tout d'abord, nous devons vérifier l'argument reçu :

Code : JavaScript

```

function num2Letters(number) {

  if (isNaN(number) || number < 0 || 999 < number) { // Si
l'argument n'est pas un nombre (isNaN), ou si le nombre est
inférieur à 0, ou si il est supérieur à 999.
    return 'Veuillez entrer un nombre entier compris entre 0 et

```

```

999.']; // Alors on retourne un message d'avertissement.
    }
}

```

Puis nous déclarons les variables nécessaires à la bonne exécution de notre fonction :

Code : JavaScript

```

function num2Letters(number) {

    // Code de vérification de l'argument [...]

    /*
    Ci-dessous on déclare deux tableaux contenant les nombres en toutes
    lettres, un tableau pour les unités et un autre pour les dizaines.
    Le tableau des
    unités va du chiffre 1 à 19 afin de simplifier quelques opérations
    lors de la conversion du nombre en lettres. Vous comprendrez ce
    système par la suite.
    */

    var units2Letters = ['', 'un', 'deux', 'trois', 'quatre', 'cinq',
    'six', 'sept', 'huit', 'neuf', 'dix', 'onze', 'douze', 'treize',
    'quatorze', 'quinze', 'seize', 'dix-sept', 'dix-huit', 'dix-neuf'],
        tens2Letters = ['', 'dix', 'vingt', 'trente', 'quarante',
    'cinquante', 'soixante', 'soixante', 'quatre-vingt', 'quatre-
    vingt'];

    /*
    Ci-dessous on calcule le nombre d'unités, de dizaines et de
    centaines à l'aide du modulo.
    Le principe est simple : si on prend 365 % 10 on obtient le reste de
    la division par 10 qui est : 5. Voilà les unités.
    Ensuite, sur 365 % 100 on obtient 65, on soustrait les unités à ce
    nombre 65 - 5 = 60, et on divise par 10 pour obtenir 6, voilà les
    dizaines !
    Le principe est le même pour les centaines sauf que l'on ne
    soustrait pas seulement les unités mais aussi les dizaines.
    */

    var units = number % 10,
        tens = (number % 100 - units) / 10,
        hundreds = (number % 1000 - number % 100) / 100;

    // Et enfin on crée les trois variables qui contiendront les
    unités, les dizaines et les centaines en toutes lettres.

    var unitsOut, tensOut, hundredsOut;

}

```



Vous remarquerez que j'ai réduit le code de vérification de l'argument écrit précédemment à un simple commentaire. Je préfère faire ça afin que vous puissiez vous focaliser directement sur le code actuellement étudié, donc ne vous étonnez pas de voir des commentaires de ce genre.

Conversion du nombre en toutes lettres

Maintenant que notre fonction est entièrement initialisée, il ne nous reste plus qu'à attaquer le cœur de notre script : la conversion. Comment allons nous procéder ? Eh bien nous avons les unités, dizaines et centaines dans trois variables séparées ainsi que deux tableaux contenant les nombres en toutes lettres. Toutes ces variables vont nous permettre de nous simplifier la vie dans notre code, par exemple, si on souhaite obtenir les unités en toutes lettres, il ne nous reste qu'à faire ça :

Code : JavaScript

```
unitsOut = units2Letters[units];
```

Si ça paraît aussi simple c'est parce que notre code a été bien pensé dès le début et organisé de façon à pouvoir travailler le plus facilement possible. Il y a sûrement moyen de faire mieux, mais ce code simplifie quand même grandement les choses non ? Maintenant notre plus grande difficulté va être de se plier à toutes les règles orthographiques de la langue française.

Bref, continuons !

Vous remarquerez que dans nos tableaux il n'y a pas le chiffre "zéro", non, on va l'ajouter nous-même à l'aide d'une condition :

Code : JavaScript

```
function num2Letters(number) {  
    // Code de vérification de l'argument [...]  
    // Code d'initialisation [...]  
  
    if (number === 0) {  
        return 'zéro'; // Tout simplement ! Si le nombre vaut "0" alors  
on retourne "zéro", normal =p  
    }  
}
```

Maintenant nous devons nous occuper des unités :

Code : JavaScript

```
function num2Letters(number) {  
    // Code de vérification de l'argument [...]  
    // Code d'initialisation [...]  
  
    if (number === 0) {  
        return 'zéro'; // Tout simplement ! Si le nombre vaut 0 alors on  
retourne "zéro", normal =p  
    } else { // Si "number" est différent de 0 alors on lance la  
conversion complète du nombre  
  
        /*  
Ci-dessous on calcule les unités. La dernière partie du code (après  
le "+") ne doit normalement pas vous poser de problèmes. Mais pour  
la
```

```

condition ternaire je pense que vous voyez assez peu son utilité. En fait, elle va permettre d'ajouter "et-" à l'unité quand cette dernière vaudra 1 et que les dizaines seront supérieures à 0 et différentes de 8. Pourquoi ? Tout simplement parce que l'on ne dit pas "vingt-un" mais "vingt-et-un", cela s'applique à toutes les dizaines sauf à "quatre-vingt-un" (d'où le "tens !== 8").
*/

    unitsOut = (units === 1 && tens > 0 && tens !== 8 ? 'et-' : '')
+ units2Letters[units];

    }
}

```



Afin de vous simplifier la lecture du code, je vous ai mis toutes les ternaires entre parenthèses, même si en temps normal ce n'est pas vraiment nécessaire.

Viennent ensuite les dizaines. Attention, ça se corse pas mal !

Code : JavaScript

```

function num2Letters(number) {

    // Code de vérification de l'argument [...]

    // Code d'initialisation [...]

    if (number === 0) {

        return 'zéro'; // Tout simplement ! Si le nombre vaut 0 alors on retourne "zéro", normal =p

    } else { // Si "number" est différent de 0 alors on lance la conversion complète du nombre

        // Conversion des unités [...]

        /*
        La condition qui suit se charge de convertir les nombres allant de 11 à 19. Pourquoi cette tranche de nombres ? Parce qu'ils ne peuvent pas se décomposer (essayez de décomposer en toutes lettres le nombre "treize", vous m'en direz des nouvelles), ils nécessitent donc d'être mis un peu à part. Bref, leur conversion en lettres s'effectue donc dans la partie concernant les dizaines. Ainsi donc on va se retrouver avec, par exemple, "tensOut = 'treize';" donc au final on va effacer la variable "unitsOut" vu qu'elle ne servira à rien.
        */

        if (tens === 1 && units > 0) {

            tensOut = units2Letters[10 + units]; // Avec "10 + units" on obtient le nombre souhaité entre 11 et 19.
            unitsOut = ''; // Cette variable ne sert plus à rien, on la vide.

        }

    }
}

```

```

    /*
    La condition suivante va s'occuper des dizaines égales à 7 ou 9.
    Pourquoi ? Eh bien un peu pour la même raison que la
    précédente condition : on retrouve les nombres allant de 11 à 19. En
    effet, on dit bien "soixante-treize" et
    "quatre-vingt-treize" et non pas "soixante-dix-trois" ou autre
    bêtise du genre. Bref, cette condition est donc chargée,
    elle aussi, de convertir les dizaines et les unités. Elle est aussi
    chargée d'ajouter la conjonction "et-" si l'unité
    vaut 1, car on dit "soixante-et-onze" et non pas "soixante-onze".
    */

    else if (tens === 7 || tens === 9) {

        tensOut = tens2Letters[tens] + '-' + (tens === 7 && units === 1
? 'et-' : '') + units2Letters[10 + units];
        unitsOut = ''; // Cette variable ne sert plus à rien ici non
plus, on la vide.

    }

    //Et enfin la condition "else" s'occupe des dizaines que l'on
peut qualifier de "normales".

    else {

        tensOut = tens2Letters[tens];

    }

    // Dernière étape, "quatre-vingt" sans unité prend un "s" à la
fin : "quatre-vingts".

    tensOut += (units === 0 && tens === 8 ? 's' : '');

    }
}

```

Un peu tordu tout ça n'est-ce pas ? Rassurez-vous, vous venez de terminer le passage le plus difficile. On s'attaque maintenant aux centaines qui sont plus simples :

Code : JavaScript

```

function num2Letters(number) {

    // Code de vérification de l'argument [...]

    // Code d'initialisation [...]

    if (number === 0) {

        return 'zéro'; // Tout simplement ! Si le nombre vaut 0 alors on
retourne "zéro", normal =p

    } else { // Si "number" est différent de 0 alors on lance la
conversion complète du nombre

        // Conversion des unités [...]

```



```

        // Conversion des dizaines [...]

        /*
        La conversion des centaines se fait en une ligne et trois ternaires.
        Ces trois ternaires se chargent d'afficher un
        chiffre si nécessaire avant d'écrire "cent" (exemple : "trois-
        cents"), d'afficher ou non la chaîne "cent" (si il
        n'y a pas de centaines, on ne l'affiche pas, normal), et enfin
        d'ajouter un "s" à la chaîne "cent" si il n'y a ni
        dizaines, ni unités et que les centaines sont supérieures à 1.
        */

        hundredsOut = (hundreds > 1 ? units2Letters[hundreds] + '-' :
        '') + (hundreds > 0 ? 'cent' : '') + (hundreds > 1 && tens == 0 &&
        units == 0 ? 's' : '');
    }
}

```

Retour de la valeur finale

Et voilà ! Le système de conversion est maintenant terminé ! Il ne nous reste plus qu'à retourner toutes ces valeurs concaténées les unes aux autres avec un tiret.

Code : JavaScript

```

function num2Letters(number) {

    // Code de vérification de l'argument [...]

    // Code d'initialisation [...]

    if (number === 0) {

        return 'zéro'; // Tout simplement ! Si le nombre vaut 0 alors on
        retourne "zéro", normal =p

    } else { // Si "number" est différent de 0 alors on lance la
        conversion complète du nombre

        // Conversion des unités [...]

        // Conversion des dizaines [...]

        // Conversion des centaines [...]

        /*
        Cette ligne de code retourne toutes les valeurs converties en les
        concaténant les unes aux autres avec un tiret. Pourquoi y a-t-il
        besoin de ternaires ? Parce que si on n'en met pas alors on risque
        de retourner des valeurs du genre "-quatre-vingt-" juste parce
        qu'il n'y avait pas de centaines et d'unités.
        */

        return hundredsOut + (hundredsOut && tensOut ? '-': '') +
        tensOut + (hundredsOut && unitsOut || tensOut && unitsOut ? '-': '')
        + unitsOut;
    }
}

```

Conclusion

La correction et les explications de l'exercice sont enfin terminées ! Si vous avez trouvé ça dur, je vous rassure : ça l'était 😊 . Bon, certes, il existe des codes beaucoup plus évolués et compliqués que ça, mais pour quelqu'un qui débute c'est déjà beaucoup que de faire ça ! Si vous n'avez toujours pas tout compris, je ne peux que vous encourager à relire les explications ou bien refaire l'exercice par vous-même. Si, malgré tout, vous n'y arrivez pas, n'oubliez pas que le SdZ possède [une rubrique dédiée au Javascript dans le forum](#), vous y trouverez facilement de l'aide pour peu que vous expliquiez correctement votre problème.

Partie 2 : Modeler vos pages web

Le Javascript est un langage qui permet de créer ce que l'on appelle des pages DHTML. Ce dernier terme désigne les pages web qui modifient par elle-même leur propre contenu sans aucun appel à un serveur. C'est cette modification de la structure d'une page web que nous allons étudier dans cette partie.

Manipuler le code HTML (Partie 1/2)

Dans ce premier chapitre dédié à la manipulation du code HTML, nous allons voir le concept de DOM et comprendre comment naviguer entre les différents nœuds qui composent une page HTML. Il sera aussi question d'éditer le contenu et les attributs des éléments HTML, avant d'aborder le deuxième chapitre.

Le Document Object Model

Le Document Object Model (abrégé *DOM*) est une interface de programmation pour les documents XML et HTML.



Une interface de programmation, qu'on appelle aussi une **API** (pour *Application Programming Interface*), est un ensemble d'outils qui permettent de faire communiquer entre eux plusieurs programmes ou, dans le cas présent, différents langages. Le terme **API** reviendra souvent, quel que soit le langage de programmation que vous apprendrez.

Le DOM est donc une API qui s'utilise avec les documents XML et HTML, et qui va nous permettre, via Javascript, d'accéder au code XML et/ou HTML d'un document. C'est grâce au DOM que nous allons pouvoir modifier des *éléments HTML* (afficher ou masquer un `<div>` par exemple), en ajouter, en déplacer ou même en supprimer.

Petite note de vocabulaire : dans un cours sur le HTML, on parlera de balises HTML (une paire de balises en réalité : une balise ouvrante et une balise fermante). Ici, en Javascript, on parlera d'*élément HTML*, pour la simple raison que chaque paire de balises (ouvrante et fermante) est vue comme un objet. Par commodité et pour ne pas confondre, on parle d'élément HTML.

Petit historique

À l'origine, quand le Javascript a été intégré dans les premiers navigateurs (Internet Explorer et Netscape Navigator), le DOM n'était pas unifié, c'est-à-dire que les deux navigateurs possédaient un DOM différent. Et donc, pour accéder à un élément HTML, la manière de faire différait d'un navigateur à l'autre, ce qui obligeait les développeurs Web à coder différemment en fonction du navigateur. En bref, c'était un peu la jungle.

Le W3C a mis de l'ordre dans tout ça, et a publié une nouvelle spécification appelée **DOM 1** (DOM Level 1). Cette nouvelle spécification définit clairement ce qu'est le DOM, et surtout comment un document HTML ou XML est schématisé. Depuis lors, un document HTML ou XML est représenté sous la forme d'un arbre, ou plutôt hiérarchiquement. Ainsi, l'élément `<html>` contient deux éléments enfants : `<head>` et `<body>`, qui à leur tour contiennent d'autres éléments enfants.

Ensuite, la spécification DOM 2 a été publiée. La grande nouveauté de cette version 2 est l'introduction de la méthode `getElementById()` qui permet de récupérer un élément HTML ou XML en connaissant son ID.

L'objet window

Avant de véritablement parler du document, c'est-à-dire de la page Web, nous allons parler de l'objet `window`. L'objet `window` est ce que l'on appelle un objet global qui représente la fenêtre du navigateur. C'est à partir de cet objet que le Javascript est exécuté.

Si nous reprenons notre Hello World du début, nous avons :

Code : HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>

    <script type="text/javascript">

      alert('Hello world!');

    </script>

  </body>
```

```
</html>
```

Contrairement à ce qui a été dit dans ce cours, **alert()** n'est pas vraiment une fonction. Il s'agit en réalité d'une méthode appartenant à l'objet `window`. Mais l'objet `window` est dit *implicite*, c'est-à-dire qu'il n'y a généralement pas besoin de le spécifier. Ainsi, ces deux instructions produisent le même effet, à savoir, ouvrir une boîte de dialogue :

Code : JavaScript

```
window.alert('Hello world!');  
alert('Hello world!');
```

Puisqu'il n'est pas nécessaire de spécifier l'objet `window`, on ne le fait généralement pas sauf si cela est nécessaire, par exemple si on manipule des *frames* (nous verrons plus loin de quoi il s'agit).



Ne faites pas de généralisation hâtive : si **alert()** est une méthode de l'objet `window`, toutes les fonctions ne font pas nécessairement partie de l'objet `window`. Ainsi, les fonctions comme **isNaN()**, **parseInt()** ou encore **parseFloat()** ne dépendent pas d'un objet. Ce sont des **fonctions globales**.

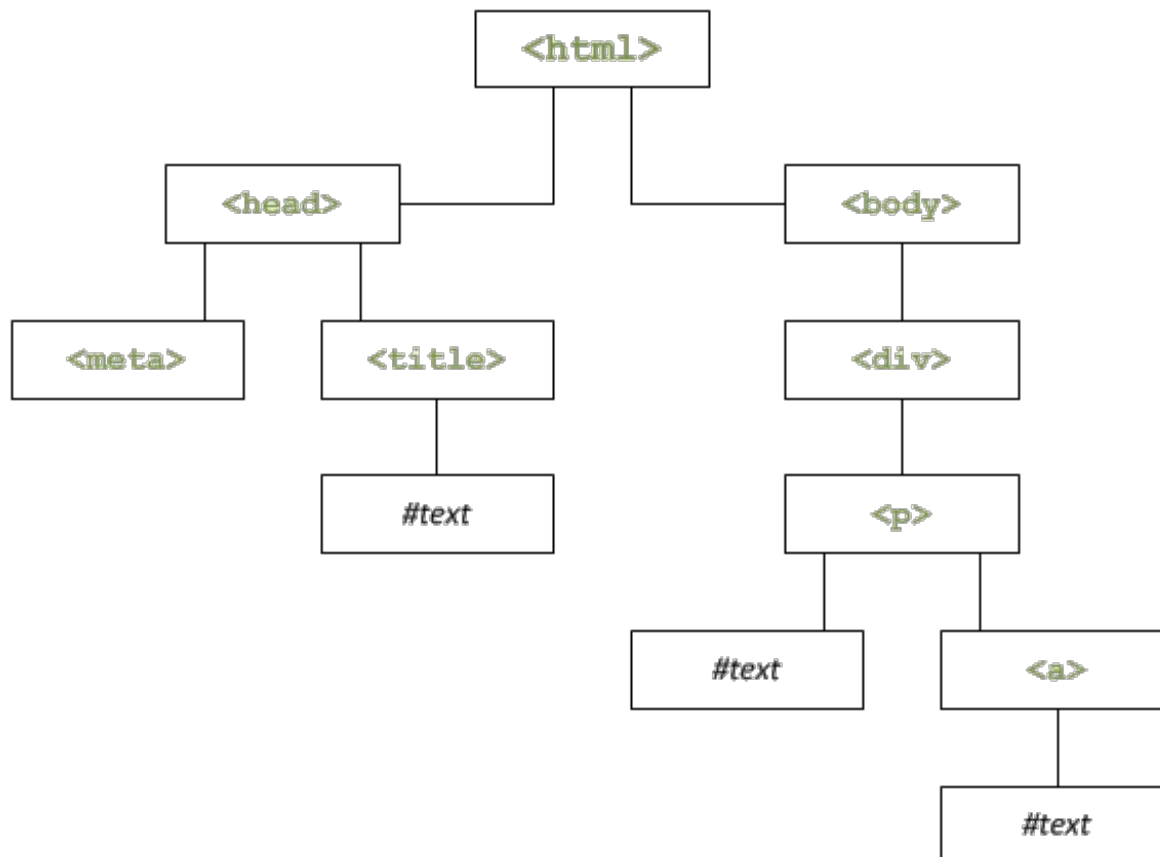
Le document

L'objet `document` est un sous-objet de `window`, l'un des plus utilisés. Et pour cause, il représente la *page Web*, ce que l'internaute voit dans la fenêtre de son navigateur. C'est grâce à cet élément là que nous allons pouvoir accéder aux éléments HTML et les modifier. Voyons donc, dans la sous-partie suivante, comment naviguer dans le document.

Naviguer dans le document

La structure DOM

Comme il a été dit précédemment, le DOM pose comme concept que la page Web est vue comme un arbre, comme une hiérarchie d'éléments. On peut donc schématiser une page Web simple comme ceci :



Voici le code source de la page :

Code : HTML

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>
  <body>
    <div>
      <p>Un peu de texte <a>et un lien</a></p>
    </div>
  </body>
</html>
```

Le schéma est plutôt simple : l'élément `<html>` contient deux éléments, appelés **enfants** : `<head>` et `<body>`. Pour ces deux enfants, `<html>` est l'élément **parent**. Chaque élément est appelé **nœud** (**node** en anglais). L'élément `<head>` contient lui aussi deux enfants : `<meta>` et `<title>`. `<meta>` ne contient pas d'enfant tandis que `<title>` en contient un, qui s'appelle `#text`. Comme son nom l'indique, `#text` est un élément qui contient du texte.

C'est important de bien saisir cette notion : le texte présent dans une page Web est vu par le DOM comme un nœud de type **#text**. Dans le schéma ci-dessus, l'exemple du paragraphe qui contient du texte et un lien illustre bien cela :

Code : HTML

```
<p>
  Un peu de texte
  <a>et un lien</a>
</p>
```

Si on va à la ligne après chaque nœud, on remarque clairement que l'élément **<p>** contient deux enfants : **#text** qui contient "Un peu de texte" et **<a>**, qui lui-même contient un enfant **#text** représentant "et un lien".

Accéder aux éléments

L'accès aux éléments HTML via le DOM est assez simple, mais demeure, actuellement, plutôt limité. L'objet document possède deux méthodes principales : **getElementById()** et **getElementsByTagName()**.

getElementById()

Cette méthode permet d'accéder à un élément en connaissant son ID qui est simplement l'attribut id de l'élément. Ça fonctionne comme ceci :

Code : HTML

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>

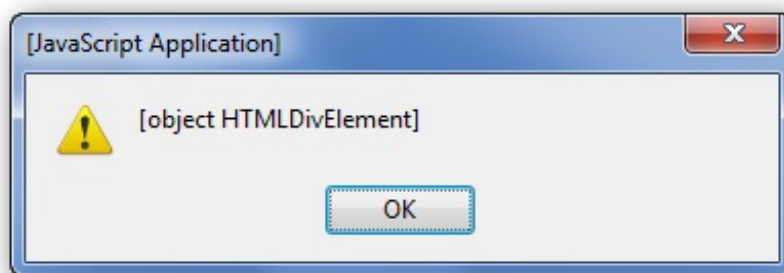
  <body>
    <div id="myDiv">
      <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
      var div = document.getElementById('myDiv');

      alert(div);
    </script>
  </body>
</html>
```

Essayer !

En exécutant ce code, le navigateur affiche ceci :



Il nous dit que **div** est un objet de type **HTMLDivElement**. En clair, c'est un élément HTML qui se trouve être un **<div>**, ce qui nous montre que le script fonctionne correctement.

getElementsByTagName()



Faites très attention dans le nom de cette méthode : il y a un *s* à *Elements*. C'est une source fréquente d'erreur.

Cette méthode permet de récupérer, sous la forme d'un tableau, tous les éléments de la famille. Si dans une page on veut récupérer tous les **<div>**, il suffit de faire comme ceci :

Code : JavaScript

```
var divs = document.getElementsByTagName('div');  
  
for (var i = 0, c = divs.length; i < c; i++) {  
    alert('Element n° ' + (i + 1) + ' : ' + divs[i]);  
}
```

Essayer !

La méthode retourne un tableau qui contient chaque élément, et pour accéder à chaque élément, il est nécessaire de parcourir le tableau avec une petite boucle.

Deux petites astuces :



- 1/ Cette méthode est accessible sur n'importe quel élément HTML et non pas seulement sur le `document`.
- 2/ En paramètre de cette méthode vous pouvez mettre une chaîne de caractères contenant une astérisque "*" qui récupèrera tous les éléments HTML.

getElementByName()

Cette méthode est semblable à **getElementByTagName()** et permet de ne récupérer que les éléments qui possèdent un attribut `name` que vous spécifiez. L'attribut `name` n'est utilisé qu'au sein des formulaires, et est déprécié depuis la spécification HTML 5 dans tout autre élément que celui d'un formulaire. Par exemple, vous pouvez vous en servir pour un élément **<input>** mais pas pour un élément **<map>**.

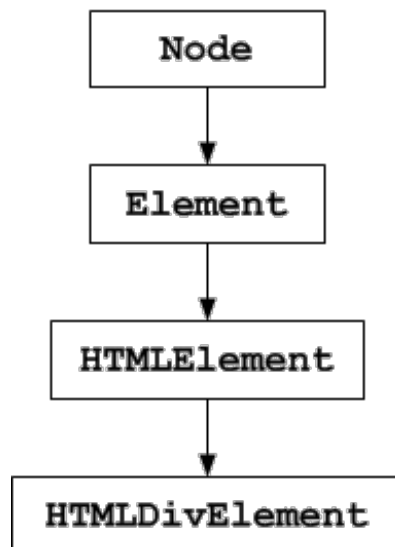
Sachez aussi que cette méthode est dépréciée en xHTML mais est maintenant standardisée pour l'HTML 5.

L'héritage des propriétés et des méthodes

Javascript voit les éléments HTML comme étant des objets, cela veut donc dire que chaque élément HTML possède des propriétés et des méthodes. Cependant faites bien attention parce que tous ne possèdent pas les mêmes propriétés et méthodes. Certaines sont néanmoins communes à tous les éléments HTML, car tous les éléments HTML sont d'un même type : le type **Node**, qui signifie nœud en anglais.

Notion d'héritage

On a vu qu'un élément `<div>` est un objet **HTMLDivElement**, mais un objet, en Javascript, peut appartenir à différents groupes. Ainsi, notre `<div>` est un **HTMLDivElement**, qui est un sous-objet d'**HTMLElement** qui est lui-même un sous-objet d'**Element**. **Element** est enfin un sous-objet de **Node**. Ce schéma est plus parlant :



L'objet **Node** apporte un certain nombre de propriétés et de méthodes qui pourront être utilisées depuis un de ses sous-objets. En clair, les sous-objets héritent des propriétés et méthodes de leurs objets parents.

Editer les éléments HTML

Maintenant que nous avons vu comment accéder à un élément, nous allons voir comment l'éditer. Les éléments HTML sont souvent composés d'attributs (l'attribut href d'un `<a>` par exemple), et d'un contenu, qui est de type `#text`. Le contenu peut aussi être un autre élément HTML.

Comme ça a été dit précédemment, un élément HTML est un objet qui appartient à plusieurs objets, et de ce fait, qui hérite des propriétés et méthodes de ses objets parents.

Les attributs

Via l'objet Element

Pour interagir avec les attributs, l'objet **Element** nous fournit deux méthodes, `getAttribute()` et `setAttribute()` permettant respectivement de récupérer et d'éditer un attribut. Le premier paramètre est le nom de l'attribut, et le deuxième, dans le cas de `setAttribute()` uniquement, est la nouvelle valeur à donner à l'attribut. Petit exemple :

Code : HTML

```
<body>
  <a id="myLink" href="http://www.un_lien_quelconque.com">Un lien
  modifié dynamiquement</a>

  <script>
    var link = document.getElementById('myLink');
    var href = link.getAttribute('href'); // On récupère l'attribut
    HREF

    alert(href);

    link.setAttribute('href', 'http://www.siteduzero.com'); // On
    édite l'attribut HREF
  </script>
</body>
```

On commence par récupérer l'élément `#myLink`, et on lit son attribut `href` via `getAttribute()`. Ensuite on modifie la valeur de l'attribut `href` avec `setAttribute()`. Le lien pointe maintenant vers <http://www.siteduzero.com>.

Les attributs accessibles

En fait, pour la plupart des éléments courants comme `<a>`, il est possible d'accéder à un attribut via une propriété. Ainsi, si on veut modifier la destination d'un lien, on peut utiliser la propriété `href`, comme ceci :

Code : HTML

```
<body>
  <a id="myLink" href="http://www.un_lien_quelconque.com">Un lien
  modifié dynamiquement</a>

  <script>
    var link = document.getElementById('myLink');
    var href = link.href;

    alert(href);

    link.href = 'http://www.siteduzero.com';
```

```
</script>
</body>
```

Essayer !



C'est cette façon de faire que l'on utilisera majoritairement pour les formulaires : pour récupérer/modifier la valeur d'un champ, on utilisera la propriété **value**.

La classe

On peut penser que pour modifier l'attribut class d'un élément HTML, il suffit d'utiliser `element.class`. Ce n'est pas possible, car le mot-clé **class** est réservé en Javascript, bien qu'il n'ait aucune utilité. A la place de **class**, il faudra utiliser **className**.

Code : HTML

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
    <style type="text/css">
      .blue {
        background: blue;
        color: white;
      }
    </style>
  </head>

  <body>
    <div id="myColoredDiv">
      <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
      document.getElementById('myColoredDiv').className = 'blue';
    </script>
  </body>
</html>
```

Dans l'exemple ci-dessus, on définit la classe CSS **blue** à l'élément **#myColoredDiv**, ce qui fait que cet élément sera affiché avec un arrière-plan bleu et une lettrine blanche.



Toujours dans le même cas, le nom **for** est réservé lui aussi en Javascript (pour les boucles). Vous ne pouvez donc pas modifier l'attribut HTML **for** d'un **<label>** en écrivant `element.for`, il faudra utiliser `element.htmlFor` à la place.

Faites attention : si votre élément comporte plusieurs classes (exemple : ``) et que vous récupérez la classe avec **className**, cette propriété ne retournera pas un tableau avec les différentes classes, mais bien la chaîne "external red u", ce qui n'est pas vraiment le comportement souhaité. Il vous faudra alors couper cette chaîne, avec la méthode **split()** pour obtenir un tableau, comme ceci :

Code : JavaScript

```
var classes = document.getElementById('myLink').className;
var classesNew = [];
classes = classes.split(' ');

for (var i = 0, c = classes.length; i < c; i++) {
    if (classes[i]) {
        classesNew.push(classes[i]);
    }
}

alert(classesNew);
```

Essayer !

Là, on récupère les classes, on découpe la chaîne, mais comme il se peut que plusieurs espaces soient présents entre chaque nom de classe, on vérifie chaque élément pour voir s'il contient quelque chose (s'il n'est pas vide). On en profite pour créer un nouveau tableau, **classesNew**, qui contiendra les noms des classes, sans "parasites".

Le contenu : innerHTML

La propriété **innerHTML** est spéciale et demande une petite introduction. Elle a été créée par Microsoft pour les besoins d'Internet Explorer. Cette propriété vient tout juste d'être standardisée au sein d'HTML 5. Bien que non-standard pendant des années, elle est devenue un standard parce que tous les navigateurs supportaient déjà cette propriété, et pas l'inverse comme c'est généralement le cas.

Récupérer du HTML

innerHTML permet de récupérer le code HTML enfant d'un élément sous forme de texte. Ainsi, si des balises sont présentes, **innerHTML** les retournera sous forme de texte :

Code : HTML

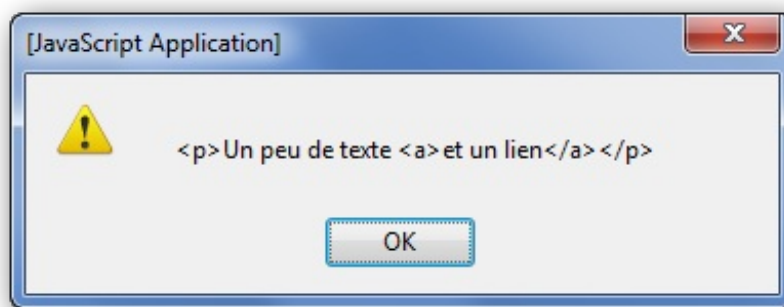
```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>

  <body>
    <div id="myDiv">
      <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
      var div = document.getElementById('myDiv');

      alert(div.innerHTML);
    </script>
  </body>
</html>
```

Nous avons donc bien une boîte de dialogue qui affiche le contenu de **myDiv**, sous forme de texte :



Ajouter ou éditer du HTML

Pour éditer ou ajouter du contenu HTML, il suffit de faire l'inverse, c'est-à-dire de définir un nouveau contenu :

Code : JavaScript

```
document.getElementById('myDiv').innerHTML = '<blockquote>Je mets  
une citation à la place du paragraphe</blockquote>';
```

Si vous voulez ajouter du contenu, et ne pas modifier le contenu déjà en place, il suffit d'utiliser += à la place de l'opérateur d'affectation :

Code : JavaScript

```
document.getElementById('myDiv').innerHTML += ' et <strong>une  
portion mise en emphase</strong>.';
```

Toutefois, une petite mise en garde : Il ne faut pas utiliser le += dans une boucle ! En effet, **innerHTML** ralentit considérablement l'exécution du code si l'on opère de cette manière, il vaut mieux donc concaténer son texte dans une variable pour ensuite ajouter le tout via **innerHTML**. Exemple :

Code : JavaScript

```
var text = '';  
  
while( /* condition */ ) {  
    text += 'votre_texte'; // On concatène dans la variable "text"  
}  
  
element.innerHTML = text; // Une fois la concaténation terminée, on  
ajoute le tout à 'element' via innerHTML
```

innerHTML et textContent

Penchons-nous maintenant sur deux propriétés analogues à `innerHTML` : `innerText` pour Internet Explorer et `textContent` pour les autres navigateurs.

innerText

`innerText` a aussi été introduite dans Internet Explorer, mais à la différence de sa propriété sœur `innerHTML`, elle n'a jamais été standardisée et n'est pas supportée par tous les navigateurs. Internet Explorer (pour toute version antérieure à la 9^e) ne supporte que cette propriété et non pas la version standardisée que nous verrons par la suite.

Le fonctionnement d'`innerText` est le même qu'`innerHTML` excepté le fait que seul le texte est récupéré, et non les balises. C'est pratique pour récupérer du contenu sans le balisage, petit exemple :

Code : HTML

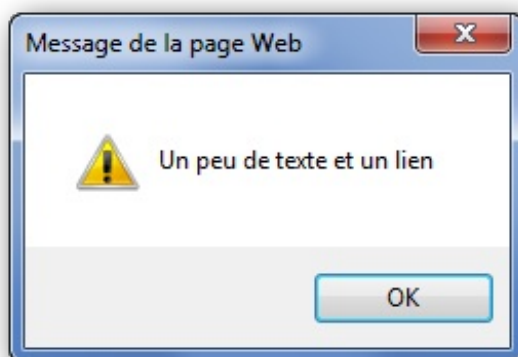
```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>

  <body>
    <div id="myDiv">
      <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
      var div = document.getElementById('myDiv');

      alert(div.innerText);
    </script>
  </body>
</html>
```

Et ce qui nous donne bien "Un peu de texte et un lien", sans les balises :



textContent

`textContent` est la version standardisée d'`innerText`, laquelle est reconnue par tous les navigateurs à l'exception d'Internet Explorer, bien que la version 9 la prenne aussi en charge. Le fonctionnement est évidemment le même. Il se pose maintenant

une question : comment faire un script qui fonctionne à la fois pour Internet Explorer et les autres navigateurs ? C'est ce que nous allons voir !

Tester le navigateur

Il est possible via une simple condition de tester si le navigateur prend en charge telle ou telle méthode ou propriété.

Code : HTML

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>

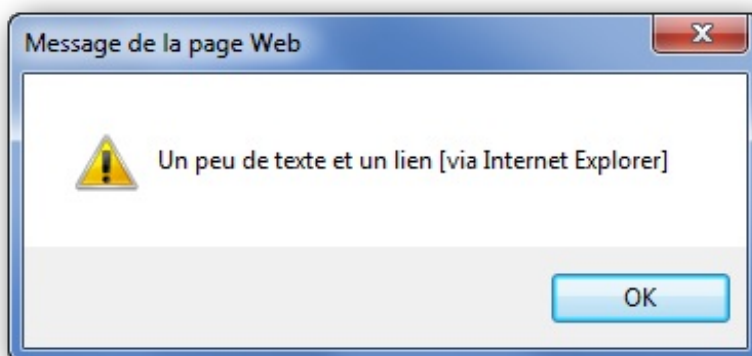
  <body>
    <div id="myDiv">
      <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
      var div = document.getElementById('myDiv');
      var txt = '';

      if (div.textContent) { // "textContent" existe ? Alors on s'en
sert !
        txt = div.textContent;
      } else if (div.innerText) { // "innerText" existe ? Alors on
doit être sous IE.
        txt = div.innerText + ' [via Internet Explorer]';
      } else { // Si aucun des deux n'existe, cela est sûrement dû
au fait qu'il n'y a pas de texte.
        txt = ''; // On met une chaîne de caractère vide.
      }

      alert(txt);
    </script>
  </body>
</html>
```

Il suffit donc de tester par le biais d'une condition si l'instruction fonctionne. Si **innerText** ne fonctionne pas, pas de soucis, on prend **textContent** :



Cela dit, ce code est quand même très long et redondant. Il est possible de le raccourcir de façon considérable :

Code : JavaScript

```
txt = div.textContent || div.innerText || '';
```

Manipuler le code HTML (Partie 2/2)

innerHTML a comme principale qualité d'être facile et rapide à utiliser et c'est la raison pour laquelle il est généralement privilégié par les débutants et même certains développeurs expérimentés. **innerHTML** a longtemps été une propriété non-standardisée, mais depuis HTML 5, elle est reconnue par le W3C et peut donc être utilisée.

Dans ce deuxième chapitre dédié à la manipulation du contenu, nous allons aborder la modification via DOM. On l'a déjà fait dans le premier chapitre, avec notamment **setAttribute()**, mais ici, il va s'agir de créer, de supprimer, de déplacer des éléments HTML. C'est un gros morceau du Javascript, pas toujours facile à assimiler. Vous allez me dire, si **innerHTML** suffit, pourquoi devoir s'embêter avec le DOM ? Tout simplement car le DOM est plus puissant et est nécessaire pour traiter du XML (ce que nous ferons par la suite).

Naviguer entre les nœuds

Nous avons vu précédemment qu'on utilisait les méthodes `getElementById()` et `getElementsByTagName()` pour accéder aux éléments HTML. Une fois qu'on a atteint un élément, il est possible de se déplacer de façon un peu plus précise, avec toute une série de méthodes et de propriétés que nous allons étudier ici.

La propriété `parentNode`

La propriété `parentNode` permet d'accéder à l'élément parent d'un élément. Regardez le code ci-dessous :

Code : HTML

```
<blockquote>
  <p id="myP">Ceci est un paragraphe !</p>
</blockquote>
```

Admettons qu'on doive accéder à `myP`, et que pour une autre raison on doive accéder à l'élément `<blockquote>`, qui est le parent de `myP`, il suffit d'accéder à `myP` puis à son parent, avec `parentNode` :

Code : JavaScript

```
var paragraph = document.getElementById('myP');
var blockquote = paragraph.parentNode;
```

`nodeType` et `nodeName`

`nodeType` et `nodeName` servent respectivement à vérifier le type d'un nœud et le nom d'un nœud.

`nodeType` retourne un nombre, qui correspond à un type de nœud. Voici un tableau qui liste les types possibles, ainsi que leurs numéros (les types courants sont mis en gras) :

Numéro	Type de nœud
1	Nœud élément
2	Nœud attribut
3	Nœud texte
4	Nœud pour passage CDATA (relatif au XML)
5	Nœud pour référence d'entité
6	Nœud pour entité
7	Nœud pour instruction de traitement
8	Nœud pour commentaire
9	Nœud document
10	Nœud type de document
11	Nœud de fragment de document

`nodeName` quant à lui, retourne simplement le nom de l'élément, en majuscule. Il est toutefois conseillé d'utiliser `toLowerCase()` (ou `toUpperCase()`) pour forcer un format de casse.

Code : JavaScript

```
var paragraph = document.getElementById('myP');
alert(paragraph.nodeType + '\n\n' +
paragraph.nodeName.toLowerCase());
```

Essayer !

Lister et parcourir des nœuds enfants

firstChild et lastChild

Comme leur nom le laisse présager, `firstChild` et `lastChild` servent respectivement à accéder au premier et au dernier enfant d'un nœud.

Code : HTML

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
  </head>

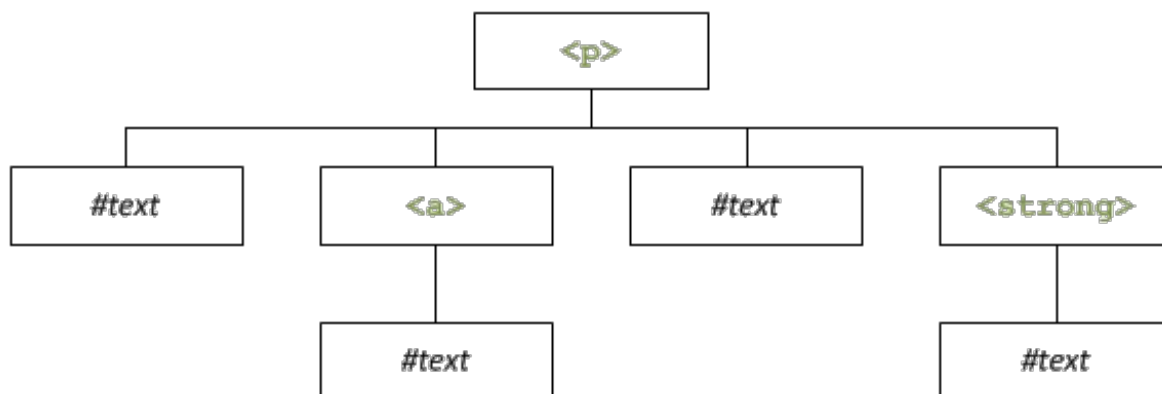
  <body>
    <div>
      <p id="myP">Un peu de texte, <a>un lien</a> et <strong>une
portion en emphase</strong></p>
    </div>

    <script type="text/javascript">
      var paragraph = document.getElementById('myP');
      var first = paragraph.firstChild;
      var last = paragraph.lastChild;

      alert(first.nodeName.toLowerCase());
      alert(last.nodeName.toLowerCase());
    </script>
  </body>
</html>
```

Essayer !

En schématisant l'élément `myP` ci-dessus, on obtient ceci :



Le premier enfant de `<p>` est un nœud textuel, alors que le dernier enfant est un élément ``.

nodeValue et data

Changeons de problème : il faut récupérer le texte du premier enfant, et le texte contenu dans l'élément ``, mais comment faire ?

Il faut soit utiliser la propriété `nodeValue` soit la propriété `data`. Si on recode le script ci-dessus, ça donne ceci :

Code : JavaScript

```

var paragraph = document.getElementById('myP');
var first = paragraph.firstChild;
var last = paragraph.lastChild;

alert(first.nodeValue);
alert(last.firstChild.data);

```

Essayer !

`first` contient le premier nœud, un nœud textuel. Il suffit de lui appliquer la propriété `nodeValue` (ou `data`) pour récupérer son contenu ; pas de difficulté ici. En revanche il y a une petite différence avec notre élément `` : vu que les attributs `nodeValue` et `data` ne s'appliquent que sur des nœuds textuels, il nous faut d'abord accéder au nœud textuel que contient notre élément, c'est-à-dire son nœud enfant. Pour cela, on utilise `firstChild`, et ensuite on récupère le contenu avec `nodeValue` ou `data`.

childNodes

La propriété `childNodes` retourne un tableau contenant la liste des enfants d'un élément. L'exemple suivant illustre le fonctionnement de cette propriété, de manière à récupérer le contenu des éléments enfants :

Code : HTML

```

<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var paragraph = document.getElementById('myP');
    var children = paragraph.childNodes;
  </script>

```

```

    for (var i = 0, c = children.length; i < c; i++) {

        if (children[i].nodeType === 1) { // C'est un élément HTML
            alert(children[i].firstChild.data);
        } else { // C'est certainement un noeud textuel
            alert(children[i].data);
        }
    }
</script>
</body>

```

Essayer !

nextSibling et previousSibling

nextSibling et **previousSibling** sont deux attributs qui permettent d'accéder respectivement au nœud suivant et au nœud précédent.

Code : HTML

```

<body>
  <div>
    <p id="myP">Un peu de texte, <a>un lien</a> et <strong>une
portion en emphase</strong></p>
  </div>

  <script type="text/javascript">
    var paragraph = document.getElementById('myP');
    var first = paragraph.firstChild;
    var next = first.nextSibling;

    alert(next.firstChild.data); // Affiche 'un lien'
  </script>
</body>

```

Essayer !

Dans l'exemple ci-dessus, on récupère le premier enfant de **myP**, et sur ce premier enfant, on utilise **nextSibling**, qui permet de récupérer l'élément **<a>**. Avec ça, il est même possible de parcourir les enfants d'un élément, en utilisant une boucle **while** :

Code : HTML

```

<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var paragraph = document.getElementById('myP');
    var child = paragraph.lastChild; // On prend le dernier
enfant

    while (child) {

        if (child.nodeType === 1) { // C'est un élément HTML
            alert(child.firstChild.data);
        } else { // C'est certainement un noeud textuel

```

```
    }  
    alert (child.data);  
  }  
  
  child = child.previousSibling; // A chaque tour de boucle, on  
  prend l'enfant précédent  
}  
</script>  
</body>
```

Essayer !

Pour changer un peu, la boucle tourne "à l'envers", car on commence par récupérer le dernier enfant et on chemine à reculons.

Attention aux nœuds vides

En considérant le code HTML suivant, on peut penser que l'élément `<div>` ne contient que 3 enfants `<p>` :

Code : HTML

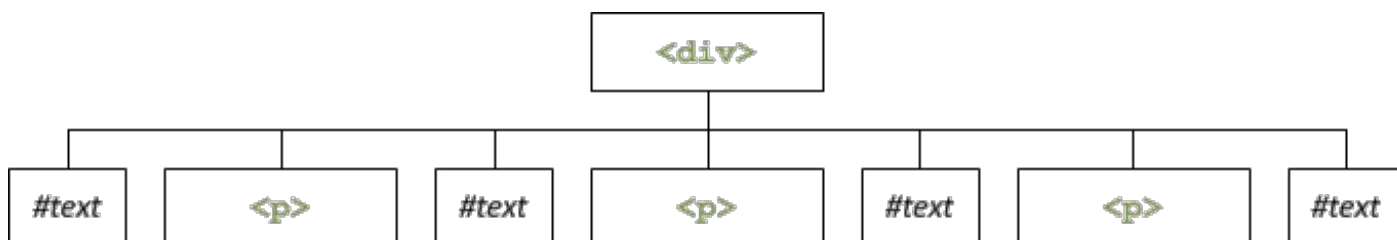
```
<div>  
  <p>Paragraphe 1</p>  
  <p>Paragraphe 2</p>  
  <p>Paragraphe 3</p>  
</div>
```

Mais attention, car le code ci-dessus est radicalement différent de celui-ci :

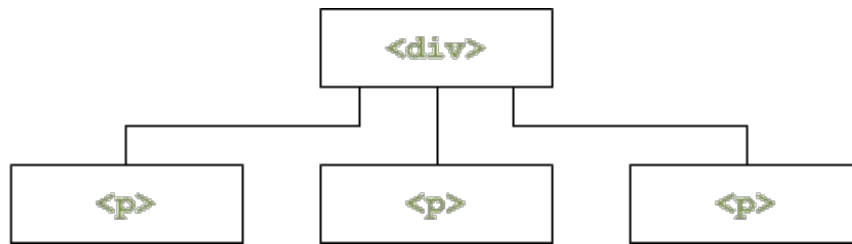
Code : HTML

```
<div><p>Paragraphe 1</p><p>Paragraphe 2</p><p>Paragraphe 3</p></div>
```

En fait, les espaces blancs entre les éléments, tout comme les retours à la ligne sont considérés comme des nœuds textuels (enfin, cela dépend des navigateurs) ! Ainsi donc, si l'on schématise le premier code, on obtient ceci :



Alors que le deuxième code peut être schématisé comme ça :



Heureusement, il existe une solution à ce problème ! Les attributs **firstElementChild**, **lastElementChild**, **nextElementSibling** et **previousElementSibling** ne retournent que des éléments HTML et permettent donc d'ignorer les nœuds textuels, ils s'utilisent exactement de la même manière que les attributs de base (**firstChild**, etc...). Cependant, ces attributs ne sont pas supportés par Internet Explorer 8 et ses versions antérieures et il n'existe pas d'alternative.

Créer et insérer des éléments

Ajouter des éléments HTML

Avec DOM, l'ajout d'un élément HTML se fait en 3 temps :

1. on crée l'élément ;
2. on lui affecte des attributs ;
3. on l'insère dans le document, et ce n'est qu'à ce moment-là qu'il sera "ajouté".

Création de l'élément

La création d'un élément se fait avec la méthode **createElement()**, un sous objet de l'objet racine, c'est-à-dire `document` (dans la majorité des cas) :

Code : JavaScript

```
var newLink = document.createElement('a');
```

On crée ici un nouvel élément `<a>` . Cet élément est créé, mais n'est PAS inséré dans le document, il n'est donc pas visible. Cela dit, on peut déjà "travailler" dessus, en lui ajoutant des attributs ou même des événements (que nous verrons dans le chapitre suivant).



Si vous travaillez dans une page Web, l'élément racine sera toujours `document`, sauf dans le cas de frames. La création d'éléments au sein de fichiers XML sera abordée plus tard.

Affectation des attributs

Ici, c'est comme nous avons vu précédemment : on définit les attributs, soit avec **setAttribute()**, soit directement avec les propriétés adéquates.

Code : JavaScript

```
newLink.id      = 'sdz_link';  
newLink.href   = 'http://www.siteduzero.com';  
newLink.title  = 'Découvrez le Site du Zéro !';  
newLink.setAttribute('tabindex', '10');
```

Insertion de l'élément

On utilise la méthode **appendChild()** pour insérer l'élément. "append child" signifie "ajouter un enfant", ce qui signifie qu'il nous faut connaître l'élément auquel on va ajouter l'élément créé. Considérons donc le code suivant :

Code : HTML

```
<!doctype html>  
<html>
```

```
<head>
  <meta charset="utf-8" />
  <title>Le titre de la page</title>
</head>

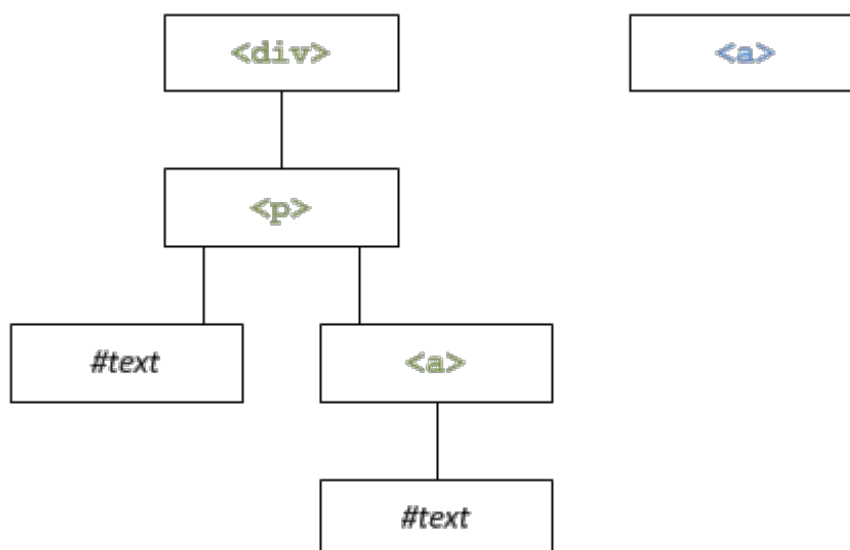
<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>
</body>
</html>
```

On va ajouter notre `<a>` dans l'élément `<p>` portant l'ID `myP`. Pour ce faire, il suffit de récupérer cet élément, et d'ajouter notre élément `<a>` via `appendChild()` :

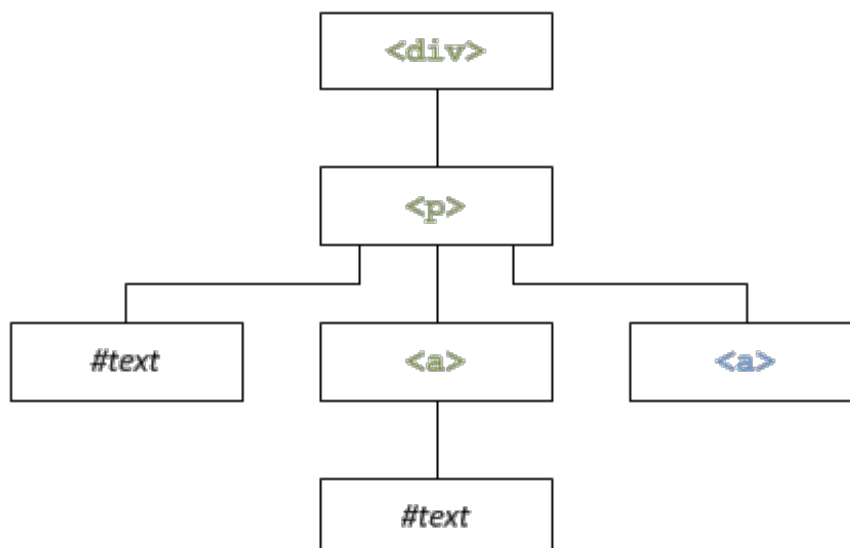
Code : JavaScript

```
document.getElementById('myP').appendChild(newLink);
```

Une petite explication s'impose ! Avant d'insérer notre élément `<a>`, la structure DOM du document ressemble à ceci :



On voit que l'élément `<a>` existe, mais n'est pas relié. C'est comme s'il était "libre" dans le document : il n'est pas encore placé. Le but est de le placer comme enfant de l'élément `myP`. La méthode `appendChild()` va alors déplacer notre `<a>` pour le placer en tant que dernier enfant de `myP` :



Cela veut dire qu'**appendChild()** insérera toujours l'élément en tant que dernier enfant, ce qui n'est pas toujours très pratique. Nous verrons après comment insérer un élément avant ou après un enfant donné.

Ajouter des nœuds textuels

L'élément a été inséré, seulement, il manque quelque chose : le contenu textuel ! La méthode **createTextNode()** sert à créer un nœud textuel (de type #text), qu'il nous suffira d'ajouter à notre élément fraîchement inséré, comme ceci :

Code : JavaScript

```

var newLinkText = document.createTextNode("Le Site du Zéro");
newLink.appendChild(newLinkText);

```

L'insertion se fait ici aussi avec **appendChild()**, sur l'élément newLink. Afin d'y voir plus clair, résumons le code :

Code : HTML

```

<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var newLink = document.createElement('a');

    newLink.id    = 'sdz_link';
    newLink.href  = 'http://www.siteduzero.com';
    newLink.title = 'Découvrez le Site du Zéro !';
    newLink.setAttribute('tabindex', '10');

    document.getElementById('myP').appendChild(newLink);

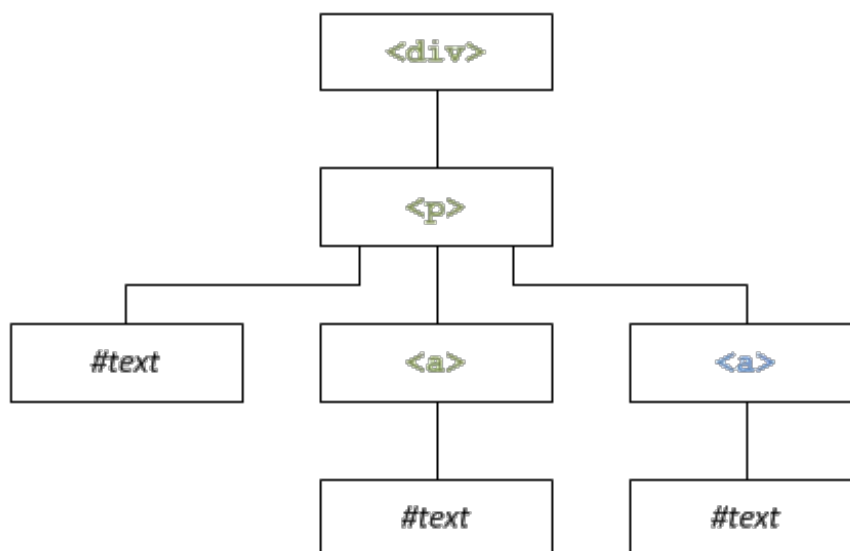
    var newLinkText = document.createTextNode("Le Site du Zéro");

    newLink.appendChild(newLinkText);
  </script>
</body>

```

Essayer !

Voici donc ce qu'on obtient, sous forme schématisée :



Il y a quelque chose à savoir : le fait d'insérer, via **appendChild()**, n'a aucune incidence sur l'ordre d'exécution des instructions. Cela veut donc dire qu'on peut travailler sur les éléments HTML et les nœuds textuels sans qu'ils soient au préalable insérés dans le document. Par exemple, on pourrait ordonner le code ci-dessus comme ceci :

Code : JavaScript

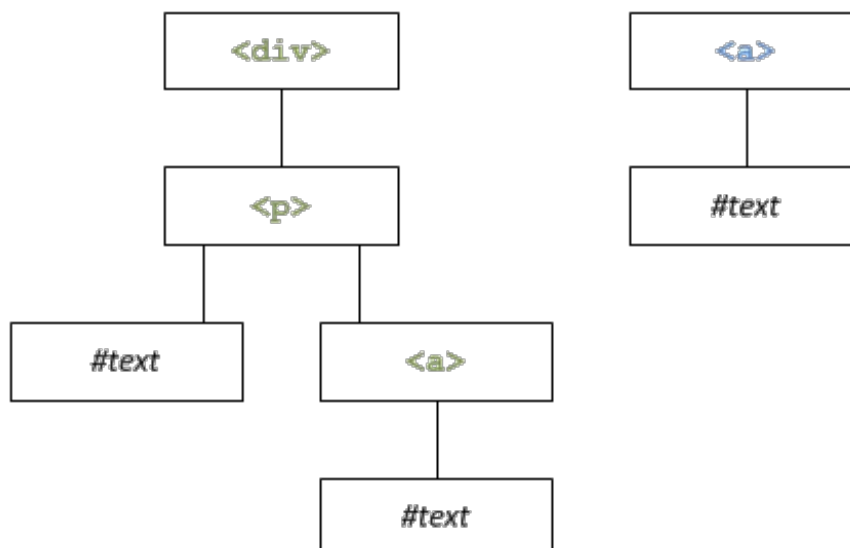
```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode("Le Site du Zéro");

newLink.id = 'sdz_link';
newLink.href = 'http://www.siteduzero.com';
newLink.title = 'Découvrez le Site du Zéro !';
newLink.setAttribute('tabindex', '10');

newLink.appendChild(newLinkText);

document.getElementById('myP').appendChild(newLink);
```

Ici, on commence par créer les deux éléments (le lien, et le nœud de texte), puis on affecte les variables au lien et on lui ajoute le nœud textuel. A ce stade-ci, l'élément HTML contient le nœud textuel, mais cet élément n'est pas encore inséré dans le document :



La dernière instruction insère alors le tout.



Nous vous conseillons d'organiser votre code comme le dernier exemple ci-dessus, c'est-à-dire avec la création de tous les éléments au début, puis les différentes opérations d'affection, et enfin, l'insertion des éléments les uns dans les autres, et pour terminer, l'insertion dans le document. Au moins comme ça c'est structuré, clair et surtout bien plus performant !

appendChild() retourne une *référence* (voir ci-après pour plus de détails) pointant sur l'objet qui vient d'être inséré. Cela peut servir dans le cas où vous n'avez pas déclaré de variable intermédiaire lors du processus de création de votre élément. Par exemple, le code ci-dessous ne pose pas de problème :

Code : JavaScript

```

var span = document.createElement('span');
document.body.appendChild(span);

span.innerHTML = 'Du texte en plus !';
  
```

En revanche, si vous retirez l'étape intermédiaire (la première ligne) pour gagner une ligne de code alors vous allez être embêté pour modifier le contenu :

Code : JavaScript

```

document.body.appendChild(document.createElement('span'));

span.innerHTML = 'Du texte en plus !'; // La variable "span"
n'existe plus... Le code plante.
  
```

La solution à ce problème est d'utiliser la référence retournée par **appendChild()** de la façon suivante :

Code : JavaScript

```

var span =
document.body.appendChild(document.createElement('span'));

span.innerHTML = 'Du texte en plus !'; // Là, tout fonctionne !
  
```



Notions sur les références

En Javascript et comme dans beaucoup de langages, le contenu des variables est "passé par valeur". Cela veut donc dire que si une variable **nick1** contient le prénom "Clarisse" et qu'on affecte cette valeur à une autre variable, la valeur est copiée dans la nouvelle. On obtient alors deux variables distinctes, contenant la même valeur :

Code : JavaScript

```
var nick1 = 'Clarisse';
var nick2 = nick1;
```

Si on modifie la valeur de **nick2**, la valeur de **nick1** reste inchangée : normal, les deux variables sont bien distinctes.

Les références

Outre le "passage par valeur", Javascript possède un "passage par référence". En fait, quand une variable est créée, sa valeur est mise en mémoire par l'ordinateur. Pour pouvoir retrouver cette valeur, elle est associée à une adresse que seul l'ordinateur connaît et manipule (on ne s'en occupe pas).

Quand on passe une valeur par référence, on transmet l'adresse de la valeur, ce qui va permettre d'avoir deux variables qui contiennent une même valeur !

Malheureusement, un exemple théorique d'un passage par référence n'est pas vraiment envisageable à ce stade du tutoriel, il faudra attendre d'aborder le chapitre sur la création d'objets. Cela dit, quand on manipule une page Web avec le DOM, on est confronté à des références, tout comme dans le chapitre suivant sur les événements.

Les références avec le DOM

Schématiser le concept de référence avec le DOM est assez simple : deux variables peuvent accéder au même élément. Regardez cet exemple :

Code : JavaScript

```
var newLink      = document.createElement('a');
var newLinkText = document.createTextNode('Le Site du Zéro');

newLink.id      = 'sdz_link';
newLink.src     = 'http://www.siteduzero.com';

newLink.appendChild(newLinkText);

document.getElementById('myP').appendChild(newLink);

// On récupère, via son ID, l'élément fraîchement inséré
var sdzLink = document.getElementById('sdz_link');

sdzLink.src = 'http://www.siteduzero.com/forum.html';

// newLink.src affiche bien la nouvelle URL :
alert(newLink.src);
```

La variable **newLink** contient en réalité une référence vers l'élément **<a>** qui a été créé. **newLink** ne contient pas l'élément, il contient une adresse qui pointe vers ce fameux **<a>** .

Une fois que l'élément HTML est inséré dans la page, on peut y accéder de nombreuses autres façons, comme avec

`getElementById()`. Quand on accède à un élément via `getElementById()`, on le fait aussi au moyen d'une référence.

Ce qu'il faut retenir de tout ça, c'est que les objets du DOM sont **toujours** accessibles par référence, et c'est la raison pour laquelle ce code ne fonctionne pas :

Code : JavaScript

```
var newDiv1 = document.createElement('div');  
var newDiv2 = newDiv1; // On tente de copier le <div>
```

Et oui, si vous avez tout suivi, **newDiv2** contient une référence qui pointe vers le même **<div>** que **newDiv1**. Mais comment dupliquer un élément alors ? Et bien il faut le cloner, et c'est ce que nous allons voir maintenant !

Cloner, remplacer, supprimer...

Cloner un élément

Pour cloner un élément, rien de plus simple : **cloneNode()**. Cette méthode accepte un paramètre booléen (**true** ou **false**) : si vous désirez cloner le nœud avec (**true**) ou sans (**false**) ses petits-enfants.

Petit exemple très simple : on crée un élément `<hr />` , et on en veut un deuxième, et donc on clone le premier :

Code : JavaScript

```
// On va cloner un élément créé :
var hr1 = document.createElement('hr');
var hr2 = hr1.cloneNode(false); // Il n'a pas d'enfants...

// Ici, on clone un élément existant :
var paragraph1 = document.getElementById('myP');
var paragraph2 = paragraph1.cloneNode(true);

// Et attention, l'élément est cloné, mais pas "inséré" tant que
l'on n'a pas appelé appendChild() :
paragraph1.parentNode.appendChild(paragraph2);
```

[Essayer !](#)



cloneNode() peut être utilisée tant pour cloner des nœuds textuels que des éléments HTML.

Remplacer un élément par un autre

Pour remplacer un élément par un autre, rien de plus simple, il y a **replaceChild()**. Cette méthode accepte deux paramètres : le premier est le nouvel élément, et le deuxième est l'élément à remplacer. Tout comme **cloneNode()**, cette méthode s'utilise sur tous les types de nœuds (éléments, nœuds textuels...).

Dans l'exemple suivant, le contenu textuel (pour rappel, il s'agit du premier enfant de `<a>`) du lien va être remplacé par un autre. La méthode **replaceChild()** est exécutée sur le `<a>`, c'est-à-dire le nœud parent du nœud à remplacer.

Code : HTML

```
<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var link = document.getElementsByTagName('a')[0];
    var newLabel= document.createTextNode('et un hyperlien');

    link.replaceChild(newLabel, link.firstChild);
  </script>
</body>
```

[Essayer !](#)

Supprimer un élément

Pour insérer un élément, on utilise **appendChild()**, et pour en supprimer un, on utilise **removeChild()**. Cette méthode prend en paramètre le nœud enfant à retirer. Si on se calque sur le code HTML de l'exemple précédent, le script ressemble à ceci :

Code : JavaScript

```
var link = document.getElementsByTagName('a')[0];  
link.parentNode.removeChild(link);
```



Il n'y a pas besoin de récupérer **myP** (l'élément parent) avec **getElementById()**, autant le faire directement avec **parentNode**.

Autres actions

Vérifier la présence d'éléments enfants

Rien de plus facile pour vérifier la présence d'éléments enfants : **hasChildNodes()**. Il suffit d'utiliser cette méthode sur l'élément de votre choix : si cet élément possède au moins un enfant, la méthode renverra **true** :

Code : HTML

```
<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var paragraph = document.getElementsByTagName('p')[0];

    alert(paragraph.hasChildNodes()); // Affiche true
  </script>
</body>
```

Insérer à la bonne place : insertBefore()

La méthode **insertBefore()** permet d'insérer un élément avant un autre. Elle reçoit deux paramètres : le premier est l'élément à insérer, tandis que le deuxième est l'élément avant lequel l'élément va être inséré :

Code : HTML

```
<body>
  <div>
    <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>

  <script type="text/javascript">
    var paragraph = document.getElementsByTagName('p')[0];
    var emphasis = document.createElement('em'),
        emphasisText = document.createTextNode(' en emphase légère
');

    emphasis.appendChild(emphasisText);

    paragraph.insertBefore(emphasis, paragraph.lastChild);
  </script>
</body>
```

Essayer !



Comme pour **appendChild()**, cette méthode s'applique sur l'élément parent.

Une bonne astuce : insertAfter()

Javascript met à disposition **insertBefore()**, mais pas **insertAfter()**. C'est dommage, car parfois c'est assez utile. Qu'à cela ne tienne, créons donc une telle fonction.

Malheureusement, il ne nous est pas possible, à ce stade-ci du tutoriel, de créer une méthode, qui s'appliquerait comme ceci :

Code : JavaScript

```
element.insertAfter(newElement, afterElement)
```

Non, il va falloir nous contenter d'une "simple" fonction :

Code : JavaScript

```
insertAfter(newElement, afterElement)
```

Algorithme

Pour insérer après un élément, on va d'abord récupérer l'élément parent. C'est logique, puisque l'insertion de l'élément va se faire soit via **appendChild()**, soit via **insertBefore()** : si on veut ajouter notre élément après le dernier enfant, c'est simple, il suffit d'appliquer **appendChild()**. Par contre, si l'élément après lequel on veut insérer notre élément n'est pas le dernier, on va utiliser **insertBefore()** en ciblant l'enfant suivant, avec **nextSibling** :

Code : JavaScript

```
function insertAfter(newElement, afterElement) {  
    var parent = afterElement.parentNode;  
  
    if (parent.lastChild === afterElement) { // Si le dernier élément  
        est le même que l'élément après lequel on veut insérer, il suffit de  
        faire appendChild()  
        parent.appendChild(newElement);  
    } else { // Dans le cas contraire, on fait un insertBefore() sur  
        l'élément suivant  
        parent.insertBefore(newElement, afterElement.nextSibling);  
    }  
}
```

Minis-TD : recréer une structure DOM

Afin de s'entraîner à jouer avec le DOM, voici quatre petits exercices. Pour chaque exercice, une structure DOM sous forme de code HTML vous est donnée, et il vous est demandé de re-créez cette structure en utilisant le DOM.



Notez que la correction donnée est une solution possible, et qu'il en existe d'autres. Chaque codeur possède un style de code, une façon de réfléchir, d'organiser et de présenter son code, surtout ici, où les possibilités sont nombreuses.

Premier exercice

Énoncé

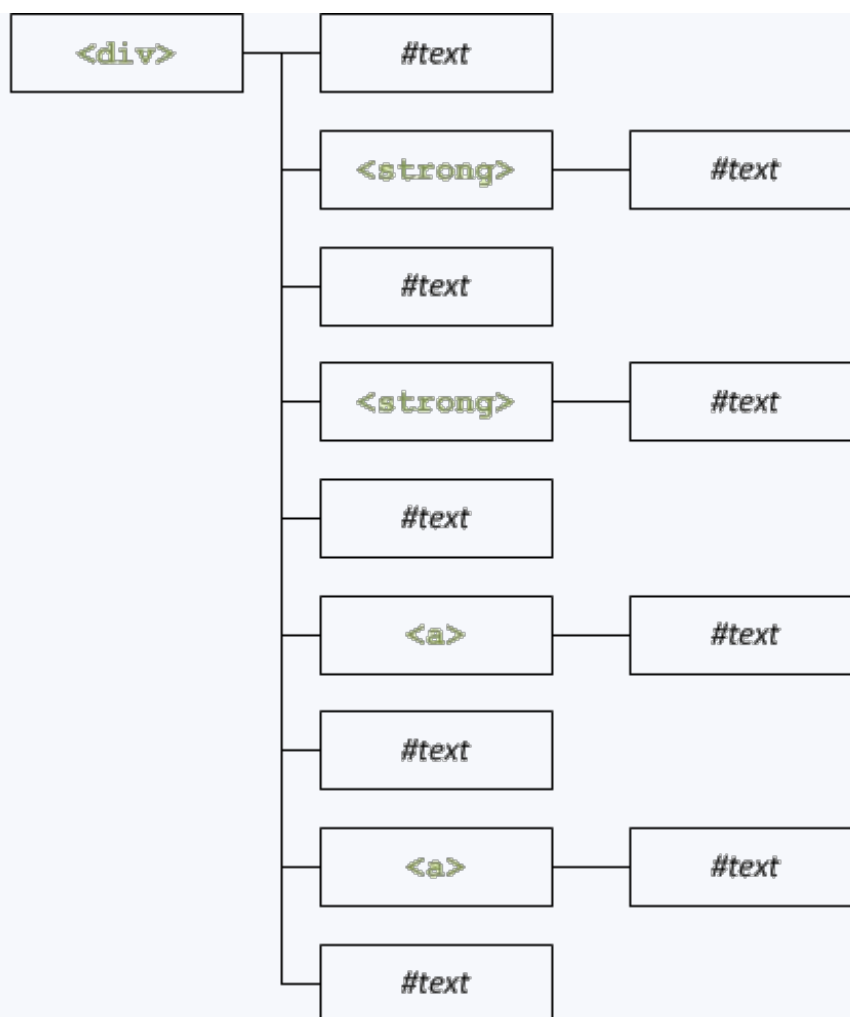
Pour ce premier exercice, nous vous proposons de recréer "du texte" mélangé à divers éléments tels des `<a>` et des ``. C'est assez simple, mais pensez bien à ne pas vous emmêler les pinceaux avec tous les nœuds textuels !

Code : HTML

```
<div id="divTD1">
  Le <strong>World Wide Web Consortium</strong>, abrégé par le sigle
  <strong>W3C</strong>, est un
  <a href="http://fr.wikipedia.org/wiki/Organisme_de_normalisation"
  title="Organisme de normalisation">organisme de standardisation</a>
  à but non-lucratif chargé de promouvoir la compatibilité des
  technologies du <a
  href="http://fr.wikipedia.org/wiki/World_Wide_Web" title="World Wide
  Web">World Wide Web</a>.
</div>
```

Corrigé

Secret (cliquez pour afficher)

**Code : JavaScript**

```

// On crée l'élément conteneur
var mainDiv = document.createElement('div');
mainDiv.id = 'divTD1';

// On crée tous les noeuds textuels, pour plus de facilité
var textNodes = [
  document.createTextNode('Le '),
  document.createTextNode('World Wide Web Consortium'),
  document.createTextNode(', abrégé par le sigle '),
  document.createTextNode('W3C'),
  document.createTextNode(', est un '),
  document.createTextNode('organisme de standardisation'),
  document.createTextNode(' à but non-lucratif chargé de
promouvoir la compatibilité des technologies du '),
  document.createTextNode('World Wide Web'),
  document.createTextNode('.')
];

// On crée les deux <strong> et les deux <a>
var w3cStrong1 = document.createElement('strong');
var w3cStrong2 = document.createElement('strong');

w3cStrong1.appendChild(textNodes[1]);
w3cStrong2.appendChild(textNodes[3]);

var orgLink = document.createElement('a');
var wwwLink = document.createElement('a');

orgLink.href =

```

```
'http://fr.wikipedia.org/wiki/Organisme_de_normalisation';
orgLink.title = 'Organisme de normalisation';
orgLink.appendChild(textNodes[5]);

wwwLink.href = 'http://fr.wikipedia.org/wiki/World_Wide_Web';
wwwLink.title = 'World Wide Web';
wwwLink.appendChild(textNodes[7]);

// On insère le tout dans mainDiv
mainDiv.appendChild(textNodes[0]);
mainDiv.appendChild(w3cStrong1);
mainDiv.appendChild(textNodes[2]);
mainDiv.appendChild(w3cStrong2);
mainDiv.appendChild(textNodes[4]);
mainDiv.appendChild(orgLink);
mainDiv.appendChild(textNodes[6]);
mainDiv.appendChild(wwwLink);
mainDiv.appendChild(textNodes[8]);

// On insère mainDiv dans le <body>
document.body.appendChild(mainDiv);
```

Essayer !

Par mesure de facilité, tous les nœuds textuels sont contenus dans le tableau **textNodes**. Ça évite de faire 250 variables différentes. Une fois les nœuds textuels créés, on crée les éléments **<a>** et ****. Une fois que tout cela est fait, on insère le tout, un élément après l'autre, dans le div conteneur.

Deuxième exercice

Énoncé

Code : HTML

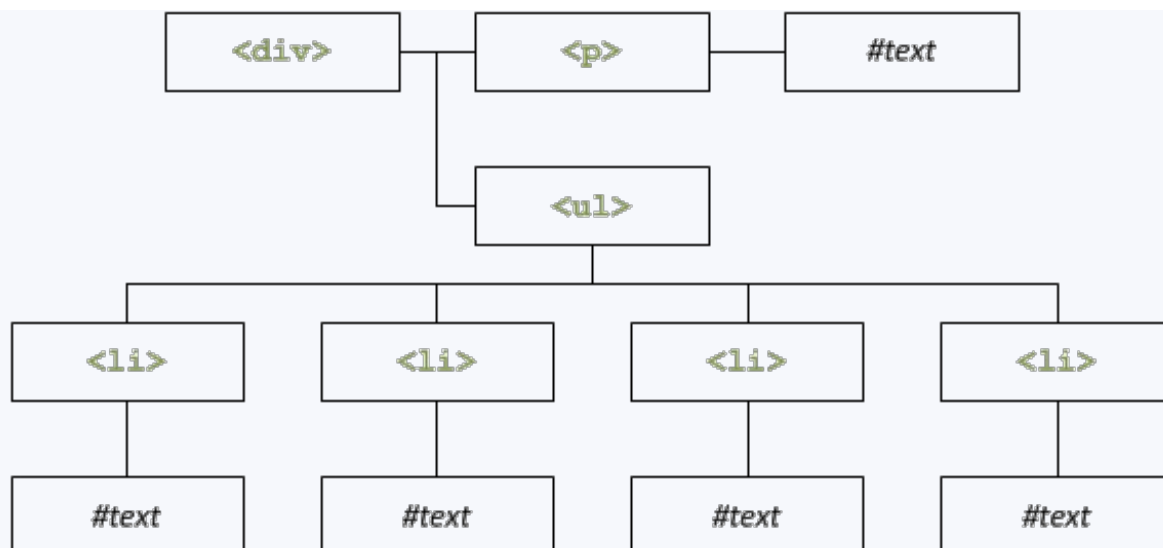
```
<div id="divTD2">
  <p>Langages basés sur ECMAScript :</p>

  <ul>
    <li>JavaScript</li>
    <li>JScript</li>
    <li>ActionScript</li>
    <li>EX4</li>
  </ul>
</div>
```

On ne va tout de même pas créer 4 éléments **** "à la main"... Utilisez une boucle **for** ! Et souvenez-vous, les éléments textuels dans un tableau.

Corrigé

Secret (cliquez pour afficher)

**Code : JavaScript**

```

// On crée l'élément conteneur
var mainDiv = document.createElement('div');
mainDiv.id = 'divTD2';

// On crée tous les nœuds textuels, pour plus de facilité
var languages = [
  document.createTextNode('JavaScript'),
  document.createTextNode('JScript'),
  document.createTextNode('ActionScript'),
  document.createTextNode('EX4')
];

// On crée le paragraphe
var paragraph = document.createElement('p');
var paragraphText = document.createTextNode('Langages basés sur
ECMAScript :');
paragraph.appendChild(paragraphText);

// On crée la liste, et on boucle pour ajouter chaque item
var uList = document.createElement('ul'),
    uItem;

for (var i = 0, c=languages.length; i < c; i++) {
  uItem = document.createElement('li');

  uItem.appendChild(languages[i]);
  uList.appendChild(uItem);
}

// On insère le tout dans mainDiv
mainDiv.appendChild(paragraph);
mainDiv.appendChild(uList);

// On insère mainDiv dans le <body>
document.body.appendChild(mainDiv);

```

Essayer !

Les nœuds textuels de la liste à puces sont créés par le biais du tableau **languages**, et pour créer chaque élément , il suffit de boucler sur le nombre d'items du tableau.

Troisième exercice

Énoncé

Voici une version légèrement plus complexe de l'exercice précédent. Le schéma de fonctionnement est le même, mais ici, le tableau **languages** contiendra **des objets littéraux**, et chacun de ces objets contiendra deux propriétés : le nœud du **<dt>** et le nœud du **<dd>**.

Code : HTML

```
<div id="divTD3">
  <p>Langages basés sur ECMAScript :</p>

  <dl>
    <dt>JavaScript</dt>
    <dd>JavaScript est un langage de programmation de scripts
    principalement utilisé dans les pages web interactives mais aussi
    coté serveur.</dd>

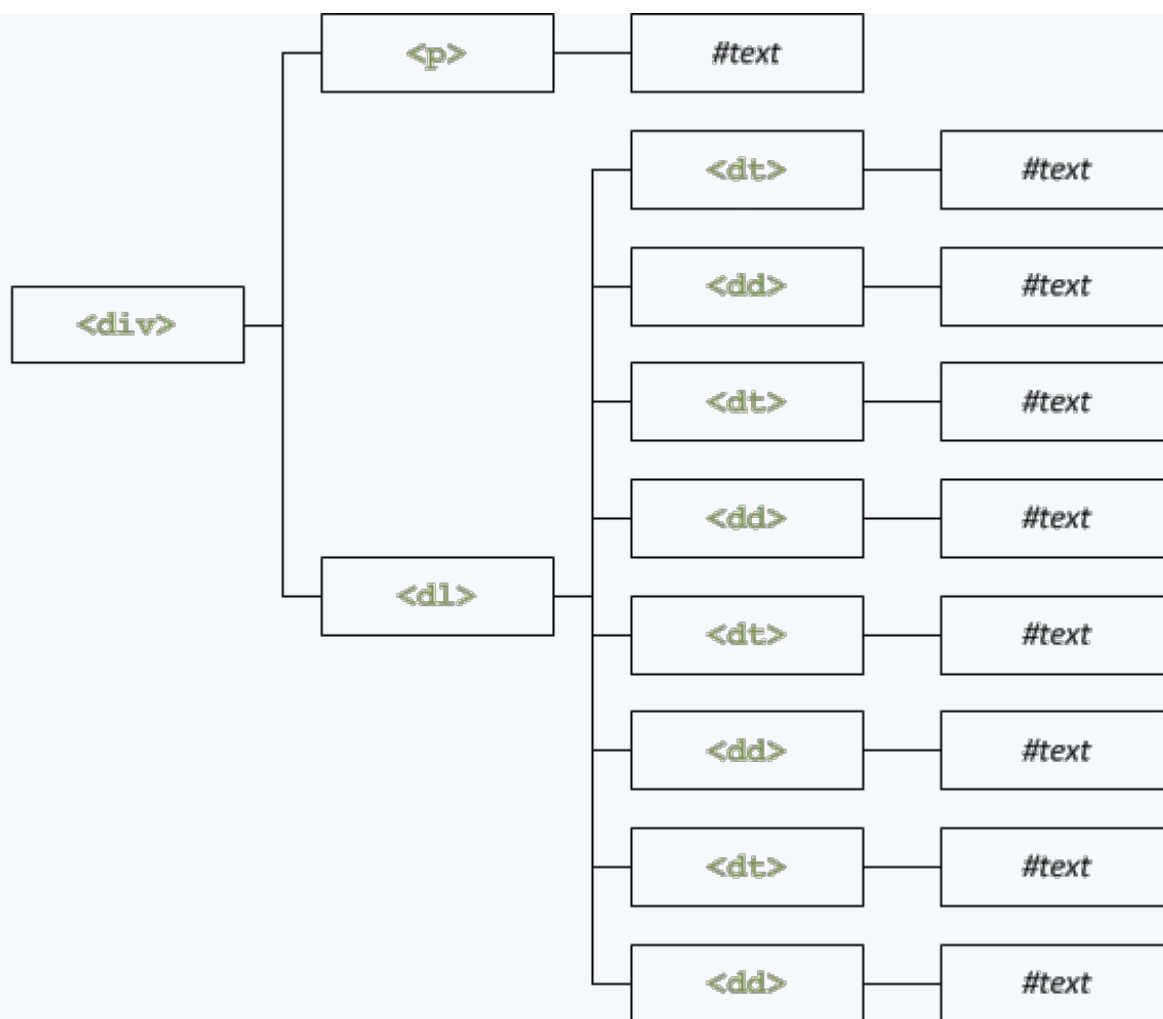
    <dt>JScript</dt>
    <dd>JScript est le nom générique de plusieurs implémentations
    d'ECMAScript 3 créées par Microsoft.</dd>

    <dt>ActionScript</dt>
    <dd>ActionScript est le langage de programmation utilisé au sein
    d'applications clientes (Adobe Flash, Adobe Flex) et serveur (Flash
    media server, JRun, Macromedia Generator).</dd>

    <dt>EX4</dt>
    <dd>ECMAScript for XML (E4X) est une extension XML au langage
    ECMAScript.</dd>
  </dl>
</div>
```

Corrigé

Secret (cliquez pour afficher)

**Code : JavaScript**

```

// On crée l'élément conteneur
var mainDiv = document.createElement('div');
mainDiv.id = 'divTD3';

// On place le texte dans des objets, eux-même placés dans un
// tableau
// Par facilité, la création des noeuds textuels se fera dans la
// boucle
var languages = [
  { t: 'JavaScript',
    d: 'JavaScript est un langage de programmation de scripts
    principalement utilisé dans les pages web interactives mais aussi
    coté serveur.' },
  { t: 'JScript',
    d: 'JScript est le nom générique de plusieurs
    implémentations d\'ECMAScript 3 créées par Microsoft.' },
  { t: 'ActionScript',
    d: 'ActionScript est le langage de programmation utilisé au
    sein d\'applications clientes (Adobe Flash, Adobe Flex) et serveur
    (Flash media server, JRun, Macromedia Generator).' },
  { t: 'EX4',
    d: 'ECMAScript for XML (E4X) est une extension XML au
    langage ECMAScript.' }
];

// On crée le paragraphe
var paragraph = document.createElement('p');
var paragraphText = document.createTextNode('Langages basés sur
ECMAScript :');

```



```

paragraph.appendChild(paragraphText);

// On crée la liste, et on boucle pour ajouter chaque item
var defList = document.createElement('dl'),
    defTerm, defTermText,
    defDefn, defDefnText;

for (var i = 0, c=languages.length; i < c; i++) {
    defTerm = document.createElement('dt');
    defDefn = document.createElement('dd');

    defTermText = document.createTextNode(languages[i].t);
    defDefnText = document.createTextNode(languages[i].d);

    defTerm.appendChild(defTermText);
    defDefn.appendChild(defDefnText);

    defList.appendChild(defTerm);
    defList.appendChild(defDefn);
}

// On insère le tout dans mainDiv
mainDiv.appendChild(paragraph);
mainDiv.appendChild(defList);

// On insère mainDiv dans le <body>
document.body.appendChild(mainDiv);

```

Le tableau contient des objets comme ceci :

Code : JavaScript

```

{
  t: 'Terme',
  d: 'Définition'}

```

[Essayer !](#)

Créer une liste de définitions (`<dl>`) n'est pas plus compliqué qu'une liste à puces normale, la seule chose qui diffère est que `<dt>` et `<dd>` sont ajoutés conjointement au sein de la boucle.

Quatrième exercice

Énoncé

Un rien plus corsé... quoique. Ici, la difficulté réside dans le nombre important d'éléments à imbriquer les uns dans les autres. Si vous procédez méthodiquement, vous avez peu de chance de vous planter.

Code : HTML

```

<div id="divTD4">
  <form enctype="multipart/form-data" method="post"
action="upload.php">
  <fieldset>

```

```

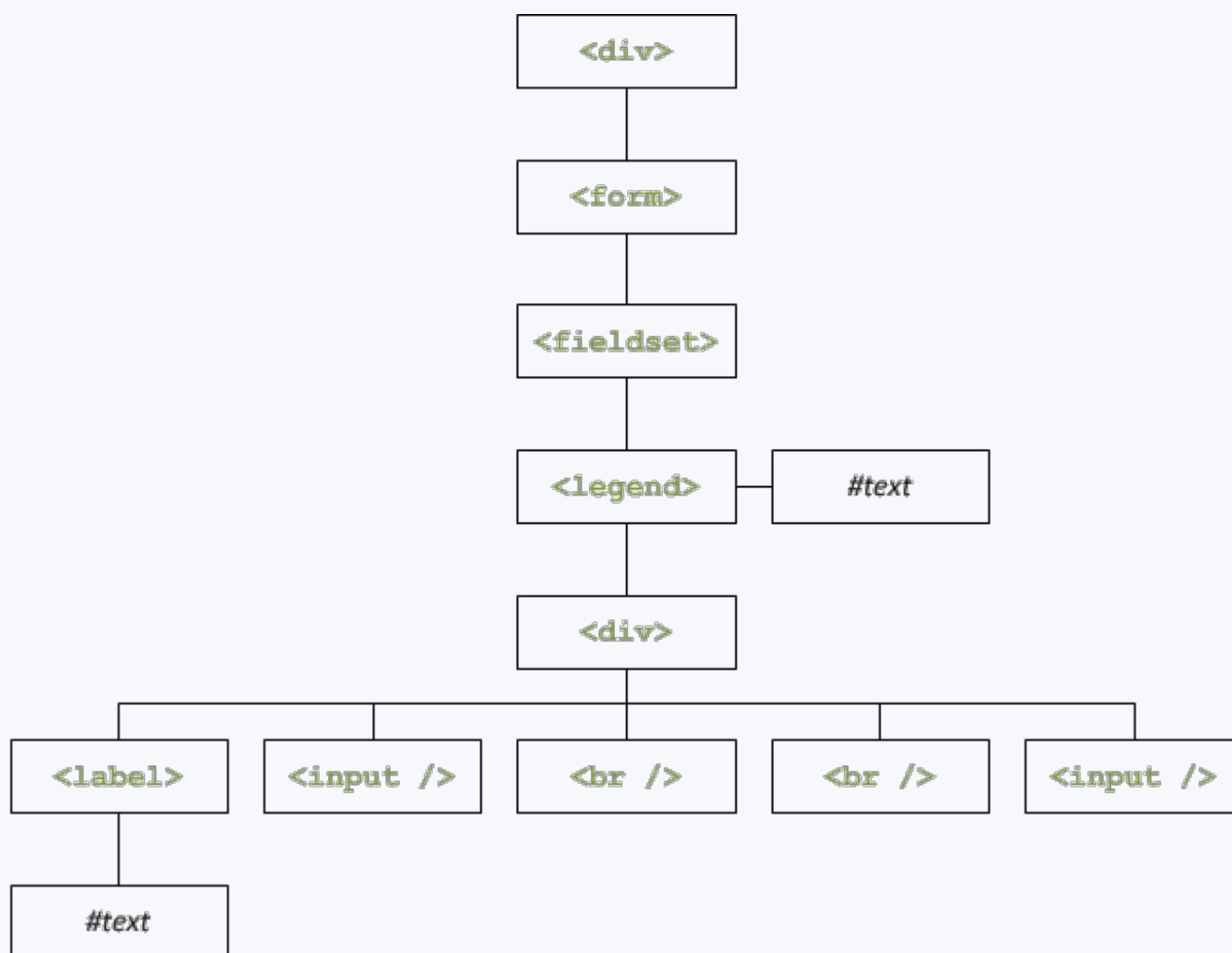
<legend>Uploader une image</legend>

<div style="text-align: center">
  <label for="inputUpload">Image à uploader :</label>
  <input type="file" name="inputUpload" id="inputUpload" />
  <br /><br />
  <input type="submit" value="Envoyer" />
</div>
</fieldset>
</form>
</div>

```

Corrigé

Secret (cliquez pour afficher)



Code : JavaScript

```

// On crée l'élément conteneur
var mainDiv = document.createElement('div');
mainDiv.id = 'divTD4';

// Création de la structure du formulaire
var form = document.createElement('form');
var fieldset = document.createElement('fieldset');
var legend = document.createElement('legend'),

```

```
        legendText = document.createTextNode('Uploader une image');
    var center      = document.createElement('div');

    form.action     = 'upload.php';
    form.enctype    = 'multipart/form-data';
    form.method     = 'post';

    center.setAttribute('style', 'text-align: center');

    legend.appendChild(legendText);

    fieldset.appendChild(legend);
    fieldset.appendChild(center);

    form.appendChild(fieldset);

    // Création des champs
    var label = document.createElement('label'),
        labelText = document.createTextNode('Image à uploader :');
    var input = document.createElement('input');
    var br = document.createElement('br');
    var submit = document.createElement('input');

    input.type = 'file';
    input.id = 'inputUpload';
    input.name = input.id;

    submit.type = 'submit';
    submit.value = 'Envoyer';

    label.appendChild(labelText);

    center.appendChild(label);
    center.appendChild(input);
    center.appendChild(br);
    center.appendChild(br.cloneNode(false)); // On clone, pour mettre
    un 2e <br />
    center.appendChild(submit);

    // On insère le formulaire dans mainDiv
    mainDiv.appendChild(form);

    // On insère mainDiv dans le <body>
    document.body.appendChild(mainDiv);
```

[Essayer !](#)

Comme il y a beaucoup d'éléments à créer, pourquoi ne pas diviser le script en deux : la structure du formulaire, et les champs. C'est plus propre, et on s'y retrouve mieux.

Conclusion des TD

Il est très probable que vous n'avez pas organisé votre code comme dans les corrections, ou que vous n'avez pas utilisé les mêmes idées, comme utiliser un tableau, ou même un tableau d'objets. Votre code est certainement bon, mais retenez une chose : essayez d'avoir un code clair et propre, tout en étant facile à comprendre, ça vous simplifiera la tâche !

Après un gros chapitre comme celui-ci, quoi de mieux que d'enchaîner sur un autre, le petit frère du maniement du DOM : les événements. Accrochez-vous, les choses sérieuses commencent !

Après l'introduction au DOM, il est temps d'approfondir ce domaine en abordant les événements en Javascript.

Au programme, l'utilisation des événements sans le DOM, avec le [DOM-0](#) (inventé par Netscape) puis avec le DOM-2. Nous allons voir comment mettre en place ces événements, les utiliser, modifier leur comportement, etc...

Après ce chapitre, vous pourrez déjà commencer à interagir avec l'utilisateur, accrochez-vous bien, ça va devenir de plus en plus intéressant !

Que sont les événements ?

Les événements, comme leur nom l'indique, permettent de déclencher une fonction selon qu'une action s'est produite ou non. Autrement dit, on peut, par exemple, faire apparaître une fenêtre `alert()` lorsque l'utilisateur survole une zone d'une page web.

"Une zone" est un terme un peu vague, il vaut mieux parler d'un élément (HTML dans la plupart des cas). Ainsi, vous pouvez très bien ajouter un événement à un élément de votre page web (par exemple, une balise `div`) pour faire en sorte de déclencher un code Javascript lorsque l'utilisateur fera une action sur l'élément en question.

La théorie

Liste des événements

Il existe de nombreux événements, tous plus ou moins utiles. Voici la liste des événements principaux, ainsi que les actions nécessaires à effectuer pour qu'ils se déclenchent :

Nom de l'événement	Action pour le déclencher
click	Cliquer (appuyer puis relâcher) sur l'élément
dblclick	Double-cliquer sur l'élément
mouseover	Faire entrer le curseur sur l'élément
mouseout	Faire sortir le curseur de l'élément
mousedown	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
mouseup	Relâcher le bouton gauche de la souris sur l'élément
mousemove	Faire déplacer le curseur sur l'élément
keydown	Appuyer (sans relâcher) sur une touche clavier sur l'élément
keyup	Relâcher une touche clavier sur l'élément
keypress	Frapper (appuyer puis relâcher) sur une touche clavier sur l'élément
focus	"Cibler" l'élément
blur	Annuler le "ciblage" de l'élément
change	Changer la valeur d'un élément spécifique aux formulaires (input, checkbox, etc...)
select	Sélectionner le contenu d'un champ de texte (input, textarea, etc...)

Toutefois, ce n'est pas tout, il existe aussi deux événements spécifiques à la balise `<form>`, les voici :

Nom de l'événement	Action pour le déclencher
submit	Envoyer le formulaire
reset	Réinitialiser le formulaire

Tout cela est pour le moment très théorique, je ne fais que vous lister quelques événements existants mais nous allons rapidement apprendre à les utiliser après un dernier petit passage concernant ce qu'on appelle le *focus*.

[Retour sur le focus](#)

Le **focus** définit ce que j'appelle le "ciblage" d'un élément. Lorsqu'un élément est ciblé, il va recevoir tous les événements de votre clavier, un exemple simple est d'utiliser une balise **input** de type **text**, si vous cliquez dessus alors l'input possède le focus, autrement dit : il est ciblé, et si vous tapez des caractères sur votre clavier vous allez les voir s'afficher dans l'input en question.

Le focus peut s'appliquer à de nombreux éléments, ainsi, si vous tapez sur la touche Tabulation de votre clavier alors que vous êtes sur une page web, vous allez avoir un élément de ciblé/sélectionné qui recevra alors tout ce que vous tapez sur votre clavier. Par exemple, si vous avez un lien de ciblé et que vous tapez sur la touche Entrée de votre clavier alors vous serez redirigé vers l'url contenue dans ce lien.

La pratique

Utiliser les événements

Bien, maintenant que vous avez vu le côté théorique (et barbant) des événements, nous allons pouvoir passer un peu à la pratique. Toutefois, dans un premier temps, il n'est que question de vous faire découvrir à quoi sert tel ou tel événement et comment il réagit, nous allons donc voir comment les utiliser sans le DOM, ce qui est considérablement plus limité.

Bien, commençons par l'événement **click** sur une simple balise **span** :

Code : HTML

```
<span onclick="alert('Hello !');">Cliquez-moi !</span>
```

[Essayer !](#)

Comme vous pouvez le constater, il suffit de cliquer sur le texte pour que la boîte de dialogue s'affiche. Afin d'obtenir ce résultat nous avons ajouté à notre balise **span** un attribut contenant les deux lettres "on" et le nom de notre événement "click", nous obtenons donc **onclick**.

Cet attribut possède une valeur qui est un code Javascript, vous pouvez y écrire quasiment tout ce que souhaitez mais tout doit tenir entre les guillemets de l'attribut.

La propriété *this*

Cette propriété n'est, normalement, pas censée vous servir dès maintenant, cependant il est toujours bon de la connaître pour les événements sans utilisation du DOM. Il s'agit d'une propriété pointant sur l'objet actuellement en cours d'utilisation. Donc, si vous faites appel à cette propriété lorsqu'un événement est déclenché, l'objet pointé sera l'élément qui a déclenché l'événement. Exemple :

Code : HTML

```
<span onclick="alert('Voici le contenu de l\'élément que vous avez cliqué :\n\n' + this.innerHTML);">Cliquez-moi !</span>
```

[Essayer !](#)



Cette propriété ne vous servira, dans l'immédiat, que pour l'utilisation des événements sans le DOM ou avec le DOM-1. Si vous tentez de vous en servir avec le DOM-2 vous allez avoir des surprises !

Retour sur le focus

Afin de bien vous montrer ce qu'est le focus, voici un exemple qui vous montrera ce que ça donne sur un input classique et un lien :

Code : HTML

```
<input id="input" type="text" size="50" value="Cliquez ici !"
onfocus="this.value='Appuyez maintenant sur votre touche de
tabulation.';" onblur="this.value='Cliquez ici !';"/>
<br /><br />
<a href="#" onfocus="document.getElementById('input').value = 'Vous
avez maintenant le focus sur le lien, bravo !';">Un lien bidon</a>
```

[Essayer !](#)

Comme vous pouvez le constater, lorsque vous cliquez sur l'input, celui-ci possède le focus : il exécute donc l'événement et affiche alors un texte différent vous demandant d'appuyer sur votre touche de tabulation. L'appui sur la touche de tabulation permet de faire passer le focus à l'élément suivant. En clair, en appuyant sur cette touche vous faites perdre le focus à l'input, ce qui déclenche l'événement **blur** (qui signifie la perte du focus) et fait passer le focus sur le lien qui affiche alors son message grâce à son événement.

Bloquer l'action par défaut de certains événements

Passons maintenant à un petit problème : Quand vous souhaitez appliquer un événement **click** sur un lien, que se passe-t-il ? Regardez donc par vous-même :

Code : HTML

```
<a href="http://www.siteduzero.com" onclick="alert('Vous avez cliqué
!');">Cliquez-moi !</a>
```

[Essayer !](#)

Si vous avez essayé le code, vous avez sûrement remarqué que la fonction **alert()** a bien fonctionné mais qu'après vous avez été redirigé vers le Site du Zéro, or on souhaite bloquer cette redirection. Pour cela, il suffit juste d'ajouter le code **return false;** dans votre événement **click** :

Code : HTML

```
<a href="http://www.siteduzero.com" onclick="alert('Vous avez cliqué
!'); return false;">Cliquez-moi !</a>
```

[Essayer !](#)

Ici, le **return false;** sert juste à bloquer l'action par défaut de l'événement qui le déclenche.



À noter que l'utilisation de **return true;** permet de faire fonctionner l'événement comme si de rien n'était. En clair, comme si on n'utilisait pas de **return false;**



Cela peut avoir son utilité si vous utilisez, par exemple, la fonction **confirm** dans votre événement.

L'utilisation de "javascript:" dans les liens : Une technique prohibée

Dans certains cas, vous allez devoir créer des liens juste pour leur attribuer un événement **click** et non pas pour leur fournir un lien vers lequel rediriger. Dans ce genre de cas, il est courant de voir ce type de code :

Code : HTML

```
<a href="javascript: alert('Vous avez cliqué !');">Cliquez-moi !</a>
```



Je vous interdis de faire cela ! Ou alors, si vous le faites, ne venez pas dire que vous avez appris le JS grâce à ce cours, ce serait la honte 😊 !

Bref, plus clairement, il s'agit d'une vieille méthode qui permet d'insérer du Javascript directement dans l'attribut **href** de votre lien juste en ajoutant **javascript:** au début de l'attribut. Cette technique est maintenant obsolète et je serais déçu de vous voir l'utiliser, je vous en déconseille donc très fortement l'utilisation et vous propose même une méthode alternative :

Code : HTML

```
<a href="#" onclick="alert('Vous avez cliqué !'); return false;">Cliquez-moi !</a>
```

Essayer !

Concrètement, qu'est-ce qui change ? On a tout d'abord remplacé l'immonde **javascript:** par un **dièse (#)** puis on a mis notre code JS dans l'événement approprié (**click**). Par ailleurs on libère l'attribut **href**, ce qui nous permet, si besoin de laisser un lien pour ceux qui n'activent pas le Javascript ou bien encore pour ceux qui aiment bien ouvrir leurs liens dans un nouvel onglet.



Ok, j'ai compris, mais pourquoi un **return false;** ?

Tout simplement parce que le **dièse (#)** redirige tout en haut de la page web, ce qui n'est pas ce que l'on souhaite. On bloque donc cette redirection avec notre petit bout de code.



Vous savez maintenant que l'utilisation de "javascript:" dans les liens est prohibée et c'est déjà une bonne chose. Cependant, gardez bien à l'esprit que l'utilisation d'un lien uniquement pour le déclenchement d'un événement **click** n'est pas une bonne chose, préférez plutôt l'utilisation d'une balise **<button>** à laquelle vous aurez retiré le style CSS.

Les événements au travers du DOM

Bien, maintenant que nous avons vu l'utilisation des événements sans le DOM, nous allons passer à leur utilisation au travers de l'interface implémentée par Netscape que l'on appelle le **DOM-0** puis au standard de base actuel : le **DOM-2**.

Le DOM-0

Cette interface est vieille, je ne vais vous en apprendre que les bases, histoire que vous sachiez comment vous en servir et surtout pour que vous puissiez en reconnaître la syntaxe dans un code qui pourrait ne pas vous appartenir.

Commençons par créer un simple code avec un événement **click** :

Code : HTML

```
<span id="clickme">Cliquez-moi !</span>

<script type="text/javascript">

  var el = document.getElementById('clickme');

  el.onclick = function() {
    alert("Vous m'avez cliqué !");
  };

</script>
```

[Essayer !](#)

Alors, voyons par étapes ce que nous avons fait dans ce code :

- On récupère tout d'abord l'élément HTML dont l'ID est **clickme**.
- On accède ensuite à son attribut **onclick** auquel on assigne une fonction anonyme.
- Dans cette même fonction, on fait un appel à la fonction **alert()** avec un texte en paramètre.

Comme vous le voyez, on définit maintenant les événements non plus dans le code HTML mais directement en Javascript. Chaque événement standard possède donc un attribut dont le nom est, à nouveau, précédé par les deux lettres "on". Cet attribut ne prend plus pour valeur un code Javascript brut mais : soit le nom d'une fonction, soit une fonction anonyme. Bref, dans tous les cas, il faut lui fournir une fonction qui contiendra le code à exécuter en cas de déclenchement de l'événement.

Concernant la suppression d'un événement avec le DOM-0, il suffit tout simplement de lui attribuer une fonction anonyme vide :

Code : JavaScript

```
el.onclick = function() {};
```

Voilà tout pour les événements DOM-0, nous pouvons maintenant passer au cœur des événements : le DOM-2 et l'**objet Event**.

Le DOM-2

Nous y voici enfin ! Alors, tout d'abord, pourquoi le DOM-2 et non pas le DOM-0 voir pas de DOM du tout ? Concernant la méthode sans le DOM, c'est simple : On ne peut pas y utiliser l'objet **Event** qui est pourtant une mine d'informations sur l'événement déclenché, donc je vous conseille de mettre cette méthode de côté dès maintenant (je vous l'ai enseignée juste

pour que vous sachiez la reconnaître).

En ce qui concerne le DOM-0, il a deux problèmes majeurs : il est vieux, et il ne permet pas de créer plusieurs fois le même événement.

Quant au DOM-2, lui, permet la création multiple d'un même événement et gère aussi l'objet **Event**. Autrement dit, le DOM-2 c'est le bien, mangez-en !



En clair, il faut constamment utiliser le DOM-2 ?

Non, pas vraiment. Autant la technique sans le DOM est à abolir (je vous l'ai montrée juste pour que vous sachiez la reconnaître dans un code), autant l'utilisation du DOM-0 est largement possible, tout dépend de votre code.

D'ailleurs, dans la majorité des cas, vous choisirez le DOM-0 pour sa simplicité d'utilisation et sa rapidité de mise en place. Ce n'est, généralement, que lorsque vous aurez besoin de créer plusieurs événements d'un même type (**click**, par exemple) que vous utiliserez le DOM-2.



Cependant, attention ! Si il y a bien une situation où je conseille de toujours utiliser le DOM-2, c'est lorsque vous créez un script JS que vous souhaitez distribuer. Pourquoi ? Eh bien admettons que votre script ajoute (avec le DOM-0) un événement **click** à votre élément `<body>`, si la personne qui décide d'ajouter votre script à sa page a déjà créé cet événement (avec le DOM-0 bien sûr) alors il va y avoir une collision puisque cette version du DOM n'accepte qu'un seul événement d'un même type !

Le DOM-2 selon les standards du web

Comme pour les autres interfaces événementielles, voici un exemple avec l'événement **click** :

Code : HTML

```
<span id="clickme">Cliquez-moi !</span>

<script type="text/javascript">

  var el = document.getElementById('clickme');

  el.addEventListener('click', function() {
    alert("Vous m'avez cliqué !");
  }, false);

</script>
```

[Essayer !](#)

Concrètement, qu'est-ce qui change du DOM-0 ? Alors tout d'abord nous n'utilisons plus un attribut mais une méthode nommée **addEventListener()** (qui ne fonctionne pas sous IE8 et antérieur, mais on en reparlera par la suite).

Cette méthode prend trois paramètres :

- Le nom de l'événement (sans les lettres "on").
- La fonction à exécuter.
- Un booléen pour spécifier si l'on souhaite utiliser la phase de capture ou bien celle de bouillonnement. Je vous expliquerai ce concept un peu plus tard dans ce chapitre. Sachez simplement que l'on utilise généralement la valeur **false** pour ce paramètre.

Une petite explication pour ceux qui n'arriveraient éventuellement pas à comprendre le code ci-dessus : J'ai bel et

bien utilisé la méthode `addEventListener()`, je l'ai simplement écrite sur 3 lignes :

- La première ligne contient l'appel à la méthode `addEventListener()`, le premier paramètre, et l'initialisation de la fonction anonyme pour le deuxième paramètre.
- La deuxième ligne contient le code de la fonction anonyme.
- La troisième ligne contient l'accolade fermante de la fonction anonyme, puis le troisième paramètre.



Ce code revient à écrire la même chose que ci-dessous mais de façon plus rapide :

Code : JavaScript

```
var el = document.getElementById('clickme');

var myFunction = function() {
    alert("Vous m'avez cliqué !");
};

el.addEventListener('click', myFunction, false);
```

Comme je vous l'ai dit plus haut, le DOM-2 permet la création multiple d'événements identiques pour un même élément, ainsi, vous pouvez très bien faire ceci :

Code : HTML

```
<span id="clickme">Cliquez-moi !</span>

<script type="text/javascript">

    var el = document.getElementById('clickme');

    // Premier événement click
    el.addEventListener('click', function() {
        alert("Et de un !");
    }, false);

    // Deuxième événement click
    el.addEventListener('click', function() {
        alert("Et de deux !");
    }, false);

</script>
```

[Essayer !](#)

Si vous avez exécuté ce code, vous avez peut-être eu les événements déclenchés dans l'ordre de création, cependant ce ne sera pas forcément le cas à chaque essai. En effet, l'ordre de déclenchement est un peu aléatoire...

Venons-en maintenant à la suppression des événements ! Celle-ci s'opère avec la méthode `removeEventListener()` et se fait de manière très simple :

Code : JavaScript

```
el.addEventListener('click', myFunction, false); // On crée
l'événement

el.removeEventListener('click', myFunction, false); // On supprime
```

l'événement en lui repassant les mêmes paramètres

Toute suppression d'événement avec le DOM-2 se fait avec les mêmes paramètres que lors de sa création ! Cependant, cela ne fonctionne pas avec les fonctions anonymes ! Tout événement DOM-2 créé avec une fonction anonyme ne peut pas être supprimé !

Le DOM-2 selon Internet Explorer

Ah ! Le fameux Internet Explorer ! Jusqu'à présent nous n'avions quasiment aucun problème avec lui, il respectait les standards, mais cela ne pouvait pas durer éternellement et voilà que les méthodes **addEventListener()** et **removeEventListener()** ne fonctionnent pas !

Alors que faut-il utiliser du coup ? Eh bien les méthodes **attachEvent()** et **detachEvent()** ! Celles-ci s'utilisent de la même manière que les méthodes standards sauf que le troisième paramètre n'existe pas et que l'événement doit être préfixé par "on" (encore, oui...), exemple :

Code : JavaScript

```
// On crée l'événement
el.attachEvent('onclick', function() {
    alert('Tadaaaaam !');
});

// On supprime l'événement en lui repassant les mêmes paramètres
el.detachEvent('onclick', function() {
    alert('Tadaaaaam !');
});
```

Internet Explorer à partir de sa 9ème version supporte les méthodes standards. En revanche, pour les versions antérieures il vous faudra jongler entre les méthodes standards et les méthodes de IE. Généralement, on utilise un code de ce genre pour gérer la compatibilité entre navigateurs :

Code : JavaScript

```
function addEvent(el, event, func) {

    if(el.addEventListener) { // Si notre élément possède la méthode
addEventListener()
        el.addEventListener(event, func, false);
    } else { // Si notre élément ne possède pas la méthode
addEvent()
        el.attachEvent('on'+event, func);
    }

}

addEvent(element, 'click', function() {
    // Votre code
});
```

Essayer une adaptation de ce code !

Du côté de la théorie

Vous vous souvenez que notre méthode `addEventListener()` prend trois paramètres ? Je vous avais dit que j'allais revenir sur l'utilisation de son troisième paramètre plus tard. Eh bien ce "plus tard" est arrivé !



Capture ? Bouillonnement ? De quoi tu parles ?

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 ait été déclenché. Toutes deux permettent de définir le sens de propagation des événements.

Mais qu'est-ce que la propagation d'un événement ? Pour expliquer cela, prenons un exemple avec ces deux éléments HTML :

Code : HTML

```
<div>
  <span>Du texte !</span>
</div>
```

Si j'attribue une fonction à l'événement **click** de chacun de ces deux éléments et que je clique sur le texte, quel événement va se déclencher en premier à votre avis ? Bonne question n'est-ce pas ?

Notre réponse se trouve dans les phases de capture et de bouillonnement. Si vous décidez d'utiliser la capture, alors l'événement du **div** se déclenchera en premier puis viendra ensuite l'événement du **span**. En revanche, si vous utilisez le bouillonnement, ce sera d'abord l'événement du **span** qui se déclenchera, puis viendra par la suite celui du **div**.

Voici un petit code qui met en pratique l'utilisation de ces deux phases :

Code : HTML

```
<div id="capt1">
  <span id="capt2">Cliquez-moi pour la phase de capture.</span>
</div>

<div id="boul1">
  <span id="boul2">Cliquez-moi pour la phase de
bouillonnement.</span>
</div>

<script type="text/javascript">
  var capt1 = document.getElementById('capt1'),
      capt2 = document.getElementById('capt2'),
      boul1 = document.getElementById('boul1'),
      boul2 = document.getElementById('boul2');

  capt1.addEventListener('click', function() {
    alert("L'événement du div vient de se déclencher.");
  }, true);

  capt2.addEventListener('click', function() {
    alert("L'événement du span vient de se déclencher.");
  }, true);

  boul1.addEventListener('click', function() {
    alert("L'événement du div vient de se déclencher.");
  }, false);

  boul2.addEventListener('click', function() {
```

```
    alert ("L'événement du span vient de se déclencher.");  
  }, false);  
</script>
```

Essayer !

Et pour finir, un lien vers [la spécification du W3C concernant ces phases](#) si vous avez envie d'aller plus loin. Je vous conseille de regarder ce lien ne serait-ce que pour voir le schéma fourni qui explique bien le concept de ces phases.

Du côté de la pratique

Bien, vous savez maintenant à quoi servent ces deux phases et pourtant, même si en théorie ce système se révèle utile, en pratique vous ne vous en servirez quasiment jamais pour la simple et bonne raison que la méthode **attachEvent** d'Internet Explorer (pour les versions antérieures à la 9ème) ne gère que la phase de bouillonnement, ce qui explique que l'on mette généralement le dernier paramètre de **addEventListener** à **false**.

Autre chose, les événements sans le DOM ou avec le DOM-0 ne gèrent, eux aussi, que la phase de bouillonnement.

Cela dit, ne vous en faites pas, vous n'avez pas appris ça en vain. Il est toujours bon de savoir cela, à la fois pour la connaissance globale du Javascript mais aussi pour comprendre ce que vous faites quand vous codez.

L'objet Event

Maintenant que nous avons vu comment créer et supprimer des événements, nous pouvons passer à l'objet **Event** !

Généralités sur l'objet Event

Tout d'abord, à quoi sert cet objet ? À vous fournir une multitude d'informations sur l'événement actuellement déclenché. Par exemple, vous pouvez récupérer quelles sont les touches actuellement enfoncées, les coordonnées du curseur, l'élément qui a déclenché l'événement, etc... Les possibilités sont nombreuses !

Cet objet est bien particulier dans le sens où il n'est accessible que lorsqu'un événement est déclenché. Son accès ne peut se faire que dans une fonction exécutée par un événement, cela se fait de la manière suivante avec le DOM-0 :

Code : JavaScript

```
el.onclick = function(e) { // L'argument "e" va récupérer une
  référence vers l'objet Event.
  alert(e.type) // Ceci affiche le type de l'événement (click,
  mouseover, etc...)
};
```

Et de cette façon là avec le DOM-2 :

Code : JavaScript

```
el.addEventListener('click', function(e) { // L'argument "e" va
  récupérer une référence vers l'objet Event.
  alert(e.type) // Ceci affiche le type de l'événement (click,
  mouseover, etc...)
}, false);
```

Il est important de préciser que **l'objet Event peut se récupérer dans un argument autre que e** ! Vous pouvez très bien le récupérer dans un argument nommé **test**, **hello**, ou autre... Après tout, l'objet **Event** est tout simplement passé en référence à l'argument de votre fonction, ce qui vous permet de choisir le nom que vous souhaitez.



Concernant Internet Explorer (dans toutes ses versions antérieures à la 9ème), si vous souhaitez utiliser le DOM-0 vous constaterez que l'objet **Event** n'est accessible qu'en utilisant **window.event**, ce qui signifie qu'il n'est pas nécessaire (pour IE bien entendu) d'utiliser un argument dans la fonction exécutée par l'événement. À contrario, si vous utilisez le DOM-2, vous n'êtes pas obligés d'utiliser **window.event**.

Afin de garder la compatibilité avec les autres navigateurs, on utilisera généralement ce code dans la fonction exécutée par l'événement : `e = e || window.event;`

Les fonctionnalités de l'objet Event

Vous avez déjà découvert ci-dessus l'attribut **type** qui permet de savoir quel type d'événement s'est déclenché. Passons maintenant à la découverte des autres attributs et méthodes que possède cet objet (attention, tout n'est pas présenté, seulement l'essentiel) :

Récupérer l'élément de l'événement actuel

Un des plus importants attributs de notre objet se nomme **target**. Celui-ci permet de récupérer une référence vers l'élément

dont l'événement a été déclenché (exactement comme le **this** pour les événements sans le DOM ou avec DOM-1), ainsi vous pouvez très bien modifier le contenu d'un élément qui a été cliqué :

Code : HTML

```
<span id="clickme">Cliquez-moi !</span>

<script type="text/javascript">
  var clickme = document.getElementById('clickme');

  clickme.addEventListener('click', function(e) {
    e.target.innerHTML = 'Vous avez cliqué !';
  }, false);
</script>
```

Comme il y a toujours un problème quelque part, voilà qu'Internet Explorer (dans toutes ses versions antérieures à la 9ème) ne supporte pas cette propriété. Ou plutôt, il la supporte à sa manière avec la propriété **srcElement**, voici le même code que ci-dessus mais compatible avec tous les navigateurs (dont IE) :

Code : HTML

```
<span id="clickme">Cliquez-moi !</span>

<script type="text/javascript">
  function addEvent(el, event, func) { // On réutilise notre
fonction de compatibilité pour les événements DOM-2
    if (el.addEventListener) {
      el.addEventListener(event, func, false);
    } else {
      el.attachEvent('on'+event, func);
    }
  }

  var clickme = document.getElementById('clickme');

  addEvent(clickme, 'click', function(e) {
    var target = e.srcElement || e.target; // Si vous avez oublié
cette spécificité de l'opérateur OU, allez voir le chapitre des
conditions
    target.innerHTML = 'Vous avez cliqué !';
  });
</script>
```

[Essayer !](#)



À noter qu'ici j'ai fait un code compatible pour Internet Explorer parce que le but était de gérer la compatibilité, mais pour le reste des autres codes je ne le ferai que si cela s'avère nécessaire. Je vous conseille donc d'avoir un navigateur à jour pour pouvoir tester tous les codes de ce cours.

Récupérer l'élément à l'origine du déclenchement de l'événement



Hum, ce n'est pas un peu la même chose ?

Eh bien non ! Pour expliquer cela de façon simple, certains événements appliqués à un élément parent peuvent se propager

d'eux-mêmes aux éléments enfants, c'est le cas des événements **mouseover**, **mouseout** et **mousemove**. Regardez donc cet exemple pour mieux comprendre :

Code : HTML

```
<p id="result"></p>

<div id="parent1">
  Parent
  <div id="child1">Enfant N°1</div>
  <div id="child2">Enfant N°2</div>
</div>

<script type="text/javascript">
  var parent1 = document.getElementById('parent1'),
      result = document.getElementById('result');

  parent1.addEventListener('mouseover', function(e) {
    result.innerHTML = "L'élément déclencheur de l'événement
    \"mouseover\" possède l'ID : " + e.target.id;
  }, false);
</script>
```

Essayer !

En testant cet exemple, vous avez sûrement remarqué que l'attribut **target** renvoyait toujours l'élément déclencheur de l'événement, or nous on souhaite obtenir l'élément sur lequel a été appliqué l'événement. Autrement dit, on veut connaître l'élément originaire de cet événement, pas ses enfants.

Le méthode est simple : utiliser l'attribut **currentTarget** au lieu de **target**. Essayez donc par vous-même après modification de cette seule ligne, l'ID affiché ne changera jamais :

Code : JavaScript

```
result.innerHTML = "L'élément déclencheur de l'événement
\"mouseover\" possède l'ID : " + e.currentTarget.id;
```

Essayer le code complet !



Bien que cette propriété semble intéressante pour certains cas d'application, il est assez difficile de la mettre en pratique car Internet Explorer (<9) ne la supporte pas et il n'y a pas d'alternative (mis à part avec l'utilisation du mot-clé **this** avec le DOM-0 ou sans le DOM).

Récupérer la position du curseur

La position du curseur est une information très importante, beaucoup de monde s'en sert pour de nombreux scripts comme le [Drag & Drop](#). Généralement, on récupère la position du curseur par rapport au coin supérieur gauche de la page web, cela dit il est aussi possible de récupérer sa position par rapport au coin supérieur gauche de l'écran. Toutefois, dans ce tuto, nous allons nous limiter à la page web, regardez donc [la doc de l'objet Event](#) si vous souhaitez en apprendre plus 😊.

Pour récupérer la position de notre curseur, il existe deux attributs : **clientX** pour la position horizontale et **clientY** pour la position verticale. Vu que la position du curseur change à chaque déplacement de la souris, il est donc logique de dire que l'événement le plus adapté à la majorité des cas est **mousemove**.



Je vous déconseille très fortement d'essayer d'exécuter la fonction **alert()** dans un événement **mousemove** ou bien vous allez rapidement être submergé de fenêtres !

Comme d'habitude, voici un petit exemple pour que vous compreniez bien :

Code : HTML

```
<div id="position"></div>

<script type="text/javascript">
  var position = document.getElementById('position');

  document.addEventListener('mousemove', function(e) {
    position.innerHTML = 'Position X : ' + e.clientX + 'px<br
/>Position Y : ' + e.clientY + 'px';
  }, false);
</script>
```

[Essayer !](#)

Pas très compliqué n'est-ce pas ? Bon, là je peux comprendre que vous trouviez l'intérêt de ce code assez limité, mais quand vous saurez manipuler le CSS des éléments vous pourrez, par exemple, faire en sorte que des éléments HTML suivent votre curseur, ce sera déjà bien plus sympathique 😊 !

Récupérer l'élément en relation avec un événement de souris

Cette fois nous allons étudier un attribut un peu plus "exotique" qui est assez peu utilisé mais peut pourtant se révéler très utile ! Il s'agit de **relatedTarget** et il ne s'utilise qu'avec les événements **mouseover** et **mouseout**.

Cet attribut remplit deux fonctions différentes selon l'événement utilisé. Avec l'événement **mouseout**, il vous fournira l'objet de l'élément sur lequel le curseur vient d'entrer ; avec l'événement **mouseover**, il vous fournira l'objet de l'élément dont le curseur vient de sortir.

Voici un exemple qui illustre son fonctionnement :

Code : HTML

```
<p id="result"></p>

<div id="parent1">
  Parent N°1<br />
  Mouseover sur l'enfant
  <div id="child1">Enfant N°1</div>
</div>

<div id="parent2">
  Parent N°2<br />
  Mouseout sur l'enfant
  <div id="child2">Enfant N°2</div>
</div>

<script type="text/javascript">
  var child1 = document.getElementById('child1'),
      child2 = document.getElementById('child2'),
      result = document.getElementById('result');

  child1.addEventListener('mouseover', function(e) {
    result.innerHTML = "L'élément quitté juste avant que le curseur
```

```
n'entre sur l'enfant N°1 est : " + e.relatedTarget.id;
}, false);

child2.addEventListener('mouseout', function(e) {
    result.innerHTML = "L'élément survolé juste après que le curseur
n'ait quitté l'enfant N°2 est : " + e.relatedTarget.id;
}, false);
</script>
```

Essayer !

Concernant Internet Explorer (toutes versions antérieures à la 9ème), il vous faut utiliser les attributs **fromElement** et **toElement** de la façon suivante :

Code : JavaScript

```
child1.attachEvent('onmouseover', function(e) {
    result.innerHTML = "L'élément quitté juste avant que le curseur
n'entre sur l'enfant N°1 est : " + e.fromElement.id;
});

child2.attachEvent('onmouseout', function(e) {
    result.innerHTML = "L'élément survolé juste après que le curseur
n'ait quitté l'enfant N°2 est : " + e.toElement.id;
});
```

Récupérer les touches frappées par l'utilisateur

Tout comme la position du curseur, il est possible de récupérer les touches frappées par l'utilisateur. Pour cela, nous allons utiliser les attributs **keyCode** et **which** qui contiennent chacun un code pour chaque caractère tapé.



Mais pourquoi deux attributs ?

Tout simplement parce qu'ils ont chacun une utilisation spécifique :

- Les événements **keyup** et **keydown** se déclenchent à chaque touche. Ainsi, si on prend l'exemple de l'événement **keydown** et que vous souhaitez faire une combinaison de touches telle que **Maj + A** vous déclencherez deux fois l'événement. Pour ces deux événements, vous pouvez utiliser soit l'attribut **keyCode** soit l'attribut **which** qui récupéreront tous les deux le code de chaque touche frappée. Cependant, l'attribut **which** n'est pas supporté par Internet Explorer (pour les versions antérieures à la 9ème), je vous conseille donc l'utilisation de **keyCode**.
- Concernant l'événement **keypress**, celui-ci aussi se déclenche à chaque touche mais peut aussi détecter les combinaisons de touches. Cela veut dire que si vous appuyez simultanément sur les touches **Maj + A**, il va analyser la combinaison effectuée et en déduire un code adéquat, dans notre cas il va deviner qu'il s'agit d'un A majuscule. Pour récupérer le code obtenu par **keypress**, il vous faut utiliser l'attribut **which**. Cependant, pour les touches dont la représentation écrite n'est pas possible (comme les touches fléchées par exemple), il vous faudra utiliser l'attribut **keyCode**.

Concernant Internet Explorer dans ses versions antérieures à la 9ème, il vous faudra utiliser **keyCode** dans tous les cas.

Bref, plutôt qu'un long charabia inutile, passons aux exemples :

Code : HTML

```
<p id="keyCode"></p>
<p id="which1"></p>
<p id="which2"></p>

<script type="text/javascript">
  var keyCode = document.getElementById('keyCode');
  var which1 = document.getElementById('which1');
  var which2 = document.getElementById('which2');

  document.addEventListener('keyup', function(e) {
    keyCode.innerHTML = 'keyCode : ' + e.keyCode;
  }, false);

  document.addEventListener('keyup', function(e) {
    which1.innerHTML = 'which avec keyup : ' + e.which;
  }, false);

  document.addEventListener('keypress', function(e) {
    which2.innerHTML = 'which avec keypress : ' + e.which;
  }, false);
</script>
```

Essayer !

Je ne pense pas avoir besoin de vous expliquer plus en détail comment fonctionne ce code, vous avez toutes les informations listées avant ce code.

Bloquer l'action par défaut de certains événements

Eh oui, on y revient ! Nous avons vu qu'il est possible de bloquer l'action par défaut de certains événements, comme la redirection d'un lien vers une page web. Sans le DOM-2, cette opération était très simple vu qu'il suffisait d'écrire **return false**; . Avec l'objet **Event**, c'est tout aussi simple vu qu'il suffit juste d'appeler la méthode **preventDefault()** !

Reprenons l'exemple que nous avons utilisé pour les événements sans le DOM et utilisons donc cette méthode :

Code : HTML

```
<a id="link" href="http://www.siteduzero.com">Cliquez-moi !</a>

<script type="text/javascript">
  var link = document.getElementById('link');

  link.addEventListener('click', function(e) {
    e.preventDefault(); // On bloque l'action par défaut de cet
    événement
    alert('Vous avez cliqué !');
  }, false);
</script>
```

Essayer !

C'est simple comme bonjour et ça fonctionne sans problème avec tous les navigateurs, que demander de plus ? Enfin, tous les navigateurs sauf les versions d'Internet Explorer antérieures à la 9ème, comme d'habitude... Pour IE, il va vous falloir utiliser la propriété **returnValue** et lui attribuer la valeur **false** pour bloquer l'action par défaut :

Code : JavaScript

```
e.returnValue = false;
```

Pour avoir un code compatible entre tous les navigateurs, utilisez donc ceci :

Code : JavaScript

```
e.returnValue = false;  
if (e.preventDefault) {  
    e.preventDefault();  
}
```

Déclencher soi-même les événements

Nous allons aborder ici une partie assez avancée sur l'utilisation des événements : leur déclenchement manuel, sans intervention de l'utilisateur ! Enfin nous allons surtout aborder leur déclenchement quand on les utilise avec le DOM-2.

En effet, alors qu'il est plutôt simple d'exécuter manuellement un événement DOM-0 en écrivant, par exemple, `el.onclick()` ; il est autrement plus compliqué de le faire en DOM-2 pour la simple et bonne raison qu'il faut gérer les valeurs à passer à l'objet **Event** lors de l'exécution de l'événement.

La procédure standard

Alors comment s'y prendre avec le DOM-2 ? Tout d'abord il nous faut deux méthodes : **createEvent()** et **dispatchEvent()**. La première permet de créer notre événement fictif, la seconde permet de l'appliquer à un élément HTML.

Comme vous le savez déjà, certains événements possèdent des attributs dans l'objet **Event** qui sont spécifiques à leur propre utilisation, par exemple, l'événement **keypress** qui possède les attributs **keyCode** et **which**. Afin de gérer ces différences entre chaque événement, ceux-ci sont classés par modules, ainsi nous retrouvons les événements **keypress**, **keyup** et **keydown** dans un seul et même module puisque ces trois événements ont des attributs et méthodes identiques.

Ces modules sont au nombre de cinq mais nous n'allons en étudier que quatre, que voici :

- **Events** :
Module général. Il englobe tous les événements existants mais ne permet pas de gestion affinée pour les événements possédant des caractéristiques particulières.
- **HTMLEvents** :
Sous-module de **Events**, il est dédié aux modifications HTML. Nous y retrouvons les événements **abort**, **blur**, **change**, **error**, **focus**, **load**, **reset**, **resize**, **scroll**, **select**, **submit**, **unload**.
- **UIEvents** :
Sous-module de **Events**, il est dédié à l'interface utilisateur. Il gère les événements **DOMActivate**, **DOMFocusIn**, **DOMFocusOut** et, par extension, les événements **keypress**, **keyup** et **keydown**. Ces trois derniers événements possèdent un module qui leur est propre dans la spécification du DOM-3 (qui est encore loin d'être implémentée sur tous les navigateurs) mais pas dans le DOM-2, donc, par défaut, ils sont accessibles dans le module **UIEvents**.
- **MouseEvent** :
Sous-module de **UIEvents**, il est dédié à la gestion de la souris. Il gère les événements **click**, **mouseout**, **mouseover**, **mousemove**, **mousedown** et **mouseup**. L'événement **dblclick** n'est pas supporté.

Le cinquième module se nomme **MutationEvents** mais il reste quelque peu inutile en raison de son implémentation hasardeuse et de son utilisation extrêmement faible.



Très bien, je sais qu'il existe des modules, mais il servent à quoi ?

Chacun de ces modules peut être passé en paramètre à la méthode **createEvent()**, de cette manière :

Code : JavaScript

```
// On crée un événement classé dans le module Events et on le
renvoie dans la variable fakeEvent :

var fakeEvent = document.createEvent('Events');
```

Une fois l'événement créé, nous allons pouvoir l'initialiser de la manière suivante :

Code : JavaScript

```
fakeEvent.initEvent('focus', false, false);
```

Comme vous pouvez le constater, nous avons ici utilisé la méthode **initEvent()**, celle-ci est associée au module **Events** et **HTMLEvents**. Chaque module possède sa propre méthode d'initialisation, voici un tableau récapitulatif avec les liens vers la documentation de Mozilla :

Module	Méthode associée
Events	initEvent()
HTMLEvents	initEvent()
UIEvents	initUIEvent()
MouseEvents	initMouseEvent()

Si vous avez consulté les liens que j'ai fourni ci-dessus, vous constaterez que chacune de ces méthodes possède des arguments différents, ces arguments sont, pour la plupart d'entre eux, les attributs que vous retrouverez dans l'objet **Event** lorsque votre événement sera exécuté. Ainsi, pour la méthode **initMouseEvent** nous voyons qu'il existe des arguments nommés **clientX** et **clientY**, ils ne vous rappellent rien ? Oui, ce sont bien les attributs de l'objet **Event** qui définissent la position du curseur. Si vous avez bien suivi, vous pouvez maintenant en déduire qu'un événement fictif, en plus de déclencher l'événement souhaité, permet de faire passer des informations personnalisées comme la position du curseur ou bien d'autres choses.

Bien, détaillons maintenant l'utilisation de chacune de ces méthodes :

Code : JavaScript

```
initEvent(type, canBubble, cancelable);
```

Ici, rien de bien compliqué, vous définissez le type de votre événement (**focus**, **blur**, etc...), si la phase bouillonnement doit se déclencher et enfin si l'événement peut être annulé ou non (avec **preventDefault()**).



Cette méthode peut servir à initialiser tous les événements existants, mais vous ne pourrez pas personnaliser certaines valeurs spécifiques à quelques événements comme la position du curseur par exemple.

Au passage, notez bien que cette méthode s'utilise avec deux modules différents : **Events** et **HTMLEvents**.

Code : JavaScript

```
initUIEvent(type, canBubble, cancelable, view, detail);
```

Nous avons ici deux nouveaux arguments : **view** et **detail**. Pour le premier, il s'agit juste de lui fournir l'objet associé à notre objet **document**, dans les navigateurs web il s'agit toujours de l'objet **window** et rien d'autre. Pour le deuxième argument, nous devons normalement lui fournir le nombre de clics effectués par la souris au moment de l'événement, cela paraît stupide mais vu que le module **MouseEvents** et un sous-module de **UIEvents** cela explique (plus ou moins) la présence de cet argument. Bref, pour cet argument, contentez-vous de le mettre constamment à 1 (même pour la méthode **initMouseEvent()**), la valeur la plus courante.

Code : JavaScript

```
initMouseEvent (type, canBubble, cancelable, view,
               detail, screenX, screenY, clientX, clientY,
               ctrlKey, altKey, shiftKey, metaKey,
               button, relatedTarget);
```

Ici nous avons de nombreux arguments supplémentaires (qui ne sont pas facultatifs). Nous retrouvons les arguments **screenX** et **screenY** qui définissent la position du curseur par rapport au coin supérieur gauche de l'écran, idem pour **clientX** et **clientY** mais par rapport au coin supérieur gauche de la page. Les quatre arguments suivants sont des booléens et définissent si les touches Ctrl, Alt, Shift et "Meta" ont été enfoncées.

Nous avons ensuite l'argument **button** qui définit quel bouton a été cliqué, la valeur habituelle est 1 ([plus d'infos sur MDC](#)). Et pour terminer nous avons l'argument **relatedTarget** dont je ne ré-expliquerai pas le fonctionnement. Si vous n'avez pas besoin de cet argument, mettez-le à **null**.



Comme vous avez pu le constater, il n'existe aucun argument du style **keyCode** ou **which** dans aucune des méthodes présentées. Cela est dû au fait que les événements liés aux appuis de touches ne sont pas entièrement supportés par le DOM-2, il n'y a malheureusement pas vraiment de solution à l'heure actuelle.

Bien, après ce long passage sur les méthodes d'initialisation, revenons à notre événement fictif nommé **fakeEvent** :

Code : JavaScript

```
var fakeEvent = document.createEvent('Events');
fakeEvent.initEvent('focus', false, false);
```

Maintenant que notre événement est créé et initialisé, il ne nous reste plus qu'à l'appliquer à un élément HTML qui possède un événement **focus** avec la méthode **dispatchEvent()** :

Code : JavaScript

```
el.dispatchEvent(fakeEvent); // On applique l'événement fictif nommé
                               "fakeEvent" sur un élément "el"
```

Bien, après la théorie, il est temps de passer à un exemple concret :

Code : HTML

```
<input id="text" type="text" value="Faites moi croire que vous me
cliquez avec un curseur en-dehors de la page =" size="80" />
  <br /><br />
<input type="button" value="Activer l'événement du clique en faisant
croire à une position absurde du curseur !" onclick="applyEvent();"
/>

<script type="text/javascript">

  var text = document.getElementById('text');

  text.addEventListener('click', function(e) {
```



```
e.target.value = 'La position du curseur par rapport à la page
est :\nX = ' + e.clientX + 'px\nY = ' + e.clientY + 'px';
}, false);

function applyEvent() {
    var fakeEvent = document.createEvent('MouseEvents');

    fakeEvent.initMouseEvent('click', false, false, window, 1, 0, 0,
-10000, -10000, false, false, false, false, 0, null);
    text.dispatchEvent(fakeEvent);
}

</script>
```

Essayer !

Et voilà ! Alors je reconnais que mon exemple n'a rien d'incroyable mais je pense qu'il vous suffira amplement pour comprendre le concept des événements fictifs.



Si vous avez testé ce code, vous aurez sûrement remarqué que lorsque vous cliquez sur le champ de texte le focus lui est donné, en revanche ce n'est pas le cas lorsque vous déclenchez l'événement fictif, pourquoi ? Cela fait parti de la sécurité, si vous déclenchez manuellement un événement (avec un événement fictif bien sûr) alors l'action par défaut de l'événement ne s'exécutera pas. Il est donc, par exemple, impossible de simuler un clique sur un champ de type **file** ou autre.

La procédure selon Internet Explorer < 9

Comme vous pouvez vous en douter, la manière de faire sous Internet Explorer (antérieur à la v9) est encore une fois différente de la manière standard. Cependant, la façon de faire d'IE n'est pas mauvaise non plus, la méthode qu'il utilise n'exige pas une flopée d'arguments pour pouvoir fonctionner, juste le type de l'événement à exécuter et un objet en argument facultatif qui contient les attributs spécifiques à l'événement (ex: clientX, screenX, etc...).

Sur Internet Explorer, les étapes sont considérablement raccourcies et une seule méthode suffit pour déclencher un événement sans aucun paramètre superflu :

Code : JavaScript

```
el.fireEvent('onmousemove'); // On déclenche manuellement
l'événement "mousemove" sur l'élément "el"
```

Après, pour passer des attributs à l'événement, rien de bien compliqué :

Code : JavaScript

```
var attr = document.createEventObject(); // On déclare un objet de
type "Event" dans lequel on va stocker nos informations

attr.clientX = 100; // On définit la position horizontale du curseur
à 100px dans notre objet "attr"
attr.clientY = 100; // On définit la position verticale du curseur à
100px dans notre objet "attr"

el.fireEvent('onmousemove', attr); // Et on déclenche notre
événement avec en second argument les informations de l'événement
```

Et voilà ! La méthode d'IE n'est pas mal non plus n'est-ce pas ? L'avantage est que l'on n'est pas obligé de spécifier chaque attribut de l'événement comme avec la méthode standard du DOM-2.

Les formulaires

Après l'étude des événements, il est temps de passer aux formulaires ! Ici commence l'interaction avec l'utilisateur grâce aux nombreux attributs et méthodes dont sont dotés les éléments HTML utilisés dans les formulaires.

Il s'agit cette fois d'un très court chapitre, ça vous changera un peu du bourrage de crâne habituel !

Les attributs

Les formulaires sont simples à utiliser, cependant il faut d'abord mémoriser quelques attributs de base.

Comme vous le savez déjà, il est possible d'accéder à n'importe quel attribut d'un élément HTML juste en tapant son nom, il en va donc de même pour des attributs spécifiques aux éléments d'un formulaire comme **value**, **disabled**, **checked**, etc... Nous allons voir ici comment utiliser ces attributs spécifiques aux formulaires.

Un attribut classique : value

Commençons par l'attribut le plus connu et le plus utilisé : **value** ! Pour ceux qui ne se souviennent pas, cet attribut permet de définir une valeur à différents éléments d'un formulaire comme les **input**, les **button**, etc...

Son fonctionnement est simple comme bonjour, on lui assigne une valeur (un nombre ou une chaîne de caractères) et elle est immédiatement affichée sur votre élément HTML, exemple :

Code : HTML

```
<input id="text" type="text" size="60" value="Vous n'avez pas le
focus !" />

<script type="text/javascript">
  var text = document.getElementById('text');

  text.addEventListener('focus', function(e) {
    e.target.value = "Vous avez le focus !";
  }, true);

  text.addEventListener('blur', function(e) {
    e.target.value = "Vous n'avez pas le focus !";
  }, true);
</script>
```

[Essayer !](#)

Alors par contre, une petite précision ! Cet attribut s'utilise aussi avec un **textarea** ! En effet, en HTML, on prend souvent l'habitude de mettre du texte dans un textarea en écrivant :

Code : HTML

```
<textarea>Et voilà du texte !</textarea>
```

Du coup, en Javascript, on est souvent tenté d'utiliser **innerHTML** pour récupérer le contenu de notre textarea, cependant cela ne fonctionne pas : il faut utiliser **value** à la place !

Les booléens avec : disabled, checked et readonly

Contrairement à l'attribut **value**, les trois attributs **disabled**, **checked** et **readonly** ne s'utilisent pas de la même manière qu'en XHTML où il vous suffit d'écrire, par exemple, `<input type="text" disabled="disabled" />` pour désactiver un champ de texte. En Javascript, ces trois attributs deviennent des booléens. Ainsi, il vous suffit de faire comme ceci pour désactiver un champ de texte :

Code : HTML

```

<input id="text" type="text" />

<script type="text/javascript">
  var text = document.getElementById('text');

  text.disabled = true;
</script>

```

Je ne pense pas qu'il soit nécessaire de vous expliquer comment fonctionne l'attribut **checked** avec une checkbox, il suffit d'opérer de la manière qu'avec l'attribut **disabled** ci-dessus. En revanche, je préfère détailler son utilisation avec les boutons de type **radio**. Chaque bouton radio coché se verra attribuer la valeur **true** à son attribut **checked**, il va donc nous falloir utiliser une boucle **for** pour vérifier quel bouton **radio** a été sélectionné :

Code : HTML

```

<label><input type="radio" name="check" value="1" /> Case n°
1</label><br />
<label><input type="radio" name="check" value="2" /> Case n°
2</label><br />
<label><input type="radio" name="check" value="3" /> Case n°
3</label><br />
<label><input type="radio" name="check" value="4" /> Case n°
4</label>
<br /><br />
<input type="button" value="Afficher la case cochée"
onclick="check();" />

<script type="text/javascript">
  function check() {
    var inputs = document.getElementsByTagName('input'),
        inputsLength = inputs.length;

    for (var i = 0 ; i < inputsLength ; i++) {
      if (inputs[i].type == 'radio' && inputs[i].checked) {
        alert('La case cochée est la n°'+ inputs[i].value);
      }
    }
  }
</script>

```

Essayer !

Voilà donc pour les boutons radio, le principe est simple, il suffit juste d'y penser !

Les listes déroulantes avec : **selectedIndex** et **options**

Les listes déroulantes possèdent elles aussi leurs propres attributs. Nous allons en retenir seulement deux parmi tous ceux qui existent : **selectedIndex**, qui nous donne l'index (l'identifiant) de la valeur sélectionnée, et **options** qui liste dans un tableau les éléments **<option>** de notre liste déroulante. Leur principe de fonctionnement est on ne peut plus classique :

Code : HTML

```

<select id="list">
  <option>Sélectionnez votre sexe</option>
  <option>Homme</option>
  <option>Femme</option>

```

```
</select>

<script type="text/javascript">
  var list = document.getElementById('list');

  list.addEventListener('change', function() {

    // On affiche le contenu de l'élément <option> ciblé par la
    propriété selectedIndex
    alert(list.options[list.selectedIndex].innerHTML);

  }, true);
</script>
```

Essayer !

Les méthodes et un retour sur quelques événements

Les formulaires ne possèdent pas que des attributs mais aussi des méthodes dont certaines sont bien pratiques ! Tout en abordant leur utilisation, nous en profiterons pour revenir sur certains événements étudiés au chapitre précédent.

Les méthodes spécifiques à l'élément <form>

Un formulaire, ou plus exactement l'élément **form**, possède deux méthodes intéressantes. La première, **submit()**, permet d'effectuer l'envoi d'un formulaire sans l'intervention de l'utilisateur. La deuxième, **reset()**, permet de réinitialiser tous les champs d'un formulaire.

Si vous êtes un habitué des formulaires HTML, vous aurez deviné que ces deux méthodes ont le même rôle que les éléments **input** de type **submit** ou **reset**.

L'utilisation de ces deux méthodes est simple comme bonjour, il vous suffit juste de les appeler sans aucun paramètre (elles n'en ont pas) et c'est fini :

Code : JavaScript

```
var el = document.getElementById('un_id_de_formulaire');

el.submit(); // Le formulaire est expédié
el.reset(); // Le formulaire est réinitialisé
```

Maintenant revenons sur deux événements : **submit** et **reset**, encore les mêmes noms ! Je suppose qu'il n'y a pas besoin de vous expliquer quand est-ce que l'un et l'autre se déclenchent, cela paraît évident. Cependant, il est important de préciser une chose : envoyer un formulaire avec la méthode **submit()** de Javascript ne déclenchera jamais l'événement **submit** ! Mais, dans le doute, voici un exemple complet dans le cas où vous n'auriez pas tout compris :

Code : HTML

```
<form id="myForm">
  <input type="text" value="Entrez un texte" />
  <br /><br />
  <input type="submit" value="Submit !" />
  <input type="reset" value="Reset !" />
</form>

<script type="text/javascript">
  var myForm = document.getElementById('myForm');

  myForm.addEventListener('submit', function(e) {
    alert('Vous avez envoyé le formulaire !\n\nMais celui-ci a été
bloqué pour que vous ne changiez pas de page.');
```

Essayer !

La gestion du focus et de la sélection

Vous vous souvenez des événements pour détecter l'activation ou la désactivation du focus sur un élément ? Eh bien il existe aussi deux méthodes, **focus()** et **blur()**, permettant respectivement de donner et retirer le focus à un élément. Leur utilisation est très simple :

Code : HTML

```
<input id="text" type="text" value="Entrez un texte" />
<br /><br />
<input type="button" value="Donner le focus"
onclick="document.getElementById('text').focus();" /><br />
<input type="button" value="Retirer le focus"
onclick="document.getElementById('text').blur();" />
```

[Essayer !](#)

Dans le même genre, il existe la méthode **select()** qui, en plus de donner le focus à l'élément, sélectionne le texte de celui-ci si cela est possible :

Code : HTML

```
<input id="text" type="text" value="Entrez un texte" />
<br /><br />
<input type="button" value="Sélectionner le texte"
onclick="document.getElementById('text').select();" />
```

[Essayer !](#)

Bien sûr, cette méthode ne fonctionne que sur des champs de texte comme un **input** de type **text** ou bien un **textarea**.

Explications sur l'événement change

Je pense qu'il est important de revenir sur cet événement afin de clarifier quelques petits problèmes que vous pourrez rencontrer en l'utilisant. Tout d'abord, il est bon de savoir que cet événement attend que l'élément auquel il est attaché perde le focus avant de se déclencher (si il y a eu modification du contenu de l'élément). Donc, si vous souhaitez vérifier l'état d'un input à chacune de ses modifications sans attendre la perte de focus, il vous faudra plutôt utiliser d'autres événements du style **keyup** (et ses variantes) ou **click**, cela dépend du type d'élément vérifié.

Et, deuxième et dernier point, cet événement est bien entendu utilisable sur n'importe quel input dont l'état peut changer, par exemple une **checkbox** ou un **input file**, n'allez surtout pas croire que cet événement est réservé seulement aux champs de texte !

Manipuler le CSS

Le Javascript est un langage permettant de rendre une page web dynamique du côté du client. Seulement, quand on pense à "dynamique", on pense aussi à "animations". Or, pour faire des animations, il faut savoir accéder au CSS et le modifier. C'est ce que nous allons étudier dans ce chapitre.

Au programme, l'édition du CSS et son analyse. Initialement, il était aussi prévu d'étudier les feuilles de style mais comme leurs méthodes de manipulation changent d'un navigateur à l'autre, nous allons faire l'impasse dessus d'autant plus qu'elles ne servent réellement pas à grand chose en Javascript.

Enfin, pour terminer le chapitre, nous étudierons comment réaliser un petit système de Drag & Drop : Enfin un sujet intéressant !

Éditer les propriétés CSS

Avant de s'attaquer immédiatement à la manipulation du CSS, rafraîchissons-nous un peu la mémoire :

Quelques rappels sur le CSS

CSS est l'abréviation de **Cascading Style Sheets**, c'est un langage qui permet d'éditer l'aspect graphique des éléments HTML et XML. Il est possible d'éditer le CSS d'un seul élément comme on le ferait en HTML de la manière suivante :

Code : HTML

```
<div style="color:red;">Le CSS de cet élément a été modifié avec  
l'attribut STYLE. Il n'y a donc que lui qui possède un texte de  
couleur rouge.</div>
```

Mais on peut tout aussi bien éditer les feuilles de style qui se présentent de la manière suivante :

Code : CSS

```
div {  
  color: red; /* Ici on modifie la couleur du texte de tous les  
  éléments DIV. */  
}
```

Je pense qu'il est de bon ton de vous le rappeler : les propriétés CSS de l'attribut **style** sont prioritaires sur les propriétés d'une feuille de style ! Ainsi, dans le code d'exemple ci-dessous, le texte n'est pas rouge mais bleu :

Code : HTML

```
<style type="text/css">  
  div {  
    color: red;  
  }  
</style>  
  
<div style="color:blue;">I'm blue ! DABADIDABADA !</div>
```

Voilà tout pour les rappels sur le CSS. Oui, c'était très rapide, mais je voulais simplement insister sur cette histoire de priorité des styles CSS, parce que ça va vous servir !

Éditer les styles CSS d'un élément

Comme dit ci-dessus, il y a deux manières de modifier le CSS d'un élément HTML, nous allons ici aborder la méthode la plus simple et la plus utilisée : l'utilisation de l'attribut **style**. L'édition des feuilles de style ne sera pas abordée car elle est complètement inutile en plus d'être mal gérée par de nombreux navigateurs.

Alors comment accéder à l'attribut **style** de notre élément ? Eh bien de la même manière que pour accéder à n'importe quel attribut de notre élément :

Code : JavaScript

```
el.style; // On accède à l'attribut "style" de l'élément "el"
```

Une fois que l'on a accédé à notre attribut, comment modifier les propriétés CSS ? Eh bien tout simplement en écrivant leur nom et en leur attribuant une valeur, **width** (pour la largeur) par exemple :

Code : JavaScript

```
el.style.width = '150px'; // On modifie la largeur de notre élément à 150px
```

Pensez bien à écrire l'unité de votre valeur, il est fréquent que moi-même je l'oublie et généralement ça pose de nombreux problèmes dans un code !

Maintenant, j'ai une question pour vous : Comment accède-t-on à une propriété CSS qui possède un nom composé ? En Javascript, les tirets sont interdits dans les noms des propriétés, ce qui fait que ce code ne fonctionne pas :

Code : JavaScript

```
el.style.background-color = 'blue'; // Ce code ne fonctionne pas, les tirets sont interdits
```

La solution est simple : On supprime les tirets et chaque mot suivant un tiret voit sa première lettre devenir une majuscule. Ainsi, notre code précédent doit s'écrire de la manière suivante pour fonctionner correctement :

Code : JavaScript

```
el.style.backgroundColor = 'blue'; // Après avoir supprimé le tiret et ajouté une majuscule au deuxième mot, le code fonctionne
```

Comme vous pouvez le constater, l'édition du CSS d'un élément n'est pas bien compliqué. Cependant, il y a une limitation de taille : la lecture des propriétés CSS !

Prenons un exemple :

Code : HTML

```
<style type="text/css">
  #myDiv {
    background-color: orange;
  }
</style>

<div id="myDiv">Je possède un fond orange.</div>

<script type="text/javascript">
  var myDiv = document.getElementById('myDiv');

  alert('Selon Javascript, la couleur de fond de ce DIV est : ' +
myDiv.style.backgroundColor); // On affiche la couleur de fond
</script>
```

Essayer !

Et on obtient : rien ! Pourquoi ? Parce que notre code va lire uniquement les valeurs contenues dans l'attribut **style**. C'est-à-dire, rien du tout dans notre exemple, car on a modifié les styles CSS depuis une feuille de style, et non pas depuis l'attribut **style**.

En revanche, en modifiant le CSS avec l'attribut **style**, on retrouve sans problème la couleur de notre fond :

Code : HTML

```
<div id="myDiv" style="background-color: orange">Je possède un fond
orange.</div>

<script type="text/javascript">
  var myDiv = document.getElementById('myDiv');

  alert('Selon Javascript, la couleur de fond de ce DIV est : ' +
myDiv.style.backgroundColor); // On affiche la couleur de fond
</script>
```

Essayer !

C'est gênant n'est-ce pas ? Malheureusement, on ne peut pas y faire grand chose à partir de l'attribut **style**, pour cela nous allons devoir utiliser la méthode **getComputedStyle()** !

Récupérer les propriétés CSS

La fonction `getComputedStyle()`

Comme vous avez pu le constater, il n'est pas possible de récupérer les valeurs des propriétés CSS d'un élément par le biais de l'attribut `style` vu que celui-ci n'intègre pas les propriétés CSS des feuilles de style, ce qui nous limite énormément dans nos possibilités d'analyse... Heureusement, il existe une fonction permettant de remédier à ce problème : `getComputedStyle()` !

Cette fonction va se charger de récupérer, à notre place, la valeur de n'importe quelle propriété CSS ! Que la propriété soit déclarée dans l'attribut `style`, une feuille de style ou bien même encore calculée automatiquement, cela importe peu : `getComputedStyle()` la récupérera sans problème.

Son fonctionnement est très simple et se fait de cette manière :

Code : HTML

```
<style type="text/css">
  #text {
    color: red;
  }
</style>

<span id="text"></span>

<script type="text/javascript">
  var text = document.getElementById('text'),
      color = getComputedStyle(text, null).color;

  alert(color);
</script>
```

[Essayer !](#)



À quoi sert ce deuxième argument que tu as mis à `null` ?

Il s'agit en fait d'un argument facultatif qui permet de spécifier une pseudo-classe à notre élément, je ne vais cependant pas m'y attarder plus longtemps car on ne s'en servira pas. En effet, Internet Explorer (pour toute version antérieure à la 9ème) ne supporte pas l'utilisation de la méthode `getComputedStyle` et utilise à la place l'attribut `currentStyle` qui, lui, ne supporte pas l'utilisation des pseudo-classes.

Il n'y a rien qui vous choque dans mon précédent paragraphe ? J'ai dit qu'il s'agissait d'un argument facultatif alors que pourtant on l'a spécifié ! Il ne s'agit pas d'une erreur de ma part mais tout simplement parce que cette fois c'est Firefox qui nous embête : Il considère cet argument comme étant obligatoire. Ce comportement perdure jusqu'à la version 4 de Firefox.

L'alternative `currentStyle` pour Internet Explorer < 9

Pour Internet Explorer, il existe encore une petite différence car la méthode `getComputedStyle()` n'existe pas ! À la place on va se servir de `currentStyle` :

Code : HTML

```
<style type="text/css">
  #text {
    color: red;
  }
</style>
```

```

<span id="text"></span>

<script type="text/javascript">
  var text = document.getElementById('text'),
      color = text.currentStyle.color;

  alert(color);
</script>

```

Essayer (Internet Explorer seulement) !

Pas bien compliqué n'est-ce pas ?



Toutes les valeurs obtenues par le biais de `getComputedStyle()` ou `currentStyle` sont en lecture seule !

Les attributs de type "offset"

Certaines valeurs de positionnement ou de taille des éléments ne pourront pas être obtenues de façon "simple" avec `getComputedStyle()`, pour pallier à ce problème il existe les attributs "offset" qui sont, dans notre cas, au nombre de cinq :

Nom de l'attribut	Contient...
offsetWidth	Contient la largeur complète (width + padding + border) de l'élément.
offsetHeight	Contient la hauteur complète (height + padding + border) de l'élément.
offsetLeft	Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord gauche de son élément parent.
offsetTop	Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord supérieur de son élément parent.
offsetParent	Utile que pour un élément en position absolue ou relative ! Contient l'objet de l'élément parent par rapport auquel est positionné l'élément actuel.

Leur utilisation ne se fait pas de la même manière que n'importe quelle propriété CSS tout d'abord parce que ce ne sont pas des propriétés CSS ! Ce sont juste des attributs (en lecture seule) mis à jour dynamiquement qui concernent certains états physiques de votre élément.

Pour les utiliser, on oublie l'attribut **style** vu qu'il ne s'agit pas de propriétés CSS et on les lit directement sur l'objet de notre élément HTML :

Code : JavaScript

```

alert(el.offsetHeight); // On affiche la hauteur complète de notre
élément HTML

```



Faites bien attention : Les valeurs contenues dans ces attributs (à part **offsetParent**) sont exprimées en pixels et sont donc de type **Number**, pas comme les propriétés CSS qui sont de type **String** et pour lesquelles les unités sont spécifiées (px, cm, em, etc...).

L'attribut `offsetParent`

Concernant l'attribut **`offsetParent`**, celui-ci contient l'objet de l'élément parent par rapport auquel est positionné votre élément actuel. C'est bien, mais qu'est-ce que ça veut dire ?!

Ce que je vais vous expliquer concerne les connaissances HTML et CSS et non pas Javascript ! Seulement, je pense que certains d'entre vous ne connaissent pas ce fonctionnement particulier du positionnement absolu, je préfère donc vous le rappeler.

Lorsque vous décidez de mettre un de vos éléments HTML en positionnement absolu, celui-ci est sorti du positionnement par défaut des éléments HTML et va aller se placer tout en haut à gauche de votre page web, par dessus tous les autres éléments. Seulement, ce principe n'est applicable que lorsque votre élément n'est pas déjà lui-même placé dans un élément en positionnement absolu. Si cela arrive, alors votre élément se positionnera non plus par rapport au coin supérieur gauche de la page web, mais par rapport au coin supérieur gauche du précédent élément placé en positionnement absolu (ou relatif/fixe).

Ce système de positionnement est clair ? Bon, nous pouvons alors revenir à notre attribut **`offsetParent`** ! S'il existe, c'est parce que les attributs **`offsetTop`** et **`offsetLeft`** contiennent le positionnement de votre élément par rapport à son précédent élément parent et non pas par rapport à la page ! Si on veut obtenir son positionnement par rapport à la page, il faudra alors aussi ajouter les valeurs de positionnement de son (ses) élément(s) parent(s).

Voici le problème mis en pratique ainsi que sa solution :

Code : HTML

```
<style type="text/css">
  #parent, #child {
    position: absolute;
    top: 50px; left: 100px;
  }

  #parent {
    width: 200px; height: 200px;
    background-color: blue;
  }

  #child {
    width: 50px; height: 50px;
    background-color: red;
  }
</style>

<div id="parent">
  <div id="child"></div>
</div>

<script type="text/javascript">
  var parent = document.getElementById('parent');
  var child = document.getElementById('child');

  alert("Sans la fonction de calcul, la position de l'élément enfant
est : \n\n" +
      'offsetTop : ' + child.offsetTop + 'px\n' +
      'offsetLeft : ' + child.offsetLeft + 'px');

  function getOffset(el) { // Notre fonction qui calcul le
positionnement complet
    var top = 0, left = 0;

    do {
      top += el.offsetTop;
```

```

    left += el.offsetLeft;
  } while (el = el.offsetParent); // Tant que "el" reçoit un
  "offsetParent" valide alors on additionne les valeurs des offset.

  return { // On retourne un objet, ça nous permet de retourner
  les deux valeurs calculées.
    top: top,
    left: left
  };
}

alert("Avec la fonction de calcul, la position de l'élément enfant
est : \n\n" +
      'offsetTop : ' + getOffset(child).top + 'px\n' +
      'offsetLeft : ' + getOffset(child).left + 'px');
</script>

```

Essayer !

Comme vous pouvez le constater, les valeurs seules de positionnement de notre élément enfant ne sont pas correctes si on souhaite connaître son positionnement par rapport à la page et non pas par rapport à l'élément parent. On est finalement obligé d'utiliser une fonction personnelle pour calculer le positionnement par rapport à la page.

Concernant cette fonction, je vais insister sur la boucle qu'elle contient car j'ai peur que le principe ne soit pas clair pour vous :

Code : JavaScript

```

do {
  top += el.offsetTop;
  left += el.offsetLeft;
} while (el = el.offsetParent);

```

Si on utilise ce code HTML :

Code : HTML

```

<body>
  <div id="parent" style="position:absolute; top:200px;
left:200px;">
    <div id="child" style="position:absolute; top:100px;
left:100px;"></div>
  </div>
</body>

```

Son schéma de fonctionnement est le suivant pour le calcul des valeurs de positionnement de l'élément **child** :

- La boucle s'exécute une première fois en ajoutant les valeurs de positionnement de l'élément **child** à nos deux variables **top** et **left**. Le calcul effectué est donc :

Code : JavaScript

```

top = 0 + 100; // 100
left = 0 + 100; // 100

```

- Ligne 4, on attribue à **el** l'objet de l'élément parent de **child**. En gros, on monte d'un cran dans l'arbre DOM. L'opération est donc la suivante :

Code : JavaScript

```
el = child.offsetParent; // Le nouvel élément est "parent"
```

- Toujours ligne 4, **el** possède une référence vers un objet valide (qui est l'élément **parent**), la condition est donc vérifiée (l'objet est évalué à **true**) et la boucle s'exécute de nouveau.
- La boucle se répète en ajoutant cette fois les valeurs de positionnement de l'élément **parent** à nos variables **top** et **left**. Le calcul effectué est donc :

Code : JavaScript

```
top = 100 + 200; // 300  
left = 100 + 200; // 300
```

- Ligne 4, cette fois l'objet parent de **parent** est **body**. La boucle va donc se répéter avec **body** qui est un objet valide. Comme on n'a pas touché à ses propriétés CSS il ne possède pas de valeurs de positionnement, le calcul effectué est donc :

Code : JavaScript

```
top = 300 + 0; // 300  
left = 300 + 0; // 300
```

- Ligne 4, **body** a un attribut **offsetParent** qui est à **undefined**, la boucle s'arrête donc.

Voilà tout pour cette boucle ! Son fonctionnement n'est pas bien compliqué mais peut en dérouter certains, c'est pourquoi j'ai préféré vous l'expliquer.



Avant que tu finisses : Pourquoi avoir écrit "hauteur complète (width + padding + border)" ? Qu'est-ce que ça veut dire ?

Il faut savoir qu'en HTML, la largeur (ou hauteur) complète d'un élément correspond à la valeur de width + le padding + les bordures.

Par exemple, sur ce code :

Code : HTML

```
<style type="text/css">  
  #offsetTest {  
    width: 100px; height: 100px;  
    padding: 10px;  
    border: 2px solid black;  
  }  
</style>  
  
<div id="offsetTest"></div>
```

La largeur complète de notre élément **offsetTest** vaut : **100** (width) + **10** (padding-left) + **10** (padding-right) + **2** (border-left) +

2 (border-right) = 124px

Et il s'agit bien de la valeur retournée par notre **offsetWidth** :

Code : JavaScript

```
var offsetTest = document.getElementById('offsetTest');  
alert(offsetTest.offsetWidth);
```

[Essayer le code complet !](#)

Votre premier script interactif !

Soyons francs : Les exercices que nous avons fait précédemment n'étaient quand même pas très utiles sans une réelle interaction avec l'utilisateur. Les **alert()**, **confirm()** et **prompt()** c'est bien sympa un moment, mais on en a vite fait le tour ! Il est donc temps de passer à quelque chose de plus intéressant : un système de Drag & drop ! Enfin... une version très simple !

Il s'agit ici d'un mini-TP, ce qui veut dire qu'il n'est pas très long à réaliser mais il demande un peu de réflexion quand même. Ce TP vous fera utiliser les événements et les manipulations CSS. Tant qu'on y est, si vous vous sentez motivés vous pouvez essayer de faire en sorte que votre script fonctionne aussi sous Internet Explorer (si vous avez bien suivi le cours, vous n'aurez aucun mal).

Présentation de l'exercice

Tout d'abord, qu'est-ce que le Drag & drop ? Il s'agit d'un système permettant le déplacement d'éléments par un simple déplacement de souris. Pour faire simple, c'est comme lorsque vous avez un fichier dans un dossier et que vous le déplacez dans un autre dossier en le faisant glisser avec votre souris.



Et je suis vraiment capable de faire ça ?

Bien évidemment ! Bon, il faut avoir suivi attentivement le cours et se démener un peu, mais c'est parfaitement possible, vous en êtes capables !

Avant de se lancer dans le code, listons les étapes de fonctionnement d'un système de Drag & Drop :

- L'utilisateur enfonce (et ne relâche pas) le bouton de sa souris sur un élément. Le Drag & Drop s'initialise alors en sachant qu'il va devoir gérer le déplacement de cet élément. Pour info, l'événement à utiliser ici est **mousedown**.
- L'utilisateur, tout en laissant le bouton de sa souris enfoncé, commence à déplacer son curseur, l'élément ciblé suit ses mouvements à la trace. L'événement à utiliser est **mousemove** et je vous conseille d'appliquer cet événement à l'élément **document**, je vous expliquerai pourquoi par la suite.
- L'utilisateur relâche le bouton de sa souris. Le Drag & Drop prend alors fin et l'élément ne suit plus le curseur de la souris. L'événement utilisé est **mouseup**.

Alors ? Ça n'a pas l'air si tordu que ça, n'est-ce pas ?

Maintenant que vous savez à peu près ce qu'il faut faire, je vais vous fournir le code HTML de base ainsi que le CSS, ça vous évitera de vous embêter à faire cela vous-même :

Code : HTML

```
<div class="draggableBox">1</div>
<div class="draggableBox">2</div>
<div class="draggableBox">3</div>
```

Code : CSS

```
.draggableBox {
  position: absolute;
  width: 80px; height: 60px;
  padding-top: 10px;
  text-align: center;
  font-size: 40px;
  background-color: #222;
  color: #CCC;
```

```
}
```

Juste deux dernières petites choses, il serait bien :

- Que vous utilisiez un objet dans lequel vous allez placer toutes les fonctions et variables nécessaires au bon fonctionnement de votre code, ce sera bien plus propre.
- Que votre code ne s'applique pas à tous les `<div>` existants mais uniquement à ceux qui possèdent la classe `draggableBox`.

Sur ce, bon courage !

Correction

Vous avez terminé l'exercice ? J'espère que vous l'avez réussi, mais si ce n'est pas le cas ce n'est pas grave ! Si la correction existe, c'est bien parce que personne n'y arrivera parfaitement du premier coup 😊. D'ailleurs, la voici :

Code : JavaScript

```
var storage = []; // Contient l'objet de la div en cours de
déplacement

function addEvent(el, event, func) { // Une fonction pour gérer les
événements sous tous les navigateurs
  if(el.attachEvent) {
    el.attachEvent('on'+event, func);
  } else {
    el.addEventListener(event, func, true);
  }
}

function init() { // La fonction d'initialisation
  var els = document.getElementsByTagName('div'),
      elsLength = els.length;

  for(var i = 0 ; i < elsLength ; i++) {
    if(els[i].className == 'draggableBox') {

      addEvent(els[i], 'mousedown', function(e) { // Initialise le
drag & drop
        var s = storage;
        s['target'] = e.target || event.srcElement;
        s['offsetX'] = e.clientX - s['target'].offsetLeft;
        s['offsetY'] = e.clientY - s['target'].offsetTop;
      });

      addEvent(els[i], 'mouseup', function() { // Termine le drag &
drop
        storage = [];
      });
    }
  }

  addEvent(document, 'mousemove', function(e) { // Permet le suivi
du drag & drop
    var target = storage['target'];

    if(target) { // Si target n'existe pas alors la condition va
renvoyer "undefined", ce qui n'exécutera pas les deux lignes
```

```
suivantes
    target.style.top = e.clientY - storage['offsetY'] + 'px';
    target.style.left = e.clientX - storage['offsetX'] + 'px';
  }
});
}

init(); // On initialise le code avec notre fonction toute prête
```

[Essayer le code complet !](#)

Pour la fonction **addEventListener()**, pas besoin de vous expliquer comment elle fonctionne, vous avez déjà vu ça au chapitre sur les événements, je vous laisse le relire au besoin. Maintenant, votre seul problème dans ce code doit être la fonction **init()**. Quant à la variable **storage**, il ne s'agit que d'un espace de stockage dont je vais vous expliquer le fonctionnement au cours de l'étude de la fonction **init()**.

Commençons !

L'exploration du code HTML

Notre fonction **init** commence par le code suivant :

Code : JavaScript

```
var els = document.getElementsByTagName('div'),
    elsLength = els.length;

for(var i = 0 ; i < elsLength ; i++) {
  if(els[i].className == 'draggableBox') {

    // Code...

  }
}
```

Dans ce code, j'ai volontairement caché les codes d'ajout d'événements car ce qui nous intéresse c'est cette boucle et la condition. Cette boucle couplée à la méthode **getElementsByTagName()**, vous l'avez déjà vue dans le chapitre sur la manipulation du code HTML, elle permet de parcourir tous les éléments HTML d'un type donné. Dans notre cas, nous parcourons tous les éléments **<div>**.

À chaque élément **<div>** trouvé, on vérifie que sa classe correspond bien à **draggableBox**. Pourquoi faire ça ? Parce que si vous ajoutez d'autres éléments **<div>** ils ne seront pas pris en compte sauf si vous leur attribuez la bonne classe, ainsi vous pouvez utiliser des **<div>** sans qu'elles soient forcément draggable.

L'ajout des événements mousedown et mouseup

Dans notre boucle qui parcourt le code HTML, nous avons deux ajouts d'événements que voici :

Code : JavaScript

```
addEventListener(els[i], 'mousedown', function(e) { // Initialise le drag &
drop
  var s = storage;
  s['target'] = e.target || event.srcElement;
  s['offsetX'] = e.clientX - s['target'].offsetLeft;
  s['offsetY'] = e.clientY - s['target'].offsetTop;
```

```
});  
  
addEvent(els[i], 'mouseup', function() { // Termine le drag & drop  
    storage = [];  
});
```

Comme vous pouvez le voir, ces deux événements ne font qu'accéder à la variable **storage**. À quoi nous sert donc cette variable ? Il s'agit tout simplement d'un tableau qui nous sert d'espace de stockage, il permet de mémoriser l'élément actuellement en cours de déplacement ainsi que la position du curseur par rapport à notre élément (je reviendrai sur ce dernier point plus tard).

Bref, dans notre événement **mousedown** (qui initialise le Drag & Drop), on ajoute l'événement ciblé dans l'attribut `storage['target']` puis les positions du curseur par rapport à notre élément dans `storage['offsetX']` et `storage['offsetY']`.

En ce qui concerne notre événement **mouseup** (qui termine le Drag & Drop), on attribue juste un tableau vide à notre attribut **storage**, comme ça tout est vidé !

La gestion du déplacement de notre élément

Jusqu'à présent, notre code ne fait que enregistrer dans la variable **storage** l'élément ciblé pour notre Drag & Drop. Cependant, notre but c'est de faire bouger cet élément. Voilà pourquoi notre événement **mousemove** intervient !

Code : JavaScript

```
addEvent(document, 'mousemove', function(e) { // Permet le suivi du  
    drag & drop  
    var target = storage['target'];  
  
    if(target) { // Si target n'existe pas alors la condition va  
        renvoyer "undefined", ce qui n'exécutera pas les deux lignes  
        suivantes  
        target.style.top = e.clientY - storage['offsetY'] + 'px';  
        target.style.left = e.clientX - storage['offsetX'] + 'px';  
    }  
});
```



Pourquoi notre événement est-il appliqué à l'élément **document** ?

Réfléchissons ! Si j'applique cet événement à mon élément ciblé, que va-t-il se passer ? Dès que l'on bougera la souris, l'événement se déclenchera et tout se passera comme on le souhaite, mais si je me mets à bouger la souris trop rapidement, le curseur va alors sortir de notre élément avant que celui-ci n'ait eu le temps de se déplacer, ce qui fait que l'événement ne se déclenchera plus tant que l'on ne replacera pas notre curseur sur l'élément. La probabilité pour que cela se produise est assez faible mais ça peut arriver !

Un autre problème peut aussi surgir : Dans notre code actuel, nous ne gérons pas la propriété CSS **z-index**, ce qui fait que lorsqu'on déplace le premier élément et qu'on place notre curseur sur un deux autres éléments, le premier élément se retrouve alors en-dessous d'eux. En quoi est-ce un problème ? Eh bien si on a appliqué le **mousemove** sur notre élément au lieu du **document** alors cet événement ne se déclenchera pas vu que l'on bouge notre curseur sur un deux autres éléments et non pas sur notre élément en cours de déplacement.

La solution est donc de mettre l'événement **mousemove** sur notre **document**. Vu que cet événement se propage aux enfants, on est sûr qu'il se déclenchera à n'importe quel déplacement du curseur sur la page.

Le reste du code n'est pas bien sorcier :

- On utilise une condition qui vérifie qu'il existe bien un indice **target** dans notre espace de stockage. Si il n'y en a pas c'est qu'il n'y a aucun Drag & Drop en cours d'exécution.
- On assigne à notre élément cible (**target**) ses nouvelles coordonnées par rapport au curseur.

Alors revenons sur un point important du précédent code : Il nous a fallu enregistrer [la position du curseur par rapport au coin supérieur gauche de notre élément](#) dès l'initialisation du Drag & Drop. Pourquoi ? Car si vous ne le faites pas, à chaque fois que vous déplacerez votre élément, celui-ci placera son bord supérieur gauche sous votre curseur et ce n'est clairement pas ce que l'on souhaite.

Essayez donc par vous-même pour vérifier !

Et voilà pour ce mini-TP, j'espère qu'il vous a plu !

TP : Un formulaire interactif

Nous sommes presque au bout de cette deuxième partie du cours ! Cette dernière aura été très conséquente et il se peut que vous ayez oublié pas mal de choses depuis votre lecture, ce TP va se charger de vous rappeler l'essentiel de ce que nous avons appris ensemble.

Le sujet va porter sur la création d'un formulaire dynamique. Qu'est-ce que j'entends par formulaire dynamique ? Eh bien un formulaire dont une partie des vérifications est effectuée par le Javascript, côté client. On peut par exemple vérifier que l'utilisateur a bien complété tous les champs, ou bien qu'ils contiennent des valeurs valides (si le champ "âge" ne contient pas des lettres au lieu de chiffres par exemple).

À ce propos, je vais tout de suite faire une précision très importante pour ce TP et tous vos codes en Javascript :



Une vérification des informations côté client ne dispensera jamais de faire cette même vérification côté serveur. Le Javascript est un langage qui s'exécute côté client, or le client peut très bien modifier son comportement ou bien carrément le désactiver, ce qui annulera les vérifications. Bref, continuez à faire comme vous l'avez toujours fait sans le Javascript : faites des vérifications côté serveur !

Bien, nous pouvons maintenant commencer !

Présentation de l'exercice

Faire un formulaire c'est bien, mais encore faut-il savoir quoi demander à l'utilisateur. Dans notre cas, nous allons faire simple et classique : un formulaire d'inscription. Notre formulaire d'inscription aura besoin de quelques informations concernant l'utilisateur, cela nous permettra d'utiliser un peu tous les éléments HTML spécifiques aux formulaires que nous avons vus jusqu'à présent. Voici les informations à récupérer ainsi que les types d'éléments HTML :

Information à relever	Type de balise à utiliser
Sexe	Input radio
Nom	Input text
Prénom	Input text
Âge	Input text
Pseudo	Input text
Mot de passe	Input password
Mot de passe (confirmation)	Input password
Pays	Select
Si l'utilisateur souhaite recevoir des mails	Input checkbox

Bien sûr, chacune de ces informations devra être traitée afin que l'on sache si le contenu est bon. Par exemple, si l'utilisateur a bien spécifié son sexe ou bien si il n'a pas entré de chiffres dans son prénom, etc... Dans notre cas, nos vérifications de contenu ne seront pas très poussées pour la simple et bonne raison que nous n'avons pas encore étudié les [Regex](#) à ce stade du cours, nous nous limiterons donc à la vérification de la longueur de la chaîne ou bien à la présence de certains caractères. Bref, rien d'incroyable, mais ça suffira amplement car le but de ce TP n'est pas vraiment de vous faire analyser le contenu mais plutôt de gérer les événements et le CSS de votre formulaire.

Voici donc les conditions à respecter pour chaque information :

Information à relever	Condition à respecter
Sexe	Un sexe doit être sélectionné
Nom	Pas moins de 2 caractères
Prénom	Pas moins de 2 caractères
Âge	Un chiffre compris entre 5 et 140
Pseudo	Pas moins de 4 caractères
Mot de passe	Pas moins de 6 caractères
Mot de passe (confirmation)	Doit être identique au premier mot de passe
Pays	Un pays doit être sélectionné
Si l'utilisateur souhaite recevoir des mails	Pas de condition

Concrètement, l'utilisateur n'est pas censé connaître toutes ces conditions quand il arrive sur votre formulaire, il faudra donc les lui indiquer avant même qu'il ne commence à entrer ses informations, comme ça il ne perdra pas de temps à corriger ses fautes. Pour cela, il va vous falloir afficher chaque condition d'un champ de texte quand l'utilisateur fera une erreur. Pourquoi parlons-nous ici uniquement des champs de texte ? Tout simplement parce que nous n'allons pas dire à l'utilisateur "Sélectionnez votre sexe" alors qu'il n'a qu'une case à cocher, cela paraît évident.

Autre chose, il faudra aussi faire une vérification complète du formulaire lorsque l'utilisateur aura cliqué sur le bouton de soumission. À ce moment là, si l'utilisateur n'a pas coché de case pour son sexe on pourra lui dire qu'il manque une information, pareil s'il n'a pas sélectionné de pays.

Vous voilà avec toutes les informations nécessaires pour vous lancer dans ce TP. Je vous laisse concevoir votre propre code HTML mais vous pouvez très bien utiliser celui de la correction si vous le souhaitez.

Correction

Bien, vous avez probablement terminé si vous lisez cette phrase. Ou bien vous n'avez pas réussi à aller jusqu'au bout, ce qui peut arriver !

Le corrigé au grand complet : HTML, CSS et Javascript

Nous pouvons maintenant passer à la correction. Pour ce TP, il vous fallait créer la structure HTML de votre page en plus du code Javascript, voici le code que j'ai réalisé pour ce TP :

Code : HTML

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>TP : Un formulaire interactif</title>
  </head>

  <body>

    <form id="myForm">

      <span class="form_col">Sexe :</span>
      <label><input name="sex" type="radio" value="H"
/>Homme</label>
      <label><input name="sex" type="radio" value="F"
/>Femme</label>
      <span class="tooltip">Vous devez sélectionner votre
sexe</span>
      <br /><br />

      <label class="form_col" for="lastName">Nom :</label>
      <input name="lastName" id="lastName" type="text" />
      <span class="tooltip">Un nom ne peut pas faire moins de 2
caractères</span>
      <br /><br />

      <label class="form_col" for="firstName">Prénom :</label>
      <input name="firstName" id="firstName" type="text" />

      <span class="tooltip">Un prénom ne peut pas faire moins de 2
caractères</span>
      <br /><br />

      <label class="form_col" for="age">Âge :</label>
      <input name="age" id="age" type="text" />
      <span class="tooltip">L'âge doit être compris entre 5 et
140</span>
      <br /><br />

      <label class="form_col" for="login">Pseudo :</label>
      <input name="login" id="login" type="text" />
      <span class="tooltip">Le pseudo ne peut pas faire moins de 4
caractères</span>
      <br /><br />

      <label class="form_col" for="pwd1">Mot de passe :</label>
      <input name="pwd1" id="pwd1" type="password" />
      <span class="tooltip">Le mot de passe ne doit pas faire moins
de 6 caractères</span>

      <br /><br />

```

```

        <label class="form_col" for="pwd2">Mot de passe (confirmation)
: </label>
        <input name="pwd2" id="pwd2" type="password" />
        <span class="tooltip">Le mot de passe de confirmation doit
être identique à celui d'origine</span>
        <br /><br />

        <label class="form_col" for="country">Pays :</label>

        <select name="country" id="country">
                <option value="none">Sélectionnez votre pays de
résidence</option>
                <option value="en">Angleterre</option>
                <option value="us">États-Unis</option>
                <option value="fr">France</option>
        </select>
        <span class="tooltip">Vous devez sélectionner votre pays de
résidence</span>

        <br /><br />

        <span class="form_col"></span>
        <label><input name="news" type="checkbox" /> Je désire
recevoir la newsletter chaque mois.</label>
        <br /><br />

        <span class="form_col"></span>
        <input type="submit" value="M'inscrire" /> <input type="reset"
value="Réinitialiser le formulaire" />

    </form>

</body>
</html>

```

Vous remarquerez que de nombreuses balises `` possèdent une classe nommée `tooltip`, elles contiennent le texte à afficher lorsque le contenu du champ les concernant ne correspond pas à ce qui est souhaité.

Nous allons maintenant passer au CSS. D'habitude je ne vous le fournis pas directement mais cette fois il fait partie intégrante de ce TP, donc le voici :

Code : CSS

```

body {
    padding-top: 50px;
}

.form_col {
    display: inline-block;
    margin-right: 15px;
    padding: 3px 0px;
    width: 200px;
    min-height: 1px;
    text-align: right;
}

input {
    padding: 2px;
    border: 1px solid #CCC;
    -moz-border-radius: 2px;
    -webkit-border-radius: 2px;
}

```

```
border-radius: 2px;
}

input:focus {
border-color: rgba(82, 168, 236, 0.75);
-moz-box-shadow: 0 0 8px rgba(82, 168, 236, 0.5);
-webkit-box-shadow: 0 0 8px rgba(82, 168, 236, 0.5);
box-shadow: 0 0 8px rgba(82, 168, 236, 0.5);
}

.correct {
border-color: rgba(68, 191, 68, 0.75);
}

.correct:focus {
border-color: rgba(68, 191, 68, 0.75);
-moz-box-shadow: 0 0 8px rgba(68, 191, 68, 0.5);
-webkit-box-shadow: 0 0 8px rgba(68, 191, 68, 0.5);
box-shadow: 0 0 8px rgba(68, 191, 68, 0.5);
}

.incorrect {
border-color: rgba(191, 68, 68, 0.75);
}

.incorrect:focus {
border-color: rgba(191, 68, 68, 0.75);
-moz-box-shadow: 0 0 8px rgba(191, 68, 68, 0.5);
-webkit-box-shadow: 0 0 8px rgba(191, 68, 68, 0.5);
box-shadow: 0 0 8px rgba(191, 68, 68, 0.5);
}

.tooltip {
display: inline-block;
margin-left: 20px;
padding: 2px 4px;
border: 1px solid #555;
background-color: #CCC;
-moz-border-radius: 4px;
-webkit-border-radius: 4px;
border-radius: 4px;
}
```

Notez bien les deux classes `.correct` et `.incorrect` : Elles seront appliquées aux `<input>` de type `text` et `password` afin de bien montrer si un champ est correctement rempli ou non.

Nous pouvons maintenant passer au plus compliqué, le code Javascript :

Code : JavaScript

```
// Fonction de désactivation de l'affichage des "tooltips"

function deactivateTooltips() {

var spans = document.getElementsByTagName('span'),
    spansLength = spans.length;

for (var i = 0 ; i < spansLength ; i++) {
    if (spans[i].className == 'tooltip') {
        spans[i].style.display = 'none';
    }
}
}
```

```
// La fonction ci-dessous permet de récupérer la "tooltip" qui
correspond à notre input

function getTooltip(el) {

    while (el = el.nextSibling) {
        if (el.className == 'tooltip') {
            return el;
        }
    }

    return false;
}

// Fonctions de vérification du formulaire, elles renvoient "true"
si tout est ok

var check = {}; // On met toutes nos fonctions dans un objet
littéral

check['sex'] = function() {

    var sex = document.getElementsByName('sex'),
        tooltipStyle = getTooltip(sex[1].parentNode).style;

    if (sex[0].checked || sex[1].checked) {
        tooltipStyle.display = 'none';
        return true;
    } else {
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['lastName'] = function(id) {

    var name = document.getElementById(id),
        tooltipStyle = getTooltip(name).style;

    if (name.value.length >= 2) {
        name.className = 'correct';
        tooltipStyle.display = 'none';
        return true;
    } else {
        name.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['firstName'] = check['lastName']; // La fonction pour le
prénom est la même que celle du nom

check['age'] = function() {

    var age = document.getElementById('age'),
        tooltipStyle = getTooltip(age).style,
        ageValue = parseInt(age.value);

    if (!isNaN(ageValue) && ageValue >= 5 && ageValue <= 140) {
        age.className = 'correct';
        tooltipStyle.display = 'none';
    }
};
```

```
        return true;
    } else {
        age.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['login'] = function() {

    var login = document.getElementById('login'),
        tooltipStyle = getTooltip(login).style;

    if (login.value.length >= 4) {
        login.className = 'correct';
        tooltipStyle.display = 'none';
        return true;
    } else {
        login.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['pwd1'] = function() {

    var pwd1 = document.getElementById('pwd1'),
        tooltipStyle = getTooltip(pwd1).style;

    if (pwd1.value.length >= 6) {
        pwd1.className = 'correct';
        tooltipStyle.display = 'none';
        return true;
    } else {
        pwd1.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['pwd2'] = function() {

    var pwd1 = document.getElementById('pwd1'),
        pwd2 = document.getElementById('pwd2'),
        tooltipStyle = getTooltip(pwd2).style;

    if (pwd1.value == pwd2.value && pwd2.value != '') {
        pwd2.className = 'correct';
        tooltipStyle.display = 'none';
        return true;
    } else {
        pwd2.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

check['country'] = function() {

    var country = document.getElementById('country'),
        tooltipStyle = getTooltip(country).style;

    if (country.options[country.selectedIndex].value != 'none') {
        tooltipStyle.display = 'none';
    }
};
```

```
        return true;
    } else {
        tooltipStyle.display = 'inline-block';
        return false;
    }
};

// Mise en place des événements

(function() { // Utilisation d'une fonction anonyme pour éviter les
variables globales.

    var myForm = document.getElementById('myForm'),
        inputs = document.getElementsByTagName('input'),
            inputsLength = inputs.length;

    for (var i = 0 ; i < inputsLength ; i++) {
        if (inputs[i].type == 'text' || inputs[i].type == 'password') {

            inputs[i].onkeyup = function() {
                check[this.id](this.id); // "this" représente l'input
actuellement modifié
            };

        }
    }

    myForm.onsubmit = function() {

        var result = true;

        for (var i in check) {
            result = check[i](i) && result;
        }

        if (result) {
            alert('Le formulaire est bien rempli.');
        }

        return false;
    };

    myForm.onreset = function() {

        for (var i = 0 ; i < inputsLength ; i++) {
            if(inputs[i].type == 'text' || inputs[i].type == 'password') {
                inputs[i].className = '';
            }
        }

        deactivateTooltips();
    };

})();

// Maintenant que tout est initialisé, on peut désactiver les
"tooltips"

deactivateTooltips();
```

Essayer le code complet de ce TP !

Les explications

Les explications vont essentiellement porter sur le code Javascript qui est, mine de rien, plutôt conséquent (plus de 200 lignes de code, ça commence à faire pas mal).

La désactivation des bulles d'aide

Dans notre code HTML nous avons créé des balises `` avec la classe `tooltip`. Ce sont des balises qui permettent d'afficher des bulles d'aide, pour que l'utilisateur sache quoi entrer comme contenu. Seulement, elles sont affichées par défaut et il nous faut donc les cacher par le biais du Javascript.



Et pourquoi ne pas les cacher par défaut puis les afficher grâce au Javascript ?

Si vous faites cela, vous prenez le risque qu'un utilisateur ayant désactivé le Javascript ne puisse pas voir les bulles d'aide, c'est plutôt fâcheux non ? Après tout, afficher les bulles d'aide par défaut et les cacher avec le Javascript ne coûte pas grand chose, autant le faire... De plus, nous allons avoir besoin de cette fonction plus tard quand l'utilisateur voudra réinitialiser son formulaire.

Venons-en donc au code :

Code : JavaScript

```
function deactivateTooltips() {  
  
    var spans = document.getElementsByTagName('span'),  
        spansLength = spans.length;  
  
    for (var i = 0 ; i < spansLength ; i++) {  
        if (spans[i].className == 'tooltip') {  
            spans[i].style.display = 'none';  
        }  
    }  
}
```

Est-il vraiment nécessaire de vous expliquer ce code en détail ? Il ne s'agit que d'un simple parcours de balises comme vous en avez déjà vu. Il est juste important de préciser qu'il y a une condition à la ligne 7 qui permet de ne sélectionner que les balises `` qui ont une classe `tooltip`.

Récupérer la bulle d'aide correspondant à un `<input />`

Il est facile de parcourir toutes les bulles d'aide, mais il est un peu plus délicat de récupérer celle correspondant à l'`input` que l'on est actuellement en train de traiter. Si on regarde bien la structure de notre code HTML, on remarque que les bulles d'aide sont toujours placées après l'`input` auquel elles correspondent, nous allons donc partir du principe qu'il suffit de chercher la bulle d'aide la plus "proche" après l'`input` que nous sommes actuellement en train de traiter. Voici le code :

Code : JavaScript

```
function getTooltip(el) {  
  
    while (el = el.nextSibling) {  
        if (el.className == 'tooltip') {
```

```
        return el;
    }
}

return false;
}
```

Notre fonction prend en argument l'**input** actuellement en cours de traitement. Notre boucle **while** se charge alors de vérifier tous les éléments suivants notre **<input>** (d'où l'utilisation du **nextSibling**). Une fois qu'un élément avec la classe **tooltip** a été trouvé, il ne reste plus qu'à le retourner.

Analyser chaque valeur entrée par l'utilisateur

Nous allons enfin aborder le vif du sujet : l'analyse des valeurs et la modification du style du formulaire en conséquence. Tout d'abord, qu'elles valeurs faut-il analyser ? Toutes, sauf la case à cocher pour l'inscription à la newsletter. Maintenant que ceci est clair, passons à la toute première ligne de code :

Code : JavaScript

```
var check = {};
```

Alors oui, au premier abord, cette ligne de code ne sert vraiment pas à grand chose, mais en vérité elle a une très grande utilité : l'objet créé va nous permettre d'y stocker toutes les fonctions permettant de "checker" (d'où le nom de l'objet) chaque valeur entrée par l'utilisateur. L'intérêt de cet objet est triple :

- Nous allons pouvoir exécuter la fonction correspondant à un champ de cette manière :
`check['id_du_champ']()` ; Cela va grandement simplifier notre code lors de la mise en place des événements.
- Il sera possible d'exécuter toutes les fonctions de "check" juste en parcourant l'objet, ce sera très pratique lorsque l'utilisateur cliquera sur le bouton d'inscription et qu'il faudra alors revérifier tout le formulaire.
- L'ajout d'un champ de texte et de sa fonction d'analyse devient très simple si on concentre tout dans cet objet, vous comprendrez très rapidement pourquoi !

Nous n'allons pas étudier toutes les fonctions d'analyse, elles se ressemblent beaucoup, nous allons donc uniquement étudier deux fonctions afin de mettre les choses au clair :

Code : JavaScript

```
check['login'] = function() {

    var login = document.getElementById('login'),
        tooltipStyle = getTooltip(login).style;

    if (login.value.length >= 4) {
        login.className = 'correct';
        tooltipStyle.display = 'none';
        return true;
    } else {
        login.className = 'incorrect';
        tooltipStyle.display = 'inline-block';
        return false;
    }
};
```


Il est très important que vous constatiez que notre fonction est contenue dans l'attribut d'un objet nommé "login", ce dernier n'est autre que l'identifiant du champ de texte auquel la fonction appartient. Le code n'est pas bien compliqué : on récupère l'**input** et l'attribut **style** de la bulle d'aide qui correspondent à notre fonction et on passe à l'analyse du contenu.

Si le contenu remplit bien la condition, alors on attribue à notre **<input>** la classe **.correct**, on désactive l'affichage de la bulle d'aide et on retourne **true**.

Si le contenu ne remplit pas la condition, notre **<input>** se voit alors attribuer la classe **.incorrect** et la bulle d'aide est affichée. En plus de cela, on renvoie la valeur **false**.

Passons maintenant à une deuxième fonction que je tenais à aborder :

Code : JavaScript

```
check['lastName'] = function(id) {  
  
    var name = document.getElementById(id),  
        tooltipStyle = getTooltip(name).style;  
  
    if (name.value.length >= 2) {  
        name.className = 'correct';  
        tooltipStyle.display = 'none';  
        return true;  
    } else {  
        name.className = 'incorrect';  
        tooltipStyle.display = 'inline-block';  
        return false;  
    }  
};
```

Cette fonction diffère de la précédente sur un seul point : elle possède un argument **id** ! Cet argument sert à récupérer l'identifiant de l'input à analyser, pourquoi ? Tout simplement parce qu'elle va nous servir à analyser deux champs de texte différents : celui du nom et celui du prénom. Puisqu'ils ont tous les deux la même condition, il aurait été stupide de créer deux fois la même fonction.

Donc, au lieu de faire appel à cette fonction sans aucun argument, il faut lui passer l'identifiant du champ de texte à analyser, ce qui donne deux possibilités :

Code : JavaScript

```
check['lastName'] ('lastName');  
check['lastName'] ('firstName');
```

Cependant, ce fonctionnement pose un problème car nous étions parti du principe que nous allions faire appel à nos fonctions d'analyse selon le principe suivant :

Code : JavaScript

```
check['id_du_champ'] ();
```

Or, si nous faisons ça, cela veut dire que nous ferons aussi appel à la fonction `check['firstName']()` qui n'existe pas... Nous n'allons pas créer cette fonction sinon nous perdrons l'intérêt de notre système d'argument sur la fonction

`check['lastName']()`. La solution est donc de faire une référence de la manière suivante :

Code : JavaScript

```
check['firstName'] = check['lastName'];
```

Ainsi, lorsque nous appellerons la fonction `check['firstName']()`, implicitement ce sera la fonction `check['lastName']()` qui sera appelée.

Si vous n'avez pas encore tout à fait compris l'utilité de ce système, vous allez voir que tout cela va se montrer redoutablement efficace dans la partie ci-dessous.

La mise en place des événements : Partie 1

La mise en place des événements se décompose en deux parties : les événements à appliquer aux champs de texte et les événements à appliquer aux deux boutons en bas de page pour envoyer ou réinitialiser le formulaire.

Nous allons commencer par les champs de textes. Tout d'abord, voici le code :

Code : JavaScript

```
var inputs = document.getElementsByTagName('input'),
    inputsLength = inputs.length;

for (var i = 0 ; i < inputsLength ; i++) {
    if (inputs[i].type == 'text' || inputs[i].type == 'password') {

        inputs[i].onkeyup = function() {
            check[this.id](this.id); // "this" représente l'input
            actuellement modifié
        };

    }
}
```

Comme je l'ai déjà fait plus haut, je ne vais pas prendre la peine de vous expliquer le fonctionnement de cette boucle et de la condition qu'elle contient, il ne s'agit que de parcourir les `<input>` et d'agir seulement sur ceux qui sont de type `text` ou `password`.

En revanche, il va falloir des explications sur les lignes 7 à 9. Les lignes 7 et 9 permettent d'assigner une fonction anonyme à l'événement `onkeyup` de l'input actuellement que nous traitons. Quant à la ligne 8, elle fait appel à la fonction d'analyse qui correspond à l'input qui a exécuté l'événement. Ainsi, si l'input dont l'id est "login" déclenche son événement, il appellera alors la fonction `check['login']()`.

Cependant, un argument est passé à chaque fonction d'analyse que l'on exécute, pourquoi ? Eh bien il s'agit de l'argument nécessaire à la fonction `check['lastName']()`, ainsi lorsque les `<input>` `lastName` et `firstName` déclencheront leur événement, ils exécuteront alors respectivement les lignes de codes suivantes :

Code : JavaScript

```
check['lastName']('lastName');
```

et

Code : JavaScript

```
check['firstName']('firstName');
```



Mais là on passe l'argument à toutes les fonctions d'analyse, ça ne pose pas de problème normalement ?

Pourquoi cela en poserait un ? Imaginons que l'input **login** déclenche son événement, il exécutera alors la ligne de code suivante :

Code : JavaScript

```
check['login']('login');
```

Cela fera passer un argument inutile dont la fonction ne tiendra pas compte, c'est tout.

La mise en place des événements : Partie 2

Nous pouvons maintenant aborder l'attribution des événements sur les boutons en bas de page :

Code : JavaScript

```
var myForm = document.getElementById('myForm');  
myForm.onsubmit = function() {  
    var result = true;  
    for (var i in check) {  
        result = check[i](i) && result;  
    }  
    if (result) {  
        alert('Le formulaire est bien rempli.');    }  
    return false;  
};  
myForm.onreset = function() {  
    for (var i = 0 ; i < inputsLength ; i++) {  
        if(inputs[i].type == 'text' || inputs[i].type == 'password') {  
            inputs[i].className = '';  
        }  
    }  
    deactivateTooltips();  
};
```

Comme vous pouvez le constater, je n'ai pas appliqué des événements **onclick** sur les boutons mais j'ai directement appliqué **onsubmit** et **onreset** sur le formulaire, ce qui est bien plus pratique dans notre cas.

Alors concernant notre événement **onsubmit**, celui-ci va parcourir notre tableau **check** et exécuter toutes les fonctions qu'il contient (y compris celles qui ne sont pas associées à un champ de texte comme `check['sex']()` et `check['country']()`). Chaque valeur retournée par ces fonctions est "ajoutée" à la variable **result**, ce qui fait que si une des fonctions a renvoyé **false** alors **result** sera aussi à **false** et l'exécution de la fonction **alert()** ne se fera pas.

Concernant notre événement **onreset**, c'est très simple : on parcourt les champs de texte, on retire leur classe et ensuite on désactive toutes les bulles d'aide grâce à notre fonction **deactivateTooltips()**.

Voilà, ce TP est maintenant terminé, j'espère que vous aurez pris du plaisir à le réaliser 😊.

Partie 3 : Les objets

Les objets

Nous avons vu dans la première partie du cours un chapitre sur les objets et les tableaux. Ce chapitre en est la suite et permet de mettre les deux pieds au sein de la création et de la modification d'objets en Javascript. C'est un gros chapitre, alors courage !

Petite problématique

Javascript possède des objets natifs, comme String, Boolean et Array, mais nous permet aussi de créer nos propres objets, avec leurs propres méthodes et propriétés.



Mais quel en est l'intérêt ?

L'intérêt est généralement une propriété de code ainsi qu'une facilité de développement. Les objets sont là pour nous faciliter la vie, mais leur création peut prendre du temps.

Rappelez-vous de l'exemple des tableaux avec les prénoms :

Code : JavaScript

```
var myArray = ['Sébastien', 'Laurence', 'Ludovic', 'Pauline',  
'Guillaume'];
```

Ce tableau sert juste à stocker les prénoms, rien de plus. Imaginons qu'il faille faire un tableau contenant énormément de données, et ce, pour chaque personne. Par exemple, pour chaque personne, on aurait les données suivantes : prénom, âge, sexe, parenté, travail... Comment structurer tout cela ?

Avec une très grosse dose de motivation, il est possible de réaliser quelque chose comme ceci :

Code : JavaScript

```
var myArray = [  
  {  
    nick: 'Sébastien',  
    age: 23,  
    sex: 'm',  
    parent: 'aîné',  
    work: 'Javascipteur'  
  },  
  {  
    nick: 'Laurence',  
    age: 19,  
    sex: 'f',  
    parent: 'soeur',  
    work: 'Sous-officier'  
  },  
  // et ainsi de suite...  
];
```

Ce n'est pas encore trop compliqué car les données restent relativement simplistes. Maintenant, pour chaque personne, on va ajouter un tableau qui contiendra ses amis, et pour chaque ami, les mêmes données que ci-dessus. Là, c'est déjà plus compliqué...

Profitons de cette problématique pour étudier les objets !

Objet constructeur

Nous avons vu que Javascript nous permettait de créer des objets littéraux et nous allons voir maintenant comment créer de véritables objets qui possèdent des propriétés et des méthodes tout comme les objets natifs.

Un objet représente quelque chose, un concept. Ici, suite à l'exemple de la famille, nous allons créer un objet appelé **Person** et qui contiendra des données, à savoir le prénom, l'âge, le sexe, le lien de parenté, le travail et la liste des amis (ce sera un tableau).

L'utilisation de tels objets se fait en deux temps :

1. On définit l'objet via un constructeur : cette étape permet de définir un objet qui pourra être réutilisé par la suite. Cet objet ne sera pas directement utilisé car nous en utiliserons une "copie" : on parle d'**instance**.
2. A chaque fois qu'on a besoin d'utiliser notre objet, on crée une instance de celui-ci, c'est-à-dire qu'on le copie.

Définition via un constructeur

Le constructeur (ou *objet constructeur* ou *constructeur d'objet*) va contenir la structure de base de notre objet. Si vous avez déjà fait de la programmation orientée objet dans des langages tels que le C++, le C# ou le Java, sachez que ce constructeur ressemble, sur le principe, à une classe.

La syntaxe d'un constructeur est la même que celle d'une fonction :

Code : JavaScript

```
function Person() {  
    // Code du constructeur  
}
```



De manière générale on met une majuscule à la première lettre d'un constructeur. Ça permet de mieux le différencier d'une fonction "normale" et ça le fait ressembler aux noms des objets natifs qui portent tous une majuscule (Array, Date, String...).

C'est donc pareil qu'une fonction. Cela dit, le code du constructeur va contenir une petite particularité : le mot-clé **this**. Ce mot-clé fait référence à l'objet dans lequel il est exécuté, c'est-à-dire ici, le constructeur **Person**. Si on utilise **this** au sein du constructeur **Person**, **this** pointe vers **Person**. Grâce à **this**, nous allons pouvoir définir les propriétés de l'objet **Person** :

Code : JavaScript

```
function Person(nick, age, sex, parent, work, friends) {  
    this.nick = nick;  
    this.age = age;  
    this.sex = sex;  
    this.parent = parent;  
    this.work = work;  
    this.friends = [];  
}
```

Utilisation de l'objet

L'objet **Person** a été défini grâce au constructeur qu'il ne nous reste plus qu'à utiliser :

Code : JavaScript

```
// Définition de l'objet Person via un constructeur
function Person(nick, age, sex, parent, work, friends) {
    this.nick = nick;
    this.age = age;
    this.sex = sex;
    this.parent = parent;
    this.work = work;
    this.friends = [];
}

// On crée des variables qui vont contenir une instance de l'objet
Person :
var seb = new Person('Sébastien', 23, 'm', 'aîné', 'Javascipteur',
[]);
var lau = new Person('Laurence', 19, 'f', 'soeur', 'Sous-officier',
[]);

alert(seb.nick); // Affiche 'Sébastien'
alert(lau.nick); // Affiche 'Laurence'
```

Que s'est-il passé ici ? L'objet **Person** a été défini comme nous l'avons vu plus haut. Pour pouvoir utiliser cet objet, on définit une variable qui va contenir une instance de l'objet **Person**, c'est-à-dire une "copie". Pour indiquer à Javascript qu'il faut utiliser une instance, on utilise le mot-clé **new**.

Retenez bien que ce mot-clé **new** ne signifie pas "créer un nouvel objet", mais signifie "créer une nouvelle instance de l'objet", ce qui est très différent puisque dans le deuxième cas on ne fait que créer une instance, une copie, de l'objet initial, ce qui nous permet de conserver l'objet en question.

Il est possible de faire un test pour savoir si la variable **seb** est une instance de **Person**. Pour ce faire, il convient d'utiliser le mot-clé **instanceof**, comme ceci :

**Code : JavaScript**

```
alert(lau instanceof Person); // Affiche true
```

Dans les paramètres de l'objet, on transmet les différentes informations pour la personne. Ainsi donc, en transmettant **Sébastien** comme premier paramètre, celui-ci ira s'enregistrer dans la propriété **this.nick**, et il sera possible de le récupérer en faisant **seb.nick**.

Modifier les données

Une fois la variable définie, on peut modifier les propriétés, exactement comme s'il s'agissait d'un simple objet littéral comme vu dans la première partie du cours :

Code : JavaScript

```
var seb = new Person('Sébastien', 23, 'm', 'aîné', 'Javascipteur',
[]);

seb.nick = 'Bastien'; // On change le prénom
seb.age = 18; // On change l'age

alert(seb.nick + ' a ' + seb.age + 'ans'); // Affiche 'Bastien a 18'
```



```
ans'
```

Au final, si on reprend la problématique du début, on peut ré-écrire **myArray** comme contenant des éléments de type **Person** :

Code : JavaScript

```
var myArray = [  
  new Person('Sébastien', 23, 'm', 'aîné', 'Javascripateur', []),  
  new Person('Laurence', 19, 'f', 'soeur', 'Sous-officier', []),  
  new Person('Ludovic', 9, 'm', 'frère', 'Etudiant', []),  
  new Person('Pauline', 16, 'f', 'cousine', 'Etudiante', []),  
  new Person('Guillaume', 16, 'm', 'cousin', 'Dessinateur', []),  
];
```

Il sera ainsi possible d'accéder aux différents membres de la famille de cette manière : `myArray[i].work` // *Pour récupérer le job.*

Ajouter des méthodes

L'objet vu précédemment est simple. Il y a moyen de l'améliorer en lui ajoutant des méthodes. Les méthodes, vous savez ce que c'est car vous en avez déjà croisées dans les chapitres sur les tableaux.

Si on reprend l'exemple précédent et si on veut ajouter un ami, il faut faire comme ceci :

Code : JavaScript

```
var seb = new Person('Sébastien', 23, 'm', 'aîné', 'Javascripateur',
[]);

// On ajoute un ami dans la tableau 'friends'
seb.friends.push(new Person('Johann', 19, 'm', 'aîné',
'Javascripateur aussi', []));

alert(seb.friends[0].nick); // Affiche Johann
```

Avec ça, on peut aussi ajouter un ami à Johann :

Code : JavaScript

```
seb.friends[0].friends.push(new Person('Victor', 19, 'm', 'aîné',
'Internet Hero', []));
```

Les possibilités sont infinies... 😊

Mais tout ça reste long à écrire. Pourquoi ne pas ajouter une méthode **addFriend()** à l'objet **Person** de manière à pouvoir ajouter un ami comme ceci :

Code : JavaScript

```
seb.addFriend('Johann', 19, 'm', 'aîné', 'Javascripateur aussi', []);
```

Ajouter une méthode

Il y a deux manières de définir une méthode pour un objet : dans le constructeur ou via prototype. Définir les méthodes directement dans le constructeur est facile puisque c'est nous qui créons le constructeur. La définition de méthodes via prototype est utile surtout si on n'a pas créé le constructeur : ce sera alors utile pour ajouter des méthodes à des objets natifs, comme String ou Array.

Définir une méthode dans le constructeur

Pour cela, rien de plus simple, on procède comme pour les propriétés, sauf qu'il s'agit d'une fonction :

Code : JavaScript

```
function Person(nick, age, sex, parent, work, friends) {
  this.nick = nick;
  this.age = age;
  this.sex = sex;
```

```
    this.parent = parent;
    this.work = work;
    this.friends = [];

    this.addFriend = function(nick, age, sex, parent, work, friends)
    {
        this.friends.push(new Person(nick, age, sex, parent, work,
friends));
    };
}
```

Le code de cette méthode est simple : il ajoute un objet **Person** dans le tableau des amis.



N'aurions-nous pas pu utiliser `new this (/* ... */)` à la place de `new Person (/* ... */)` ?

Non car comme on l'a vu plus haut, `this` fait référence à l'objet dans lequel il est exécuté, c'est à dire le constructeur **Person**. Si on avait fait `new this (/* ... */)` ça aurait équivalu à insérer le constructeur dans lui-même. Mais de toute façon, en tentant de faire ça, vous obtenez une erreur du type "this n'est pas un constructeur", donc l'interpréteur Javascript ne plante heureusement pas.

Ajouter une méthode via prototype

Quand vous définissez un objet, il possède automatiquement un sous-objet appelé `prototype`.



`prototype` est un objet, mais c'est aussi le nom d'une librairie Javascript : **Prototype**. Faites donc attention si vous faites de recherches avec les mots "javascript prototype", car vous risquez de tomber sur les pages de cette librairie.

Cet objet `prototype` va nous permettre d'ajouter des méthodes à un objet. Voici comment ajouter une méthode `addFriend()` à notre objet **Person** :

Code : JavaScript

```
Person.prototype.addFriend = function(nick, age, sex, parent, work,
friends) {
    this.friends.push(new Person(nick, age, sex, parent, work,
friends));
}
```

Le `this` fait ici aussi référence à l'objet dans lequel il s'exécute, c'est-à-dire l'objet **Person**.

L'ajout de méthodes par `prototype` a l'avantage d'être indépendant de l'objet, c'est-à-dire que vous pouvez définir votre objet dans un fichier, et ajouter des méthodes dans un autre fichier (pour autant que les deux fichiers soient inclus dans la même page Web).

Ajouter des méthodes aux objets natifs

Une grosse particularité du Javascript est qu'il est *orienté objet par prototype* ce qui le dote de certaines caractéristiques que d'autres langages orientés objet ne possèdent pas. Avec Javascript, il est possible de modifier les objets natifs, ce qui est impossible dans des langages comme le C# ou le Java. En fait, les objets natifs possèdent aussi un objet prototype qui permet de les modifier !

Ajout de méthodes

Ce n'est parfois pas facile de visualiser le contenu d'un tableau avec une `alert()`. Pourquoi ne pas créer une méthode qui afficherait le contenu d'un objet littéral via `alert()` mais de façon plus élégante (un peu comme la fonction `var_dump` de PHP si vous connaissez).

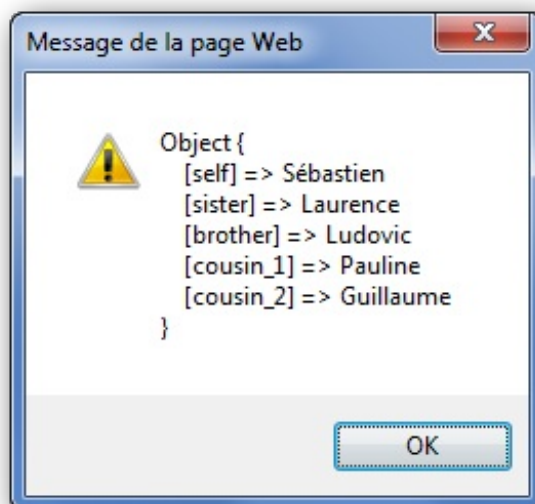
Voici le type d'objet à afficher proprement :

Code : JavaScript

```
var family = {
  self:      'Sébastien',
  sister:    'Laurence',
  brother:   'Ludovic',
  cousin_1:  'Pauline',
  cousin_2:  'Guillaume'
};

family.debug(); // Nous allons créer cette méthode debug
```

La méthode `debug()` affichera ceci :



Comme il s'agit d'un objet, le type natif est `Object`. Comme vu précédemment, on va utiliser son sous-objet prototype pour lui ajouter la méthode voulue :

Code : JavaScript

```
// Testons si cette méthode n'existe pas déjà !
if (!Object.prototype.debug) {

  // Créons la méthode
  Object.prototype.debug = function() {
```

```
var text = 'Object {\n';

for (var i in this) {
  if (i !== 'debug') {
    text += ' [' + i + '] => ' + this[i] + '\n';
  }
}

alert(text + '}');
```



Mais pourquoi tester si `i` est différent de "debug" ?

Parce qu'en ajoutant la méthode `debug()` aux objets, elle s'ajoute même aux objets littéraux : autrement dit, `debug()` va se lister lui-même ce qui n'a pas beaucoup d'intérêt. Exemple en enlevant cette condition :



Rien de bien méchant, mais ça perturbe la lecture : le code de la méthode `debug()` est affiché.

Remplacer des méthodes

Comme vous avez dû le remarquer, quand on utilise prototype, on affecte une fonction. Cela veut donc dire qu'il est possible de modifier les méthodes natives des objets en leur affectant une nouvelle méthode. Ce n'est pas spécialement utile, mais c'est faisable.

Limitations

Dans Internet Explorer

En théorie, chaque objet peut se voir attribuer des méthodes via prototype. Mais en pratique, si ça marche bien avec les objets natifs génériques comme `String`, `Date`, `Array`, `Object`, `Number` et `Boolean`, ça marche moins bien avec les objets natifs liés au DOM comme `Node`, `Element` ou encore `HTMLElement` en particulier dans Internet Explorer.

Chez les éditeurs

Sachez également que si vos scripts sont destinés à être utilisés sous la forme d'extensions pour Firefox et que vous soumettez votre extension sur le site de [Mozilla Addons](#), celui-ci sera refusé car Mozilla trouve qu'il est dangereux de modifier les objets natifs. C'est un peu vrai puisque si vous modifiez une méthode native, elle risque de ne pas fonctionner correctement pour une autre extension... D'ailleurs, à ce propos, il est de temps de parler des **namespaces**, ou plutôt des espaces de noms !

Les namespaces

En informatique, un **namespace**, ou *espace de nom* en français, est un ensemble fictif qui contient des informations, généralement des propriétés et des méthodes, ainsi que des sous-namespaces. Le but d'un namespace est de s'assurer de l'unicité des informations qu'il contient.

Par exemple, Sébastien et Johann habitent tous deux au numéro 42. Sébastien dans la rue de Belgique et Johann dans la rue de France. Le numéro de leur maison peut être confondu, puisqu'il s'agit du même. Ainsi, si Johann dit "J'habite au numéro 42", c'est ambigu, car Sébastien aussi. Alors, pour différencier les deux numéros, on va toujours donner le nom de la rue. Ce nom de rue peut être vu comme un namespace : il permet de différencier deux données identiques.

En programmation, quelque chose d'analogue peut se produire : imaginez que vous développiez une fonction **myBestFunction()**, et vous trouvez un script tout fait pour réaliser un effet quelconque. Vous ajoutez alors ce script au vôtre. Problème : dans ce script tout fait, une fonction **myBestFunction()** est aussi présente... Votre fonction se retrouve écrasée par l'autre, et votre script ne fonctionnera plus correctement.

Pour éviter ce genre de désagrément, on va utiliser un namespace !



Le Javascript, au contraire de langages comme le C# ou le Java ne propose pas de vrai système de namespace. Ce que l'on étudie ici est un système pour reproduire plus ou moins correctement un tel système.

Définir un namespace

Un namespace est une sorte de catégorie : vous allez vous créer un namespace, et au sein de celui-ci, vous allez placer vos fonctions. De cette manière, vos fonctions seront en quelque sorte "préservées" d'éventuels écrasements. Comme Javascript ne gère pas naturellement les namespaces (comprenez : il n'y a pas de structure dédiée à ça), on va devoir se débrouiller seuls, et utiliser un simple objet littéral. Premier exemple :

Code : JavaScript

```
var myNamespace = {
  myBestFunction: function () {
    alert('Ma meilleure fonction !');
  }
};

// On exécute la fonction :
myNamespace.myBestFunction();
```

On commence par créer un objet littéral appelé **myNamespace**. Ensuite on définit une méthode : **myBestFunction()**. Souvenez-vous, dans le chapitre sur les objets littéraux, nous avons vu que nous pouvions définir des propriétés, et il est aussi possible de définir des méthodes, en utilisant la même syntaxe.

Pour appeler **myBestFunction()**, il faut obligatoirement passer par l'objet **myNamespace**, ce qui limite très fortement la probabilité de voir votre fonction écrasée par une autre. Bien évidemment, votre namespace doit être original pour être certain qu'un autre développeur n'utilise pas le même... cette technique n'est donc pas infaillible, mais réduit considérablement les problèmes.

Un style de code

Utiliser un namespace est aussi "joli" car ça permet d'avoir un code visuellement propre et structuré. Une grande majorité des "gros" scripts sont organisés via un namespace, notamment car il est possible de décomposer le script en catégories. Par exemple, vous faites un script qui gère un webmail (comme Hotmail ou Gmail) :

Code : JavaScript

```

var thundersebWebMail = {
  // Propriétés
  version: 1.42,
  lang: 'english',

  // Initialisation
  init : function() { /* initialisation */ },

  // Gestion des mails
  mails: {
    list: function() { /* affiche la liste des mails */ },
    show: function() { /* affiche un mail */ },
    trash: function() { /* supprime un mail */ },
    // et cætera...
  },

  // Gestion des contacts
  contacts: {
    list: function() { /* affiche la liste des contacts */ },
    edit: function() { /* édite un contact */ },
    // et cætera...
  }
};

```

Ce code fictif comprend une méthode d'initialisation, et deux "sous-namespaces" : mails et contacts, chacun d'eux sert respectivement à gérer les mails et les contacts. Chacun de ces sous-namespaces contient les méthodes qui lui sont propres.

Structurer son code de cette manière est propre et lisible, ce qui n'est pas toujours le cas d'une succession de fonctions. L'exemple ci-dessus sans namespace :

Code : JavaScript

```

var webmailVersion = 1.42,
    webmailLang = 'english';

function webmailInit() { /* initialisation */ }

function webmailMailsList() { /* affiche la liste des mails */ }
function webmailMailsShow() { /* affiche un mail */ }
function webmailMailsTrash() { /* supprime un mail */ }

function webmailContactsList() { /* affiche la liste des contacts */ }
function webmailContactsEdit() { /* édite un contact */ }

```

C'est tout de suite plus confus, il n'y a pas de hiérarchie, c'est brouillon. Bien évidemment, ça dépend du codeur : un code en namespace peut être brouillon, alors qu'un code "normal" peut être très propre ; mais de manière générale, un code en namespace est accessible, plus lisible et plus compréhensible. C'est évidemment une question d'habitude.

L'emploi de this

Le mot-clé **this** s'utilise ici exactement comme dans les objets vus précédemment. Mais attention, si vous utilisez **this** dans un sous-namespace, celui-ci pointera vers ce sous-namespace, et non vers le namespace parent. Ainsi, l'exemple suivant ne fonctionnera pas correctement, car en appelant la méthode **init()** on lui demande d'exécuter **this.test()**. Or, **this** pointe vers **subNamespace**, et il n'existe aucune méthode **test()** au sein de **subNamespace**.

Code : JavaScript

```
var myNamespace = {
  test: function() { alert('Test'); },
  subNamespace: {
    init: function() {
      this.test();
    }
  }
};

myNamespace.subNamespace.init();
```

Pour accéder à l'objet parent, il n'y a malheureusement pas de solution si ce n'est écrire son nom entièrement :

Code : JavaScript

```
var myNamespace = {
  test: function() { alert('Test'); },
  subNamespace: {
    init: function() {
      myNamespace.test();
    }
  }
};

myNamespace.subNamespace.init();
```

Vérifier l'unicité du namespace

Une sécurité supplémentaire est de vérifier l'existence du namespace : s'il n'existe pas, on le définit et dans le cas contraire, on ne fait rien pour ne pas risquer d'écraser une version déjà existant, tout en retournant un message d'erreur.

Code : JavaScript

```
// On vérifie l'existence de l'objet myNamespace
if ("undefined" == typeof(myNamespace)) {

  var myNamespace = {
    // Tout le code
  };

} else {
  alert('myNamespace existe déjà !');
}
```

Le cas des constructeurs

Le namespace vu ci-dessus fonctionne très bien pour les objets littéraux. Mais comment utiliser un namespace avec un constructeur ? C'est plutôt simple, mais pas nécessairement évident. Il faut commencer par créer un objet auquel on va ajouter une méthode qui fera office de constructeur :

Code : JavaScript

```
if ("undefined" == typeof(myNamespace)) {

    var myNamespace = {}; // Objet vide

    // Création du constructeur au sein de myNamespace
    myNamespace.Person = function(nick, age, sex, parent, work,
friends) {
        this.nick = nick;
        this.age = age;
        this.sex = sex;
        this.parent = parent;
        this.work = work;
        this.friends = [];
    };

    // Ajout d'une méthode par prototype :
    myNamespace.Person.prototype.addFriend = function(nick, age,
sex, parent, work, friends) {
        this.friends.push(new myNamespace.Person(nick, age, sex,
parent, work, friends));
    };
}
```

Et pour l'utiliser, il suffit de faire :

Code : JavaScript

```
var seb = new myNamespace.Person('Sébastien', 23, 'm', 'aîné',
'Javascipteur', []);

// On ajoute un ami dans la tableau 'friends'
seb.friends.addFriend('Johann', 19, 'm', 'aîné', 'Javascipteur
aussi', []);
```

Mais... faire ceci :

Code : JavaScript

```
var myNamespace = {};

myNamespace.Person = function() { /* ... */};
```



Equivaut à faire cela, non ?

Code : JavaScript

```
var myNamespace = {  
    Person: function() { /* ... */ }  
};
```

C'est exact, et ça fonctionne parfaitement... sauf dans Internet Explorer ! Cela dit, ce genre de namespace de constructeur n'est pas très courant. Rares sont les cours qui pensent même à l'aborder.

Les chaînes de caractères

Les chaînes de caractères, représentées par l'objet `String` ont déjà été manipulées au sein de ce cours, mais il reste assez bien de choses à voir à leur propos ! Nous allons dans un premier temps découvrir les types primitifs car cela nous sera nécessaire pour vraiment parler des objets comme `String`, `RegExp` et autres que nous verrons par la suite.

Les types primitifs

Nous avons vu, tout au long du cours, que les chaînes de caractères étaient des objets `String`, les tableaux des `Array` etc. C'est toujours vrai mais il convient de nuancer ces propos en introduisant le concept de type primitif.

Pour créer une chaîne de caractères, on utilise cette syntaxe :

Code : JavaScript

```
var myString = "Chaîne de caractères primitive";
```

Cet exemple crée ce que l'on appelle une chaîne de caractère primitive, qui n'est pas un objet `String`. Pour instancier un objet string, on fait comme ceci :

Code : JavaScript

```
var myRealString = new String("Chaîne");
```

Ceci est valable pour les autres objets :

Code : JavaScript

```
var myArray = []; // Tableau primitif
var myRealArray = new Array();

var myObject = {}; // Objet primitif
var myRealObject = new Object();

var myBoolean = true;
var myRealBoolean = new Boolean("true");

var myNumber = 42;
var myRealNumber = new Number("42");
```

Ce que nous avons utilisé jusqu'à présent étaient en fait des types primitifs, et non des instances d'objets.



Quelle est la différence entre un type primitif et une instance ?

La différence est minime pour nous, développeurs. Prenons l'exemple de la chaîne de caractères : à chaque fois que nous allons faire une opération sur une chaîne primitive, Javascript va automatiquement convertir cette chaîne en une instance temporaire de `String`, de manière à pouvoir utiliser les propriétés et méthodes fournies par l'objet `String`. Une fois les opérations terminées, l'instance temporaire est détruite.

Au final, utiliser un type primitif ou une instance revient au même du point de vue de l'utilisation. Mais, il subsiste de légères différences avec l'opérateur `instanceof` qui peut retourner de drôles de résultats...

Pour une raison ou une autre, imaginons qu'on veuille savoir de quelle instance est issu un objet :

Code : JavaScript

```
var myString = 'Chaîne de caractères';
```

```
if (myString instanceof String) {  
    // Faire quelque chose  
}
```

La condition sera fausse ! Et c'est bien normal puisque **myString** est une chaîne primitive et non une instance de **String**.

Pour tester le type primitif, il convient d'utiliser l'opérateur **typeof** :

Code : JavaScript

```
if (typeof myString === 'string') {  
    // Faire quelque chose  
}
```

typeof permet de vérifier le type primitif (en anglais on parle de *datatype*). Mais ici aussi faites attention au piège, car la forme primitive d'une instance de **String** est... **object** :

Code : JavaScript

```
alert(typeof myRealString); // Retourne 'object'
```

Il faudra donc faire bien attention en testant le type ou l'instance d'un objet ! Il est même d'ailleurs déconseillé de faire ce genre de tests vu le nombre de problèmes que cela peut apporter. La seule valeur retournée par **typeof** dont on peut être sûr c'est **"undefined"**. Nous allons étudier, plus tard dans ce chapitre, une solution pour tester si une variable contient une chaîne de caractères ou non.



Retenez bien une chose dans l'ensemble : il est plus simple en tous points d'utiliser directement les types primitifs !

L'objet String

L'objet `String` est l'objet que vous manipulez depuis le début du tutoriel : c'est lui qui gère les chaînes de caractères.

Propriétés

`String` ne possède qu'une seule propriété, `length` qui retourne le nombre de caractères contenu dans une chaîne. Les espaces, les signes de ponctuation, les nombres... sont considérés comme des caractères. Ainsi, cette chaîne de caractères contient 21 caractères :

Code : JavaScript

```
var myString = 'Ceci est une chaîne !';  
alert(myString.length);
```

Essayer !

Au fait, saviez-vous qu'il n'était pas toujours obligatoire de déclarer une variable ? Vous pouvez en effet écrire directement le "contenu" de votre variable et lui appliquer une propriété ou une méthode, comme ceci :



Code : JavaScript

```
alert('Ceci est une chaîne !'.length);
```

Méthodes

`String` possède quelques méthodes qui sont pour la plupart assez intéressantes mais basiques. Javascript est un langage assez simple, qui ne contient pas énormément de méthodes de base. C'est un peu l'inverse de PHP qui contient une multitude de fonctions pour réaliser un peu tout et n'importe quoi. Par exemple, PHP possède une fonction pour mettre la première lettre d'une chaîne en majuscule, alors qu'en Javascript il faudra nous-même récupérer le premier caractère et le mettre en majuscule. Le Javascript fournit juste quelques méthodes de base, et ce sera à vous de coder vous-même d'autres méthodes ou fonctions selon vos besoins.

`String` embarque aussi toute une série de méthodes obsolètes destinées à ajouter des balises HTML à une chaîne de caractères. Ne soyez donc pas étonnés si un jour vous rencontrez de telles méthodes dans de vieux scripts au détour d'internet. Ces méthodes, `anchor()`, `big()`, `blink()`, `bold()`, `fixed()`, `fontcolor()`, `fontsize()`, `link()`, `small()`, `strike()`, `sub()`, `sup()` s'utilisent de cette manière :

Code : JavaScript

```
var myString = 'Chaîne à insérer';  
document.body.innerHTML += myString.bold();
```

Essayer (ce code n'est pas standard et peut ne pas fonctionner) !

Ce qui a pour effet d'ajouter ce code HTML : `Chaîne à insérer`. Firefox, Opera et Chrome ajoutent la balise en minuscule ``, tandis qu'Internet Explorer l'ajoute en majuscule ``.

Sachez que ça existe, mais que ce n'est plus utilisé et que ce n'est pas standard.

Toujours dans le registre des méthodes dépassées, on peut citer la méthode **concat ()** qui permet de concaténer une chaîne dans une autre. C'est exactement comme utiliser l'opérateur + :

Code : JavaScript

```
var string_1 = 'Ceci est une chaîne',  
    string_2 = ' de caractères',  
    string_3 = ' ma foi assez jolie'  
  
var result = string_1.concat(string_2, string_3);
```

concat () est donc équivalente à :

Code : JavaScript

```
var result = string_1 + string_2 + string_3;
```

Passons maintenant en revue les autres méthodes de **String** !

La casse et les caractères

toLowerCase () et toUpperCase ()

toLowerCase () et toUpperCase () permettent respectivement de convertir une chaîne en minuscules et en majuscules. C'est pratique pour réaliser différents tests ou pour uniformiser une chaîne de caractères. Leur utilisation est simplisme :

Code : JavaScript

```
var myString = 'tim berners-lee';  
myString = myString.toUpperCase(); // Retourne TIM BERNERS-LEE
```

Accéder aux caractères

La méthode charAt () permet de récupérer un caractère en fonction de sa position dans la chaîne de caractères. La méthode reçoit un paramètre : la position du caractère :

Code : JavaScript

```
var myString = 'Pauline';  
var first = myString.charAt(0); // P  
var last = myString.charAt(myString.length - 1); // e
```

En fait, les chaînes de caractères peuvent être imaginées comme des tableaux, à la différence qu'il n'est pas possible d'accéder aux caractères en utilisant les crochets : à la place, il faut utiliser charAt () .



Certains navigateurs permettent toutefois d'accéder aux caractères d'une chaîne comme s'il s'agissait d'un tableau, c'est-à-dire comme ceci : myString[0]. Ce n'est pas standard, donc il est fortement conseillé d'utiliser myString.charAt(0)

Obtenir le caractère en ASCII

La méthode charCodeAt () fonctionne comme charAt () à la différence que ce n'est pas le caractère qui est retourné mais le code ASCII du caractère.

Créer une chaîne de caractères depuis une chaîne ASCII

fromCharCode () permet de faire, entre guillemets, l'inverse de charCodeAt () : instancier une nouvelle chaîne de caractères sur base d'une chaîne ASCII, dont chaque code est séparé par une virgule. Son fonctionnement est quelque peu étrange puisqu'il est nécessaire d'utiliser l'objet String lui-même :

Code : JavaScript

```
var myString = String.fromCharCode(74, 97, 118, 97, 83, 99, 114,  
105, 112, 116); // le mot JavaScript en ASCII  
alert(myString); // Affiche JavaScript
```




Mais quel est l'intérêt d'utiliser les codes ASCII ?

Ce n'est pas très fréquent mais ça peut être utile dans un cas bien particulier : détecter les touches du clavier. Admettons qu'il faille savoir quelle touche du clavier a été pressée par l'utilisateur. Pour cela, on utilise la propriété `keyCode` de l'élément `event`. On peut aussi utiliser le gestionnaire d'événement `onkeyup` pour savoir quand une touche vient d'être pressée :

Code : HTML

```
<textarea onkeyup="listenKey(event)"></textarea>
```

La fonction `listenKey()` peut être écrite comme suit :

Code : JavaScript

```
function listenKey(event) {  
    var key = event.keyCode;  
    alert('La touche numéro ' + key + ' a été pressée. Le caractère '  
+ String.fromCharCode(key) + ' a été inséré.');
```

[Essayer le code complet !](#)

`event.keyCode` retourne le code ASCII qui correspond au caractère inséré par la touche. Pour la touche `J`, le code ASCII est 74 par exemple.



Cela fonctionne bien pour les touches de caractères, mais ça ne fonctionne pas pour les touches de fonction comme Delete, Enter, Shift, Caps... Ces touches retournent bien un code ASCII mais celui-ci ne peut être converti en un caractère "lisible par un humain". Par exemple, le `keyCode` de la barre d'espace est 32. Donc pour savoir si l'utilisateur a pressé ladite barre, il suffira de tester si le `keyCode` vaut 32 et d'agir en conséquence.

Supprimer les espaces blancs avec `trim()`

`trim()` est une petite exception, car c'est une méthode "jeune" : `trim()` n'était pas présente lors des débuts du Javascript et qui a été ajoutée en 2009 avec la sortie de la version 1.8.1 de Javascript. `trim()` n'est de ce fait pas supportée par les navigateurs anciens comme Internet Explorer 7, Firefox 3 etc.

`trim()` sert à supprimer les espaces blancs avant et après une chaîne de caractères. C'est particulièrement utile quand on récupère des données saisies dans une zone de texte, car l'utilisateur est susceptible d'avoir laissé des espaces avant et après son texte.



Cette méthode étant relativement jeune, elle n'est pas disponible dans les anciens navigateurs et n'apparaît dans Internet Explorer qu'à partir de la version 8.

Variantes de `trim()`

Firefox supporte deux variantes de `trim()` : `trimLeft()` et `trimRight()` qui permettent respectivement d'enlever les espaces à gauche ou à droite. Ces deux méthodes ne sont pas standard et ne sont donc pas disponibles, pour le moment, dans les autres navigateurs.

Rechercher, couper et extraire

Connaître la position avec `indexOf()` et `lastIndexOf()`

La méthode `indexOf()` est utile dans deux cas de figure :

- savoir si une chaîne de caractères contient un caractère ou un morceau de chaîne
- savoir à quelle position se trouve le premier caractère de la chaîne recherchée

`indexOf()` retourne la position du premier caractère trouvé, et s'il n'y en a pas, la valeur -1 est retournée.

Code : JavaScript

```
var myString = 'Le JavaScript est plutôt cool';
var result   = myString.indexOf('JavaScript')

if (result > -1) {
  alert('La chaîne contient le mot "Javascript" qui débute à la
position ' + result);
}
```

[Essayer !](#)

Le code ci-dessus a pour but de savoir si la chaîne **myString** contient la chaîne *JavaScript*. La position de la première occurrence de la chaîne recherchée est stockée dans la variable **result**. Si **result** vaut -1, alors la chaîne n'a pas été trouvée. Si, en revanche, **result** vaut 0 (le premier caractère) ou une autre valeur, la chaîne est trouvée.

Si `indexOf()` retourne la position de la première occurrence trouvée, `lastIndexOf()` retourne la position de la dernière.

Extraire une chaîne avec `substring()`, `substr()` et `slice()`

On a vu comment trouver la position d'une chaîne de caractères dans une autre, et il est temps de voir comment extraire une portion de chaîne, sur base de cette position.

Considérons cette chaîne :

Code : JavaScript

```
var myString = 'Thunderseb et Nesquik69';
```

Le but du jeu va être de récupérer, dans deux variables différentes, les deux pseudonymes contenus dans `myString`. Pour ce faire, nous allons utiliser `substring()`. `substring(a, b)` permet d'extraire une chaîne à partir de la position `a` jusqu'à la position `b`.

Pour extraire `Thunderseb`, il suffit de connaître la position du premier espace blanc, puisque la position de départ vaut 0 :

Code : JavaScript

```
var nick_1 = myString.substring(0, myString.indexOf(' '));
```

Pour Nesquik69, il suffit de connaître la position du dernier espace blanc : c'est à ce moment que commencera la chaîne. Comme Nesquik69 termine la chaîne, il n'y a pas besoin de spécifier de deuxième paramètre pour `substring()`, la méthode va automatiquement aller jusqu'au bout :

Code : JavaScript

```
var nick_2 = myString.substring(myString.lastIndexOf(' ') + 1); //  
Ne pas oublier d'ajouter 1, pour débiter au N, et non à l'espace  
blanc
```

Une autre manière de procéder serait d'utiliser `substr()`, la méthode sœur de `substring()`. `substr(a, n)` accepte deux paramètres : le premier est la position de début, et le deuxième le nombre de caractères à extraire. Cela pré-suppose donc de connaître le nombre de caractères à extraire. Ça limite son utilisation, et c'est une méthode que vous ne rencontrerez pas fréquemment, au contraire de `substring()`.

Une dernière méthode d'extraction existe : `slice()`. `slice()` ressemble très fortement à `substring()`, mais avec une option en plus. Une valeur négative est transmise pour la position de fin, `slice()` va extraire la chaîne jusqu'à la fin, en décomptant le nombre de caractères indiqué. Par exemple, si on ne veut récupérer que *Thunder* hors de *Thunderseb*, on peut faire comme ceci :

Code : JavaScript

```
var nick_1 = 'Thunderseb'.slice(0, -3);
```

Couper une chaîne en un tableau avec `split()`

La méthode `split()` permet de couper une chaîne de caractères à chaque fois qu'une sous-chaîne est rencontrée. Les "morceaux" résultats de la coupe de la chaîne sont placés dans un tableau.

Code : JavaScript

```
var myCSV = 'Pauline,Guillaume,Clarisse'; // CSV = Comma-separated  
values  
  
var splitted = myCSV.split(','); // On coupe à chaque fois qu'une  
virgule est rencontrée  
  
alert(splitted.length); // 3
```

[Essayer !](#)

Dans l'exemple ci-dessus, `splitted` contient un tableau contenant 3 éléments : Pauline, Guillaume et Clarisse.



`split()` peut aussi couper une chaîne à chaque fois qu'un retour à la ligne est rencontré : `myString.split('\n')`. C'est très pratique pour créer un tableau où chaque item contient une ligne, hors d'un texte issu d'un formulaire par exemple.

Tester l'existence d'une chaîne de caractères

Nous l'avons vu plus haut, l'instruction `typeof` est utile, mais la seule valeur de confiance qu'elle retourne c'est `"undefined"`. En effet, lorsqu'une variable contient le type primitif d'une chaîne de caractères, `typeof` retourne bien la valeur `"string"`, mais si la variable contient une instance de `String` alors on obtient en retour la valeur `"object"`. Ce qui fait que `typeof` ne fonctionne que dans un cas sur deux, il nous faut donc une autre solution pour gérer le deuxième cas.

Et cette solution existe ! Il s'agit de la méthode `valueOf()` qui est héritée de `Object`. Cette méthode renvoie la valeur primitive de n'importe quel objet. Ainsi, si on crée une instance de `String` :

Code : JavaScript

```
var string_1 = new String('Test');
```

Et que l'on récupère le résultat de sa méthode `valueOf()` dans la variable `string_2` :

Code : JavaScript

```
var string_2 = string_1.valueOf();
```

Alors l'instruction `typeof` montre bien que `string_1` est une instance de `String` et que `string_2` est une valeur primitive :

Code : JavaScript

```
alert(typeof string_1); // Affiche "object".  
alert(typeof string_2); // Affiche "string".
```

[Essayer le code complet !](#)

Grâce à cette méthode, il devient bien plus simple de vérifier si une variable contient une chaîne de caractères ou non, voici notre code final :

Code : JavaScript

```
function isString(variable) {  
    return typeof variable.valueOf() === 'string'; // Si le type de la  
    valeur primitive est 'string' alors on retourne "true".  
}
```



D'accord, cette fonction va s'exécuter correctement si on envoie une instance de `String`. Mais que va renvoyer `valueOf()` si on passe une valeur primitive à notre fonction ?

Eh bien, tout simplement la même valeur. Je m'explique : `valueOf()` retourne la valeur primitive d'un objet, mais si cette méthode est utilisée sur une valeur qui est déjà de type primitif, alors elle va retourner la même valeur primitive, ce qui convient très bien vu que dans tous les cas il s'agit de la valeur primitive que nous souhaitons analyser !

Pour vous convaincre, testez donc par vous-même :

Code : JavaScript

```
alert(isString('Test')); // Affiche "true".  
alert(isString(new String('Test'))); // Affiche "true".
```

Essayer le code complet !

Notre fonction marche dans les deux cas 😊.

Alors, est-ce qu'en pratique vous aurez besoin d'une fonction pareille ? La réponse est : assez peu... Mais il est toujours bon de savoir comment s'en servir non ?

Et enfin, à titre d'information, sachez qu'il est aussi possible d'obtenir une instantiation d'objet à partir d'un type primitif (autrement dit, on fait l'inverse `valueOf()`), il suffit de procéder de cette manière :

Code : JavaScript

```
var myString = Object('Mon texte');
```

Pour rappel, `Object` est l'objet dont tous les autres objets (tel que `String`) héritent. Ainsi, en créant une instance de `Object` avec un type primitif en paramètre, l'objet instancié sera de même type que la valeur primitive. En clair, si vous passez en paramètre un type primitif **string** alors vous obtiendrez une instance de l'objet `String` avec la même valeur passée en paramètre.

Les expressions régulières [Partie 1]

Dans ce chapitre, nous allons aborder quelque chose d'assez complexe : les expressions régulières. C'est complexe, mais aussi très puissant ! Ce n'est pas un concept lié à Javascript, car les expressions régulières, souvent surnommées *regex*, trouvent leur place dans bon nombre de langages, comme Perl, Python ou encore PHP.

Les regex sont une sorte de langage "à part" qui sert à manipuler les chaînes de caractères. Voici quelques exemples de ce que les regex sont capables de faire :

- Vérifier si une URL entrée par un utilisateur ressemble effectivement à une URL. On peut faire pareil pour les adresses email, les numéros de téléphone et toute autre syntaxe structurée ;
- Rechercher et extraire des informations hors d'une chaîne de caractères (bien plus puissant que de jouer avec `indexOf()` et `substring()`);
- Supprimer certains caractères, et au besoin, les remplacer par d'autres ;
- Pour les forums, convertir des langages comme le BBCode en HTML lors des prévisualisations pendant la frappe ;
- Et bien d'autres choses...

Les regex en Javascript

La syntaxe des regex en Javascript découle de la syntaxe des regex [Perl](#). Perl est un langage assez utilisé pour l'analyse et le traitement des données textuelles (des chaînes de caractères donc), en raison de la puissance de ses expressions régulières. Javascript hérite donc d'une grande partie de la puissance des expressions régulières de Perl.



Si vous avez déjà appris le [PHP](#), vous avez certainement vu que ce langage supporte deux types de regex : les regex POSIX et les regex PCRE. Ici, oubliez les POSIX ! En effet, les regex PCRE sont semblables aux regex Perl (avec quelques nuances), donc à celles de Javascript.

Utilisation

Les regex ne s'utilisent pas seules, et il y a deux manières de les utiliser : soit par le biais de l'objet `RegExp` qui est l'objet qui gère les expressions régulières, soit par le biais de certaines méthodes de l'objet `String` :

- `match()` : retourne un tableau contenant toutes les occurrences recherchées
- `search()` : retourne la position d'une portion de texte (semblable à `indexOf()` mais avec une regex)
- `split()` : la fameuse méthode `split()`, mais avec une regex en paramètre
- `replace()` : effectue un rechercher/remplacer

Nous n'allons pas commencer par ces 3 méthodes car nous allons d'abord nous entraîner à écrire et tester des regex et pour ce faire, nous utiliserons la méthode `test()` fournie par l'objet `RegExp`. L'instanciation d'un objet `RegExp` se fait comme ceci :

Code : JavaScript

```
var myRegex = /contenu_à_rechercher/;
```

Ça ressemble à une chaîne de caractères à l'exception qu'elle est encadrée par deux slashes (/) au lieu des apostrophes ou guillemets traditionnels.

Si votre regex contient elle-même des slashes, n'oubliez pas de les échapper en utilisant un anti-slash comme suit :

Code : JavaScript



```
var regex_1 = /contenu_/contenu/; // la syntaxe est FAUSSE
car le slash n'est pas échappé
var regex_2 = /contenu_\/_contenu/; // ici, le slash est
échappé avec un anti-slash
```

L'utilisation de la méthode `test()` est très simple. En cas de réussite du test, `true` est renvoyé. Dans le cas contraire, c'est `false`.

Code : JavaScript

```
if (myRegex.test('Chaîne de caractères dans laquelle effectuer la
recherche')) {
  // Retourne true sur le test est réussi
} else {
  // Retourne false dans le cas contraire
```

```
}
```

Pour vos tests, n'hésitez pas à utiliser une syntaxe plus concise, comme ceci :

Code : JavaScript

```
if (/contenu_à_rechercher/.test('Chaîne de caractères bla bla bla'))
```


Recherches de mots

Le sujet étant complexe, nous allons commencer par des choses simples, c'est-à-dire des recherches de mots. Ce n'est pas si anodin que ça, car il y a déjà moyen de faire beaucoup de choses. Comme expliqué ci-dessus, une regex s'écrit comme suit :

```
/contenu_de_la_regex/
```

Où contenu_de_la_regex sera à remplacer par ce que nous allons rechercher. Comme nous faisons du Javascript, nous avons de bons goûts et donc nous allons parler de raclette savoyarde. Écrivons donc une regex qui va regarder si dans une phrase, le mot raclette apparaît :

```
/raclette/
```

Si on teste ceci, voilà ce que ça donne :

Code : JavaScript

```
if (/raclette/.test('Je mangerais bien une raclette savoyarde !')) {  
  alert('Ça semble parler de raclette');  
} else {  
  alert('Pas de raclette à l\'horizon');  
}
```

[Essayer !](#)

Résumons tout ça. Le mot raclette a été trouvé dans la phrase *Je mangerais bien une raclette savoyarde !*. Si on change le mot recherché, tartiflette, par exemple, le test retourne **false**, puisque ce mot n'est pas contenu dans la phrase.

Si on change notre regex et qu'on met une majuscule au mot raclette, comme ceci : `/Raclette/`, le test renverra **false**, car le mot raclette présent dans la phrase ne comporte pas de majuscule. C'est relativement logique en fait. Il est possible, grâce aux options, de dire à la regex d'ignorer la casse, c'est-à-dire de rechercher indifféremment les majuscules et les minuscules. Cette option s'appelle **i**, et comme chaque option (nous en verrons d'autre), elle se place juste après le slash de fermeture de la regex :

```
/Raclette/i
```

Avec cette option, la regex reste utilisable comme nous l'avons vu précédemment, à savoir :

Code : JavaScript

```
if (/Raclette/i.test('Je mangerais bien une raclette savoyarde !'))  
{  
  alert('Ça semble parler de raclette');  
} else {  
  alert('Pas de raclette à l\'horizon');  
}
```

[Essayer !](#)

Ici, majuscule ou pas, la regex n'en tient pas compte, et donc le mot *Raclette* est trouvé, même si le mot présent dans la phrase ne comporte pas de majuscule.

À la place de *Raclette*, la phrase pourrait contenir le mot *Tartiflette*. Pouvoir écrire une regex qui rechercherait soit *Raclette* soit *Tartiflette* serait donc intéressant. Pour ce fait, nous disposons de l'opérateur OU, représenté par la barre verticale *pipe* (|)

). Son utilisation est très simple puisqu'il suffit de le placer entre chaque mot recherché, comme cela :

```
/Raclette|Tartiflette/i
```

Code : JavaScript

```
if (/Raclette|Tartiflette/i.test('Je mangerais bien une tartiflette savoyarde !')) {  
  alert('Ça semble parler de trucs savoyards');  
} else {  
  alert('Pas de plats légers à l\'horizon');  
}
```

[Essayer !](#)

La recherche peut inclure plusieurs possibilités, pas seulement deux :

```
/Raclette|Tartiflette|Fondue|Croziflette/i
```

Avec cette regex, on saura si la phrase contient au choix une des 4 spécialités savoyardes !

Début et fin de chaîne

Les symboles `^` et `$` permettent respectivement de représenter le début et la fin d'une chaîne de caractères. Si un de ces symboles est présent, il indique à la regex que ce qui est recherché commence ou termine la chaîne. Cela délimite la chaîne en quelque sorte :

```
/^raclette savoyarde$/
```

Voici un tableau avec divers tests qui sont effectués pour montrer l'utilisation de ces deux symboles :

Chaîne	Regex	Résultat
Raclette savoyarde	<code>/^Raclette savoyarde\$/</code>	true
Une raclette savoyarde	<code>/^Raclette/</code>	false
Une raclette savoyarde	<code>/savoyarde\$/</code>	true
Une raclette savoyarde !	<code>/raclette savoyarde\$/</code>	false

Les caractères et leurs classes

Jusqu'à présent les recherches étaient plutôt basiques. Nous allons maintenant étudier les classes de caractères qui permettent de spécifier plusieurs caractères ou types de caractères à rechercher. Ça reste encore assez simple.

```
/gr[ao]s/
```

Une classe de caractères est écrite entre crochets, et sa signification est simple : une des lettres qui se trouve entre les crochets peut convenir. Cela veut donc dire que l'exemple ci-dessus va trouver les mots *gras* et *gros*, car la classe, à la place de la voyelle, contient, aux choix, les lettres a et o. Beaucoup de caractères peuvent être utilisés au sein d'une classe :

```
/gr[aèio]s/
```

Ici, la regex trouvera les mots *gras*, *grès*, *gris* et *gros*. Ainsi donc, en parlant d'une tartiflette, qu'elle soit grosse ou grasse, cette regex le saura :

Chaîne	Regex	Résultat
Cette tartiflette est grosse	/Cette tartiflette est gr[ao]sse/	true
Cette tartiflette est grasse	/Cette tartiflette est gr[ao]sse/	true

Les intervalles de caractères

Toujours au sein des classes de caractères, il est possible de spécifier un intervalle de caractères. Si on veut trouver les lettres allant de a à o, on écrira `[a-o]`. Si n'importe quelle lettre de l'alphabet peut convenir, il est donc inutile de les écrire toutes : écrire `[a-z]` suffit.

Plusieurs intervalles peuvent être écrits au sein d'une même classe. Ainsi, la classe `[a-zA-Z]` va rechercher toutes les lettres de l'alphabet, qu'elles soient minuscules ou majuscules. Si un intervalle fonctionne avec des lettres, il fonctionne aussi avec des chiffres ! La classe `[0-9]` trouvera donc un chiffre allant de 0 à 9. Il est bien évidemment possible de combiner des chiffres et des lettres : `[a-z0-9]` trouvera un lettre minuscule ou un chiffre.

Exclure des caractères

Si au sein d'une classe on peut inclure des caractères, on peut aussi en exclure ! Pour ce faire, il suffit de faire figurer un accent circonflexe au début de la classe, juste après le premier crochet : `[^aeuyio]`. Cette classe trouvera une consonne, puisqu'elle ignorera les voyelles. L'exclusion d'un intervalle est possible aussi : `[^b-y]` qui exclura les lettres allant de b à y.



Il faut prendre en compte que la recherche n'ignore pas les caractères accentués. Ainsi, `[a-z]` trouvera *a*, *b*, *i*, *o*... mais ne trouvera pas *â*, ni *ï* ou encore *ê*. S'il s'agit de trouver un caractère accentué, il faut l'indiquer explicitement : `[a-zââààèèùèëîïôöçñ]`

Trouver un caractère quelconque

Le point permet de trouver n'importe quel caractère, à l'exception des sauts de ligne (les retours à la ligne). Ainsi, la regex ci-dessous trouvera les mots *gras*, *grès*, *gris*, *gros*, et d'autres mots inexistantes comme *grys*, *grus*, *grps*... Le point symbolise donc un caractère quelconque :

```
/gr.s/
```

Les quantificateurs

Les quantificateurs permettent de dire combien de fois un caractère doit être recherché. Il est possible de dire qu'un caractère peut apparaître 0 ou 1 fois, une fois ou une infinité de fois, ou même, avec des accolades, de dire qu'un caractère peut être répété 3, 4, 5 ou 10 fois.

A partir d'ici, la syntaxe des regex va devenir plus complexe !

Les 3 symboles quantificateurs

- `?` : ce symbole indique que le caractère qui le précède peut apparaître 0 ou 1 fois ;
- `+` : ce symbole indique que le caractère qui le précède peut apparaître 1 ou plusieurs fois ;
- `*` : ce symbole indique que le caractère qui le précède peut apparaître 0, 1 ou plusieurs fois.

Reprenons notre raclette. Il y a deux *t*, mais il se pourrait que l'utilisateur ait fait une faute de frappe et n'en ait mis qu'un seul. On va donc écrire une regex capable de gérer ce cas de figure :

```
/raclett?e/
```

Ici, le premier *t* sera trouvé, et derrière le deuxième se trouve le point d'interrogation, ce qui signifie que le deuxième *t* peut apparaître 0 ou 1 fois. Cette regex gère donc notre cas de figure.

Un cas saugrenu serait qu'il y ait beaucoup de *t* : *racletttttttte*. Pas de panique, il suffit d'utiliser le quantificateur `+` :

```
/raclet+e/
```

Le `+` indique que le *t* sera présent une fois ou un nombre infini de fois. Avec le symbole `*`, la lettre est facultative mais peut être répétée un nombre infini de fois. En utilisant `*`, la regex ci-dessus peut s'écrire :

```
/raclett*e/
```

Les accolades

À la place des 3 symboles vus précédemment, on peut utiliser des accolades pour définir explicitement combien de fois un caractère peut être répété. Trois syntaxes sont disponibles :

- `{n}` : le caractère est répété *n* fois ;
- `{n,m}` : le caractère est répété de *n* à *m* fois. Par exemple, si on a `{0,5}`, le caractère peut être présent de 0 à 5 fois ;
- `{n, }` : le caractère est répété de *n* fois à l'infini

Si la tartiflette possède un, deux ou trois *t*, la regex peut s'écrire :

```
/raclet{1,3}e/
```

Les quantificateurs, accolades ou symboles, peuvent aussi être utilisés avec les classes de caractères. Si on mange une *racleffe* au lieu d'une raclette, on peut imaginer la regex suivante :

```
/racle[tf]+e/
```

Voici, pour clôturer ce chapitre, quelques exemples de regex qui utilisent tout ce qui a été vu :

Chaîne	Regex	Résultat
Hellooooooooo	<code>/Hello+/?</code>	true
Goooooooo	<code>/Go{2,}gle/?</code>	true
Le 1er septembre	<code>/Le [1-9] [a-z]{1,2} septembre/?</code>	true
Le 1er septembre	<code>/Le [1-9] [a-z]{1,2} [a-z]+/?</code>	false

La dernière regex est fautive à cause de l'espace blanc. En effet, la classe `[a-z]` ne trouvera pas l'espace blanc. Nous verrons ça dans un prochain chapitre sur les regex.

Les métacaractères

Nous avons vu précédemment que la syntaxe des regex est définie par un certain nombre de caractères spéciaux, comme `^`, `$`, `[` et `]` ou encore `+` et `*`. Ces caractères sont ce que l'on appelle des métacaractères, et en voici la liste complète :

`! ^ $ () [] { } ? + * . / \ |`

Un problème se pose si on veut chercher la présence d'une accolade dans une chaîne de caractères. En effet, si on a ceci, la regex ne fonctionnera pas :

Chaîne	Regex	Résultat
Une accolade {comme ceci}	<code>/accolade {comme ceci}/</code>	false

C'est normal, car les accolades sont des métacaractères qui définissent un nombre de répétition : en clair, cette regex n'a aucun sens pour l'interpréteur Javascript ! Pour palier à ce problème, il suffit d'échapper les accolades au moyen d'un anti-slash : `\` :

`/accolade \{comme ceci}\`

De cette manière, les accolades seront vues par l'interpréteur comme étant des accolades "dans le texte", et non comme des métacaractères. Il en va de même pour tous les métacaractères cités précédemment. Il faut même penser à échapper l'anti-slash, avec... un anti-slash :

Chaîne	Regex	Résultat
Un slash / et un anti-slash \	<code>// et un anti-slash \</code>	erreur de syntaxe
Un slash / et un anti-slash \	<code>/\ / et un anti-slash \</code>	true

Ici, pour pouvoir trouver le `/` et le `\`, il convient également de les échapper.



Il est à noter que si le logiciel que vous utilisez pour rédiger en Javascript fait bien son job, la première regex ci-dessus provoquera une mise en commentaire (à cause des deux `//` au début) : c'est un bon indicateur pour dire qu'il y a un problème. Si votre logiciel détecte aussi les erreurs de syntaxe, il peut vous être d'une aide précieuse.

Les métacaractères au sein des classes

Au sein d'une classe de caractères, il n'y a pas besoin de d'échapper les métacaractères, à l'exception des crochets (qui délimitent le début et la fin d'une classe), du tiret (qui est utilisé pour définir un intervalle) et du slash (qui symbolise la fin de la regex) et de l'anti-slash (qui sert à échapper).



Concernant le tiret, il existe une petite exception : il n'a pas besoin d'être échappé si il est placé en début ou en fin de classe.

Ainsi, si on veut rechercher un caractère de a à z ou les métacaractères `!` et `?`, il faudra écrire ceci :

`/[a-z!?!?]/`

Et s'il faut trouver un slash ou un anti-slash, il ne faut pas oublier de les échapper :

`/[a-z!?!?\/\\]/`

Types génériques et assertions

Les types génériques

On a vu que les classes étaient pratique pour chercher un caractère au sein d'un groupe, ce qui permet de trouver un caractère sans savoir au préalable quel sera ce caractère. Seulement, utiliser des classe alourdit fortement la syntaxe des regex et les rend difficilement lisibles. Pour palier à ce petit souci, on va utiliser ce qu'on appelle des types génériques. Certains parlent aussi de "classes raccourcies", mais ce terme n'est pas tout à fait exact.

Les types génériques s'écrivent tous de la manière suivante : `\x`, où `x` représente une lettre. Voici la liste de tous les types génériques :

Type	Description
<code>\d</code>	Trouve un caractère décimal (un chiffre)
<code>\D</code>	Trouve un caractère qui n'est pas décimal (donc, pas un chiffre)
<code>\s</code>	Trouve un caractère blanc
<code>\S</code>	Trouve un caractère qui n'est pas un caractère blanc
<code>\w</code>	Trouve un caractère "de mot" : une lettre, accentuée ou non, ainsi que l'underscore
<code>\W</code>	Trouve un caractère qui n'est pas un caractère "de mot"

Les assertions

Les assertions s'écrivent comme les types génériques mais ne fonctionnent pas tout à fait de la même façon. Un type générique recherche un caractère, tandis qu'une assertion recherche entre deux caractères. C'est tout de suite plus simple avec un tableau :

Type	Description
<code>\b</code>	Trouve une limite de mot
<code>\B</code>	Ne trouve pas de limite de mot



Il faut juste faire attention avec l'utilisation de `\b`, car cette assertion reconnaît les caractères accentués comme des "limites de mots". Ça peut donc provoquer des comportements inattendus.

En plus de cela existent les caractères de type "espace blanc" :

Type	Description
<code>\n</code>	Trouve un retour à la ligne
<code>\t</code>	Trouve une tabulation

Ces deux "caractères" sont reconnus par le type générique `\s` (qui trouve, pour rappel, n'importe quel espace blanc).

Ce n'est pas fini ! Les regex reviennent d'ici quelque chapitres, où nous étudierons leur utilisation avec diverses méthodes Javascript.

Les expressions régulières [Partie 2]

Dans ce deuxième chapitre dédié aux regex, nous allons voir leur utilisation au sein de Javascript. En effet, le premier chapitre n'était là que pour enseigner la base de la syntaxe. Ici, nous allons apprendre à utiliser les regex !

Construire une regex

Le gros de la théorie sur les regex est maintenant vu, et il ne reste plus qu'un peu de pratique. Nous allons tout de même voir comment écrire une regex pas à pas, de façon à ne pas se planter.

Nous allons partir d'un exemple simple : vérifier si une chaîne de caractère correspond à une adresse e-mail. Pour rappel, une adresse mail est de cette forme : javascript@siteduzero.com.

Une adresse mail contient 3 parties distinctes :

- la partie locale, avant l'arobase. Ici, c'est javascript
- l'arobase @
- le domaine, lui même composé du label (siteduzero) et de l'extension (.com).

Pour construire une regex, il suffit de procéder par étapes : on fait comme si on lisait la chaîne de caractères et on écrit la regex au fur et à mesure. On écrit d'abord la partie locale, qui n'est composée que de lettres, de chiffres et d'éventuellement un tiret, un trait de soulignement et un point. Tous ces caractères peuvent être répétés plus d'une fois (il faut donc utiliser le quantificateur +) :

```
 /^[a-z0-9._-]+$/
```

On ajoute l'arobase. Ce n'est pas un métacaractère, donc pas besoin de l'échapper :

```
 /^[a-z0-9._-]+@$/
```

Après vient le label du nom de domaine, lui aussi composé de lettres, de chiffres, de tirets et de traits de soulignement. Ne pas oublier le point, car il peut s'agir d'un sous-domaine (par exemple @tutoriels.sitedezero.com) :

```
 /^[a-z0-9._-]+@[a-z0-9._-]+$/
```

Puis vient le point de l'extension du domaine : attention à ne pas oublier de l'échapper, car il s'agit d'un métacaractère :

```
 /^[a-z0-9._-]+@[a-z0-9._-]+\.$/
```

Et pour finir, l'extension ! Une extension de nom de domaine ne contient que des lettres, au minimum 2, maximum 6. Ce qui nous fait :

```
 /^[a-z0-9._-]+@[a-z0-9._-]+\.[a-z]{2,6}$/
```

Testons donc :

Code : JavaScript

```
var email = prompt("Entrez votre adresse e-mail :",  
"javascript@siteduzero.com");  
  
if (/^[a-z0-9._-]+@[a-z0-9._-]+\.[a-z]{2,6}$/ .test(email)) {  
    alert("Adresse e-mail valide !");  
} else {  
    alert("Adresse e-mail invalide !");  
}
```

L'adresse e-mail est détectée comme étant valide !

L'objet RegExp

L'objet `RegExp` est l'objet qui gère les expressions régulières. Il y a donc deux façons de déclarer une regex : via `RegExp` ou via son type primitif que nous avons utilisé jusqu'à présent :

Code : JavaScript

```
var myRegex1 = /^Raclette$/i;
var myRegex2 = new RegExp("^Raclette$", "i");
```

Le constructeur `RegExp` reçoit deux paramètres : le premier est l'expression régulière sous la forme d'une chaîne de caractères, et le deuxième est l'option de recherche, ici, `i`. L'intérêt d'utiliser `RegExp` est qu'il est possible d'inclure des variables dans la regex, chose impossible en passant par le type primitif :

Code : JavaScript

```
var nickname = "Sébastien";
var myRegex = new RegExp("Mon prénom est " + nickname, "i");
```

Ce n'est pas spécialement fréquent, mais ça peut se révéler particulièrement utile. Il est cependant conseillé d'utiliser la notation littérale (le type primitif) quand l'utilisation du constructeur `RegExp` n'est pas nécessaire.

Méthodes

`RegExp` ne possède que deux méthodes : `test()` et `exec()`. La méthode `test()` a déjà été utilisée et permet de tester une expression régulière, et renvoie `true` si le texte est réussi ou `false` si le test échoue. De son côté, `exec()` applique également une expression régulière mais renvoie un tableau dont le premier élément contient la portion de texte trouvée dans la chaîne de caractères. Si rien n'est trouvé, `null` est renvoyé.

Code : JavaScript

```
var sentence = "Si ton tonton";

var result = /\bton\b/.exec(sentence);

if (result) { // On vérifie que ce n'est pas null
    alert(result);
}
```

Propriétés

L'objet `RegExp` contient 9 propriétés, appelées `$1`, `$2`, `$3`... jusqu'à `$9`. Comme nous allons le voir dans la sous-partie suivant, il est possible d'utiliser une regex pour extraire des portions de texte, et ces portions sont accessibles via les propriétés de `$1` à `$9`.

Tout ceci va être mis en lumière un peu plus bas en parlant des parenthèses capturantes.

Les parenthèses capturantes

Nous avons vu pour le moment que les regex servaient à voir si une chaîne de caractère correspondait à un modèle (la regex). Mais il y a moyen de faire mieux, comme extraire des informations. Pour définir les informations à extraire, on utilise des parenthèses, que l'on appelle parenthèses capturantes car leur utilité est de capturer une portion de texte, que la regex va extraire.

Considérons cette chaîne de caractères : 'Je suis né en mars'. Au moyen de parenthèses capturantes, nous allons extraire le mois de la naissance, pour pouvoir le réutiliser :

Code : JavaScript

```
var birth = 'Je suis né en mars';

/^Je suis né en (\S+)$/.exec(birth);

alert(RegExp.$1); // Affiche 'mars'
```

Cet exemple est un peu déroutant, mais est en réalité assez simple à comprendre. Dans un premier temps, on crée la regex avec les fameuses parenthèses. Comme les mois sont faits de caractères qui peuvent être accentués, on peut directement utiliser le type générique `\S`. `\S+` indique qu'on recherche une série de caractères, jusqu'à la fin de la chaîne (délimitée, pour rappel, par `$`) : ce sera le mois. On englobe ce "mois" dans des parenthèses pour faire comprendre à l'interpréteur Javascript que leur contenu devra être extrait.

La regex est exécutée via `exec()`. Et ici, une autre explication s'impose. Quand on exécute `test()` ou `exec()`, le contenu des parenthèses capturantes est enregistré temporairement au sein de l'objet `RegExp`. Le premier couple de parenthèses sera enregistré dans la propriété `$1`, le deuxième dans `$2` et ainsi de suite, jusqu'au neuvième, dans `$9`. Cela veut donc dire qu'il ne peut y avoir que maximum que 9 couples de parenthèses. Les couples sont numérotés suivant le sens de lecture, de gauche à droite.

Et pour accéder aux propriétés, il suffit de faire `RegExp.$1`, `RegExp.$2`, etc...

Voici un autre exemple, reprenant le regex de validation de l'adresse mail. Ici, le but est de décomposer l'adresse pour récupérer les différentes parties :

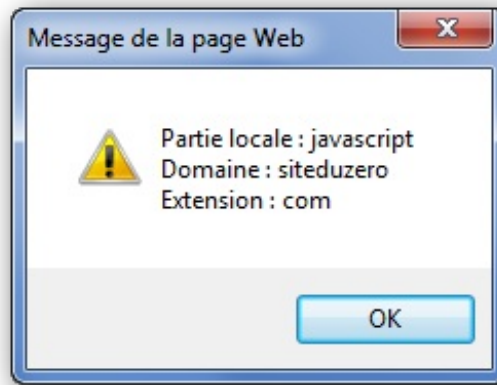
Code : JavaScript

```
var email = prompt("Entrez votre adresse e-mail :",
"javascript@siteduzero.com");

if (/^([a-z0-9._-]+)@([a-z0-9._-]+\.[a-z]{2,6})$/i.test(email)) {
    alert('Partie locale : ' + RegExp.$1 + '\nDomaine : ' +
    RegExp.$2 + '\nExtension : ' + RegExp.$3);
} else {
    alert('Adresse e-mail invalide !');
}
```

[Essayer !](#)

Ce qui nous affiche bien les 3 parties :



Remarquez que même si `test ()` et `exec ()` sont exécutés au sein d'un `if ()` le contenu des parenthèses est quand même enregistré. Pas de changement de ce côté là !

Les recherches non-greedy

Le mot anglais *greedy* signifie "gourmand". En Javascript, les regex sont généralement gourmandes, ce qui veut dire que quand on utilise un quantificateur comme le +, le maximum de caractères est recherché, alors que ce n'est pas toujours le comportement espéré. Petite mise en lumière :

Nous allons construire une regex qui va extraire l'adresse Web à partir de cette portion de HTML sous forme de chaîne de caractères :

Code : JavaScript

```
var html = '<a href="www.mon-adresse.be">Mon site</a>';
```

Voici la regex qui peut être construite :

```
</a href="(.)+"/
```

Et ça marche :

Code : JavaScript

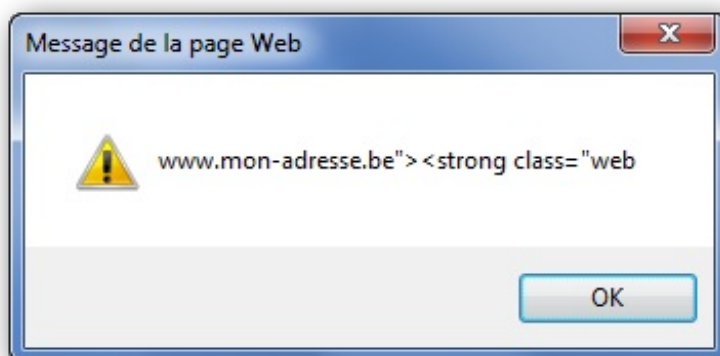
```
</a href="(.)+"/.exec(html);  
alert(RegExp.$1); // www.mon-adresse.be
```

Maintenant, supposons que la chaîne de caractères ressemble à ceci :

Code : JavaScript

```
var html = '<a href="www.mon-adresse.be"><strong class="web">Mon  
site</strong></a>';
```

Et là, c'est le drame :



En spécifiant `.+` comme quantificateur, on demande de rechercher le plus possible de caractères jusqu'à rencontrer les caractères `">`, et c'est ce que Javascript fait :

```
<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>
```

JavaScript va trouver la partie surlignée : il cherche jusqu'à ce qu'il tombe sur la dernière apparition des caractères ">". Mais ce n'est pas dramatique, fort heureusement !

Pour palier à ce problème, on va écrire le quantificateur directement suivi du point d'interrogation, comme ceci :

Code : JavaScript

```
var html = '<a href="www.mon-adresse.be"><strong class="web">Mon  
site</strong></a>';  
  
/<a href="(.*?) ">/ .exec (html) ;  
  
alert (RegExp.$1) ;
```

Le point d'interrogation va faire en sorte que la recherche soit moins gourmande, et s'arrête une fois que le minimum requis est trouvé, d'où l'appellation "non-greedy" (*non gourmande*).



Petite précision, ça ne fonctionne pas avec les accolades.

Rechercher et remplacer

Une fonctionnalité intéressante des regex est de pouvoir effectuer des recherches-remplacements. Rechercher-remplacer signifie qu'on recherche des portions de texte dans une chaîne de caractères et qu'on remplace ces portions par d'autres. C'est relativement pratique pour modifier une chaîne rapidement, ou pour convertir des données. Une utilisation fréquente est la conversion de balises BBCode en HTML pour prévisualiser le contenu d'une zone de texte.

Un recherche-remplacement se fait au moyen de la méthode `replace()` de l'objet `String`. La méthode reçoit deux arguments : la regex et une chaîne de caractères qui sera le texte de remplacement. Petit exemple :

Code : JavaScript

```
var sentence = 'Je m\'appelle Sébastien';
var result = sentence.replace(/Sébastien/, 'Johann');
alert(result); // Je m'appelle Johann
```

Très simple : `replace` va rechercher le prénom Sébastien et le remplacer par Johann.

Utilisation de `replace()` sans regex

A la place d'une regex, il est aussi possible de fournir une simple chaîne de caractères. C'est utile pour remplacer un mot ou un groupe de mots, mais ce n'est pas une utilisation fréquente, on utilisera généralement une regex. Voici toutefois un exemple :

Code : JavaScript

```
var result = 'Je m\'appelle Sébastien'.replace('Sébastien',
'Johann');
alert(result); // Je m'appelle Johann
```

L'option `g`

Nous avons vu l'option `i` qui permet aux regex d'être insensibles à la casse des caractères. Il existe une autre option, `g`, qui signifie "rechercher plusieurs fois". Par défaut, la regex donnée précédemment ne sera exécutée qu'une fois : dès que Sébastien sera trouvé, il sera remplacé et puis c'est tout. Donc si le prénom Sébastien est présent deux fois, seul le premier sera remplacé. Pour éviter ça, on utilisera l'option `g` qui va dire de continuer la recherche jusqu'à ce que plus rien ne soit trouvé :

Code : JavaScript

```
var sentence = 'Il s\'appelle Sébastien. Sébastien écrit un
tutoriel.';
var result = sentence.replace(/Sébastien/g, 'Johann');
alert(result); // Il s'appelle Johann. Johann écrit un tutoriel.
```

Ainsi, toutes les occurrences de Sébastien sont correctement remplacées par Johann. Le mot occurrence est nouveau ici, et il est maintenant temps de l'employer. A chaque fois que la regex trouve la portion de texte qu'elle recherche, on parle d'occurrence. Dans le code précédent, deux occurrences de Sébastien sont trouvées : une juste après "appelle" et l'autre après le point.

Rechercher et capturer

Il est possible d'utiliser les parenthèses capturantes pour extraire des informations et les réutiliser au sein de la chaîne de remplacement. Par exemple, on a une date au format américain : 05/26/2011 et on veut la convertir au format Jour/Mois/Année. Rien de plus simple :

Code : JavaScript

```
var date = '05/26/2011';  
date = date.replace(/^(\\d{2})\\/(\\d{2})\\/(\\d{4})$/, 'Le $2/$1/$3');  
alert(date); // Le 26/05/2011
```

Chaque nombre est capturé avec une parenthèse, et pour récupérer chaque parenthèse, il suffit d'utiliser \$1, \$2 et \$3 (directement dans la chaîne de caractères), exactement comme nous l'aurions fait avec `RegExp`. \$1.



Et si on veut juste remplacer un caractère par le signe dollar, il faut l'échapper ?

Pour placer un simple caractère \$ dans la chaîne de remplacement, il suffit de le doubler, comme ceci :

Code : JavaScript

```
var total = 'J'ai 25 dollars en liquide.';  
alert(total.replace(/dollars?/, '$$')); // J'ai 25 $ en liquide
```

Le mot dollars est effectivement remplacé par son symbole. Un point d'interrogation a été placé après le s pour pouvoir trouver soit *dollars* soit *dollar*.

Voici un autre exemple illustrant ce principe. L'idée ici est de convertir une balise BBCode de mise en gras :
[b]un peu de texte[/b] en un formatage HTML via `un peu de texte`. N'oubliez pas d'échapper les crochets qui sont, pour rappel, des métacaractères !

Code : JavaScript

```
var text = 'bla bla [b]un peu de texte[/b] bla [b]bla bla en  
gras[/b] bla bla';  
text = text.replace(/\\[b\\] ([\\s\\S]*) \\[\\b\\]/g,  
'<strong>$1</strong>');  
alert(text);
```



Mais pourquoi avoir utilisé `[\\s\\S]` et non pas juste le point ?

Il s'agit ici de trouver TOUS les caractères qui se trouvent entre les balises. Or, le point ne trouve que des caractères et des

espaces blanc hormis le retour à la ligne. C'est la raison pour laquelle on utilisera souvent la classe comprenant `\s` et `\S` quand il s'agira de trouver du texte comportant des retours à la ligne.

Cette petite regex de remplacement est la base d'un système de prévisualisation du BBCode. Il suffit décrire une regex de ce type pour chaque balise, et le tour est joué :

Code : HTML

```
<script type="text/javascript">

function preview() {
    var value = document.getElementById("text").value;

    value = value.replace(/\[b\]([\s\S]*)\[\/b\]/g,
'<strong>$1</strong>');
    value = value.replace(/\[i\]([\s\S]*)\[\/i\]/g, // Gras
'<em>$1</em>');
// Italique
    value = value.replace(/\[s\]([\s\S]*)\[\/s\]/g,
'<del>$1</del>');
    value = value.replace(/\[u\]([\s\S]*)\[\/u\]/g, // Barré
'<span
style="text-decoration: underline">$1</span>'); // Souligné

    document.getElementById("output").innerHTML = value;
}

</script>

<form>
    <textarea id="text"></textarea><br />
    <button type="button" onclick="preview()">Prévisualiser</button>
    <div id="output"></div>
</form>
```

[Essayer !](#)

Utiliser une fonction pour le remplacement

A la place d'une chaîne de caractères, il est possible d'utiliser une fonction pour gérer le ou les remplacements. Ça permet, par exemple, de réaliser des opérations sur les portions capturées (\$1, \$2, \$3...).

Les paramètres de la fonction sont soumis à une petite règle, car ils doivent respecter un certain ordre (mais leurs noms peuvent bien évidemment varier) : `function(str, p1, p2, p3 /* ... */ , offset, s)`. Les paramètres `p1`, `p2`, `p3`... représentent les fameux \$1, \$2, \$3... S'il n'y a que 3 parenthèses capturantes, il n'y aura que 3 "p". S'il y en a 5, il y aura 5. Les paramètres sont expliqués ci-dessous :

- Le paramètre `str` contient la portion de texte trouvée par la regex
- Les paramètres `p*` contiennent les portions capturées par les parenthèses
- Le paramètre `offset` contient la position de la portion de texte trouvée
- Le paramètre `s` contient la totalité de la chaîne



Si on a besoin que de `p1` et de `p2` et pas des deux derniers paramètres, ces deux derniers peuvent être omis.

Pour illustrer cela, nous allons réaliser un petit script tout simple, qui recherchera des nombres dans une chaîne et les transformera en toutes lettres. la transformation se fera au moyen de de la fonction `num2Letters()` qui a été codée lors du

tout premier TP : [Convertir un nombre en toutes lettres](#)

Code : JavaScript

```
var sentence = 'Dans 22 jours, j\'aurai 24 ans';

var result = sentence.replace(/(\d+)/g, function(str, p1) {
  p1 = parseInt(p1);

  if (!isNaN(p1)) {
    return num2Letters(p1);
  }
});

alert(result); // Dans vingt-deux jours, j\'aurai vingt-quatre ans
```

L'exemple utilise une fonction anonyme, mais il est bien évidemment possible de déclarer une fonction :

Code : JavaScript

```
function convertNumbers(p1) {
  p1 = parseInt(p1);

  if (!isNaN(p1)) {
    return num2Letters(p1);
  }
}

var sentence = 'Dans 22 jours, j\'aurai 24 ans';

var result = sentence.replace(/(\d+)/g, convertNumbers);
```

Autres recherches

Il reste deux méthodes de `String` à voir, `search()` et `match()`, plus un petit retour sur la méthode `split()`.

Rechercher la position d'une occurrence

La méthode `search()`, toujours de l'objet `String` ressemble à `indexOf()` hormis que le paramètre est une expression régulière. `search()` retourne la position de la première occurrence trouvée. Si aucune occurrence n'est trouvée, `-1` est retourné. Exactement comme `indexOf()` :

Code : JavaScript

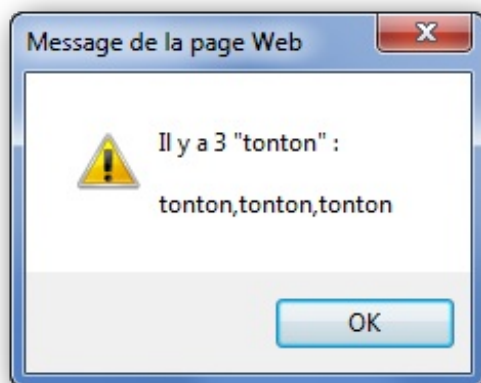
```
var sentence = 'Si ton tonton';  
var result = sentence.search(/\bton\b/);  
  
if (result > -1) { // On vérifie que quelque chose a été trouvé  
    alert('La position est ' + result); // 3  
}
```

Récupérer toutes les occurrences

La méthode `match()` de l'objet `String` fonctionne comme `search()`, à la différence qu'elle retourne un tableau de toutes les occurrences trouvées. C'est pratique pour compter le nombre de fois qu'une portion de texte est présente par exemple :

Code : JavaScript

```
var sentence = 'Si ton tonton tond ton tonton, ton tonton tondu sera';  
var result = sentence.match(/\btonton\b/g);  
  
alert('Il y a ' + result.length + ' "tonton" :\n\n' + result);
```



Couper avec une regex

Nous avons vu que la méthode `split()` recevait une chaîne de caractères en paramètre. Mais il est également possible de

transmette une regex. C'est très pratique pour découper une chaîne sur base, par exemple, de plusieurs caractères distincts :

Code : JavaScript

```
var family = 'Guillaume,Pauline;Clarisse:Arnaud;Benoît;Maxime';  
var result = family.split(/[,;:]/g);  
  
alert(result);
```

L'alert () affiche donc un tableau contenant tous les prénoms, car il a été demandé à split () de couper la chaîne dès qu'une virgule, un deux-points ou un point-virgule est rencontré.

Les données numériques

Après l'étude des chaînes de caractères et des Regex, il est temps de passer aux données numériques.

La gestion des données numériques en Javascript est assez limitée mais elle existe quand même et se fait essentiellement par le biais des objets **Number** et **Math**. Le premier est assez inintéressant mais il est bon de savoir à quoi il sert et qu'est-ce qu'il permet de faire, le deuxième est une véritable boîte à outils qui vous servira probablement un jour où l'autre.

L'objet Number

Cet objet est à la base de tout nombre et pourtant on ne s'en sert quasiment jamais de manière explicite car on lui préfère (comme pour la plupart des objets) l'utilisation de son type primitif. Cet objet possède pourtant des fonctions intéressantes comme, par exemple, faire des conversions depuis une chaîne de caractères jusqu'à un nombre en instanciant un nouvel objet `Number` :

Code : JavaScript

```
var myNumber = new Number('3'); // La chaîne de caractères "3" est
convertie en un nombre de valeur 3.
```

Cependant, cette conversion est "bancale" dans le sens où on ne sait pas si on obtiendra un nombre entier ou flottant en retour, on lui préfère donc les fonctions `parseInt()` et `parseFloat()` qui permettent d'être sûr de ce que l'on obtiendra. De plus, ces deux dernières fonctions utilisent un second argument permettant de spécifier la base (binaire, décimale, etc...) du nombre écrit dans la chaîne de caractères, ce qui permet d'enlever tout soupçon sur le résultat obtenu.

Cet objet possède des attributs accessibles directement sans aucune instanciation (on appelle cela des attributs propres à l'objet constructeur). Ils sont au nombre de cinq (liste non-exhaustive, je vous conseille [de vous référer à la documentation](#) si vous avez un doute) et bien qu'ils ne servent pas à grand chose, je préfère vous les énumérer afin que vous sachiez qu'ils existent (peut-être aurez-vous à vous en servir un jour) :

- **NaN** : Vous connaissez déjà cet attribut qui signifie **Not a Number** et qui permet, généralement, d'identifier l'échec d'une conversion de chaîne de caractères en un nombre. À noter que cet attribut est aussi disponible dans l'objet `window`, passer par l'objet `Number` pour y accéder n'a donc que peu d'intérêt, surtout qu'il est bien rare d'utiliser cet attribut car on lui préfère l'utilisation de la fonction `isNaN()`, plus fiable.
- **MAX_VALUE** : Cet attribut représente le nombre maximum pouvant être stocké dans une variable en Javascript. Cette constante peut changer selon la version du Javascript utilisée.
- **MIN_VALUE** : Identique à la constante **MAX_VALUE** sauf que là il s'agit de la valeur minimale.
- **POSITIVE_INFINITY** : Il s'agit ici d'une constante représentant l'infini positif. Vous pouvez l'obtenir en résultat d'un calcul si vous divisez une valeur positive par 0. Cependant, son utilisation est rare car on lui préfère la fonction `isFinite()`, plus fiable.
- **NEGATIVE_INFINITY** : Identique à **POSITIVE_INFINITY** sauf que là il s'agit de l'infini négatif. Vous pouvez obtenir cette constante en résultat d'un calcul si vous divisez une valeur négative par 0.

Je vous propose de faire un essai :

Code : JavaScript

```
var max = Number.MAX_VALUE, // Comme vous pouvez le constater, je
n'instancie pas mon objet, comme pour un accès au "prototype".
    inf = Number.POSITIVE_INFINITY;

if(max < inf) {
    alert("La valeur maximum en Javascript est inférieure à l'infini !
    Surprenant, n'est-ce pas ?");
}
```

Essayer !

Du côté des méthodes, l'objet `Number` n'est pas bien plus intéressant car toutes les méthodes qu'il possède sont déjà supportées par l'objet `Math`. Cependant, on va faire un petit aparté sur la méthode `toString()` qui permet de convertir un objet en chaîne de caractère, comme ceci :

Code : JavaScript

```
var myNumber = 3;

alert("Le nombre est : " + myNumber.toString());
```

Cette méthode n'est pas disponible uniquement sur l'objet `Number` mais sur tous les objets. C'est cette méthode qui est appelée lorsque vous faites `alert(un_objet)` ; ainsi la fonction `alert()` peut afficher une chaîne de caractères sans aucun problème.

Cependant, cette méthode est assez peu utilisée de manière explicite et dans le cas de notre objet `Number` on privilégiera la syntaxe suivante :

Code : JavaScript

```
var myNumber = 3,
    myString;

myString = '' + myNumber; // On concatène notre nombre à une chaîne
de caractères vide.
```


L'objet Math

Après une première sous-partie assez peu intéressante, nous passons enfin à l'objet `Math` qui va réellement nous servir ! Tout d'abord, deux petites choses :

- La liste des attributs et méthodes ne sera pas exhaustive, [consultez la documentation](#) si vous souhaitez tout connaître.
- Tous les attributs et méthodes de cet objet sont accessible directement sans aucune instantiation, on appelle cela des méthodes et attributs statiques. Considérez donc cet objet plutôt comme un *namespace* 😊.

Les attributs

Les attributs de l'objet `Math` sont des constantes qui définissent certaines valeurs mathématiques comme *le chiffre PI* ou *le nombre d'Euler*. Nous n'allons parler que de ces deux constantes car les autres ne sont clairement pas souvent utilisées en Javascript.

Pour les utiliser, rien de bien compliqué :

Code : JavaScript

```
alert(Math.PI); // Affiche la valeur du chiffre PI.  
alert(Math.E); // Affiche la valeur du nombre d'Euler.
```

Voilà tout, les attributs de l'objet `Math` ne sont pas bien durs à utiliser donc je n'ai pas grand chose à vous apprendre dessus. En revanche, les méthodes sont nettement plus intéressantes !

Les méthodes

L'objet `Math` comporte de nombreuses méthodes permettant de faire divers calculs un peu plus évolués qu'une simple division. Il existe des méthodes pour le calcul des cosinus et sinus, les méthodes d'arrondi et de troncature, etc... Elles sont assez nombreuses pour faire bon nombre d'applications pratiques.

Arrondir et tronquer

Vous aurez souvent besoin d'arrondir vos nombres en Javascript, notamment si vous faites des animations, après tout, il est impossible de dire à un élément HTML qu'il fait 23,33 pixels de largeur, il faut un nombre entier. C'est pourquoi nous allons voir les méthodes `floor()`, `ceil()` et `round()`.

La méthode `floor()` retourne le plus grand entier inférieur ou égal à la valeur que vous avez passé en paramètre :

Code : JavaScript

```
Math.floor(33.15); // Retourne : 33  
Math.floor(33.95); // Retourne : 33  
Math.floor(34); // Retourne: 34
```

Concernant la méthode `ceil()`, celle-ci retourne le plus petit entier supérieur ou égal à la valeur que vous avez passé en paramètre :

Code : JavaScript

```
Math.ceil(33.15); // Retourne : 34
Math.ceil(33.95); // Retourne : 34
Math.ceil(34); // Retourne: 34
```

Et pour finir, la méthode `round()` qui fait un arrondi tout bête :

Code : JavaScript

```
Math.round(33.15); // Retourne : 33
Math.round(33.95); // Retourne : 34
Math.round(34); // Retourne: 34
```

Calculs de puissance et de racine carrée

Bien que le calcul d'une puissance puisse paraître bien simple à coder, il existe une méthode permettant d'aller plus rapidement, celle-ci se nomme `pow()` et s'utilise de cette manière :

Code : JavaScript

```
Math.pow(3, 2); // Le premier paramètre est la base, le deuxième est
l'exposant.
// Ce calcul donne donc : 3 * 3 = 9
```

Concernant le calcul de la racine carrée d'un nombre, il existe aussi une méthode prévue pour cela, elle se nomme `sqrt()` (abréviation de "square root") :

Code : JavaScript

```
Math.sqrt(9); // Retourne : 3
```

Les cosinus et sinus

Quand on souhaite faire des calculs en rapport avec les angles, on a souvent recours aux fonctions Cosinus et Sinus. L'objet `Math` possède des méthodes qui remplissent exactement le même rôle : `cos()` et `sin()` (il existe bien entendu les méthodes `acos()` et `asin()`).

Leur utilisation est, encore une fois, très simple :

Code : JavaScript

```
Math.cos(Math.PI); // Retourne : 1
Math.sin(Math.PI); // Retourne : environ 0
```

Retrouver les valeurs maximum ou minimum

Voilà deux méthodes qui peuvent se révéler bien utiles : `max()` et `min()`, elles permettent respectivement de retrouver les valeurs maximum et minimum dans une liste de nombres, qu'importe si les nombres sont dans l'ordre croissant, décroissant ou aléatoire. Ces deux méthodes prennent autant de paramètres que de nombres à comparer :

Code : JavaScript

```
Math.max(42, 4, 38, 1337, 105); // Retourne : 1337
Math.min(42, 4, 38, 1337, 105); // Retourne : 4
```

Calculer un nombre aléatoire

Il est toujours intéressant de savoir comment calculer un nombre aléatoire pour chaque langage que l'on utilise, cela finit toujours par servir un jour où l'autre. En Javascript, la méthode qui s'occupe de ça est nommée `random()` (pour faire original). Cette fonction est utile mais n'est malheureusement pas très pratique à utiliser comparativement à celle présente, par exemple, en PHP.

En PHP, il est possible de définir entre quels nombres doit être choisi le nombre aléatoire. En Javascript, un nombre décimal aléatoire est choisi entre 0 (inclus) et 1 (exclus), ce qui nous oblige à faire de petits calculs par la suite pour obtenir un nombre entre une valeur minimum et maximum.

Venons-en à l'exemple :

Code : JavaScript

```
alert(Math.random()); // Retourne un nombre compris entre 0 et 1.
```

[Essayer !](#)

Là, notre script est un peu limité du coup, la solution est donc de créer notre propre fonction qui va gérer les nombres minimum (inclus) et maximum (exclus) :

Code : JavaScript

```
function rand(min, max, integer) {
    if(!integer) {
        return Math.random() * (max - min) + min;
    } else {
        return Math.floor(Math.random() * (max - min + 1) + min);
    }
}
```

Et voilà une fonction prête à être utilisée ! Le troisième paramètre sert à définir si l'on souhaite obtenir un nombre entier ou non.

[Essayer une adaptation de ce code !](#)

Cette fonction est assez simple en terme de réflexion : on prend le nombre minimum que l'on soustrait au maximum, on obtient alors l'intervalle de valeur qui n'a plus qu'à être multiplié au nombre aléatoire (qui est compris entre 0 et 1), le résultat obtenu sera alors compris entre 0 et la valeur de l'intervalle, il ne reste alors plus qu'à lui ajouter le nombre minimum pour obtenir une valeur comprise entre notre minimum et notre maximum !



Je pense qu'il est bon de vous informer : la méthode `random()` de l'objet `Math` ne renvoie pas vraiment un nombre aléatoire, ce n'est d'ailleurs pas réellement possible sur un ordinateur. Cette méthode est basée sur plusieurs algorithmes qui permettent de renvoyer un nombre "pseudo-aléatoire", mais le nombre n'est jamais vraiment aléatoire. À vrai dire, cela ne devrait pas vous affecter dans vos projets, mais il est toujours bon de le savoir.

Les inclassables

Bien que les objets `Number` et `Math` implémentent l'essentiel des méthodes de gestion des données numériques qui existent en Javascript, certaines fonctions globales existent et permettent de faire quelques conversions et contrôles de données un peu plus poussés.

Les fonctions de conversion

Si vous avez lu tous les chapitres précédents (ce que vous devriez avoir fait), vous avez normalement déjà vu ces deux fonctions, mais nous allons revoir leur utilisation dans le doute.

Ces deux fonctions se nomment `parseInt()` et `parseFloat()`, elles permettent de convertir une chaîne de caractères en un nombre. Ces deux fonctions possèdent chacune deux arguments :

- Le premier est obligatoire, il s'agit de la chaîne de caractère à convertir en nombre. Exemple : "303" donnera le nombre 303 en sortie.
- Le deuxième est facultatif mais il est très fortement conseillé de s'en servir. Il permet de spécifier la base que le nombre utilise dans la chaîne de caractères. Exemple : 10 pour spécifier la base *décimale*, 2 pour la base *binnaire*.

L'importance du deuxième argument est simple à démontrer avec un exemple :

Code : JavaScript

```
var myString = '08';

alert(parseInt(myString)); // Affiche "0".
alert(parseInt(myString, 10)); // Affiche "8".
```

Alors pourquoi cette différence de résultat ? La solution est simple : en l'absence d'un second argument, les fonctions `parseInt()` et `parseFloat()` vont tenter de deviner la base utilisée par le nombre écrit dans la chaîne de caractères. Ici, la chaîne de caractères commence par un zéro, ce qui est caractéristique de la base *octale*, on obtient donc 0 en retour. Afin d'éviter d'éventuelles conversions hasardeuses, il est toujours bon de spécifier la base que l'on souhaite évaluer.



Attention à une chose ! Les fonctions `parseInt()` et `parseFloat()` peuvent réussir à retrouver un nombre dans une chaîne de caractères, ainsi, la chaîne de caractères "303 pixels" renverra bien "303" après conversion. Cependant, cela ne fonctionne que si la chaîne de caractères commence par le nombre à retourner. Ainsi, la chaîne de caractères "Il y a 303 pixels" ne renverra que la valeur **NaN**. Pensez-y !

Les fonctions de contrôle

Vous souvenez-vous des valeurs **NaN** et **Infinite** ? Je vous avais parlé de deux fonctions permettant de vérifier la présence ces deux valeurs, les voici : `isNaN()` qui permet de savoir si notre variable contient un nombre ou non et `isFinite()` qui permet de déterminer si le nombre est fini ou non.

`isNaN()` renvoie **true** si la variable ne contient pas de nombre, elle s'utilise de cette manière :

Code : JavaScript

```
var myNumber = parseInt("test"); // Notre conversion sera un échec
et renverra "NaN".

alert(isNaN(myNumber)); // Affiche "true", cette variable ne
contient pas de nombre.
```

Quant à `isFinite()`, cette fonction renvoie **true** si le nombre ne tend pas vers l'infini :

Code : JavaScript

```
var myNumber = 1/0; // 1 divisé par 0 tend vers l'infini.  
  
alert(isFinite(myNumber)); // Affiche "false", ce nombre tend vers l'infini.
```



Mais pourquoi utiliser ces deux fonctions ? Je n'ai qu'à vérifier si ma variable contient la valeur **NaN** ou **Infinity**...

Admettons que vous fassiez cela ! Essayons le cas de **NaN** :

Code : JavaScript

```
var myVar = "test";  
  
if (myVar == NaN) {  
    alert('Cette variable ne contient pas de nombre.');} else {  
    alert('Cette variable contient un nombre.');}
```

Essayer !

Voyez-vous le problème ? Cette variable ne contient pas de nombre mais ce code croit pourtant que c'est le cas. Cela est dû au fait que l'on ne fait que tester la présence de la valeur **NaN**, cette valeur est présente uniquement si la tentative d'écriture d'un nombre a échoué (une conversion loupée par exemple), elle ne sera jamais présente si la variable était destinée à contenir autre chose qu'un nombre.

Un autre facteur important aussi, c'est que la valeur **NaN** n'est pas égale à elle-même ! Alors oui, ce n'est vraiment pas logique mais c'est comme ça :

Code : JavaScript

```
alert(NaN == NaN); // Affiche : false
```

Bref, la fonction `isNaN()` est utile car elle vérifie si votre variable était destinée à contenir un nombre et ensuite elle vérifie que ce nombre ne possède pas la valeur **NaN**.

Concernant `isFinite()`, un nombre peut tendre soit vers l'infini positif, soit vers l'infini négatif. Une condition de ce genre ne peut donc pas fonctionner :

Code : JavaScript

```
var myNumber = -1/0;  
  
if (myNumber == Number.POSITIVE_INFINITY) {
```

```
    alert("Ce nombre tend vers l'infini.");  
  } else {  
    alert("Ce nombre ne tend pas vers l'infini.");  
  }  
}
```

Essayer !

Ce code est faux, on ne fait que tester si notre nombre tend vers l'infini positif, alors que la fonction `isFinite()` se charge de tester aussi si le nombre tend vers l'infini négatif.

La gestion du temps

La gestion du temps est importante en Javascript ! Elle vous permet de temporiser vos codes et donc de créer, par exemple, des animations, des compteurs à rebours et bien d'autres choses qui nécessitent une temporisation dans un code.

En plus de cela, nous allons aussi apprendre à manipuler la date et l'heure.

Le système de datation

L'heure et la date sont gérés par un seul et même objet qui se nomme `Date`. Avant de l'étudier de fond en comble, nous allons d'abord comprendre comment fonctionne le système de datation en Javascript.

Introduction aux systèmes de datation

Nous, humains, lisons la date et l'heure de différentes manières, peu importe la façon dont cela est écrit, nous arrivons toujours à deviner de quelle date ou heure il s'agit. En revanche, un ordinateur, lui, possède une manière propre à son système pour lire ou écrire la date. Généralement, cette dernière est représentée sous un seul et même nombre qui est, par exemple : **1301412748510**. Vous avez ici l'heure et la date à laquelle j'ai écrit ces lignes sous forme d'un nombre assez conséquent.



Et ce nombre signifie quoi ?

Il s'agit en fait, en Javascript, du nombre de millisecondes écoulées depuis le 1er Janvier 1970 à minuit. Cette manière d'enregistrer la date est très fortement inspirée du [système d'horodatage utilisé par les systèmes Unix](#). La seule différence entre le système Unix et le système du Javascript, c'est que ce dernier stocke le nombre de millisecondes, tandis que le premier stocke le nombre de secondes. Dans les deux cas, ce nombre s'appelle un *timestamp*.

Au final, ce nombre ne nous sert vraiment qu'à peu de choses à nous, développeurs, car nous allons utiliser l'objet `Date` qui va s'occuper de faire tous les calculs nécessaires pour obtenir la date ou l'heure à partir de n'importe quel *timestamp*.

L'objet `Date`

L'objet `Date` nous fournit un grand nombre de méthodes pour lire ou écrire une date. Il y en a d'ailleurs tellement que nous n'allons en voir qu'une infime partie.

Le constructeur

Commençons par le constructeur ! Ce dernier prend en paramètre de nombreux arguments et s'utilise de différentes manières, voici les quatre manières d'utiliser notre constructeur :

Code : JavaScript

```
new Date();
new Date(timestamp);
new Date(dateString);
new Date(année, mois, jour [, heure, minutes, secondes,
millisecondes ]);
```

À chaque instantiation de l'objet `Date`, ce dernier enregistre soit la date actuelle si aucun paramètre n'est spécifié, soit une date que vous avez choisie. Les calculs effectués par les méthodes de notre objet instancié se feront à partir de la date enregistrée. Détaillons l'utilisation de notre constructeur :

- La première ligne instancie un objet `Date` dont la date est fixée à celle de l'instant même de l'instanciation.
- La deuxième ligne vous permet de spécifier le timestamp à utiliser pour notre instanciation.
- La troisième ligne prend en paramètre une chaîne de caractères qui doit être interprétable par la méthode `parse()`, nous y reviendrons.
- Enfin, la dernière ligne permet de spécifier manuellement chaque composant qui constitue une date, nous retrouvons donc en paramètres obligatoires : l'année, le mois et le jour. Les quatre derniers paramètres sont facultatifs (c'est pour cela que vous voyez des crochets, ils signifient que les paramètres sont facultatifs). Ils seront initialisés à 0 si ils ne sont pas spécifiés, nous y retrouvons : l'heure, les minutes, les secondes et les millisecondes.

Les méthodes statiques

L'objet `Date` possède deux méthodes statiques mais nous n'allons aborder l'utilisation que de la méthode `parse()`.

Vu le nom de cette méthode je pense que vous pouvez déjà plus ou moins deviner ce qu'elle permet de faire. Cette méthode prend en unique paramètre une chaîne de caractères représentant une date et renvoie le timestamp associé. Cependant, on ne peut pas écrire n'importe quelle chaîne de caractères, il faut respecter une certaine syntaxe qui se présente de la manière suivante :

Code : Autre

```
Sat, 04 May 1991 20:00:00 GMT+02:00
```

La date écrite ci-dessus représente donc le *Samedi 4 mai 1991 à 20h pile* avec un décalage de deux heures supplémentaires sur l'horloge de Greenwich.

Il existe plusieurs manières d'écrire la date, cependant je ne vous fournis que celle là car les dérives sont nombreuses. Si vous voulez connaître toutes les syntaxes existantes alors allez donc jeter un coup d'œil [au document qui traite de la standardisation de cette syntaxe](#) (la syntaxe est décrite après le chapitre 7).

Enfin, pour utiliser cette syntaxe avec notre méthode, rien de plus simple :

Code : JavaScript

```
var timestamp = Date.parse('Sat, 04 May 1991 20:00:00 GMT+02:00');  
alert(timestamp); // Affiche : 67338000000
```

Les méthodes des instances de l'objet Date

Étant donné que l'objet `Date` ne possède aucun attribut standard, nous allons directement nous intéresser à ses méthodes qui sont très nombreuses. Elles sont d'ailleurs tellement nombreuses (et similaires en terme d'utilisation) que je vais vous en lister seulement quelques-unes (les plus utiles bien sûr) et en expliquer le fonctionnement d'une manière générale.

Commençons tout de suite par huit méthodes très simples qui fonctionnent toute selon l'heure locale :

- `getFullYear()` : Renvoie l'année sur 4 chiffres.
- `getMonth()` : Renvoie le mois (0 à 11).
- `getDate()` : Renvoie le jour du mois (1 à 31).
- `getDay()` : Renvoie le jour de la semaine (1 à 7, la semaine commence le Lundi).
- `getHours()` : Renvoie l'heure (0 à 23).
- `getMinutes()` : Renvoie les minutes (0 à 59).
- `getSeconds()` : Renvoie les secondes (0 à 59).
- `getMilliseconds()` : Renvoie les millisecondes (0 à 999).



Les 8 méthodes listées ci-dessus possèdent chacune une fonction de type "set". Par exemple, avec la méthode `getDay()` il existe aussi, pour le même objet `Date`, la méthode `setDay()` qui permet de définir le jour en le passant en paramètre.

Chacune de ses méthodes s'utilise d'une manière enfantine, vous instanciez un objet `Date` et il ne vous reste plus qu'à appeler les méthodes souhaitées :

Code : JavaScript

```
var myDate = new Date('Sat, 04 May 1991 20:00:00 GMT+02:00');  
  
alert(myDate.getMonth()); // Affiche : 4  
alert(myDate.getDay()); // Affiche : 6
```

Deux autres méthodes pourront aussi vous être utiles :

- La méthode `getTime()` renvoie le timestamp de la date de votre objet.
- La méthode `setTime()` vous permet de modifier la date de votre objet en lui passant en unique paramètre un timestamp.

Mise en pratique : Calculer le temps d'exécution d'un script

Dans de nombreux langages de programmation, il peut parfois être intéressant de faire des tests sur la rapidité d'exécution de différents scripts. En Javascript, ces tests sont très simples à faire notamment grâce à la prise en charge native des millisecondes avec l'objet `Date` (les secondes sont rarement très intéressantes à connaître car elles sont généralement à 0 vu la rapidité de la majorité des scripts 🤔).

Admettons que vous ayez une fonction `slow()` que vous soupçonnez d'être assez lente, vous allez vouloir vérifier sa vitesse d'exécution. Pour cela, rien de bien compliqué !

Commençons tout d'abord par exécuter notre fonction :

Code : JavaScript

```
slow();
```

Comment allons-nous opérer maintenant ? Nous allons récupérer le *timestamp* avant l'exécution de la fonction puis après son exécution, il n'y aura ensuite plus qu'à soustraire le premier timestamp au deuxième et nous aurons notre temps d'exécution ! Allons-y :

Code : JavaScript

```
var firstTimestamp = new Date().getTime(); // On obtient le  
timestamp avant l'exécution.  
  
slow(); // La fonction travaille.  
  
var secondTimestamp = new Date().getTime(), // On récupère le  
timestamp après l'exécution.  
    result = secondTimestamp - firstTimestamp; // On fait la  
soustraction.  
  
alert("Le temps d'exécution est de : " + result + "  
millisecondes.");
```

Les fonctions temporelles

Nous avons vu comment travailler avec les dates et l'heure, mais malgré ça il nous est toujours impossible d'influencer le délai d'exécution de nos scripts pour permettre, par exemple, la création d'une animation. C'est là que les fonctions `setTimeout()` et `setInterval()` interviennent : la première permet de déclencher un code au bout d'un temps donné, tandis que la seconde va déclencher un code à un intervalle régulier que vous aurez spécifié.

Utiliser `setTimeout()` et `setInterval()`

Avec un simple appel de fonction

Ces deux fonctions ont exactement les mêmes paramètres : le premier est la fonction à exécuter, le deuxième est le temps en millisecondes.

Concernant le premier paramètre, il y a trois manières de le spécifier :

- En passant la fonction en référence :

Code : JavaScript

```
setTimeout(myFunction, 2000); // myFunction sera exécutée au bout de 2 secondes.
```

- En définissant une fonction anonyme :

Code : JavaScript

```
setTimeout(function() {  
  // Code...  
}, 2000);
```

- En utilisant une sale méthode que vous devez bannir de tous vos codes :

Code : JavaScript

```
setTimeout('myFunction()', 2000);
```

Pourquoi ne pas procéder de cette manière ? Tout simplement parce que cela appelle implicitement la fonction `eval()` qui va se charger d'analyser et exécuter votre chaîne de caractères. Pour de plus amples informations, [je vous redirige vers l'excellent tuto de nod_](#).

En ce qui concerne le deuxième paramètre, il n'y a pas grand chose à dire mis à part qu'il s'agit du temps à spécifier (en millisecondes) à votre fonction. Une bonne chose à savoir c'est que ce temps n'a que peu d'intérêt à être en-dessous des 10 millisecondes (Environ... Cela dépend des navigateurs !) pour la simple et bonne raison que la plupart des navigateurs n'arriveront pas à exécuter votre code avec un temps aussi petit. En clair, si vous spécifiez un temps de 5ms, votre code sera probablement exécuté au bout de 10ms.

Avec une fonction nécessitant des paramètres

Admettons que vous souhaitiez passer des paramètres à la fonction utilisée avec `setTimeout()` ou `setInterval()`,

comment allez vous faire ?

C'est bien simple, nos deux fonctions temporelles possèdent toutes les deux 2 paramètres, mais en vérité il est possible d'en attribuer autant que l'on veut. Les paramètres supplémentaires seront alors passés à la fonction appelée par notre fonction temporelle, exemple :

Code : JavaScript

```
setTimeout(myFunction, 2000, param1, param2);
```

Ainsi, au terme du temps passé en deuxième paramètre, notre fonction `myFunction()` sera appelée de la manière suivante :

Code : JavaScript

```
myFunction(param1, param2);
```

Cependant, cette technique ne fonctionne pas sur les vieilles versions d'Internet Explorer, il nous faut donc ruser :

Code : JavaScript

```
setTimeout(function() {  
    myFunction(param1, param2);  
}, 2000);
```

Ci-dessus, nous avons créé une fonction anonyme qui va se charger d'appeler la fonction finale avec les bons paramètres, et ceci fonctionne sur tous les navigateurs !

Annuler une action temporelle

Il se peut que vous ayez parfois besoin d'annuler une action temporelle. Par exemple, vous avez utilisé la fonction `setTimeout()` pour qu'elle déclenche une alerte si l'utilisateur n'a pas cliqué sur une image dans les 10 secondes qui suivent, si l'utilisateur clique sur l'image il va alors vous falloir annuler votre action temporelle avant son déclenchement, c'est là qu'entrent en jeu les fonctions `clearTimeout()` et `clearInterval()`. Comme vous pouvez vous en douter, la première s'utilise pour la fonction `setTimeout()` et la deuxième pour `setInterval()`.

Ces deux fonctions prennent toutes les deux un seul argument : l'identifiant de l'action temporelle à annuler. Cet identifiant (qui est un simple nombre entier) est retourné par les fonctions `setTimeout()` et `setInterval()`.

Voici un exemple logique :

Code : HTML

```
<button id="myButton">Annuler le compte à rebours</button>  
  
<script type="text/javascript">  
    (function() {  
        var button = document.getElementById('myButton');  
        var timerID = setTimeout(function() { // On crée notre compte à
```

```

rebours
    alert("Vous n'êtes pas très réactif vous !");
    }, 5000);

    button.onclick = function() {
        clearTimeout(timerID); // Le compte à rebours est annulé
        alert("Le compte à rebours a bien été annulé."); // Et on
        prévient l'utilisateur
    };

    }) ();
</script>

```

Essayer !

On peut même aller un peu plus loin en gérant plusieurs actions temporelles à la fois :

Code : HTML

```

<button id="myButton">Annuler le compte à rebours (5s)</button>

<script type="text/javascript">
    (function() {

        var button = document.getElementById('myButton'),
            timeLeft = 5;

        var timerID = setTimeout(function() { // On crée notre compte à
            rebours
                clearInterval(intervalID);
                button.innerHTML = "Annuler le compte à rebours (0s)";
                alert("Vous n'êtes pas très réactif vous !");
            }, 5000);

        var intervalID = setInterval(function() { // On met en place
            l'intervalle pour afficher la progression du temps
                button.innerHTML = "Annuler le compte à rebours (" + -
            -timeLeft + "s)";
            }, 1000);

        button.onclick = function() {
            clearTimeout(timerID); // On annule le compte à rebours
            clearInterval(intervalID); // Et l'intervalle
            alert("Le compte à rebours a bien été annulé.");
        };

    }) ();
</script>

```

Essayer !

Mise en pratique : Les animations !

Venons-en maintenant à une utilisation concrète et courante des actions temporelles : les animations ! Tout d'abord, qu'est-ce qu'une animation ? C'est la modification progressive de l'état d'un objet. Ainsi, une animation peut très bien être la modification de la transparence d'une image à partir du moment où c'est fait d'une manière progressive et non pas instantanée.

Concrètement, comment peut-on créer une animation ? Reprenons l'exemple de la transparence : on veut que notre image passe d'une opacité de **1** à **0.2**.

Code : HTML

```


<script type="text/javascript">
  var myImg = document.getElementById('myImg');

  myImg.style.opacity = 0.2;
</script>
```

Le problème ici, c'est que notre opacité a été modifiée immédiatement de 1 à 0.2. Nous, nous voulons que ce soit progressif, il faudrait donc faire comme ceci :

Code : JavaScript

```
var myImg = document.getElementById('myImg');

for(var i = 0.9 ; i >= 0.2 ; i -= 0.1) {
  myImg.style.opacity = i;
}
```

Mais encore une fois, tout s'est passé en une fraction de seconde ! C'est là que les actions temporelles vont entrer en action et ceci afin de temporiser notre code et de lui laisser le temps d'afficher la progression à l'utilisateur ! Dans notre cas, nous allons utiliser la fonction `setInterval()` :

Code : JavaScript

```
var myImg = document.getElementById('myImg');

var intervalID = setInterval(function() {

  if ( (myImg.style.opacity -= 0.1) <= 0.2 ) {
    clearInterval(intervalID);
  }

}, 50);
```

Essayer !

Et voilà, cela fonctionne sans problème et ce n'est pas si compliqué qu'on ne le pense au premier abord ! Alors après il est possible d'aller bien plus loin en combinant les animations mais, maintenant que vous avez les bases, vous devriez être capables de faire toutes les animations dont vous rêvez !

Partie 4 : Annexe

Cette partie est ce que nous considérons comme étant une "partie facultative", vous y trouverez divers chapitres qui pourront vous être utiles au cours de la lecture du cours, libre à vous de les lire ou non. Sachez juste que cette partie peut être une mine d'or pour certains d'entre vous qui souhaitez pousser vos connaissances en Javascript au maximum.

Déboguer votre code

Créer des scripts paraît facile au premier abord, mais on finit toujours par tomber sur le même problème : notre code ne fonctionne pas ! On peut alors dire qu'il y a un bug, en clair il y a une erreur dans le code qui fait qu'il s'exécute mal ou ne s'exécute tout simplement pas.

Dans ce chapitre annexe, nous allons donc étudier quels sont les différents bugs que l'on peut généralement rencontrer en Javascript et surtout comment les résoudre. Pour cela, nous allons avoir besoin de différents outils que vous allez découvrir tout au long de votre lecture.

Le débogage : qu'est-ce que c'est ?

Les bugs

Avant de parler de débogage, intéressons-nous d'abord aux bugs. Ces derniers sont des erreurs humaines que vous avez laissées dans votre code, ils ne sont jamais le fruit du hasard. N'essayez pas de vous dire "je n'en ferai pas", ce n'est pas possible de rester concentré au point de ne jamais vous tromper sur plusieurs centaines de lignes de code !

Il existe deux types principaux de bugs : ceux que l'interpréteur Javascript saura vous signaler car ce sont des fautes de syntaxe, et ceux que l'interpréteur ne verra pas car ce sont des erreurs dans votre algorithme et non pas dans votre syntaxe.

Pour faire simple, voici un bug syntaxique :

Code : JavaScript

```
va myVar = 'test; // Le mot-clé "var" est mal orthographié et il
manque une apostrophe.
```

Et maintenant un bug algorithmique :

Code : JavaScript

```
// On veut afficher la valeur 6 avec les chiffres 3 et 2.
var myVar = 3 + 2;
// Mais on obtient 5 au lieu de 6 car on a fait une addition au lieu
d'une multiplication.
```

Vous voyez la différence ? Du point de vue de l'interpréteur Javascript, le premier code est faux car il est incapable de l'exécuter, or le deuxième code est exécutable mais on n'obtient pas la valeur escomptée.

Il faut bien se mettre en tête que l'interpréteur Javascript se fiche bien des valeurs retournées par votre code, lui il veut exécuter le code et c'est tout. Voici la différence entre le caractère syntaxique et algorithmique d'une erreur : la première empêche le code de s'exécuter, la seconde empêche quant-à elle le bon déroulement du script. Pourtant, les deux empêchent votre code de s'exécuter correctement.

Le débogage

Comme son nom l'indique, cette technique consiste à supprimer les bugs qui existent dans votre code. Pour chaque type de bug vous avez plusieurs solutions bien particulières.

Les bugs syntaxiques sont les plus simples à résoudre car l'interpréteur Javascript vous signalera généralement l'endroit où l'erreur est apparue dans la console de votre navigateur. Vous verrez comment vous servir de ces consoles un peu plus bas.

En ce qui concerne les bugs algorithmiques, là il va falloir faire travailler votre cerveau et chercher par vous-même où vous avez bien pu vous tromper. La méthode la plus simple consiste à remonter les couches de votre code pour trouver à quel endroit se produit l'erreur.

Par exemple, si vous avez un calcul qui affiche la mauvaise valeur, vous allez immédiatement aller vérifier ce calcul. Si ce calcul n'est pas en cause mais qu'il fait appel à des variables alors vous allez vérifier la valeur de chacune de ces variables, etc... De fil en aiguille vous allez parvenir à déboguer votre code.



Pour vérifier les valeurs de vos variables, calculs, etc... Pensez bien à utiliser la fonction **alert** !

Les consoles d'erreurs

Comme vous l'avez constaté ci-dessus, un script peut contenir des erreurs. Quand c'est le cas, l'interpréteur vous le fait savoir en vous affichant un message d'erreur qui s'affiche dans ce que l'on appelle la *console d'erreurs*.

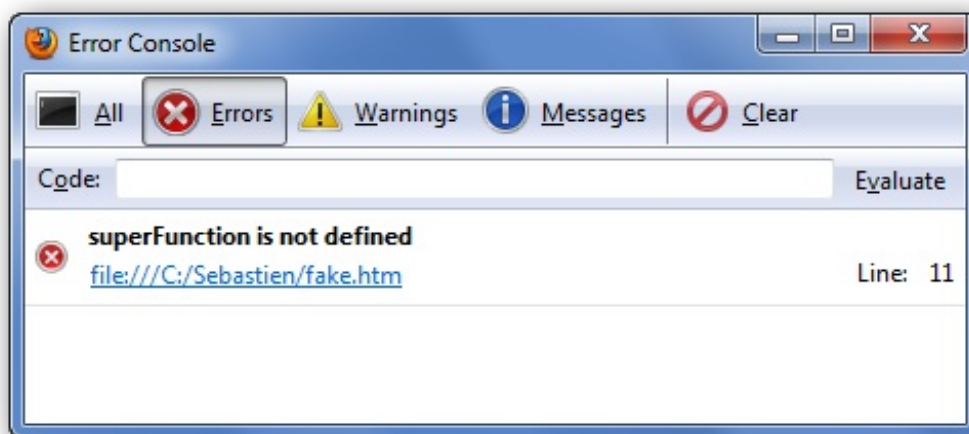
Chaque navigateur possède sa propre console d'erreurs, nous allons donc faire le tour des différents navigateurs pour voir où se trouvent chacune de ces fameuses consoles. Par la suite, si votre script ne fonctionne pas et que vous vous demandez pourquoi, ayez le bon réflexe : allez consulter la console d'erreurs !



Pour les captures d'écran ci-dessous, j'ai provoqué moi-même une erreur Javascript afin de vous montrer un exemple.

Mozilla Firefox


Allez dans le menu **Outils** puis cliquez sur **Console d'erreurs** et voici ce que vous obtenez :

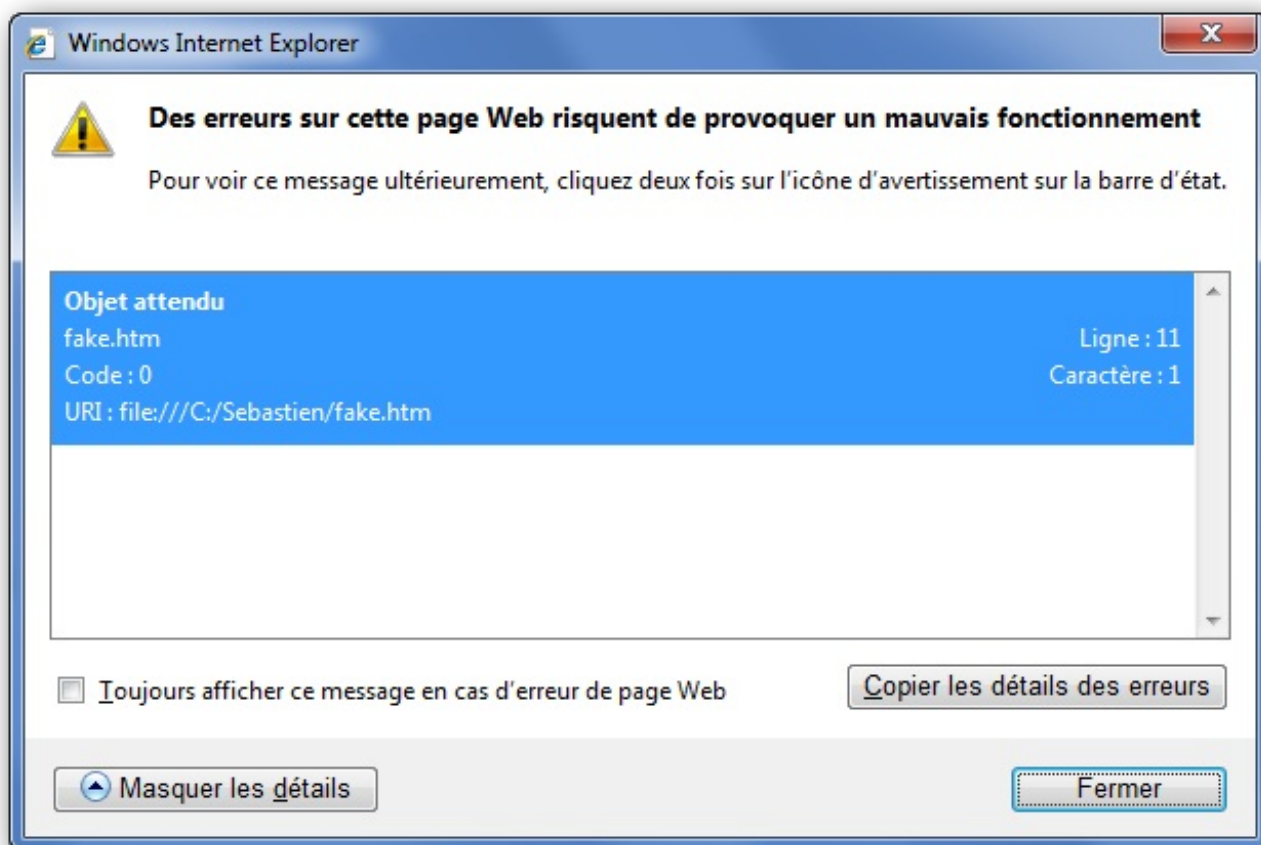


On lit ici qu'une erreur "superFunction is not defined" est apparue à la ligne 11, dans le fichier C : \Sebastien\fake .htm.

L'erreur décrite signifie qu'on a voulu exécuter la fonction `superFunction` alors qu'elle n'existe pas.


Internet Explorer

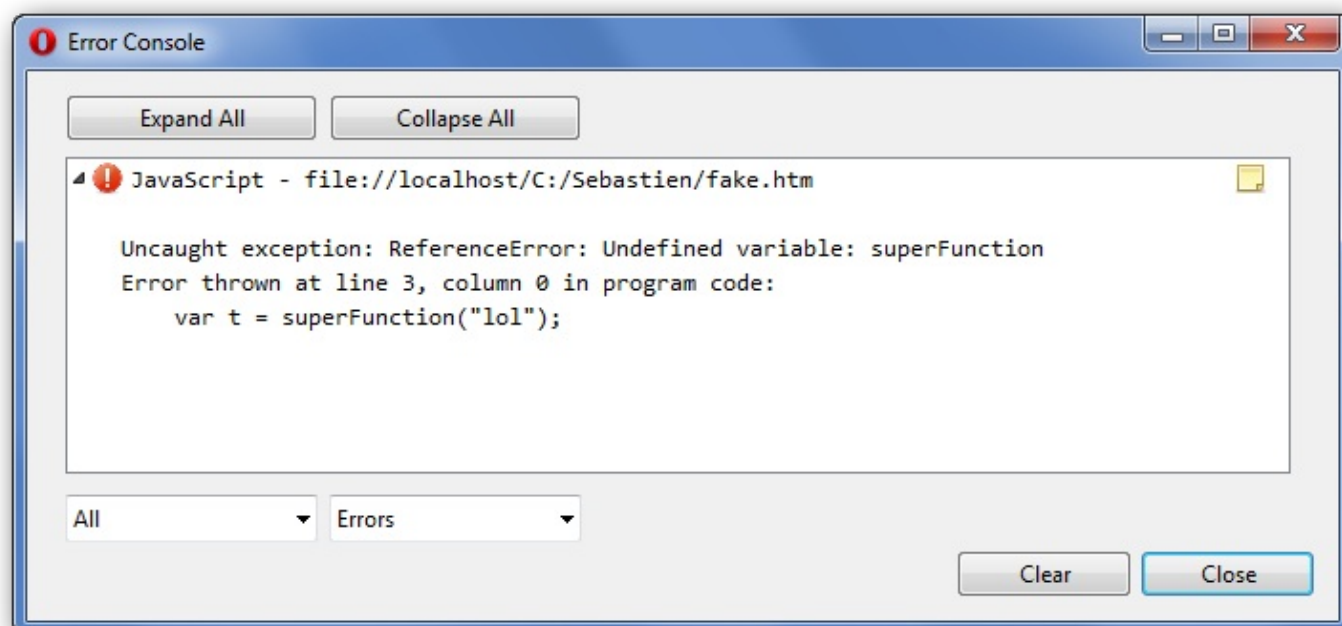
Si une erreur survient, Internet Explorer le signale à gauche de la barre d'état, via ce symbole :  Terminé. Il vous suffit de double-cliquer dessus pour afficher le détail des erreurs survenues :



Ici, on remarque que l'erreur est donnée en français, ce qui ne la rend pas spécialement plus simple à comprendre. Généralement, les erreurs affichées par Internet Explorer restent assez obscures.


Opera

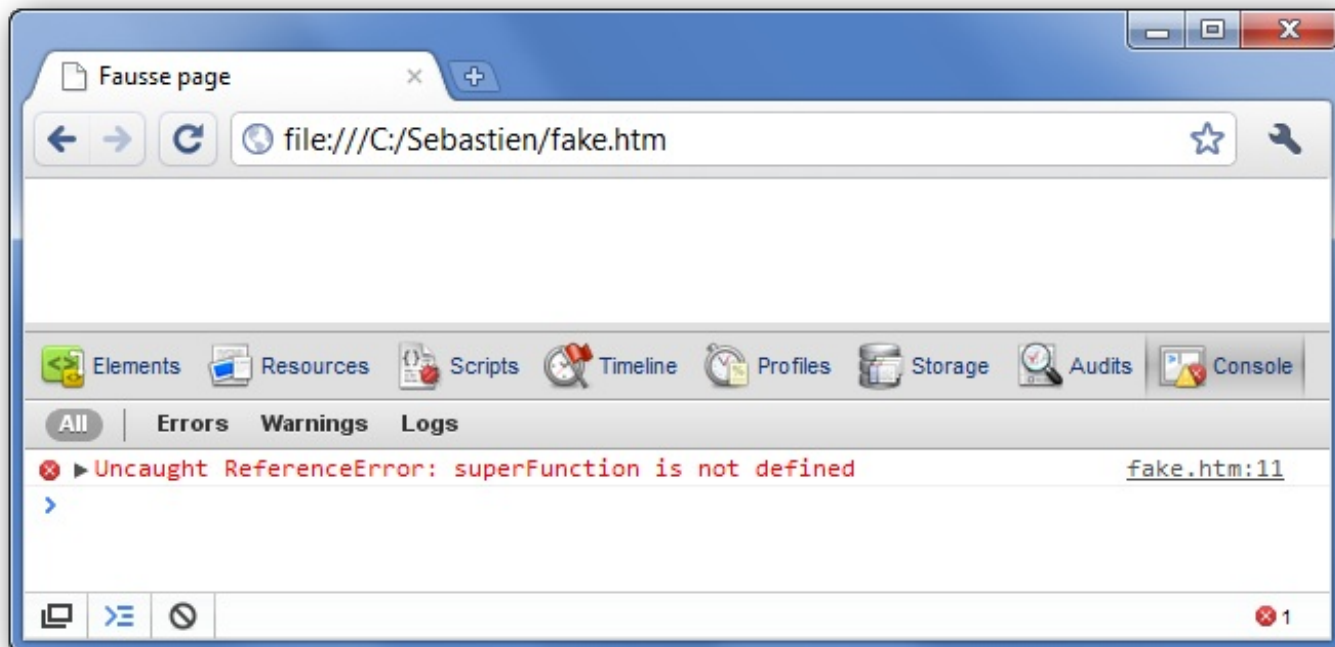
Cliquez sur le menu  puis sur **Page**, sur **Outils de développeur** et enfin sur **Console d'erreur** :



Ici le descriptif est assez conséquent et la portion de code incriminée est affichée.



Google Chrome

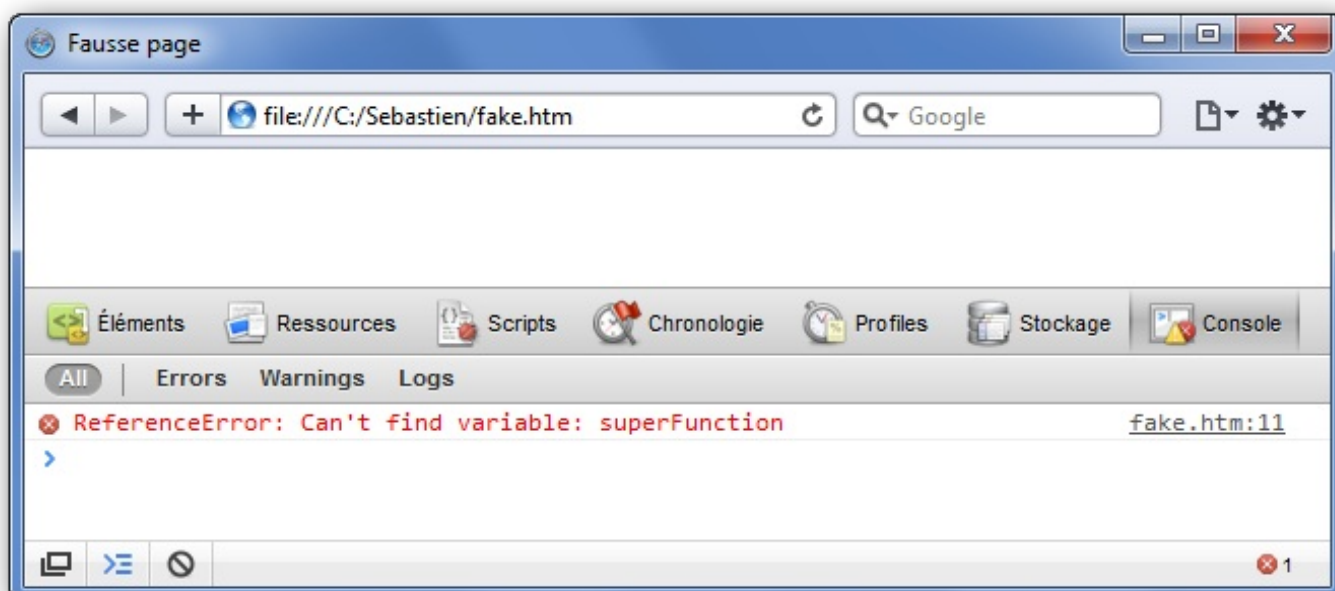
Cliquez sur l'icône  (Outils) pour aller dans le menu **Tools** puis cliquez sur **Console JavaScript**. Cliquez aussi sur l'onglet **Console** pour n'afficher que la console Javascript.



Ici, l'erreur affichée est semblable à celle de Firefox, excepté que le type de l'erreur est mentionné : `ReferenceError` .

Safari

Dans Safari, il faut dans un premier temps activer le menu **Développement**. Pour ce faire, cliquez sur  puis sur **Préférences**. Dans l'onglet **Avancées**, cochez la case **Afficher le menu Développement dans la barre des menus**. Quand c'est fait, pour ouvrir la console, cliquez sur  puis sur **Développement**, et enfin sur **Afficher la console des erreurs** :



La console de Safari est la même que celle de Google Chrome.

Les bugs les plus courants

Bien que les consoles d'erreurs soient très pratiques, certaines erreurs que vous ferez seront courantes et peuvent être résolues sans même vraiment rechercher d'où vient le problème. Voici quelques exemples d'erreurs (algorithmique ou syntaxique) que vous pourriez faire assez fréquemment :

Noms de variables et de fonctions mal orthographiés

Voici probablement l'erreur la plus courante, vous allez souvent vous tromper dans l'orthographe de vos variables ou de vos fonctions, ce qui devrait générer une erreur de type **undefined**, autrement dit votre fonction ou votre variable n'existe pas, ce qui est normal si vous avez tapé le mauvais nom. À noter que la confusion entre majuscules et minuscules est fréquente elle aussi, la casse est très importante !

Confusion entre les différents opérateurs

Il est fréquent de se tromper dans les différents types d'opérateur qui existent, surtout dans les conditions. Voici les deux cas les plus courants :

- Écrire **if** (a = b) au lieu de **if** (a == b) . Ces deux codes s'exécutent mais n'ont pas la même signification. Le premier est une affectation, le deuxième est une comparaison.
- Confondre les opérateurs binaires **if** (a & b | c) avec les opérateurs de comparaison **if** (a && b || c) . Si vous souhaitez faire des comparaisons logiques, pensez bien à doubler les caractères, sinon vous utiliserez les opérateurs binaires à la place.

Mauvaise syntaxe pour les tableaux et les objets

Créer un tableau ou un objet n'est pas bien compliqué mais il est aussi facile de faire quelques petites erreurs, voici les plus fréquentes :

- Laisser une virgule en trop à la fin de la déclaration d'un objet ou d'un tableau : **var** myArray = [1, 2, 3,];
Oui, chaque item doit être séparé des autres par une virgule mais le dernier d'entre eux ne doit pas en être suivi.
- Écrire **var** object = { a = 1 }; au lieu de **var** object = { a : 1 }; . Il faut toujours utiliser les deux points (:) pour assigner une valeur à la propriété d'un objet.

Créer une boucle infinie

Une boucle infinie ne prendra jamais fin et donc se répétera inlassablement. Ceci est un bug, et non pas une fonctionnalité comme dans d'autres langages tel que le C. Prenez donc garde à ne pas mettre une condition qui renvoie toujours une valeur équivalente à **true** comme ci-dessous :

Code : JavaScript

```
var nb1 = 4, nb2 = 5;

while (nb1 < nb2) {
  // Etc...
}
```

Dans le cas où vous auriez créé une boucle infinie sans le vouloir, le navigateur n'exécutera probablement pas votre code et retournera une erreur.

Exécuter une fonction au lieu de la passer en référence à une variable

Cette erreur est très courante notamment quand il s'agit d'attribuer une fonction à un évènement. Normalement, passer une fonction en référence à une variable consiste à faire ceci :

Code : JavaScript

```
function function1() {  
    // Code...  
}  
  
var function2 = function1; // On passe en référence la fonction  
"function1" à la variable "function2".  
  
function2(); // En tentant d'exécuter "function2" ce sera  
"function1" qui sera exécutée à la place.
```

Or, vous êtes nombreux à vous tromper et à écrire ceci lors du passage en référence :

Code : JavaScript

```
var function2 = function1();
```

Ce code est faux car vous ne passez pas à **function2** la référence vers **function1**, vous lui passez le retour de cette dernière puisque vous l'exécutez.



Dans ce paragraphe, j'ai employé plusieurs fois le terme "référence", si vous ne savez pas exactement ce que c'est, il s'agit en fait d'un "lien" vers une variable/fonction/etc. Donc, quand on dit que l'on passe la référence d'une fonction à une variable, on lui passe en fait un "lien" vers notre fonction. La fonction n'est pas copiée dans la variable, il y a juste un lien de créé entre la variable et la fonction.

Les kits de développement

En plus des consoles d'erreurs, certains navigateurs intègrent de base un kit de développement Web. Ces kits vous permettent de déboguer bien plus efficacement des erreurs complexes.

Ces kits possèdent plusieurs outils. Pour la plupart, ils sont constitués :

- d'une console d'erreur.
- d'un éditeur HTML dynamique qui vous permet de visualiser et éditer le code source de la page, qu'il ait été modifié par du Javascript ou non.
- d'un éditeur CSS, vous permettant de désactiver/activer des styles à la volée ainsi que d'en ajouter ou en supprimer.
- de différents systèmes d'analyse des performances client et réseau.

Chaque navigateur dans sa dernière version possède un kit de développement pré-intégré, la seule exception reste Firefox qui nécessite un module complémentaire nommé Firebug que vous pourrez trouver [sur le site officiel](#).



Voici un aperçu de ce que donne le panneau de Firebug.

Concernant Internet Explorer, il existe un kit de développement uniquement depuis la version 8, pas avant. Pour l'afficher appuyez sur F12, ce raccourci clavier est aussi valable pour Firebug sous Firefox.


Concrètement, à quoi peuvent réellement vous servir ces kits ? Il existe deux utilisations principales :

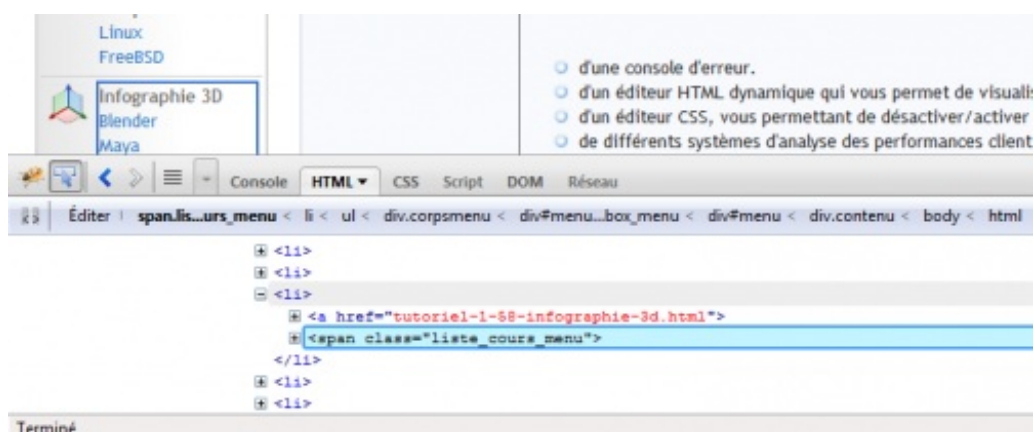
- Afficher le code HTML dynamique, afin de voir comment votre code Javascript a modifié le DOM et surtout si il a été modifié correctement !
- Mettre des points d'arrêts dans votre code Javascript pour vérifier l'état de votre code à un moment donné.



Ci-dessous je vous décris comment vous servir de ces deux outils mais uniquement sous Firebug vu que c'est le plus utilisé et surtout qu'il possède une version lite compatible sur un grand nombre de navigateurs. Si vous utilisez le kit d'un autre navigateur, ne vous inquiétez pas, le fonctionnement devrait être sensiblement le même.

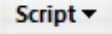
Éditer le code HTML dynamique

Pour afficher le code HTML dynamique, cliquez sur  (raccourci clavier : Ctrl+Maj+C) et déplacez votre curseur sur la page web, chaque élément survolé sera encadré de bleu, si vous cliquez sur l'un d'eux vous verrez alors le code le concernant s'afficher comme ceci :



À partir de là, vous pouvez choisir de développer les éléments pour voir ce qu'ils contiennent, vous pouvez aussi les modifier, etc...

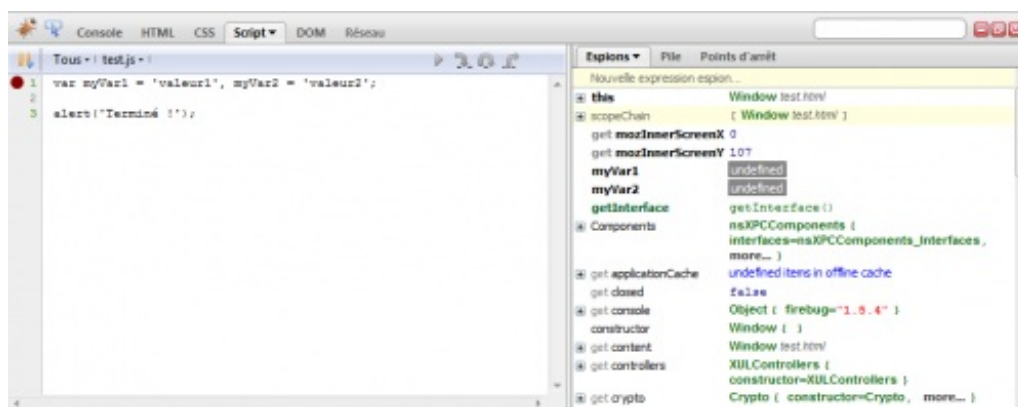
Utiliser les points d'arrêts


Concernant les points d'arrêts, activez tout d'abord le panneau de script de Firebug en cliquant sur la petite flèche à droite de l'onglet "Script"  puis en cliquant sur "Activé".




Concrètement, à quoi servent les points d'arrêts ?

À faire une pause à une ligne d'un code Javascript et afficher l'état complet des variables, méthodes, objets, etc... La quantité d'informations disponible est tout simplement immense, vous pouvez les retrouver dans le cadre de droite de Firebug dès qu'un point d'arrêt s'est déclenché :




Voyons la structure de ce panneau de débogage : à gauche vous avez le code Javascript avec un bouton  qui spécifie le fichier contenant du Javascript qui est en cours de visualisation, à droite vous retrouvez toutes les informations obtenues dès qu'un point d'arrêt se déclenche.

Pour ajouter un nouveau point d'arrêt, il vous suffit de cliquer à gauche du numéro de la ligne de code à laquelle vous souhaitez attribuer un point d'arrêt, vous verrez alors un point rouge s'afficher comme ceci :

 1 `var myVar1 = 'valeur1', myVar2 = 'valeur2';`



Notez bien que lorsque vous attribuez un point d'arrêt à une ligne de code, ce point d'arrêt mettra le code en pause avant l'exécution de la ligne en question.

Après avoir mis vos points d'arrêts, il ne vous reste plus qu'à recharger la page avec **F5** et votre code sera mis en pause à chaque point d'arrêt rencontré. Firebug listera alors, dans le cadre de droite, l'état de toutes les propriétés de votre script. Pour passer au point d'arrêt suivant, vous n'aurez qu'à cliquer sur continuer  jusqu'à ce que le code se termine.

Ce tutoriel est encore loin d'être terminé, de nombreux chapitres sont encore prévus pour vous apprendre de nombreuses choses !

Au passage, il semblerait que beaucoup d'entre vous souhaitent faire des animations, donc, afin que vous soyez au courant, sachez que le chapitre concernant cela sera disponible dans la 3ème partie de ce tutoriel.

Enfin, pour terminer, place aux remerciements :

- à [Ceriko](#) pour ses corrections stylistiques et ses feedbacks ;

- à [restimel](#), qui relit chacun des chapitres de ce cours et corrige chaque erreur qu'il trouve ;
- au groupe [Javascript²](#), qui nous soutient depuis le début dans ce projet ;
- à [ZoZor](#), pour tout.

Sur ce, bon apprentissage !

MCours.com