Partie 1 : Les bases du Javascript

Comme tout langage de programmation, le Javascript possède quelques particularités : sa syntaxe, son modèle d'objet, etc. En clair, tout ce qui permet de différencier un langage d'un autre. D'ailleurs, vous découvrirez rapidement que le Javascript est un langage relativement spécial dans sa manière d'aborder les choses. 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 d'entrer directement dans le vif du sujet, ce chapitre va vous apprendre ce qu'est le Javascript, ce qu'il permet de faire, quand il peut ou doit être utilisé et comment il a évolué depuis sa création en 1995.

Nous aborderons aussi plusieurs notions de bases telles que les définitions exactes de certains termes.

Qu'est-ce que le Javascript ?
Qu'est-ce que c'est ?

Citation : Définition

> Le Javascript est un langage de programmation de scripts orienté objet.

Dans cette description 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, appelées **instructions**, qui vont permettre de donner des ordres à l'ordinateur afin 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

Le 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 trois 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 précompilé** : 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 précompilés.
- **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. Et quand on dit que le Javascript est un langage de scripts, 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*), celui de Mozilla Firefox se nomme *SpiderMonkey* et 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 à la partie 2. 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 des images, des dates, des chaînes de caractères… mais il est également possible de créer soi-même des objets pour se 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. 

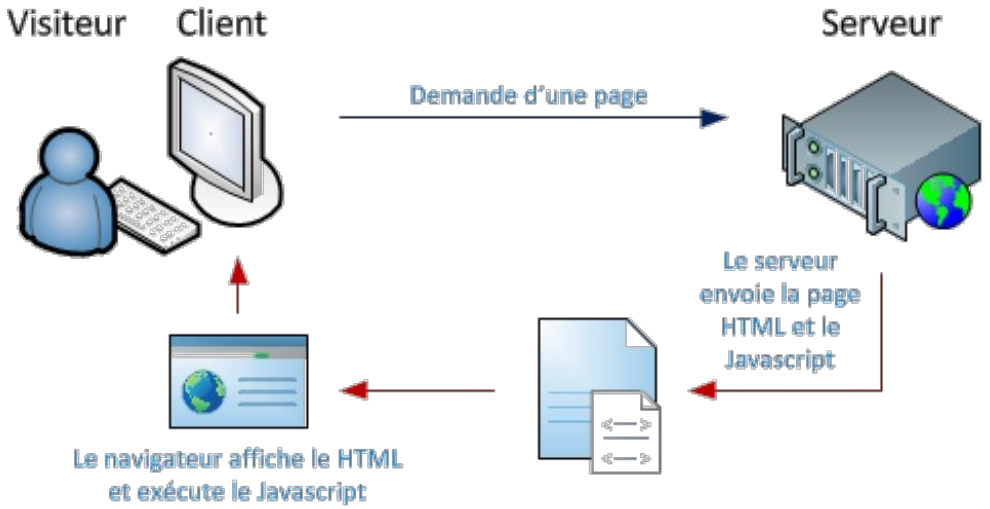
Le Javascript, le langage de scripts

Le Javascript est à ce jour utilisé majoritairement sur Internet, conjointement avec les 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.

Le 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 comme le 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 puis exécuter les scripts *client-side* tel que le Javascript. Voici un schéma reprenant ce fonctionnement :

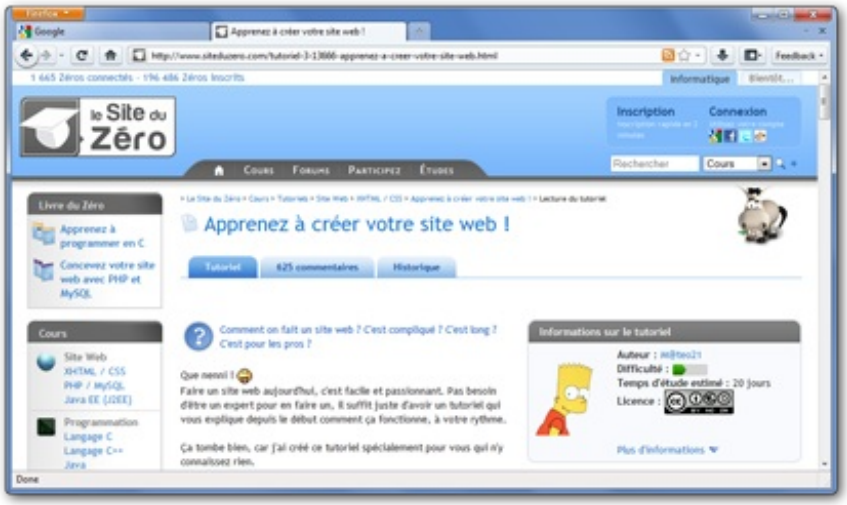


Javascript est un langage dit client-side, c'est à dire interprété par le client (le visiteur)

Le Javascript, pas que le Web

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

Le 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 le Javascript qui est utilisé pour animer l'interface. D'autres logiciels reposent également sur cette technologie, comme TomTom HOME qui sert à gérer votre GPS TomTom via votre PC.



Le navigateur Firefox 4

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 le 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 du LiveScript, qui sera renommée JavaScript en hommage au langage Java créé par la société Sun Microsystems. En effet, à cette époque, le langage Java était de plus en plus populaire, et

Brendan Eich, le papa de Javascript

appeler le LiveScript JavaScript était une manière de faire de la publicité, et au Java, et au JavaScript lui-même. Mais attention, au final, **ces deux langages sont radicalement différents** ! N'allez pas confondre le Java et le 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.

Le Javascript sort en décembre 1995 et est embarqué dans le navigateur Netscape 2. Le langage est alors un succès, si 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 (*European Computer Manufacturers Association* à l'époque, aujourd'hui *European association for standardizing information and communication systems*) 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 International 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 3, parue en décembre 1999.

L'ECMAScript et ses dérivés

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

- **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 du Javascript

Les versions du Javascript sont basées sur celles de l'ECMAScript (que nous abrégerons ES). Ainsi, il existe :

- ES 1 et ES 2, qui sont les prémices du langage Javascript ;
- ES 3 (sorti en décembre 1999), qui est fonctionnel sur tous les navigateurs (sauf les vieilles versions d'Internet Explorer) ;
- ES 4, qui a été abandonné en raison de modifications trop importantes qui ne furent pas appréciées ;
- ES 5 (sorti en décembre 2009), qui est la version la plus récemment sortie ;
- ES 6, qui est actuellement en cours de conception.

Ce cours portera sur l'ensemble des versions sorties à ce jour.

Un logo inconnu

Il n'y a pas de logo officiel pour représenter le Javascript. Cependant, le logo suivant est de plus en plus utilisé par la communauté, surtout depuis sa présentation à la JSConf EU de 2011. Vous pourrez le trouver à cette adresse sous différents formats, n'hésitez pas à en abuser en cas de besoin.

Ce logo non-officiel est de plus en plus utilisé

En résumé

- Le Javascript est un langage de programmation interprété, c'est-à-dire qu'il a besoin d'un interpréteur pour pouvoir être exécuté.
- Le Javascript est utilisé majoritairement au sein des pages Web.
- Tout comme le HTML, le Javascript est exécuté par le navigateur de l'internaute : on parle d'un comportement *client-side*, par opposition au *server-side* lorsque le code est exécuté par le serveur.
- Le Javascript est standardisé par l'ECMA International sous le nom d'ECMAScript qui constitue la référence du langage. D'autres langages découlent de l'ECMAScript, comme ActionScript, EX4 ou encore JScript.NET.
- La dernière version standardisée du Javascript est basée sur l'ECMAScript 5, sorti en 2009.

Premiers pas en Javascript

Comme indiqué précédemment, 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 durant 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 l'indémodable Notepad++ pour Windows, l'éternel Vim pour Linux et le performant TextWrangler pour Mac.

Afficher une boîte de dialogue

Le Hello World!

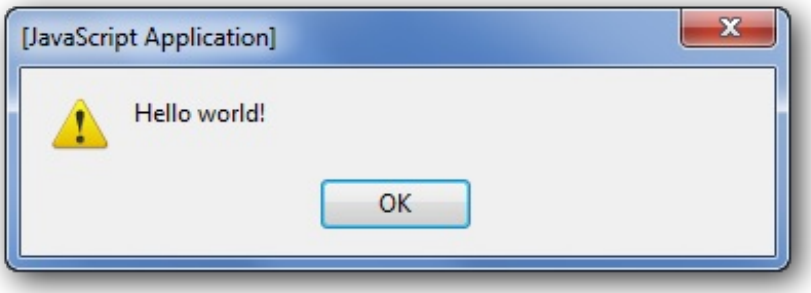
Ne dérogeons pas à la règle traditionnelle qui veut que tous les tutoriels de programmation commencent par afficher le texte « Hello World! » (« Bonjour le monde ! » en français) à l'utilisateur. Voici un code HTML simple contenant une instruction (nous allons y revenir) Javascript, placée au sein d'un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

Code : HTML - Hello World!

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

  <body>

    <script>

      alert('Hello world!');

    </script>

  </body>
</html>
```

Essayer !

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



Une boîte de dialogue s'ouvre, vous présentant le texte Hello World!

Vous remarquerez que nous vous avons 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.
Bref, nous, les auteurs, avons décidé de vous fournir des liens d'exemples 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* affiché dans les commentaires du code.

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 le 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 cet exemple, il n'y a qu'une instruction : l'appel de la fonction `alert()`.

La boîte de dialogue `alert()`

`alert()` est une instruction simple, appelée **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, retenez que l'instruction `alert()` sert juste à afficher une boîte de dialogue.

La syntaxe du Javascript

Les instructions

La syntaxe du Javascript n'est pas compliquée. 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 dans notre exemple. En revanche, si vous écrivez plusieurs instructions sur une même ligne, comme dans l'exemple suivant, 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

Le 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 instructions sont hiérarchisées en plusieurs niveaux et on utilise des espaces ou des tabulations pour les décaler 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';
    }
}
```

Ce code est indenté de quatre espaces, c'est-à-dire que le décalage est chaque fois un multiple de quatre. Un décalage de quatre espaces est courant, tout comme un décalage de deux. 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, nous n'aimons pas les indentations de deux, nous préférons celles de quatre).

Voici le même code, 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 rédigiez 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 précédemment, vous pouvez voir qu'il y a des espaces un peu partout pour aérer le code et qu'il y a une seule instruction par ligne (à l'exception des **if else**, mais nous verrons cela plus tard). 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 nous 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, bonne chance 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 slashs :

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 commentaires permet les retours à la ligne. Un commentaire multiligne commence par /* et se termine par */ :

Code : JavaScript

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

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

Code : JavaScript

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

Les fonctions

Dans l'exemple du `Hello world!`, nous avons utilisé 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 HTML5, il ne l'est pas. C'est pourquoi les exemples de ce cours, en HTML5, ne comporteront pas cet attribut.

Si vous n'utilisez pas le HTML5, sachez que 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, avec `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>

      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 HTML5, les commentaires d'encadrement sont inutiles.

Les commentaires d'encadrement servent à isoler le code Javascript pour que le validateur du W3C (*World Wide Web Consortium*) 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>
<!--

    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 src="hello.js"></script>

  </body>
</html>
```

On suppose ici 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, il est conseillé de placer les éléments **<script>** 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>
            // Un peu de code Javascript...
        </script>

        <script 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 vaut mieux 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 une documentation, voici celle que nous vous conseillons :

Mozilla Developer Network

Ce site Web est une documentation ; dans le jargon informatique il s'agit d'un espace 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 ce genre de 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 nous vous conseillons alors l'utilisation du site suivant :

jsFiddle

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 tester puis vous le collez dans la section `Javascript` en bas à gauche de la page. Une fois que vous avez copié le texte, il ne vous reste plus qu'à cliquer sur le bouton `Run` en haut à gauche et votre code sera exécuté immédiatement dans la section `Result` en bas à droite. Essayez donc avec ce code pour voir :

Code : JavaScript

```
alert('Bien, vous savez maintenant utiliser le site jsFiddle !');
```

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

En résumé

- Les instructions doivent êtres séparées par un point-virgule.
- Un code Javascript bien présenté est plus lisible et plus facilement modifiable.
- Il est possible d'inclure des commentaires au moyen des caractères //, /* et */.
- Les codes Javascript se placent dans un élément **<script>**.
- Il est possible d'inclure un fichier Javascript grâce à l'attribut `src` de l'élément **<script>**.

- Questionnaire récapitulatif

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 sur votre ordinateur 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 pas le droit d'en faire usage en tant que noms de variables. Vous trouverez sur cette page (en anglais) tous les mots réservés par le Javascript.

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

Code : JavaScript

```
var myVariable;
```

Le Javascript étant un langage sensible à la casse, faites bien attention à ne pas vous tromper sur les majuscules et minuscules utilisées car, dans l'exemple suivant, nous avons bel et bien trois variables différentes déclarées :

Code : JavaScript

```
var myVariable;
var myvariable;
var MYVARIABLE;
```

Le mot-clé **`var`** est présent pour indiquer que vous déclarez une variable. Une fois celle-ci 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 = sert à attribuer une valeur à la variable ; ici nous lui avons attribué le nombre 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 trois 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 tout nombre, 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 l'écriture décimale `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 d'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 contenue dans le mot « c'est ». À noter que ce problème est identique pour les guillemets.
En ce qui nous concerne, nous utilisons généralement les apostrophes mais quand le texte en contient trop alors les guillemets peuvent être bien utiles. C'est à vous de voir comment vous souhaitez présenter vos codes, 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 les étudierons lorsque ce sera nécessaire.

Tester l'existence de variables avec `typeof`

Il se peut que vous ayez un jour ou 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 l'instruction **`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
addition	+
soustraction	-
multiplication	*
division	/
modulo	%

Concernant le dernier opérateur, le modulo est tout simplement le reste d'une division. Par exemple, si vous divisez 5 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 nombres c'est bien, mais avec deux variables contenant elles-mêmes des nombres 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 nous avons 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 vous aurez besoin d'écrire des choses 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 cela peut devenir très vite rébarbatif, il existe donc une solution plus simple pour ajouter un nombre 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 :

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

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, comme dans cet exemple :

Code : JavaScript

```
var hi = 'Bonjour', name = 'toi', result;
result = hi + name;
alert(result); // Affiche : « Bonjourtoi »
```

Cet exemple va 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 en ajouter un à l'une des variables, comme ceci : `var hi = 'Bonjour ';`

Autre chose, vous souvenez-vous 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 ».
```

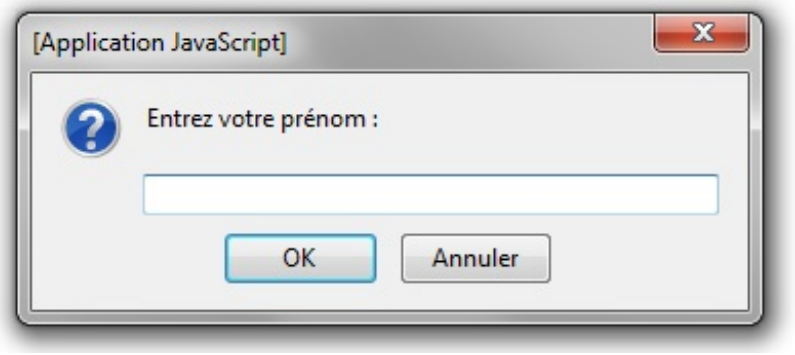
Interagir avec l'utilisateur

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 a une petite particularité. Elle renvoie ce que l'utilisateur a écrit sous forme d'une chaîne de caractères, 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 nombres 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 ? Eh bien la raison a déjà été écrite quelques lignes plus haut :

Citation

> Elle renvoie ce que l'utilisateur a écrit **sous forme d'une chaîne de caractères** […]

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 nombre. 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 chaînes de caractères.

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

En résumé

- Une variable est un moyen pour stocker une valeur.
- On utilise le mot clé **var** pour déclarer une variable, et on utilise = pour affecter une valeur à la variable.
- Les variables sont typées dynamiquement, ce qui veut dire que l'on n'a pas besoin de spécifier le type de contenu que la variable va contenir.
- Grâce à différents opérateurs, on peut faire des opérations entre les variables.
- L'opérateur + permet de concaténer des chaînes de caractères, c'est-à-dire de les mettre bout à bout.
- La fonction `prompt()` permet d'interagir avec l'utilisateur.

- Questionnaire récapitulatif
- Déclarer et initialiser une variable
- Déclarer deux variables en une fois

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 seront déjà bien plus ouvertes car vos conditions vont influer 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 nous vous avions 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. 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 de deux types d'opérateurs : un logique et un de comparaison.

Les opérateurs de comparaison

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

Nous n'allons pas vous faire un exemple pour chacun d'entre eux mais nous allons 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 la même valeur

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 le 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 **false** alors elle ne l'est pas.

Lorsqu'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
nombre et « text » 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 le *nombre* 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 qu'une table de vérité en électronique. Avant de décrire leur fonctionnement, il nous faut d'abord les lister, ils sont au nombre 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 »

result = false && 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'importent les autres valeurs. Exemple :

Code : JavaScript

```
var result = true || true;
alert(result); // Affiche : « true »

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. S'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 le reste de ce chapitre. Toutefois, avant de passer à la suite, il faudrait s'assurer que vous ayez 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 tutoriel :

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 nous écrirons dorénavant le mot « condition » qui sera quand même plus rapide à écrire et à lire.

Avant toute chose, précisons 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 n'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 influer sur l'exécution de votre code. Nous allons tout de suite entrer 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 :

- De 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 nous plaçons ici la première accolade à la fin de la première ligne de condition, mais vous pouvez très bien la placer comme vous le souhaitez (en dessous, par exemple).

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 nous décomposions toutes les étapes d'une condition dans plusieurs variables, dorénavant nous vous conseillons 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.

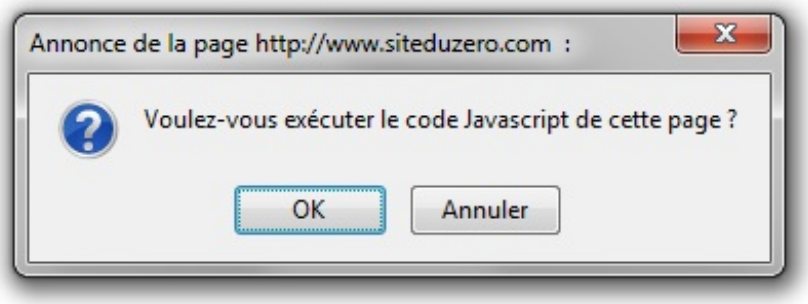
Petit intermède : la fonction `confirm()`

Afin d'aller un petit peu plus loin dans le cours, nous allons 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**, il est conseillé de procéder de la façon suivante :

Code : JavaScript

```
if ( /* condition */ ) {
    // Du code…
} else {
    // Du 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, selon les goûts, un peu plus « propre » à lire. 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 savoir 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.

Cette 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 une condition avec la structure **if** juste avant elle.

La condition « switch »

Nous venons d'étudier 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 quatre 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 est de devoir réécrire à chaque fois 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 le mot-clé **case** suivi de la valeur qu'il doit 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.

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.

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 cela reste extrêmement rare.

Avant de clore cette partie, il est nécessaire de vous faire comprendre un point essentiel : un **switch** permet de faire une action en fonction d'une valeur mais aussi *en fonction du type de la valeur* (comme l'opérateur ===), ce qui veut dire que ce code n'affichera jamais « Bravo ! » :

Code : JavaScript

```
var drawer = prompt('Entrez la valeur 1 :');
```

```
switch (drawer) {
    case 1:
        alert('Bravo !');
    break;

    default:
        alert('Perdu !');
}
```

En effet, nous avons retiré la fonction `parseInt()` de notre code, ce qui veut dire que nous passons une chaîne de caractères à notre **`switch`**. Puisque ce dernier vérifie aussi les types des valeurs, le message « Bravo ! » ne sera jamais affiché.

En revanche, si nous modifions notre premier **`case`** pour vérifier une chaîne de caractères plutôt qu'un nombre alors nous n'avons aucun problème :

Code : JavaScript

```
var drawer = prompt('Entrez la valeur 1 :');

switch (drawer) {
    case '1':
        alert('Bravo !');
    break;

    default:
        alert('Perdu !');
}
```

Les ternaires

Et voici enfin le dernier type de condition, les **ternaires**. Vous allez voir qu'elles sont très particulières, tout d'abord parce qu'elles sont très rapides à écrire (mais peu lisibles) et surtout parce qu'elles renvoient une valeur.

Pour que vous puissiez bien comprendre dans quel cas de figure vous pouvez utiliser les ternaires, nous allons 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 précédent : `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). Beaucoup de personnes en déconseillent l'utilisation, pour notre part nous vous conseillons 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 onze 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 syntaxe, vous vous en rendrez compte par la suite si vous connaissez déjà un autre langage 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 peuvent être de 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.

Bien entendu, la valeur **`undefined`** est aussi évaluée à **`false`**.

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`**. Ceci est extrêmement pratique ! Tâchez de bien vous en rappeler car nous allons nous en resservir fréquemment !

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 quatre 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 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 n'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 nombres 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** (évaluée à **false**) qui signifie *Not a Number*. Nos différentes conditions ne se vérifieront donc pas et la structure **else** sera 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**, nous vous conseillons de relire un peu ce chapitre car ils ne sont clairement pas adaptés à ce type d'utilisation.

En résumé

- Une condition retourne une valeur booléenne : `true` ou `false`.
- De nombreux opérateurs existent afin de tester des conditions et ils peuvent être combinés entre eux.
- La condition `if else` est la plus utilisée et permet de combiner les conditions.
- Quand il s'agit de tester une égalité entre une multitude de valeurs, la condition `switch` est préférable.
- Les ternaires sont un moyen concis d'écrire des conditions `if else` et ont l'avantage de retourner une valeur.

- Questionnaire récapitulatif
- Ecrire une condition
- Ecrire une condition sous forme de ternaire

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-mêmes. 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, nous allons aborder le sujet de l'incrémentation.

L'incrémentation

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. Le Javascript, comme d'autres langages de programmation, permet ce que l'on appelle l'**incrémentation**, ainsi que son contraire, la **décrémentation**.

Le fonctionnement

L'incrémentation permet d'ajouter une unité à un nombre au moyen d'une syntaxe courte. À l'inverse, la décrémentation 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 **incrémenter** et **décrémenter**), et cela nous sera particulièrement utile tout au long de ce chapitre.

L'ordre des opérateurs

Il existe deux manières d'utiliser l'incrémentation 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'incrémentation. 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 :

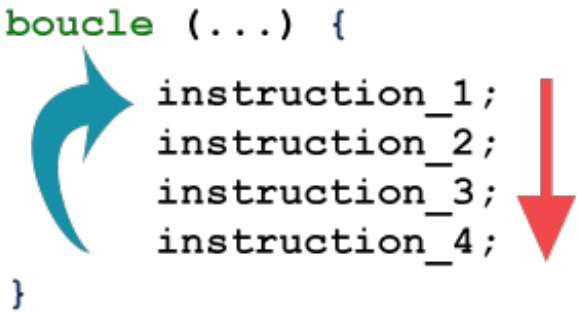
Code : JavaScript

```
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. À chaque fois que la boucle se répète on parle d'**itération** (qui est en fait un synonyme de répétition).



Fonctionnement de la boucle while

Pour faire fonctionner une boucle, il est nécessaire de définir une condition. Tant que celle-ci 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. À chaque fois que les instructions présentes dans la boucle sont exécutées, la condition de la boucle est réévaluée pour savoir s'il faut réexécuter la boucle ou non. Dans cet exemple, 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 (la fonction `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 sœurs. Ce n'est pas compliqué à faire direz-vous, puisqu'il s'agit d'afficher une boîte de dialogue à l'aide de `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 lorsque 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'une espace juste après
    } else {
        proceed = false; // Aucun prénom n'a été entré, 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. Nous avons choisi de

la nommer `proceed`, qui veut dire « poursuivre » en anglais.

À 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 une simple espace, pour séparer les prénoms. Si par contre `nick` contient la valeur **`null`** — ce qui veut dire que l'utilisateur n'a pas entré 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, nous utilisons 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) {
        nicks += nick + ' '; // Ajoute le nouveau prénom ainsi
qu'une 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 fréquente, 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 y a fort à parier que c'est celle-ci 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 trois blocs : **initialisation**, **condition**, et **incrémentation**. Ces trois blocs sont séparés par un point-virgule ; c'est un peu comme si les parenthèses contenaient trois instructions distinctes.

for, la boucle conçue pour l'incrémentation

La boucle **for** possède donc trois blocs qui la définissent. Le troisième est le bloc d'*incrémentation* 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 cinq fois une boîte de dialogue à l'aide de `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 inférieure à 5. Enfin, dans le bloc d'*incrémentation*, on indique qu'`iter` sera incrémentée à chaque itération terminée.

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, ce 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'incrémentation 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, nous pouvons 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'*incrémentation* 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 que l'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, nous allons créer une variable de boucle, nommée `i`, qui sera incrémentée à chaque passage de boucle.

Les variables de boucles **for** sont généralement nommées `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'*incrémentation* 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.

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 précédent 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 de nombreux autres langages, dans lesquels une variable déclarée dans une boucle est « détruite » une fois la boucle exécutée.

Priorité d'exécution

Les trois 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`** ;
- *Incrémentation* : 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 du 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 nous ne pouvons aborder maintenant car elle ne s'utilise que dans certains cas spécifiques.

En résumé

- L'incrémentation est importante au sein des boucles. Incrémenter ou décrémenter signifie ajouter ou soustraire une unité à une variable. Le comportement d'un opérateur d'incrémentation est différent s'il se place avant ou après la variable.
- La boucle **`while`** permet de répéter une liste d'instructions tant que la condition est vérifiée.
- La boucle **`do while`** est une variante de **`while`** qui sera exécutée au moins une fois, peu importe la condition.
- La boucle **`for`** est une boucle utilisée pour répéter une liste d'instructions un certain nombre de fois. C'est donc une variante très ciblée de la boucle **`while`**.

- Questionnaire récapitulatif
- Ecrire une boucle while
- Reconstituer une boucle for
- Ecrire une boucle while qui exécute un prompt()

Les fonctions

Voici un chapitre très important, tant par sa longueur que par les connaissances qu'il permet d'acquérir ! Vous allez y découvrir ce que sont exactement les fonctions et comment en créer vous-mêmes. Tout y passera, vous saurez gérer vos variables dans les fonctions, utiliser des arguments, retourner des valeurs, créer des fonctions dites « anonymes », bref, tout ce qu'il vous faut pour faire des fonctions utiles !

Sur ce, nous allons tout de suite commencer, parce qu'il 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 (en anglais ça l'est en tous les cas).

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'appelez 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 intégré au langage que vous utilisez. Ainsi, l'utilisation des fonctions `alert()`, `prompt()`, `confirm()`, etc. 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 nous pouvons 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 paramètre 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 « 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 (dont les 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 `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 nous le disions 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 où l'utilisation d'une fonction va se révéler utile :

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, nous avons écrit ici exactement deux fois le même code, ce qui nous donne un résultat peu efficace. Nous pouvons envisager d'utiliser une boucle mais si nous voulons 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 cette façon :

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 deux 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 d'ici peu) et surtout nous appelons deux fois notre fonction plutôt que de réécrire le code qu'elle contient.

Voilà l'utilité basique des fonctions : 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 nous allons tout d'abord devoir passer par une étude barbante 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 il est facile d'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, nous allons légèrement modifier l'ordre des instructions mais l'effet devrait 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 allons revenir sur l'affichage de cette erreur dans un instant) nous indique que la variable `ohai` n'existe pas au moment de son affichage avec la fonction `alert()` alors que nous avons pourtant 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**.

Comme je vous le disais, une erreur s'est déclenchée, mais comment avons-nous pu le savoir ? Nous avons en fait utilisé ce qui s'appelle un débogueur ! Un chapitre est consacré à cela dans la partie annexe du cours. Il est vivement conseillé de le lire.

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 votre 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) elles sont parfois à proscrire car elles peuvent rapidement vous perdre dans votre code (et engendrer des problèmes si vous souhaitez partager 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 manière suivante :

Code : JavaScript

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

calculate();
```

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

Juste un petit avertissement : beaucoup de personnes râlent 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 au premier abord 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 nous venons de le dire, 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 envoyez :
    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, voire plusieurs. Il existe aussi des arguments facultatifs que vous n'êtes pas obligés de spécifier.

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

Code : JavaScript

```
function myFunction (arg) { // Vous pouvez mettre une espace entre
```

```
    le nom de la fonction et la parenthèse ouvrante si vous le
    souhaitez, la syntaxe est libre !
        // Votre code…
}
```

Ainsi, si vous passez un argument à cette même fonction, vous retrouverez dans la variable `arg` ce qui a été passé en paramètre. 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) {
    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 `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 tous les arguments auront enfin été reçus ;
- Enfin, `myFunction()` s'exécute !

Vous l'aurez peut-être constaté mais il nous arrive de dire que nous passons des valeurs en paramètres d'une fonction. Cela veut dire que ces valeurs deviennent les arguments d'une fonction, tout simplement. Ces deux manières de désigner les choses sont couramment utilisées, mieux vaut donc savoir ce qu'elles signifient.

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); // Nous pouvons maintenant utiliser l'argument comme souhaité : 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-mêmes 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, nous pouvons 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 ce code n'est pas très bonne, nous vous conseillons de modifier la présentation quand le besoin s'en fait ressentir. Pour notre part, nous aurions 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 nous créions 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 ?

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 d'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 (eh oui ! ) 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 dans cet exemple) 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 ne souhaite 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 dans les chapitres suivants, nous n'allons pas nous 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 !';
    alert('Attention ! Le texte arrive !');
}

alert(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 choses : 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 apprendre à vous en servir tout de suite.

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 de faire comme pour une fonction classique mais sans indiquer de nom :

Code : JavaScript

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

C'est bien joli, mais du coup 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 (nous vous avions 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 ! Eh 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 nous écrivons :

Code : JavaScript

```
function structure() {
    // Du 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() {
```

```
    // Du 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'isolement 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
```

Hou là, une syntaxe bizarre ! Il est vrai que ce code peut dérouter un petit peu au premier abord, nous allons donc vous expliquer ça pas à pas.

Tout d'abord, nous distinguons une fonction anonyme :

Code : JavaScript

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

Viennent 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 nommée :

Code : JavaScript

```
function test() {
    // Du 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 permet de dire « je désigne cette fonction » pour que l'on puisse ensuite indiquer, avec le deuxième couple de parenthèses, que l'on souhaite l'exécuter. Le code évolue donc de cette manière :

Code : JavaScript

```
// Ce code :

    (function test() {

    })();

 // Devient :

    (test)();

 // Qui devient :

    test();
```

Alors, pour une fonction nommée, 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.

À titre d'information, sachez que ces fonctions immédiatement exécutées se nomment des *Immediately Executed Functions*, abrégées IEF. Nous utiliserons cette abréviation dorénavant.

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 IEF 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 une variable du même nom 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 !

Allez, une dernière chose avant de finir ce chapitre !

Les zones isolées sont pratiques (vous découvrirez bien vite pourquoi) mais parfois on aimerait bien enregistrer dans le code global une des 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 ! Il est très important, 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 !

En résumé

- Il existe des fonctions natives, mais il est aussi possible d'en créer, avec le mot-clé **function**.
- Les variables déclarées avec **var** au sein d'une fonction ne sont accessibles que dans cette fonction.
- Il faut éviter le plus possible d'avoir recours aux variables globales.
- Une fonction peut recevoir un nombre défini ou indéfini de paramètres. Elle peut aussi retourner une valeur ou ne rien retourner du tout.

- Des fonctions qui ne portent pas de nom sont des fonctions anonymes et servent à isoler une partie du code.

- Questionnaire récapitulatif
- Définir une fonction
- Écrire une fonction pour comparer deux nombres
- Ecrire une fonction qui demande un nombre

Les objets et les tableaux

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

Il s'agit ici du dernier gros chapitre de la première partie de ce cours, accrochez-vous !

Introduction aux objets

Il a été dit précédemment 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. Le Javascript met à notre disposition des objets natifs, c'est-à-dire des objets directement utilisables. Vous avez déjà manipulé 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 réalité, une variable contient surtout un objet. Par exemple, si nous créons 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 le 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, le 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. Le 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 trois choses distinctes :

- Un constructeur ;
- Des propriétés ;
- Des méthodes.

Le constructeur

Le constructeur est un code qui est exécuté quand on utilise un nouvel objet. Il 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, nous en reparlerons quand nous aborderons l'orienté objet.

Les propriétés

Toute valeur va être placée 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, elle 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

Nous allons créer une chaîne de caractères, pour ensuite afficher son nombre de caractères et la transformer en 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. Ce dernier permet d'accéder aux propriétés et aux méthodes d'un objet. Ainsi, quand nous écrivons `myString.length`, nous demandons au Javascript de fournir le nombre de caractères contenus dans `myString`. La propriété `length` contient ce nombre, qui a été défini quand nous avons 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 aussi un caractère)
```

Essayer !

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

Objets natifs déjà rencontrés

Nous en avons déjà rencontré trois :

1. `Number` : l'objet qui gère les nombres ;
2. `Boolean` : l'objet qui gère les booléens ;
3. `String` : l'objet qui gère les chaînes de caractères.

Nous allons maintenant découvrir l'objet `Array` qui, comme son nom l'indique, gère les tableaux (*array* signifie « tableau » en anglais) !

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. À 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 0. Voici un schéma représentant un tableau, qui stocke cinq 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, 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 pinceaux, et à toujours garder cela 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 au 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 cette syntaxe 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 peuvent donc 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 une 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 grâce à une méthode nommée `toString()`.

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 chaînes de caractères seront collées les unes 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'appliquent 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 fréquemment en Javascript, surtout plus tard, quand nous verrons comment interagir avec les éléments HTML.

« Parcourir un tableau » signifie passer en revue chaque item du tableau pour, par exemple, afficher les items un à un, leur faire subir des modifications ou exécuter des actions en fonction de leur contenu.

Dans le chapitre sur les boucles nous avons étudié la boucle **for**. Celle-ci va nous servir à parcourir les tableaux. La boucle **while** peut aussi être utilisée, mais **for** est la plus adaptée pour cela. Nous allons voir aussi une variante de **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 un tableau 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. À 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 que l'on n'a pas atteint le nombre d'items. `myArray.length` représente le nombre d'items du tableau, c'est-à-dire cinq.

Le nombre d'items est différent des indices. S'il y a cinq 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

Nous avons volontairement mal rédigé le code précédent. En effet, dans le chapitre sur les boucles, nous avons dit que le deuxième bloc d'une boucle **for**, le bloc de condition, était exécuté à chaque itération. Ici ça veut donc dire que `myArray.length` est utilisé 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 normalement pas (dans le cas contraire, n'utilisez pas la solution qui suit).

L'astuce est de définir une seconde variable, dans le bloc d'initialisation, qui contiendra la valeur de `length`. On utilisera cette variable pour la condition :

Code : JavaScript

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

    alert(myArray[i]);

}
```

Nous utilisons `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 `sister` pourrait retourner la valeur « Laurence ».

Pour ce faire, nous allons créer nous-mêmes un tableau 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 schématiquement 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 l'indique ce code, 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 caractères. Donc, pour récupérer le nom de ma sœur, il suffit de faire :

Code : JavaScript

```
family.sister;
```

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

Code : JavaScript

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

Cela 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 après. Exemple :

Code : JavaScript

```
var id = 'sister';

alert(family[id]); // Affiche : « Laurence »
```

Cette façon de faire convient également aux propriétés de tout objet. 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()` car elle n'existe tout simplement pas dans un objet vide, il faudrait pour cela un tableau. En revanche, il est possible d'ajouter un item en spécifiant un identifiant qui n'est pas encore présent. Par exemple, si nous voulons ajouter un oncle dans le tableau :

Code : JavaScript

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

Ou bien sous cette forme :

Code : JavaScript

```
family.uncle = 'Didier'; // Même opération mais d'une autre manière
```

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 ne nous est d'aucune utilité dans le cas d'un objet littéral puisque nous devons posséder un identifiant. En revanche, la boucle **`for in`** se révèle très intéressante !

La boucle **`for in`** est l'équivalent de la boucle `foreach` du 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 scripts tiers utilisés dans la page, 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. Vous trouverez plus d'informations à ce propos sur cet article provenant du site de développement d'Opera. Gardez cet article de côté, il ne pourra que vous resservir durant votre apprentissage.

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 retourner ce dernier. 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 commencer, nous vous proposons un petit exercice qui reprend de manière simple ce que nous avons vu dans ce chapitre.

Énoncé

Dans le chapitre sur les boucles, nous avions 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'une espace jus
s
    } else {
        break; // On quitte la boucle
    }
}

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

Ce que nous vous demandons ici, c'est de stocker les prénoms dans un tableau. Pensez à la méthode `push()`. À la fin, il faudra afficher le contenu du tableau, avec `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 une espace. Si le tableau ne contient rien, faites-le savoir à l'utilisateur, toujours avec `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 v
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 !

Nous avons donc repris le code donné dans l'énoncé, et l'avons modifié pour y faire intervenir un tableau : `nicks`. À la fin, nous vérifions si le tableau contient des items, avec la condition `nicks.length > 0`. Le contenu du tableau est alors affiché

avec la méthode `join()`, qui permet de spécifier le séparateur. Car en effet, si nous avions fait `alert(nicks)`, les prénoms auraient été séparés par une virgule.

En résumé

- Un objet contient un constructeur, des propriétés et des méthodes.
- Les tableaux sont des variables qui contiennent plusieurs valeurs, chacune étant accessible au moyen d'un indice.
- Les indices d'un tableau sont toujours numérotés à partir de 0. Ainsi, la première valeur porte l'indice 0.
- Des opérations peuvent être réalisées sur les tableaux, comme ajouter des items ou en supprimer.
- Pour parcourir un tableau, on utilise généralement une boucle **`for`**, puisqu'on connaît, grâce à la propriété `length`, le nombre d'items du tableau.
- Les objets littéraux sont une variante des tableaux où chaque item est accessible via un identifiant et non un indice.

- Questionnaire récapitulatif
- Écrire une boucle for pour parcourir un tableau
- Écrire un objet littéral et le parcourir avec for in