

Le JavaScript : c'est quoi ?

Qu'est-ce que c'est ?

Définition

Le JavaScript est un langage de programmation.

Pour être plus précis, c'est un **langage orienté objet** : quand on code en JavaScript, on se base sur des *objets*. Je n'en dis volontairement pas plus pour l'instant, car ce sujet sera abordé un peu plus loin, lorsque vous aurez déjà acquis les connaissances requises. Cependant il est qualifié de langage script car l'on peut faire du javascript dans un autre langage

Entre les balises `<script>` et `</script>`

Une nouvelle paire de balises



Et si notre script est très long, on ne va quand même pas tout mettre dans notre gestionnaire d'événements ?

Rassurez-vous, s'il existe plusieurs manières différentes d'insérer du JavaScript, ce n'est pas pour rien. La seconde solution consiste à écrire le script entre deux balises spécifiques, `<script>` et `</script>`.

Il faut commencer par préciser au navigateur que notre script est... du JavaScript. Pour cela, on rajoute la propriété `type="text/javascript"`, ce qui nous donne ceci :

Code : HTML

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

    /* votre code javascript se trouve ici
    c'est déjà plus pratique pour un script de plusieurs lignes */

</script>
```



Mais il y a un problème !

Imaginez un vieux navigateur qui ne connaît pas le JS : s'il tombe sur cette balise, il va tout simplement l'ignorer, car elle lui sera inconnue. Mais tout le code qui suit sera considéré comme du texte, il va donc l'afficher sur votre page ! Pas très élégant 😞.

Il existe heureusement une solution...

Pour faire ignorer ce texte à ce vieux navigateur, on lui fait tout simplement croire qu'il s'agit d'un commentaire en (x)HTML.

Pour un navigateur qui connaît le JavaScript, il saura qu'il n'a plus affaire à du (x)HTML, et passera donc la balise du commentaire (`<!-->`) sans en tenir compte.

Mais pour éviter qu'il soit déboussolé en rencontrant la balise de fin de commentaire (`-->`), nous allons lui dire, en JS, qu'il s'agit d'un commentaire, grâce à `//`.

Pour résumer, voici comment placer du JS dans une page en utilisant la balise `<script>` :

Code : HTML

```
<script type="text/javascript">
  <!--

    /* et vous pouvez placer votre code JS ici
    sans etre inquiete par les vieux navigateurs */

  //-->
</script>
```

Dans l'en-tête ou dans le corps de la page (x)HTML



Ces balises, elles sont à mettre dans l'en-tête, ou dans le corps de la page ?

On peut les placer soit dans l'en-tête (`<head> ... </head>`), soit dans le corps (`<body> ... </body>`) de la page (x)HTML :

- dans le corps de la page, sont à placer les scripts à exécuter au chargement de cette dernière (lorsque le navigateur "lira" le code, il l'exécutera en même temps).
C'est ce que nous allons utiliser pour cet exemple, il suffit d'écrire le code à exécuter entre les balises ;
- en revanche, sont à placer dans l'en-tête les éléments qui doivent être exécutés plus tard (lors d'un événement particulier, par exemple).
Dans ce cas, le code n'est pas écrit "en vrac", nous apprendrons plus loin comment l'organiser.

Ce qui nous fait donc deux manières d'insérer le code grâce à ces balises.

Exemple d'application

Essayez de réaliser cet exemple : dans notre page web, on veut :

- une boîte de dialogue indiquant le début du chargement de la page (donc, le code est à placer au début du corps de la page), une autre indiquant la fin du chargement de celle-ci (donc, à la fin du corps).

Voici le code complet de la page :

Code : HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"xml:lang="fr" lang="fr">
<head>

  <!-- en-tete du document -->

  <title>Un exemple</title>

  <meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1" />
</head>

<body>

  <!-- script pour le debut du chargement -->
  <script type="text/javascript">
  <!--
    alert('Debut du chargement de la page');
  //-->
  </script>

  <!-- ici se trouve le contenu de la page web -->
  <p>
    Vous testez un script..  

    Enjoy ;)
  </p>

  <!-- script pour la fin du chargement -->
  <script type="text/javascript">
  <!--
    alert('Fin du chargement de la page');
  //-->
  </script>

</body>

</html>
```



Vous remarquez que tant que la première boîte de dialogue est ouverte, la page n'est pas chargée. En effet, le navigateur exécute le JS au fur et à mesure du chargement de la page : il attend donc que le script soit terminé avant de charger la suite.

Placer le code dans un fichier séparé

Importer un fichier externe

Comme pour le CSS, on peut très bien placer notre code dans un fichier indépendant. On dit que **le code est importé depuis un fichier externe**. En CSS, l'extension de ce fichier (les deux à quatre lettres précédées d'un point à la fin d'un fichier) était **.css**.

Vous l'avez peut être deviné, l'extension du fichier externe contenant du code JavaScript est **.js**.

On va indiquer aux balises le nom et l'emplacement du fichier contenant notre (ou nos) script(s), grâce à la propriété `src` (comme pour les images).



Comme il n'y a plus de code entre les balises, la technique pour masquer le code aux anciens navigateurs n'a plus lieu d'être.

Et voici ce que cela nous donne :

Code : HTML

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

Ce même fichier qui contiendra par exemple ceci :

Code : JavaScript

```
alert('Bonjour');
```

Créer le fichier

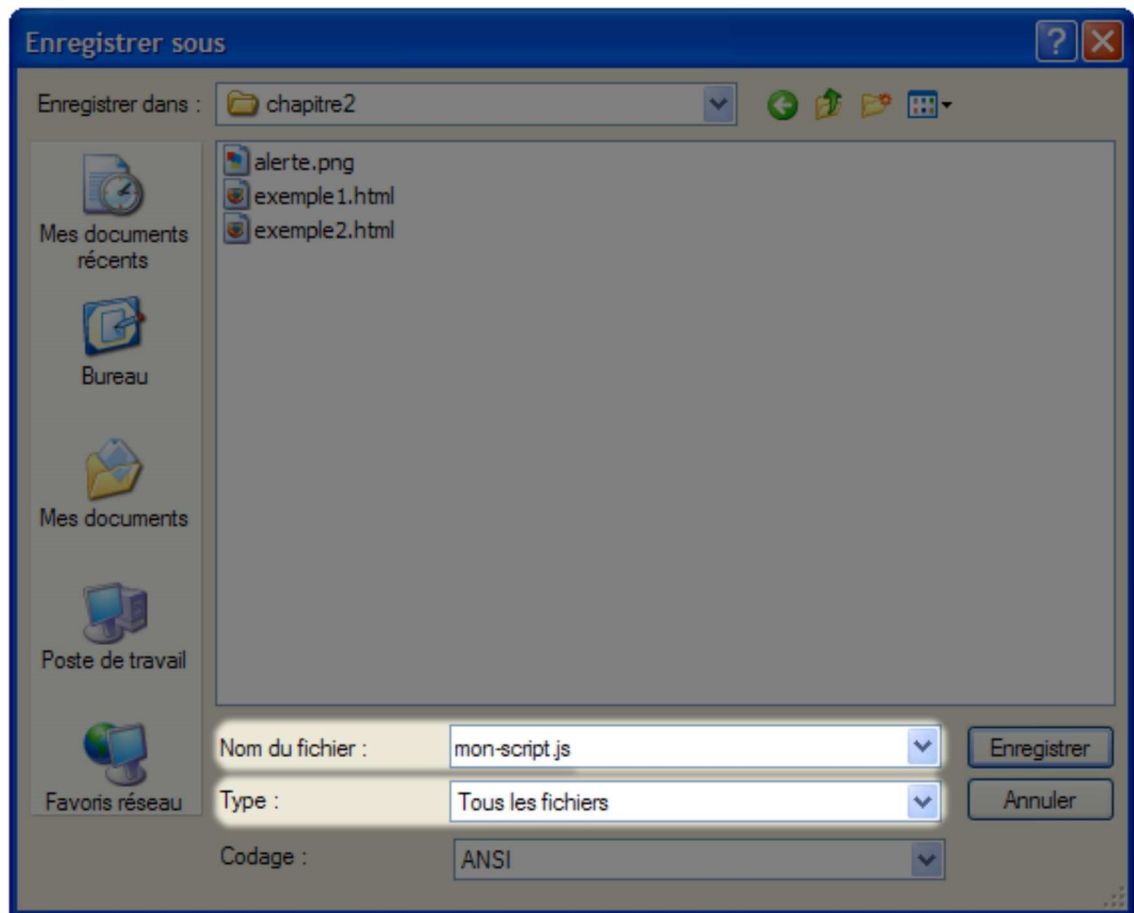
Si vous ne savez pas créer un fichier ayant l'extension `.js`, voici comment faire.

Avec Notepad++

- Créez un nouveau fichier avec le menu **Fichier**, puis
- **Nouveau**. Dans le menu **Langage**, sélectionnez...
- **Javascript !** 🤖
- Enregistrez votre fichier, en lui donnant le nom que vous voulez, suivi de `.js`. Pour ouvrir à nouveau ce fichier, clic droit, puis **Open in Notepad++**.

Avec Bloc-notes

- Ouvrez Bloc-notes.
 - Dans le menu **Fichier**, choisissez **Enregistrer sous...**
 - Dans le menu déroulant **Type** de la nouvelle fenêtre, choisissez **Tous les fichiers** (ou **All types** en anglais).
 - Entrez le nom que vous voulez, suivi de `.js`.
- Pour rouvrir ce fichier, faites un clic droit dessus, **Ouvrir avec...** puis choisissez **Bloc-notes**.



Enregistrer sous... avec Bloc-notes

De quoi se compose le "code" ?

Reprenons le "code" affichant une boîte de dialogue :

Code : JavaScript

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

Voyons voir de quoi il se compose...

Des instructions

Qu'est-ce que c'est ?

Voici un mot important en programmation ; une petite explication s'impose donc...

Une instruction est un "ordre" que l'on donne à l'ordinateur, comme on pourrait taper "format C:" dans sa console (~~les joies de Win~~ pour les adeptes du grand nettoyage de printemps 😊).

`alert('Hello world');` est donc une instruction : on ordonne au script de créer une boîte de dialogue.

Un autre exemple : un calcul est une instruction.

Une grande partie de ce que vous écrirez en JS sont des instructions, mais pas tout. Difficile de vous donner des exemples maintenant, je vous indiquerai au fur à mesure ce qui est ou n'est pas une instruction.

Séparer des instructions

Quand on donne une instruction à l'ordinateur, il faut également lui dire où est la fin de cette instruction... Pour cela, en JS, il y a deux solutions :

- **revenir à la ligne** (avec la touche Entrée ou Enter) : l'ordinateur comprendra qu'il aura ensuite affaire à une autre instruction. utiliser un **point-virgule (;)** à la fin de l'instruction, comme je l'ai
- fait dans les exemples jusqu'ici. Il est bien sûr parfaitement possible d'utiliser un point-virgule *et* un (ou plusieurs) retour(s) à la ligne...



Dans de nombreux langages de programmation (le JavaScript étant une exception), le point-virgule est **obligatoire** à la fin d'une instruction. Je vous **recommende** donc **fortement** de prendre l'habitude d'en mettre un après chaque instruction.

Les fonctions

Explication

Une fonction est une suite d'instructions ayant un rôle précis. (🧐)

Un exemple :

Code : JavaScript

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

Cette fonction affiche, lorsqu'on l'appelle (*terme à retenir*), une boîte de dialogue contenant le texte entre les apostrophes (ici : 'Hello world').

Si vous êtes un peu matheux (ou si vous avez simplement suivi en cours au collège et au lycée 🧐), le terme *fonction* ne doit pas vous être inconnu... On a dit tout à l'heure qu'un calcul est une instruction. La fonction f telle que $f(x) = 15x - 4$ est une suite de calculs, donc en quelque sorte une suite "d'instructions" (par analogie avec le JavaScript, car *instruction* est un terme purement informatique).

Eh bien en JS, on a, parmi beaucoup d'autres, la fonction `alert()`.

Rôle des parenthèses



À quoi servent les parenthèses vides que tu mets en écrivant `alert()` ?

Excellente question !

Revenons à notre exemple mathématique : entre les parenthèses, on précise la valeur de l'argument x. Avec $x = 5$, on aura $f(5) = 15 \cdot 5 - 4 = 71$.

C'est le même principe en JavaScript : par exemple, avec `alert()`, l'argument est le texte à afficher. S'il y a plusieurs arguments, ils sont séparés par des virgules.

On doit **obligatoirement** mettre ces parenthèses, même s'il n'y a aucun argument (elles seront alors vides).



Normalement, le **nom** de la fonction est `alert`, ou `f` pour la fonction mathématique citée plus haut.

Employer le nom `alert()` est un abus de langage, mais cela permet de repérer facilement qu'on parle d'une fonction.

De nouvelles fonctions



Il y a des fonctions déjà définies en JavaScript. On peut aussi créer nos propres fonctions ?

Bien sûr !

Il existe en effet des fonctions déjà définies (retenez également ce terme) en JS, qu'on va donc pouvoir utiliser directement, comme notre fonction `alert()`. Le navigateur va alors chercher la définition de cette fonction et l'exécuter, ce qui aura pour effet de faire apparaître notre message.

Mais il est également possible de créer nos propres fonctions (par exemple, une fonction qui convertit des euros en francs), ce sera l'objet d'un chapitre.

Les variables

Profiter de la mémoire de son ordinateur

En javascript, vous aurez rapidement besoin de demander à votre ordinateur de retenir des données. Qu'est-ce que j'entends par *données* ? Aussi bien des nombres que des mots, ou des phrases (et même plus). C'est ce que nous verrons un peu plus bas.



Je vous rappelle que le JS est exécuté côté client.

Un exemple : vous pouvez enregistrer l'âge d'un visiteur pour lui souhaiter un bon anniversaire le jour J (sur son PC, en enregistrant sa date de naissance).

En revanche, impossible de créer un livre d'or : on ne peut pas enregistrer les messages sur le serveur pour les rendre accessibles à tous les visiteurs.

De quelle manière sont enregistrées les données ?

Peut-être vous demandez-vous comment votre ordinateur range toutes ces informations ?

Vous vous doutez bien qu'on ne fait pas un "gros tas" de données, en vrac : imaginez la galère pour retrouver

ce qui nous intéresse 💡 .

Comme vous vous en doutez, tout cela est organisé.

Illustrons un peu le fonctionnement ...

Imaginez plein de boîtes vides : c'est la mémoire de l'ordinateur.

On veut par exemple retenir le nombre 2006, mais on ne peut pas le mettre comme ça dans une boîte vide, car on ne pourra pas le retrouver !

On va donc coller une étiquette sur une boîte, pour la nommer "annee" : on pourra donc la retrouver plus tard. On va maintenant pouvoir y ranger le nombre 2006 à l'intérieur.

Illustration



Eh bien en JS, c'est un peu ce principe :

- la *boîte* porte le nom de **variable**.
- L'*étiquette* est le **nom** de cette variable (dans notre cas, *annee*). Le nombre *2006* est la **valeur** de cette variable.

Que faire avec ces données ?

Premièrement (c'est la raison pour laquelle vous les avez enregistrées 😊), vous allez pouvoir les **lire** quand vous en aurez besoin.

Cela consiste à ouvrir la boîte, prendre l'information, l'utiliser et la remettre dans sa boîte.

Par exemple, on fait un calcul, on enregistre le résultat dans une variable, et lorsque l'utilisateur clique sur un bouton spécial, on affiche le résultat.

Pouvoir lire ces données, c'est déjà pas mal.

Mais vous aurez vite besoin de les **modifier**.

Dans ce cas, on ouvre la boîte, et on remplace l'ancienne valeur par la nouvelle.

Une application (certes stupide 😊) : un bouton qui, à chaque clic, augmente de 1 la valeur d'une variable. Vous pouvez alors afficher le nombre de fois que le visiteur a cliqué sur ce bouton (ou

"comment occuper son petit frère pendant quelques heures" 🤪).

Créer, lire et modifier une variable

Maintenant que vous avez compris (du moins je l'espère 😊) ce que c'était, voyons le fonctionnement en JavaScript.

Créer la variable

Pour créer une variable, rien de compliqué. Voici comment ça fonctionne : **Code : JavaScript**

```
var nom;
```



C'est une instruction, vous remarquez donc le point-virgule en fin de ligne...

nom est le nom de la variable (si vous avez aimé mon illustration, c'est ce qui est écrit sur l'étiquette collée sur la boîte).

Le terme exact n'est pas "créer", mais déclarer une variable.

Pour ceux qui aiment les maths, si on prend la phrase "*soit x l'âge de Toto, en années*", le `var` correspondrait au *soit*.

(**Note** : nous verrons un peu plus loin qu'il y a des cas pour lesquels on ne déclare pas une variable. Mais retenez pour l'instant qu'il ne faut pas oublier la déclaration).



Faites attention à ne pas déclarer deux fois la même variable.



On peut donner le nom qu'on veut à nos variables ?

Eh non, ce serait trop beau 🤖 .

Le nom peut contenir les 26 lettres de l'alphabet, en majuscules et en minuscules, les 10 chiffres ainsi que le *underscore* (le tiret du bas, touche 8 sur les claviers français). Mais il ne doit pas commencer par un chiffre.

Il y a également des mots "interdits" : ils existent déjà en JS, mais pour autre chose. Si on les utilise comme nom de variable, l'ordinateur ne saura plus de quoi il s'agit.

Parmi cette liste de mots, les plus courants (pour un français) seraient "break", "case", "char", "continue", "delete", "double", "final", "long", "new", "public" et "super". Les autres sont des noms anglais moins courants (je ne pense pas que vous ayez l'idée d'appeler une variable "throws" 🤖). Vous pourrez retrouver la liste complète en annexe.

Heureusement, ça vous laisse quand même du choix dans le nom de vos variables 😊 .



Le JavaScript est **sensible à la casse** : autrement dit, il fait la différence entre lettres majuscules et minuscules.

Ainsi, **nom**, **Nom** et **NOM** sont trois variables différentes !

Préférez aussi un nom relativement court et explicite pour vos variables. N'allez pas créer la variable **taratata**, **hEURÉ** ou bien

laDateDeNaissanceDuMonsieurQuiVisiteMonNouveauSiteWeb 😊 .

Dans le premier cas, on ne sait pas ce que c'est, dans le second, vous risquez de vous tromper entre majuscules et minuscules, et le dernier... un peu lourd et pas très pratique à écrire !

Préférez les noms **heure** et **dateDeNaissance** (ou **date_de_naissance** (avec des tirets) : à vous de trouver la meilleure écriture).

Notez que nous utilisons une mémoire **temporaire** : les variables sont détruites lorsque le visiteur quitte la page (lorsqu'il change de page, entre autres).

Modifier la valeur d'une variable

Pour modifier la valeur d'une variable, rien de plus simple : on utilise pour cela le symbole d'affectation **=** de cette manière :

Code : JavaScript

```
var annee;  
var message;  
  
annee = 2006;  
message = "Bonjour, visiteur";
```



Il n'est pas obligatoire de déclarer les variables au début du code, mais ça peut vous éviter de déclarer deux fois une même variable.

Il est possible, lorsque l'on crée une variable, de lui affecter immédiatement une valeur. On dit qu'on initialise cette variable.

Code : JavaScript

```
var annee = 2006;  
var message = "Bonjour, visiteur";
```



Pourquoi met-on des guillemets dans la variable **message** ?

Car ils délimitent du texte. Je n'en dis pas plus, j'y reviendrai un peu plus bas.

On peut modifier une variable autant de fois qu'on veut :

Code : JavaScript

```
var annee = 2004; // on declare et initialise la variable  
annee = 2005; // on modifie une premiere fois  
annee = 2006; // puis une deuxieme fois
```

Lire une variable

Lorsque l'ordinateur trouve un nom de variable, il fait directement référence à son contenu.

Autrement dit, si j'ai une boîte nommée `annee` contenant la valeur 2006, si je parle à l'ordinateur de `annee` (sans guillemets), il ira chercher la valeur contenue par cette boîte.

Un exemple sera plus simple : reprenons nos

alertes. **Code : JavaScript**

```
var annee = 2006;  
alert(annee);
```

ce qui affichera "2006"...

Remarquez l'absence de guillemets : en effet, on ne veut pas afficher le texte "annee", mais le contenu de cette variable.



Alors maintenant, on ne doit pas mettre de guillemets lors des alertes ?! 🤔

Je n'ai jamais dit ça : voyons plutôt quand les utiliser.

Les chaînes de caractères

"Chaîne de caractères", un nom au premier abord un peu barbare ... Ce n'est en fait pas bien méchant.

En anglais : *string* (je ne le dis pas pour le "jeu de mot" d'une finesse sans égale, mais parce que, tôt ou tard, vous en entendrez parler).

C'est une suite de caractères (un caractère est une lettre, un chiffre, un espace, un tiret...) qui a un ordre précis.

Dit autrement, ce sont des caractères mis bout-à-bout.

Cela forme donc... du texte ! (Dit comme ça, c'est tout de suite plus simple



.) C'est un **type** de donnée, au même titre qu'un nombre.



Une sous-partie entière pour nous parler du texte ? 🤔

Eh oui, vous verrez qu'il y a des choses à dire...

Délimiter une chaîne de caractères

Vous vous dites peut-être : "c'est facile, il suffit d'écrire le texte après le signe = pour enregistrer du texte dans une variable".

Que nenni !

Et cela pour plusieurs raisons : tout d'abord, comment l'ordinateur saurait où se trouve le début et la fin de votre texte ? (Le point virgule en fait-il partie, ...)

Deuxièmement, l'ordinateur analysera le type de données : est-ce que ce sont des variables ? Des chiffres ?

Il va donc falloir dire à l'ordinateur : "ceci est une chaîne de caractères, n'essaie pas de comprendre".

Et pour cela, on utilise indifféremment les guillemets " (dits "*double quotes*") ou les apostrophes ' (dites "*simple quotes*") : ils délimitent la chaîne de caractères, un peu comme une citation.



Pourquoi **deux** types de guillemets ?

Car le JavaScript peut être placé directement dans une balise (x)HTML (rappelez-vous du **onclick**) : dans ce cas, on ne peut pas mettre de *double quotes* (ils sont déjà utilisés en (x)HTML pour délimiter le script). On utilise donc les *simple quotes*.

Un exemple :

Code : HTML

```

```



Code : JavaScript

```
alert('Bonjour');
```

Vous souvenez-vous du code que je vous ai donné auparavant ?

Eh bien vous pouvez maintenant comprendre qu'entre les parenthèses, on met... une chaîne de caractères !

Voyons quelques exemples (avec les variables) :

Code : JavaScript

```
// deux chaines de caracteres
var message1 = 'Une autre chaine de caracteres;
var message2 = "C'est une chaine de caracteres ...;"

// maintenant, on les affiche
alert(message1);
alert(message2);
```

Caractères spéciaux et d'échappement



Et si je veux mettre des guillemets à l'intérieur de ma chaîne, qui est délimitée par des guillemets ? Et comment je fais un saut de ligne ?

Tout d'abord, les guillemets.



Ceci ne marchera pas, car les guillemets servent à délimiter la chaîne :

Code : JavaScript

```
var message = "Ceci est un "petit" test";
alert(message);
```

Il faut dire à l'ordinateur que les guillemets du milieu sont des caractères normaux, et non pas les délimiteurs. Pour cela, on utilise ce qu'on appelle un caractère d'échappement : on fait précéder le guillemet à afficher d'un `\` (*antislash*, touches **Alt Gr + 8** pour les claviers français, **Alt Gr + <** pour les belges).

Et pour afficher un antislash... on le fait précéder d'un antislash aussi 🤔.

Code : JavaScript

```
var message1 = "Ceci est un \"petit\" test. Mais pas besoin d'antislash \\ devant les apostrophes.";
var message2 = 'Un autre "petit" test. Cette fois, il faut penser à l\'antislash devant les apostrophes;

alert(message1);
alert(message2);
```

Les caractères spéciaux

On peut également insérer des retours à la ligne (qu'on ne peut pas insérer simplement en appuyant sur la touche Enter), ainsi que des tabulations ou autres.

Voici les caractères spéciaux les plus courants :

- `\n` qui insère un retour à la ligne.
- `\t` pour insérer une tabulation (ne marche pas dans tous les cas)
- `\b` pour insérer un backspace (comme si vous appuyez sur la touche "retour arrière", celle au-dessus de Enter qui permet d'effacer le dernier caractère).
- `\uXXXX` pour insérer le caractère donc la valeur unicode est `XXXX` (cette valeur est un nombre en hexadécimales).

Vous pouvez trouver la liste de ces caractères avec leur valeur sur [cette page wikipédia](#).

Un exemple (cela signifie "Bonjour ! - Comment vous appelez-vous ?" en allemand. Je l'ai pris pour le caractère spécial) :

Code : JavaScript

```
alert("Guten Tag !\nWie hei\u00DFen Sie ?");
```

La concaténation



Conçaquoi ?! 🤔

Pas de panique, encore un mot compliqué pour quelque chose de simple.

Partons d'un "exercice" : on a une variable `age` qui contient ... l'âge du visiteur (si, si !). On veut afficher un message annonçant : "Vous avez XX ans" (XX est l'âge).

Immédiatement, vous m'écrivez ceci :

Code : JavaScript

```
alert("Vous avez age ans");
```

Et, ô déception, vous voyez s'afficher "Vous avez age ans". 😞

Pour faire cela, on va en fait mettre bout-à-bout nos 3 morceaux de chaîne (le premier morceau est "Vous avez ", ensuite la variable `age`, et enfin " ans", sans oublier les espaces après "avez" et avant "ans").

Concaténer, c'est en fait "mettre bout-à-bout" plusieurs chaînes de caractères pour n'en former qu'une seule.

Comment faire ? On utilise simplement le symbole de concaténation `+` entre chaque morceau (pour ceux qui connaissent, en php, c'est le point qui est utilisé), comme ceci : **Code : JavaScript**

```
var age = 18; // on cree la variable pour pouvoir tester
alert("Vous avez " + age + " ans"); // on affiche les chaines mises
bout-à-bout
```

(On aurait aussi pu créer une variable **message** contenant la chaîne concaténée, et l'afficher ensuite.)

Demander une chaîne de caractère au visiteur

Vous avez peut-être envie de tester ce code, mais avec une variable **age** contenant l'âge qu'on aura demandé au visiteur ?

Voyons une première façon de le lui demander.

Vous vous souvenez sûrement de nos `alert("message")`; affichant un message dans une boîte de dialogue. Eh bien il existe un moyen très proche (encore une boîte de dialogue) pour demander au visiteur de saisir son âge.

Une boîte de dialogue demandant au visiteur de saisir un texte



Code : JavaScript

```
var age = prompt("Texte d'invite");
```

Entre les parenthèses, on met une chaîne de caractères (comme pour `alert`) qui sera affichée au-dessus du champ pour saisir son texte.

On récupère la chaîne de caractères dans la variable **age** (notez que si elle a déjà été déclarée, on ne met pas le **var** devant).

Voici un exemple complet (que je vous conseille de réaliser) : on demande l'âge du visiteur pour le lui afficher ensuite (dans une phrase).

Ce qui nous donne ceci :

Code : JavaScript

```
var age = prompt("Quel âge avez-vous ? (en années)"); // on demande
l'age
alert("Vous avez " + age + " ans"); // on affiche la phrase
```

Vous savez maintenant écrire... apprenons donc à compter 😊.

C'est vrai, après tout, un ordinateur n'est rien d'autre qu'une espèce de grosse machine à calculer.

Les nombres

Un autre type de variable

Les nombres, au même titre que les chaînes de caractères, sont un **type** de variable.

Comme ce sont des nombres (l'ordinateur sait compter), on ne met pas de guillemets : l'ordinateur va "comprendre" qu'il s'agit d'un nombre, et va l'enregistrer à sa manière (et non pas bêtement chiffre par chiffre, comme il l'aurait fait avec une chaîne de caractères).

Ils se classent en deux catégories :

- les nombres entiers : ce sont des valeurs exactes ; les nombres à virgule (les entiers très grands - beaucoup plus qu'un milliard - rentrent dans cette catégorie) : il faut garder à l'esprit que ces valeurs ne sont **pas toujours exactes** !

Pourquoi ? Car il se peut que certains nombres décimaux (avec un nombre fini de chiffres, comme 0,2) ne se "terminent pas" pour l'ordinateur (un peu comme 1/3 ne se termine pas). D'où cette imprécision.



Notez tout d'abord qu'on n'utilise pas la virgule, mais le **point**. On écrira par exemple 3,14 **3.14**.

Un petit exemple :

Code : JavaScript

```
var nombre = 1.234;  
alert(nombre);
```

Des nombres pour calculer

Comme vous le savez, avec des nombres... on peut faire des calculs. Eh oui, le JavaScript, c'est parfois aussi des maths 😊.

L'ordinateur est particulièrement doué pour ces calculs... Voici les opérateurs de base :

- **+** (addition), exemple : $52 + 19 = 71$
- **-** (soustraction), exemple : $52 - 19 = 33$
- ***** (multiplication), exemple : $5 * 19 = 95$
- **/** (division), exemple : $5 / 3 = 1,666...667$ (une petite vingtaine de "6")
- **%** (modulo) : ici, quelques explications s'impose. $a \% b$ (si b est positif, ce qui est le cas dans presque toutes les applications) est le reste de la division de a par b .
Exemple : $52 \% 19 = 14$ (car $52 = 19 * 2 + 14$)

Allons faire quelques exercices d'application...

Un programme qui demande deux nombres et qui en affiche le quotient (qui effectue une division, si vous préférez). A vos claviers 🤖.

Correction...

Secret ([cliquez pour afficher](#))

Code : JavaScript

```
// on demande les nombres
var nombre1 = prompt('Premier nombre ?');
var nombre2 = prompt('Deuxieme nombre ?');

// on calcule le quotient et on l'affiche
var resultat = nombre1 / nombre2;
alert("Le quotient de ces deux nombres est "+ resultat);
```

Notez qu'on aurait pu afficher directement le résultat, sans créer une variable pour celui-ci. On aurait alors mis le calcul à la place du nom de la variable dans le message à afficher.

Cas particuliers

Peut-être avez-vous essayé de faire planter le truc (le genre de truc que j'adore m'amuser à faire 🤖). Voici les messages que vous avez pu obtenir :

- si vous entrez "0" en deuxième nombre (division par 0 impossible...) : le résultat sera "Infinity" ou "-Infinity" (l'infini, positif ou négatif).
- Si vous avez entré du texte, vous obtenez un joli "NaN".

Ça ne veut pas dire "Nan, fais pas ça !" (bah quoi, ça aurait pu ? 🤖).

NaN signifie Not a Number ("Pas un nombre") : c'est un message que vous serez sûrement amenés à rencontrer

...

Priorités de calcul

Ces opérateurs se rangent en deux catégories :

- la première est composée des multiplications (*), division (/) et modulo (%) la seconde regroupe addition (+) et soustraction (-).

- Tout d'abord, les opérations de la première catégorie sont effectuées de gauche à droite.
- Ensuite (il ne reste donc que des additions et soustractions, les autres calculs sont déjà effectués), le calcul se fait de gauche à droite.

S'il y a des parenthèses, on effectue le calcul entre celles-ci en priorité, suivant ces règles.

Un exemple : calcul étape par étape (en bleu, le calcul à effectuer ; en gras, le résultat du calcul précédent) :

- 15 - **7 * 6** % 2 + 3 (catégorie 1, de gauche à droite)
- 15 - **42** % 2 + 3 (idem)
- **15 - 0** + 3 (ne reste que la catégorie 2 : de gauche à droite) **15 + 3** (idem) **18**.
-

D'autres opérateurs pour simplifier l'écriture

Certains calculs reviennent régulièrement.

Parmi ceux-ci, on peut citer ceux du genre :

Code : JavaScript

```
resultat = resultat + X;    // on ajoute X à la variable resultat
```

Il existe l'opérateur **+=** s'utilisant

comme ceci : **Code : JavaScript**

```
resultat += X;    // on augmente la valeur de resultat de X
```

Ces deux lignes sont strictement équivalentes.

Il existe, de la même manière, les opérateurs **-=** (on retranche la valeur de la deuxième variable à celle de la première), ***=** (on multiplie la valeur de la première variable par celle de la deuxième), **/=** (idem mais avec une division) et **%=**.

Incrémentation / décrémentation

Lorsque l'on veut augmenter de 1 la valeur d'une variable (on dit **incrémenter**), par exemple pour un compteur, on utilise la notation :

Code : JavaScript

```
variable++;
```

De même, pour **décrémenter** (diminuer la valeur de 1) une variable, le code est le suivant : **Code : JavaScript**

```
variable--;
```

Pour résumer...

Voici plusieurs lignes de codes qui sont parfaitement équivalentes.

Code : JavaScript

```
variable += X;  
variable = variable + X;
```

Code : JavaScript

```
variable -= X;  
variable = variable - X;
```

Code : JavaScript

```
variable *= X;  
variable = variable * X;
```

Code : JavaScript

```
variable /= X;  
variable = variable / X;
```

Code : JavaScript

```
variable %= X;  
variable = variable % X;
```

Les opérateurs pour incrémenter / décrémenter :

Code : JavaScript

```
variable++;  
variable += 1;  
variable = variable + 1;
```

Code : JavaScript

```
variable--;  
variable -= 1;  
variable = variable - 1;
```

Créer une fonction

À propos des fonctions

Quelques rappels

Une fonction est une suite d'instructions ayant un rôle précis (pour ne pas dire une fonction précise...).

- On lui donne (éventuellement) des **arguments** (également appelés **paramètres**) entre les parenthèses qui suivent le nom de cette fonction.
- Certaines fonctions nous **renvoient une valeur**, que l'on peut par exemple enregistrer dans une variable.

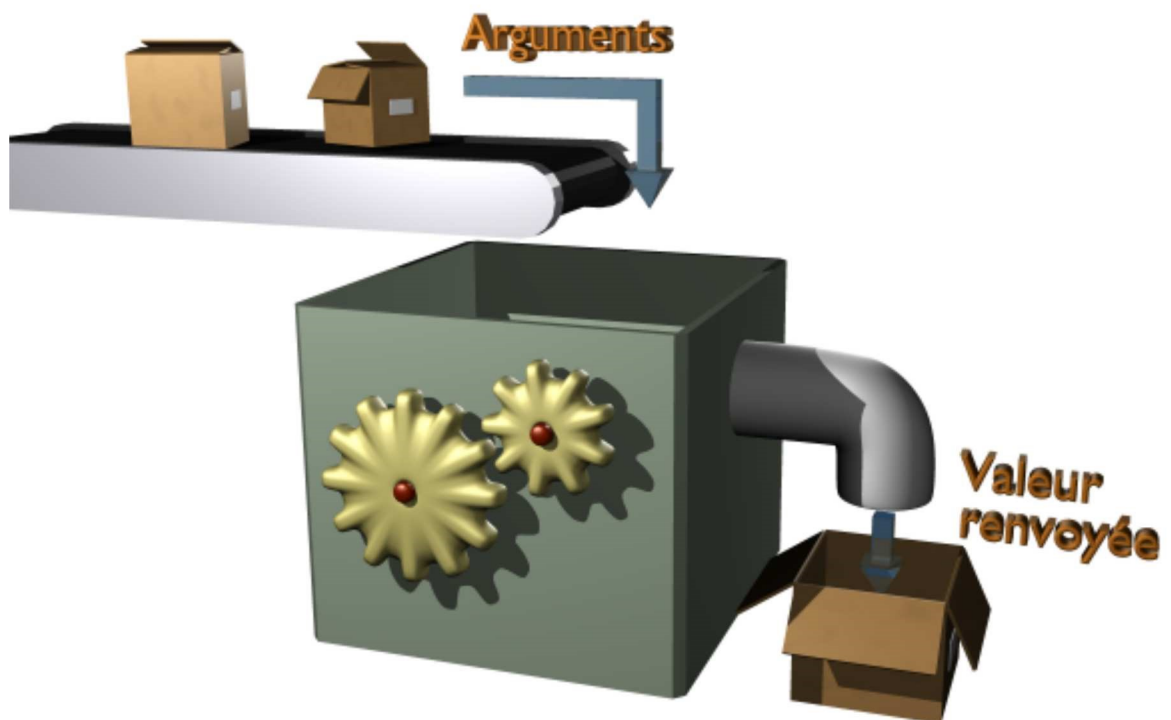
Un petit exemple :

Code : JavaScript

```
var message = prompt('Entrez un texte');
```

- On *appelle* la fonction `prompt()`
- On lui fournit un seul *argument*, qui est `'Entrez un texte'` (c'est le message d'invite qui sera affiché dans la boîte de dialogue).
- Cette fonction nous *renvoie* le texte saisi par l'utilisateur, qu'on enregistre ici dans la variable `message`.

Voici comment on pourrait schématiser une fonction :



Nos fonctions à nous

Les fonctions que nous allons créer fonctionnent selon le même principe. Dans de ce chapitre, notre fonction aura pour but de convertir des Euros en Francs (c'est un bon exemple, même s'il est un peu démodé 😊).



Mais on peut bien faire ça sans avoir besoin de faire une fonction ?

Oui, mais en créant une fonction, nous pourrons lancer le script à chaque fois que nous en aurons envie 😊. Il nous suffira pour cela d'**appeler** cette fonction.

Déclarer et appeler une fonction

Commençons par quelques mots de vocabulaire :

- on dit que l'on **déclare** une fonction lorsqu'on la "crée" : on dit à l'ordinateur qu'elle existe et ce qu'elle doit faire (on écrit le code de cette fonction).
- On pourra ensuite **appeler** notre fonction, ce qui veut dire qu'on va lui demande d'exécuter son code.

Déclarer notre fonction

Voyons comment faire notre propre fonction : nous allons utiliser l'exemple du convertisseur dont je vous ai parlé plus haut.

Déclaration

On commence par **déclarer** notre fonction : on va dire à l'ordinateur "je veux créer une fonction qui s'appelle machin et qui fait ceci".

Quelques informations à ce sujet :

- lors de la déclaration d'une fonction, celle-ci n'est pas exécutée, mais mise en mémoire, pour être exécutée plus tard.
Le code sera alors placé dans l'en-tête de la page, entre les balises `<head>` et `</head>`.
- Une fonction se déclare un peu à la manière d'une variable, à l'aide d'un **mot-clé** (qui est **var** pour les variables, souvenez-vous 😊) : ici, il s'agit de **function** (eh oui, c'est de l'anglais, d'où cette orthographe).
- Pour le nom, c'est comme pour les variables, à savoir les 26 lettres de l'alphabet, en majuscules et en minuscules (une majuscule étant en JS différente d'une minuscule), le *underscore* et les chiffres (sauf pour le premier caractère).
- Le nom doit être suivi de parenthèses (même si elles ne contiennent rien), qui vont contenir les éventuels arguments.



J'attire votre attention sur ces parenthèses. Elles sont **indispensables**, même si elles sont vides.

Si on récapitule ce que je viens de dire, on peut déjà déclarer notre fonction.

Nommons-la **conversion**. On obtient donc ceci :

Code : JavaScript

```
function conversion()
```

Contenu de la fonction

On fait suivre cette déclaration du contenu de la fonction (les instructions de celle-ci).

S'il y a plusieurs instructions (ce qui est généralement le cas 😊), on délimite le contenu de notre fonction grâce à des accolades **{ et }** : on forme alors ce qu'on appelle un **bloc d'instructions**.

Il va donc falloir écrire le code entre accolades, à la suite de la ligne :

Code : JavaScript

```
function conversion()
```

Quel code ? Eh bien celui que la fonction doit exécuter. Si on résume, on veut :

- demander une valeur en euros à l'utilisateur la convertir en francs
- (pour rappel, 1 Euro = 6,55957 Frs) afficher le résultat (pourquoi
- pas sur deux lignes, une en euros, l'autre en francs).

On ne s'occupe pas d'arrondir le résultat, ni de vérifier si l'utilisateur écrit bien un nombre (si c'est un boulet, tant pis pour lui 😊).

Essayez de coder cela vous-mêmes, ça ne devrait pas vous poser de problème...

Vous avez terminé ? Parfait, voici ce que ça nous donne :

Secret ([cliquez pour afficher](#))

```
function conversion()
{
  var somme = prompt("Entrez la valeur en Euros :");
  var resultat = somme * 6.55957;
  alert(somme + " E\n" + resultat + " Frs");
}
```

Code : JavaScript

Nous venons de créer notre première fonction.

Ça va, vous n'êtes pas trop émus ? 😊

Appeler notre fonction

Maintenant que notre fonction est créée, on va enfin pouvoir l'utiliser 😊.

Pour cela, on écrit simplement ceci (dans le corps de la page cette fois) :

Code : JavaScript

```
conversion();
```

Et notre fonction est exécutée !

Elle va, comme nous le voulions, demander une valeur (en euros) à l'utilisateur, et lui afficher le résultat de la conversion en francs.

Et vous savez quoi ? Eh bien on peut maintenant l'appeler quand on veut dans notre page !

Insérons une image qui exécutera cette fonction lorsque nous cliquerons dessus. Pour ceux qui voudraient une image un poil correcte pour notre convertisseur, je vous en ai concoctée une.

Vous n'avez qu'à faire **clic droit** > **Enregistrer l'image sous...**



Nous voulons appeler notre fonction lorsque l'utilisateur cliquera sur l'image.

Utilisons alors le gestionnaire d'événement **onclick** :

Code : HTML

```

```

Et voilà, vous venez de créer un script qui n'est pas totalement inutile 😊.

Les arguments

Revenons à notre fonction de conversion : nous allons lui faire subir quelques modifications.

L'idée est, avec une unique fonction, de pouvoir convertir tout type d'unités.

Par exemple, on peut faire un bouton qui, lorsqu'on cliquerait dessus, convertirait des mètres en centimètres, et une image convertissant les Euros en Francs.

En y réfléchissant, le code à exécuter serait presque le même, il n'y a que le taux de conversion et le nom des unités qui changent...

Il nous suffirait, au moment de l'appel de la fonction, de dire : "le taux de change est de 6.55957, et les unités sont les Euros et les Francs".

Eh bien c'est possible grâce aux **arguments** !

Pour vous en expliquer le principe, je vais prendre l'exemple d'une balise

(x)HTML : **Code : HTML**

```

```



Ce code insère une image.

L'image qui est affichée est celle dont on a indiqué le nom : on utilise la même balise (c'est) quelle que soit l'image, mais en précisant quelle image on veut afficher.

Eh bien c'est exactement le même principe : on va indiquer à notre fonction le taux de conversion à utiliser...

Créer une fonction qui possède des arguments

On va reprendre notre fonction de conversion, en y apportant quelques modifications : commençons par enlever les unités (on les ajoutera plus tard), et ajouter un argument : le taux de change.

On pourra ainsi, à partir de cette même fonction, convertir tout ce qu'on veut.

Si on veut convertir des euros en francs, on n'aura qu'à écrire

`conversion(6.55957)`. Pour convertir des mètres en centimètres, ça sera

`conversion(100)` (1 m = 100 cm).

Et on peut même faire un convertisseur belge, avec `conversion(1)`, qui convertit des francs belges (BEF) en francs belges (1 BEF = 1

BEF) ! 😞 (désolé pour les belges...).

Il nous suffit de préciser le taux entre les parenthèses. Pratique, pas vrai ? 😎

Allons-y :

tout d'abord, lors de la déclaration de la fonction, il nous faut préciser entre les parenthèses le nom de l'argument : appelons-le **taux**.

Lorsqu'on appellera la fonction `conversion(6.55957)`, cela aura pour effet de créer une variable **taux** avant d'exécuter la fonction, et cette variable aura pour valeur **6.55957**. Elle sera détruite une fois la fonction exécutée.

Voici le code modifié et commenté : **Code : JavaScript**

```
function conversion(taux)  // declaration de la fonction avec un
argument
{
    var valeur = prompt("Entrez la valeur à convertir");
    var resultat = valeur * taux; // on calcule le resultat, en
    utilisant l'argument
    alert("Valeur : "+valeur + "\nRésultat : "+resultat);
}
```

Il ne faut donc pas oublier de préciser ce taux lorsqu'on appellera la fonction : **Code : HTML**

```
<p>  </p>
```

Et si on veut convertir des mètres en centimètres, pas besoin d'une autre fonction, il suffit de faire comme ceci :

Code : HTML


```
<p> <a href="#" onclick="conversion(100)">Conversion mètres &gt; centimètres</a> </p>
```

Fonction à plusieurs arguments

Il serait tout de même plus agréable de voir s'afficher le nom des unités... Eh bien il suffit pour cela de créer deux arguments supplémentaires !

Rien de bien compliqué, il faut juste les séparer par des **virgules**. Vous pouvez les créer dans l'ordre que vous voulez (essayez cependant d'utiliser un ordre "logique"). Ensuite, lorsque vous appellerez votre fonction, il faudra donner les paramètres dans le même ordre (si le taux de conversion est le premier, on l'indiquera en premier - logique 😊).

Choisissons cet ordre : **unité 1, taux et unité 2**.

Après quelques modifications apportées à notre fonction, on obtient ceci : **Code : JavaScript**

```
function conversion(unitel1, taux, unite2)
{
    var valeur = prompt("Entrez la valeur à convertir, en "+ unitel1);
    var resultat = valeur * taux;
    alert(valeur + ' ' + unitel1 + '\n' + resultat + ' ' + unite2);
}
```

Et on l'appelle ensuite de cette manière : **Code : HTML**

```
<p>
   <br />
  <a href="#" onclick="conversion('m', 100, 'cm')">Conversion mètres &gt; centimètres</a>
</p>
```

Je vous laisse tester...

Ça commence à avoir de la gueule, pas vrai ? 🤪

Arguments facultatifs

Il est possible de créer des arguments **facultatifs** : on peut choisir de ne pas les préciser lors de l'appel de notre fonction.

Par exemple, il serait possible de créer une fonction qui additionne tous les nombres passés en arguments :

Code : JavaScript

```
addition(12, 5); // nous donnerait 17
addition(21, 4, 15, 11, 6); // nous donnerait 57
```

Ceci n'étant qu'un exemple.

Tenez, vous rappelez-vous de notre fonction **prompt()** ?

Eh bien elle possède un second paramètre facultatif : la valeur initiale du champ de saisie. Essayez plutôt ceci : **Code : JavaScript**

```
var nombre = prompt('Entrez un nombre', 'Votre nombre ici');
alert('Vous avez tapé ' + nombre);
```

Si on ne donne pas de second paramètre, le champ est initialement vide. Mais si on en indique un, alors il aura cette valeur.

Je vous parle de ces arguments pour que vous sachiez qu'ils existent. Cependant, créer des paramètres facultatifs fait appel à des notions que vous ne connaissez pas encore. Nous approfondirons donc ce sujet un peu plus tard, dès que vous aurez les éléments nécessaires.

Portée d'une variable

Fonctionnement des arguments

Comme je vous l'ai annoncé plus haut, les arguments ne sont en fait rien d'autre que des variables, mais propres à la fonction : elles sont créées lors de son appel, et elles sont détruites à la fin de celle-ci.

Voici une fonction :

Code : JavaScript

```
function essai(argument1, argument2)
{
    // code de la fonction
}
```

Et un appel de cette fonction :

Code : JavaScript

```
essai(57, 'un message');
```

Lorsqu'on appellera cette fonction ainsi, voici comment ça se passera :

- tout d'abord, les variables **argument1** et **argument2** seront créées et prendront comme valeurs respectives le nombre **57** et la chaîne de caractères '**un message**'.
- Ensuite, le code de la fonction sera exécuté (ces variables peuvent être modifiées, comme des variables classiques).
- Lorsque la fonction sera terminée, les variables **argument1** et **argument2** seront détruites.

Portée des variables

Essayez donc de créer cette fonction, de l'appeler, et une fois cette dernière terminée, d'afficher la valeur de la variable **argument1** : ça ne marche pas. C'est normal, elle a été détruite.

Essayez de renouveler l'opération, mais en créant vous-mêmes une variable à l'intérieur de la fonction, comme ceci : **Code : JavaScript**

```
function essai()  
{  
    var variable = 'Bonjour';  
}
```

Appelez ensuite cette fonction, puis affichez la valeur de cette variable, comme ceci : **Code : JavaScript**

```
essai();  
alert(variable);
```

Cette fois non plus, ça ne marche pas !

Variable locale



À la fin d'une fonction, toutes les variables **déclarées** à l'intérieur de celle-ci sont détruites. On utilise le terme de variables **locales**.

Notez que j'ai bien parlé de **déclaration**, ce qui signifie l'utilisation de **var**. Voici pourquoi, dans le dernier exemple, ça ne "*marchait pas*".



Mais c'est nul ! 🤔 Pourquoi on ne peut pas accéder quand on veut et où on veut à toutes les variables ?!

Eh bien imaginez le bazar !

Certaines variables seront peut-être utilisées plusieurs fois (si vous donnez le même nom à deux variables différentes, par exemple). De plus, il est moyennement apprécié (pour ne pas dire qu'il ne faut jamais le faire) de déclarer plusieurs fois la même variable. Enfin bref, c'est ingérable.

Au moins, avec les variables locales, c'est bien rangé : chaque fonction a ses variables à elle seule, on peut donc les modifier sans craindre de modifier une variable d'une autre fonction. Et il suffit d'utiliser les arguments pour passer des valeurs d'une fonction à l'autre.

Variables globales

Il est quand même possible de créer des variables dites **globales**, c'est-à-dire accessibles depuis n'importe quelle fonction.



Ces variables sont à utiliser le moins possible !

Si une variable locale suffit pour telle utilisation, ne créez pas une variable globale en vous disant : "comme ça, je pourrai l'utiliser de partout".

Pour créer une telle variable, il y a deux solutions :

-

Code : JavaScript

```
// on cree deux variables globales
var variable1;
var variable2 = 0;

function essai()
{
    variable1 = 1;
    variable2 = 8;
    // modification des variables globales
}
// ces deux variables seront encore accessibles une fois la
fonction terminee
```

- la première consiste à déclarer la variable en dehors de toute fonction.

La seconde est de ne pas déclarer la variable avec var : on l'utilise comme si elle avait déjà été déclarée (on parle de *déclaration implicite*). L'exemple serait le même que celui du dessus, mais sans les 3 premières lignes. Les variables sont créées directement dans la fonction. Personnellement, je préfère utiliser la première solution.

Au risque de me répéter, je vous rappelle qu'il faut privilégier au maximum l'emploi de variables locales...

Valeur renvoyée

Approche

Partons de cet exemple, que vous connaissez tous :

Code : JavaScript

```
var ma_variable = prompt('Entrez une valeur');
```

On utilise le signe `=` : mais comment une fonction peut-elle avoir une valeur ?

Prenons l'exemple des fonctions en mathématiques...

Soit f la fonction définie par $f(x) = 5x - 3$.

On a $f(x) = 5x - 3$...c'est bien ce signe `=` qui nous intéresse !

Eh bien essayons de créer une fonction f qui fait la même chose que notre fonction mathématique.

Création d'une fonction classique

On peut d'ores et déjà écrire la première ligne de code :

Code : JavaScript

```
function f(x)
```

On déclare une fonction nommée f ayant un seul attribut, nommé x .

Que va contenir cette fonction ?

On va également créer une variable pour le résultat du calcul : appelons-la **resultat** (très original, n'est-ce pas ?).

Code : JavaScript

```
var resultat = 5*x - 3;
```

Renvoi de la valeur

On a notre fonction et notre résultat, il nous reste plus qu'à dire : "je veux que ma fonction prenne cette valeur". Pour employer le terme exact, on veut que notre fonction **renvoie** (ou *retourne*) cette valeur.

Par exemple, avec **prompt**, la valeur retournée est le texte tapé par l'utilisateur. Dans notre cas, c'est le contenu de la variable **resultat**.

Pour ce faire, on fait précéder la valeur à renvoyer du mot-clé

return. Code : JavaScript

```
return resultat;
```



Lorsqu'une fonction retourne une valeur, son exécution se termine : s'il reste du code à exécuter dans cette fonction, il sera ignoré.

Pour stopper l'exécution d'une fonction, on peut donc utiliser simplement ceci, qui ne renvoie rien, mais termine l'exécution :

Code : JavaScript

```
return;
```

On assemble maintenant tous nos morceaux, voici la fonction *f* telle qu'on pourrait la coder en JS : **Code : JavaScript**

```
function f(x)
{
    var resultat = 5*x - 3;
    return resultat;
}
```

On teste de cette manière (ou en enregistrant le résultat dans une variable) :

Code : JavaScript

```
alert( f(7) );
```

Et... on voit s'afficher le résultat : 32 😊.

Quelques exemples

De cette manière, vous pouvez créer toutes sortes de fonctions (particulièrement mathématiques 😊). Voici quelques exemples :

- *triple(x)* qui renvoie le triple de l'argument
- *carre(x)* qui renvoie le carré de l'argument
- *(x²) cube(x)* qui renvoie le cube de l'argument.

Voici la correction (vous pouvez faire différemment, avec une variable par exemple) :

Secret ([cliquez pour afficher](#))

```
function triple(x)
{
    return 3*x;
}

function carre(x)
{
    return x*x;
}

function cube(x)
{
    return x*x*x;
}
```

Code : JavaScript

Un exemple de calcul dans lequel on voit clairement l'utilité de ceci :

Code : JavaScript

```
var resultat = 4*cube(5) - 7*3 / carre(6);
```

Les booléens

Un nouveau type de variable

Vous savez qu'une variable peut contenir un nombre ou une chaîne de caractères.

Mais elle peut également contenir d'autres données : celle que nous allons aborder s'appelle un **booléen**.

Qu'est-ce que c'est ?

Avant de se lancer dans des explications, un peu d'histoire.

Ce nom vient de Georges Boole, un britannique à l'origine de cette technique.

Il s'agit de traiter des informations qui peuvent prendre **deux valeurs** (par exemple "vrai" ou "faux"), ce qui est très utilisé, particulièrement en électronique et en informatique : soit il y a du courant (première valeur), soit il n'y en a pas (deuxième valeur).

Pourquoi je vous parle de cela dans un chapitre sur les conditions ?

Car ces dernières ne sont rien d'autre... que des valeurs booléennes !

Soit la condition est vraie, soit elle est fausse.

Variable booléenne

Il est possible de créer des variables contenant de telles valeurs (pour enregistrer par exemple le résultat d'un test). Ces variables prennent deux valeurs : **true** (vrai) et **false** (faux).

Un petit exemple : cette variable pourrait permettre de savoir si le visiteur est majeur ou mineur (la valeur serait à demander au visiteur) :

Code : JavaScript

```
var majeur = true;
```

Ce n'est pas plus compliqué que ça ...

Opérateurs et conditions

Il serait bien de pouvoir comparer des variables entre elles pour déterminer la valeur d'une variable booléenne. Par exemple, si l'âge du visiteur est inférieur à 18 ans, alors on retient qu'il n'est pas majeur, sinon on retient qu'il est majeur.

C'est bien sûr possible en JS, grâce à des **opérateurs de comparaison**.

Opérateurs de comparaison

Les opérateurs d'égalité (fonctionnent aussi avec des chaînes de caractères) :

- **==** : si les deux valeurs sont égales, alors on a true, sinon false.
- **!=** : si les deux valeurs sont différentes, alors on a true, sinon false.



En JS (comme dans de nombreux autres langages), le signe **!** sert à marquer la négation. **!=** veut donc dire "non égal", donc "différent".



Le signe **==** n'est pas une erreur d'écriture !
Il sert à comparer deux variables, à la différence du signe **=**, utilisé pour affecter une valeur à une variable. Confondre les deux est une erreur classique...

Retenez donc bien que **pour comparer deux valeurs, on emploie le signe ==**.

Opérateurs de comparaison de valeurs numériques (si les valeurs ne sont pas des nombres, le résultat ne sera pas celui attendu) :

- **a < b** : si **a** est inférieur à (plus petit que) **b**, alors on a true, sinon false.
- **a > b** : si **a** est supérieur à (plus grand que) **b**, alors on a true, sinon false.
- **a <= b** : si **a** est inférieur ou égal à **b**, alors on a true, sinon false.
- C'est le contraire de **>** : si **a** n'est pas supérieur à **b**, ...
- **a >= b** : si **a** est supérieur ou égal à **b**, alors on a true, sinon false.
- C'est le contraire de **<** : si **a** n'est pas inférieur à **b**, ...

Utilisation de ces opérateurs

Revenons à notre exemple : on a une variable **age** contenant l'âge du visiteur, et on veut en déduire la valeur de la variable booléenne **majeur**.

Pas très difficile avec ces opérateurs : on est majeur si on a 18 ans ou plus. On écrira donc :

Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');

// on compare l'age : s'il est supérieur ou égal à 18 ans, alors le
// visiteur est majeur
var majeur = (age >= 18);

alert('Vous êtes majeurs : ' + majeur);    // on vérifie que ça marche
^^
```



Les parenthèses autour de la condition ne sont pas nécessaires, mais elles rendent le code beaucoup plus lisible (à mon goût 😊).

Libre à vous de les mettre ou non.

Des conditions plus élaborées



Et si on veut une variable **costaud** qui sera vraie lorsque le visiteur mesurera plus de 2 mètres et qu'il pèsera plus de 90 kgs ?

Eh bien c'est ici que deux nouveaux opérateurs entrent en scène, des **opérateurs logiques**.

Vous avez vous-mêmes prononcé le nom de l'un d'eux : il s'agit de l'opérateur **ET**. Le second est l'opérateur **OU**.

En JS, on les note **&&** (ET) et **||** (OU) ; vous formez ce "|" en appuyant deux fois sur **Alt Gr + 6** sur un clavier AZERTY français, et **Alt Gr + 1** sur un belge.

ET

On veut une variable **costaud** qui sera vraie lorsque le visiteur mesurera plus de 2 mètres et qu'il pèsera plus de 90 kgs.

Commençons par demander la taille et le poids du visiteur :

Code : JavaScript

```
var taille = prompt('Combien mesurez-vous ? (en mètres)');
var poids = prompt('Combien pesez-vous ? (en kgs)');
```

(Attention à taper un point, et non une virgule pour la taille !)

Voyons maintenant quelles sont les deux conditions nécessaires :

- la première : **taille** >= 2 (on pourrait aussi prendre uniquement supérieur à) - la seconde : **poids** >= 90.

On pourrait créer deux valeurs booléennes contenant les résultats de ces deux tests, et ensuite créer notre variable **costaud**, mais faisons cela en une seule fois.

Il faut que les deux conditions soient vraies, d'où l'emploi de **ET**.

On obtient :

Code : JavaScript

```
costaud = (taille>=2 && poids>=90);
```

(Je n'ai pas mis d'espace de part et d'autre des signes >= pour mieux faire ressortir les deux conditions : encore une fois, libre à vous d'en mettre où bon vous semble.)

On assemble ces morceaux, ce qui nous donne ce script :

Code : JavaScript

```
var taille = prompt('Combien mesurez-vous ? (en mètres)');
var poids = prompt('Combien pesez-vous ? (en kgs)');

costaud = (taille>=2 && poids>=90);
alert('Vous êtes costaud : ' + costaud);
```

Voici ce qu'on appelle une **table de vérité** pour l'opérateur **&&**.

On affiche la valeur de **a && b** selon les valeurs de **a** et de **b**, ligne par ligne (false = 0 et true = 1).

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

OU

On veut maintenant faire le même script, mais avec une personne de plus de 2 mètres **OU** de plus de 90 kgs. Une petite modification : on emploie **||** au lieu de **&&**.

Ici, il suffit qu'une seule condition soit vérifiée pour que le message s'affiche, c'est pourquoi on emploie **OU**.

Notez que si les deux conditions sont vérifiées, ça marche aussi 😊.

On a donc le code suivant :

Code : JavaScript

```
var taille = prompt('Combien mesurez-vous ? (en mètres)');
var poids = prompt('Combien pesez-vous ? (en kgs)');

costaud = (taille>=2 || poids>=90);
alert('Vous êtes costaud : ' + costaud);
```

Et n'oublions pas la table de vérité de cet opérateur ...

a b		a b
0	0	0
0	1	1
1	0	1
1	1	1

Priorités de ces opérateurs

En l'absence de parenthèses, les && sont prioritaires sur les || : on commence par eux. **Code : Autre**

```
a && b || c && d      =      (a && b) || (c && d)
```

En présence de parenthèses, comme pour des calculs, les éléments entre ces dernières sont calculés avant le reste.

La négation

Symbole "NON"

Repartons (encore) de notre exemple concernant l'âge...

Cette fois-ci, nous ne voulons pas une variable booléenne "majeur", mais une "mineur" (qui sera **true** pour un mineur et **false** pour un majeur - extraordinaire, vous ne trouvez pas ? 🤔).

Vous me dites : "facile, c'est vrai si l'âge est inférieur à 18 ans".

Certes, vous n'avez pas tort.

Sauf si je rajoute qu'il faut se servir de la condition élaborée pour la variable "majeur", qui est la suivante : **Code : JavaScript**

```
var majeur = (age >= 18);
```

Après longue réflexion, vous m'affirmez que c'est simplement l'opposé de cette condition.
On a ici "si l'âge est supérieur ou égal à 18 ans".

Mais on veut "si l'âge n'est **PAS** supérieur ou égal à 18 ans".

Eh bien ce "**PAS**" existe : il s'agit du symbole **!** (comme j'ai pu vous l'annoncer avec le signe **!=**, signifiant "pas égal"). Il précède la condition, comme ceci :

Code : JavaScript

```
var mineur = !(age >= 18);
```

C'est tout 😊.

Enfin presque, j'allais oublier la table de vérité (certes

t	
o	
u	

t

e

s

i

m

p

l

e

)

:

a

!

a

0 1

1 0

Théorème de De Morgan

Ça rappelle la SI (Sciences de l'Ingénieur) du lycée... que de bons souvenirs 😎.

Bref, notez tout d'abord qu'il n'y a pas de faute de frappe, il s'agit bien du théorème de De Morgan.

Il nous vient d'un certain Auguste (ou Augustus) De Morgan, mathématicien britannique du dix-neuvième siècle, qui est, avec Boole (eh oui 😊), le fondateur de cette logique binaire (booléenne).

Mais ce qui nous intéresse maintenant plus particulièrement, ce sont les lois qu'il a formulées...

Plutôt que de vous les énoncer "bêtement", nous allons les retrouver ensemble.

Illustration

On sort de nos variables pour prendre un exemple concret.

Vous êtes végétariens : vous mangez de tout, sauf de la viande et du poisson (si ce n'est pas le cas, on va faire comme si 🙄).

Vous arrivez à un repas, vous vous dites : "je mange s'il n'y a PAS de viande ET PAS de poisson". Ou bien : "je mange s'il n'y a PAS : (du poisson OU de la viande)".

On aurait donc égalité entre ces deux expressions :

Code : Autre

```
!a && !b
!(a || b)
```

Une petite table de vérité permet de vérifier ça :

ab	(a b)	!a && !b
00	1	1
01	0	0
10	0	0
11	0	0

Et en inversant les opérateurs logiques **&&** et **||**, on obtient le même résultat. Ce qui nous prouve les lois de De Morgan (formulées ici avec les notations du JS) : **Citation : lois de De Morgan**

2) $!(a \ \&\& \ b) = !a \ || \ !b$

1) $!(a \ || \ b) = !a \ \&\& \ !b$

(Notez qu'elle est également vérifiée avec 3 variables booléennes ou plus.)



On pourrait très bien s'en passer.

Tu nous compliques la vie pour pas grand chose !

Tout d'abord, ça permet de simplifier (moins de parenthèses) les conditions. Et ça évite surtout cette erreur :



Code : Autre

```
!(a && b)    est égal à    !a && !b
```

IF et ELSE



Mais à quoi ça sert, toutes ces variables booléennes ?

Eh bien il est possible de faire un test : c'est-à-dire que si une variable booléenne vaut **true**, alors on effectue une action, et si elle vaut **false**, on n'en effectue pas (ou une autre).

IF

Pour commencer, nous allons reprendre notre code avec la variable
majeur : Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');  
var majeur = (age >= 18);
```

Ce que nous voulons maintenant, c'est afficher un message **SI** la personne est majeure. En anglais, le "si" nous donne "if"... c'est justement ce qu'on utilise pour faire un test : **Code : JavaScript**

```
if(valeur_booleene)
```

Si la valeur booléenne est **true**, alors l'instruction qui suit le test est exécutée. Sinon, elle est ignorée et on passe à la suite.



Si on veut mettre plusieurs instructions avec un seul **if**, on les place entre accolades pour former un *bloc d'instructions*, qui sera exécuté seulement si le test est vrai. Avec notre code à nous, voici le résultat :

Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');  
var majeur = (age >= 18);  
  
if(majeur)           // on effectue le test  
    alert('Vous êtes majeur'); // si la variable vaut "true", alors on  
                                affiche ce message
```

Voici ce même code, mais avec un bloc d'instructions (indispensable s'il y a plusieurs instructions à exécuter si le test est vrai, facultatif s'il n'y en a qu'une comme ici) : **Code : JavaScript**

```
var age = prompt('Quel âge avez-vous ? (en années)');  
var majeur = (age >= 18);  
  
if(majeur)  
{  
    alert('Vous êtes majeur');  
    // instructions dans le bloc à placer ici  
}
```

On n'est pas obligés de stocker le résultat du test, on peut l'utiliser directement dans le `if()` de cette manière :

Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');
if(age >= 18)
    alert('Vous êtes majeur');
```

C'est généralement comme ça qu'on fait, sauf lorsqu'on a besoin de conserver le résultat.

Et pour faire encore plus court, on peut encore se passer de la variable `age` : on teste directement la valeur entrée par l'utilisateur :

Code : JavaScript

```
if(prompt('Votre âge ?') >= 18)    // on demande l'âge et on le compare à
18
    alert('Vous êtes majeur');
```

ELSE

Si le test est faux, nous avons vu que l'instruction (ou le bloc d'instructions) qui suit est ignoré(e), et que le script continue après celui (celle)-ci.

Il est également possible d'effectuer une instruction ou un bloc d'instructions si la condition est fausse : il (elle) sera ignoré(e) à son tour si le test est vrai.

Pour cela, on rajoute simplement le mot-clé "else" (en français : "SINON").

Reprenons notre code, mais qui affichera cette fois-ci un message aux mineurs :

Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');
if(age >= 18)
    alert('Vous êtes majeur');
else
    alert('T\'es mineur');
```

Ce qui se "traduit" par : **SI** l'âge est supérieur ou égal à 18, alors on affiche que le visiteur est majeur, **SINON** on affiche qu'il est mineur.

Conversion en valeur booléenne



Et si je fais ce test, ça donne quoi ?

Code : JavaScript

```
if (age)
  // instruction ici
```

On utilise assez souvent ce type de test.

Le résultat sera **false** dans les cas suivants :

- la variable a été déclarée, mais n'a pas de valeur.

Code : JavaScript

```
var age;
```

- La variable vaut **0** (zéro - ATTENTION, la valeur '0' - zéro, mais sous forme de chaîne de caractères - ne rentre pas dans cette liste !).
- La variable vaut **null**.
- La variable vaut **null** (valeur renvoyée par certaines fonctions dans certains cas).

Dans les autres cas, le résultat sera **true**.

Ce test permet donc de savoir si l'utilisateur a entré une valeur.

Exemple : lorsqu'on demande l'âge, il y a un bouton "Annuler" à côté du bouton "Valider". En cliquant dessus, la variable **age** contiendra la valeur "" (chaîne de caractères vide).


On peut donc se servir de ce test pour vérifier que l'utilisateur n'a pas cliqué sur "Annuler".

Code : JavaScript

```
var age = prompt('Quel âge avez-vous ? (en années)');
if (!age)
  alert('Vous devez entrer votre âge');
else
{
  if (age >= 18)
    alert('Vous êtes majeur');
  else
    alert('T\'es mineur');
}
```

La fonction isNaN

Présentation

Continuons avec une fonction qui a bien sa place dans ce chapitre : **isNaN** (à première vue, un joli nom , qui prend un seul argument.

Elle a bien sa place ici, car elle retourne une valeur booléenne.

Que fait cette fonction ?

Derrière un nom qui peut sembler bizarre, il n'y a en fait rien de compliqué.



Rappel : le sigle **NaN** signifie "*Not a Number*".

Soit, après une traduction très difficile : "*pas un nombre*" 😊.

Voilà, je vous ai tout dit !

Le nom **isNaN** devient tout de suite moins barbare : "*is Not a Number*", ou bien encore "n'est pas un nombre".

En effet, cette fonction renvoie :

- **true** si l'argument n'est **pas** un nombre ;
- **false** si l'argument est un nombre (ou bien une chaîne de caractères qui "est" un nombre : un nombre entre guillemets).



Rappel : on utilise un point à la place de la virgule.

3.14 est donc un nombre, mais 3,14 n'en est pas un !

Petit exemple

On calcule le carré d'un nombre demandé à l'utilisateur.

S'il n'a pas tapé un nombre, on ~~l'insulte~~ le lui fait gentiment remarquer.

Code : JavaScript

```
var nb = prompt('Entrez un nombre');
if(isNaN(nb))
    alert('Vous devez entrer un nombre !');
else
    alert('Son carré est ' + nb*nb);
```

Vous remarquerez au passage que les valeurs **null** (lorsque vous cliquez sur "Annuler") et "" (chaîne vide - lorsque vous laissez le champ vide) prennent la valeur numérique **0**.

Distinguer plusieurs cas avec SWITCH

Exemple d'utilisation



Et si on a besoin de tester plusieurs valeurs ? On est obligé de faire plusieurs if les uns à la suite des autres ?

Il est possible d'utiliser cette solution. Un exemple :

Code : JavaScript

```
var nom = prompt("Entrez un nom d'animal");
if(nom == "chat")
    alert("Miaou !");
else if(nom == "chien")
    alert("Et PAF, le chien !");
else if(nom == "pingouin")
    alert("Bonjour, Tux");
else
    alert("Je n'ai rien à te dire...");
```

Mais cette solution devient particulièrement lourde à écrire et peu lisible lorsqu'on a beaucoup de possibilités.

C'est pourquoi il existe une autre méthode, plus adaptée.

Voyez plutôt l'exemple précédent, réalisé de cette manière :

Code : JavaScript

```
var nom = prompt("Entrez un nom d'animal");
switch(nom)
{
    case "chat":
        alert("Miaou !");
        break;
    case "chien":
        alert("Et PAF, le chien !");
        break;
    case "pingouin":
        alert("Bonjour, Tux");
        break;
    default:
        alert("Je n'ai rien à te dire...");
        break;
}
```



Cette syntaxe n'existe qu'à partir du **JavaScript 1.2**.

Autrement dit, les vieux navigateurs ne seront pas capables de la comprendre.

Rassurez-vous cependant, lorsque je dis "vieux navigateurs", il s'agit dans ce cas de *Internet Explorer 3.0* et de

Netscape Navigator 3.0 (et versions antérieures)... Ce qui permet donc de relativiser le problème de compatibilité.

Explications...

Le `switch` n'est en fait pas si différent d'un `if`...

Si on regarde la première ligne, dans un cas on a `if(truc)`, et dans l'autre, `switch(truc)`.

On pourrait dire qu'ils n'ont qu'une différence, mais qui est radicale. Dans le cas d'un `if`, le "`truc`" entre les parenthèses est un booléen (il est soit *vrai*, soit *faux*) : on a donc seulement deux cas possibles.

Pour un `switch`, c'est... une variable quelconque (dans notre cas, il s'agit d'une chaîne de caractères) : on a donc une infinité de cas possibles...

Comment ça marche ?

On commence par dire qu'on veut étudier la valeur d'une variable, grâce à `switch(variable)`, qu'on fait suivre d'accolades pour en délimiter le corps.

Pour ce qui est du corps (le "contenu" de notre `switch`), on utilise le même modèle pour chaque cas possible : **Code : JavaScript**

```
case VALEUR:
    INSTRUCTIONS
    break;
```

Le `case VALEUR` indique au navigateur qu'il doit commencer à partir de cette ligne si la variable vaut `VALEUR`. Le navigateur va donc exécuter toutes les instructions se trouvant dans les lignes qui suivent, jusqu'à trouver l'instruction `break`, qui va forcer le navigateur à sortir du `switch` (délimité par les accolades).

Il est possible de ne pas mettre ce `break` : dans ce cas, les instructions suivantes (celles du cas suivant) seront exécutées elles aussi. (Vous verrez cela dans l'exemple suivant.)

Dans le cas où il n'existe pas de `case` correspondant à la valeur de la variable, le code qui se trouve à partir de `default` (qui doit être placé après les autres cas) sera exécuté.

Pour résumer, voici
la syntaxe : **Code :**
JavaScript

```
switch(variable)
{
    case VALEUR_1:
        INSTRUCTIONS;
        break;
    case VALEUR_2:
        INSTRUCTIONS;
        break;
    /* etc... */
    default:
        INSTRUCTIONS;
        break;
}
```

Des boucles ? Pour quoi faire ?

Petit exercice

En guise de présentation, vous allez réaliser un petit exercice pour faire connaissance avec les boucles.

Votre mission, si toutefois vous l'acceptez : afficher la liste de tous les nombres plus petits que 10 (une boîte de dialogue par nombre).

Vous vous dites : "ça va être un jeu d'enfant". En effet...

Code : JavaScript

```
alert(1);
alert(2);
alert(3);
alert(4);
alert(5);
alert(6);
alert(7);
alert(8);
alert(9);
```

Mais c'est un peu long et très ennuyant 😞.

Pour éviter de s'ennuyer en programmant (on est quand même là pour s'amuser 😊), nous allons apprendre une autre méthode, beaucoup plus puissante, pour arriver au même résultat.

Utiliser une boucle

Ce qu'il faudrait, c'est pouvoir demander à l'ordinateur de "compter", et tant qu'il n'a pas atteint 10, d'afficher la valeur.

Eh bien c'est sur ce modèle que sont conçues ce qu'on appelle les **boucles** : on répète une action tant que une condition est satisfaite.

Pourquoi **les** boucles ? Car il en existe plusieurs différentes, qui s'utilisent dans des situations elles aussi différentes.

Boucle "while"

Syntaxe et fonctionnement

Commençons par la boucle la plus simple...

En voici la
synt
axe :
Cod
e :
Java
Scri
pt

```
while(condition)
{
    instructions
}
```

Fonctionnement de la boucle "while"

Tant que la condition est satisfaite, on **répète** les instructions.

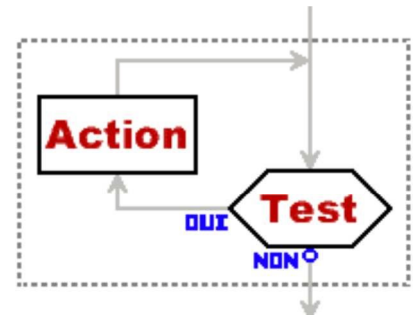


Si la condition n'est pas vérifiée au premier tour de boucle, alors les instructions ne seront pas exécutées.

Voici le fonctionnement exact de la boucle, illustré par le schéma ci-contre :

Si la condition est vérifiée,
--> Alors on exécute les instructions, et on
retourne à **#**, **--> Sinon** on passe à la suite.

Reprenons notre petit exercice...



On va créer une variable qui servira à "compter".

Tant que cette variable sera inférieure à 10, on **affichera** un message, puis on l'**incrémentera**.



Incrémenter une variable signifie augmenter sa valeur (en général, de 1).
Décrémenter signifie diminuer sa valeur.

Ce qui nous donne :

Code : JavaScript

```
var i = 1;      // on initialise le compteur
while(i < 10)   // tant que i<10 ...
{
    // ... on affiche un message
    alert(i);
    i++;
}
```



Si vous oubliez d'incrémenter votre variable, la condition sera toujours vraie !

La boucle ne s'arrêtera donc jamais...

Vous obtiendrez sûrement un message d'erreur de votre navigateur vous proposant d'interrompre le script.

Boucle "for"

Présentation de cette boucle

Un code qui revient souvent...

Les boucles **while** sont, comme vous venez de le voir, très "basiques" : la syntaxe est on ne peut plus simple !

Lors de leur utilisation, il est très fréquent d'utiliser un **compteur**, comme nous l'avons fait dans l'exemple précédent. Ce compteur, qui n'est rien d'autre qu'une variable (souvent appelée **i**, mais ce n'est pas une obligation, même si c'est pratique à écrire 😊), est ensuite incrémenté (ou décrémenté) à chaque tour de boucle.

On se retrouve souvent avec un code qui ressemble à ceci :

Code : JavaScript

```
var i;
i = 0; // initialisation
while(i < 10) // condition
{
    alert(i); // action
    i++;      // incrementation
}
```



Mais ça ne serait pas plus pratique avec une boucle qui regroupe tout ça ?

Eh bien justement si, et c'est le but de cette nouvelle boucle, que nous allons étudier dans les paragraphes suivants...

Une boucle qui va nous simplifier la vie

Voici la boucle **for**, qui regroupe tout ce que nous voulions !

Sa syntaxe ?

Code : JavaScript

```
for(initialisation ; condition ; incrementation
{
    instructions
}
```



Ce sont bien des points-virgules qui se trouvent entre les parenthèses après le **for**.

Attention à ne pas les confondre avec des virgules, qui sont utilisées pour séparer les arguments des fonctions.

Et notre code précédent se transforme comme par magie ! **Code : JavaScript**

```
var i; // on déclare notre variable
for(i=0; i<10; i++) // 3-en-1 :) initialisation + condition +
    incrémentation
    alert(i); // action
```



Il n'y a qu'une instruction à effectuer à chaque tour de boucle.

Les accolades sont donc devenues inutiles, c'est pourquoi je ne les ai pas mises.

Voilà qui est plus pratique 😊 !

Et encore plus pratique : on peut déclarer la variable **i** directement dans le **for**, comme ceci : **Code : JavaScript**

```
for(var i=0; i<10; i++)
    alert(i);
```

N'est-ce pas formidable ? 😊

Un peu de pratique



Les boucles **for** sont plus pratiques que les boucles **while**, mais **uniquement dans certains cas** ! Il est parfois plus judicieux d'utiliser une boucle **while**, notamment lorsqu'on ne retrouve pas cette structure "initialisation / condition / incrémentation" (nous en verrons des exemples par la suite).

Petit exemple

Vous devriez maintenant être capables de faire une boucle (on va utiliser **for**) qui fait la somme des nombres de 1 à 10 (inclus).

Je vous laisse faire, cela ne devrait pas vous poser de problème.

Vous avez terminé ?

Voici un corrigé :

Secret ([cliquez pour afficher](#))

```
var i;
    var somme = 0; // variable pour le resultat
    for(i=1; i<=10; i++)
        somme += i;    // on ajoute la valeur du
compteur à notre resultat
    alert(somme);
```

Code : JavaScript

Ce qu'il ne faut pas faire

Ce qui est rigolo avec les boucles **for**, c'est qu'on peut mettre tout plein de trucs dans les parenthèses. Et avec une indentation "excentrique", ainsi qu'avec des noms de variables qui ne veulent rien dire, on obtient des

résultats vraiment géniaux 🐱.

Voyez plutôt :

Code : JavaScript

```
var azerty=0,arezty;
for(arezty=0;azerty<10;arezty+= azerty=1 +azerty);alert(arezty);
```

Pas mal, vous ne trouvez pas ? 😊

Que fait ce code ? Ah... eh bien comme l'exemple juste avant, il fait la somme des nombres de 1 à 10...



Ce code est un excellent exemple... de ce qu'il **ne faut pas faire** ! À moins que vous aimiez les codes illisibles...

Voici quelques conseils concernant les boucles **for** :

- évitez de modifier la variable qui sert de compteur à l'extérieur des parenthèses du **for**, qui sont prévues pour ça.
- Inversement, évitez de mettre entre ces parenthèses du code qui ne concerne pas directement cette variable. Pour rappel, donnez des noms explicites et pas trop longs à vos variables (**i** ou **j**, parfois **cpt**, étant des noms classiques pour des compteurs).
- Pensez également à bien indenter votre code, et à mettre quelques commentaires, (uniquement) s'ils s'avèrent nécessaires.

Boucle "do ... while"

Le dernier type de boucle que nous allons aborder ressemble assez fortement à **while**.

La syntaxe

Sa syntaxe est la suivante :

Code : JavaScript

```
do
{
    instructions
}
while(condition);
```



Ici, le **while** est placé à la fin de la boucle, après les instructions qui le concernent : c'est pour ça qu'il est suivi d'un **point-virgule**.

Comment ça marche ?

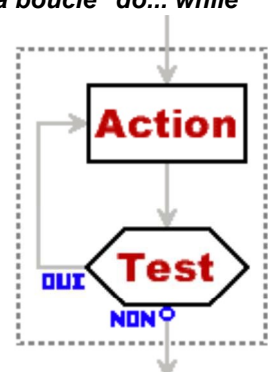
Fonctionnement de la boucle "do... while"

Vous avez pu remarquer que le **while** est placé à la fin de la boucle : c'est ce qui fait la différence avec la boucle "while" que nous avons vue plus haut.



Quelle différence ?

Eh bien le test est effectué après les instructions, ce qui veut dire que ces dernières seront exécutées **au moins une fois**, quelle que soit la condition (cf. le petit schéma ci-contre).



Un petit exemple...

Une des utilisations possibles : on demande des noms à l'utilisateur (avec **prompt()**) tant qu'il ne clique pas sur "Annuler" (ou qu'il ne laisse pas le champ vide).

Pour l'instant, on ne va pas s'occuper d'enregistrer tous les noms qu'il a tapés.



Rappel : la syntaxe est la suivante :

Code : JavaScript

```
var nom = prompt('Entrez un nom');
```

Si l'utilisateur clique sur "Annuler", la variable vaudra **null**.

Voici une manière de coder cet exemple : Code :

JavaScript

```
var saisie;
do
    saisie = prompt('Entrez un nom, ou cliquez sur Annuler pour quitter');
while(saisie != null && saisie != '');
```

ou bien, en "simplifiant" un peu :

Code : JavaScript

```
var saisie;
do
    saisie = prompt('Entrez un nom, ou cliquez sur Annuler pour quitter');
while(saisie); // ca revient au meme que dans l'exemple ci-dessus
```

Instructions de contrôle

Après avoir fait le tour des boucles (sans jeu de mot 😊), nous allons aborder un dernier point dans ce chapitre : les **instructions de contrôle**, qui ne sont rien d'autre que des instructions qui agissent sur les boucles.

break

Comment ça marche ?

La première, qui se prénomme **break** (ce qui signifie "casser" pour les non-anglophones), est très simple d'utilisation. Elle arrête immédiatement la boucle dans laquelle elle se trouve (le script continue normalement, en reprenant juste après la boucle en question).

Petit exemple :

Code : JavaScript

```
var i;
for(i=0; i<20; i++)
{
    if(3*i > 10)
        break;
    alert(3*i);
}
```

On va afficher les multiples de 3. Mais si un multiple dépasse 10, on s'arrête.

Un code moins bien "organisé"



Cette instruction est très pratique dans certains cas. Cependant, elle est à éviter autant que possible : il vaut mieux créer une condition plus complète pour votre boucle, plutôt que d'insérer des **if** avec des **break** dans toute la boucle.

L'exemple plus haut est correct du point de vue de la syntaxe, mais c'est exactement l'utilisation qu'il ne faut pas faire de **break**.

Voici une manière bien mieux structurée pour ce même exemple : Code : JavaScript

```
var i;
for(i=0; 3*i<=10; i++)
    alert(3*i);
```

Ici, tout ce qui concerne la boucle se trouve entre les parenthèses du **for**, ce qui rend le code bien mieux "organisé".

Cas de boucles imbriquées



Et si on met un **break** à l'intérieur d'une boucle, qui est elle-même à l'intérieur d'une autre boucle ?

Dans ce cas, il s'applique uniquement à la boucle la plus "imbriquée".

Dans l'exemple qui suit, le **break** s'applique seulement à la boucle avec la variable **j**. Code : JavaScript

```

var i, j;
for(i=0; i<5; i++)
{
    for(j=0; j<5; j++)
    {
        if(j == 2)
            break;
    }
}

```

continue

L'instruction de contrôle suivante, **continue**, s'utilise de manière quelque peu semblable.

Fonctionnement...

Lorsque l'instruction **continue** est rencontrée, toutes les instructions qui suivront seront ignorées, jusqu'à ce qu'on arrive à la fin des instructions de la boucle. La boucle continue ensuite normalement.

Pour simplifier, **continue** permet en quelque sorte de "sauter" certains tours de la boucle.

... et exemple

Vous allez tout de suite comprendre avec cet exemple. **Code : JavaScript**

```

var i;
for(i=-5; i<=5; i++)
{
    if(i == 0)
        continue;
    alert("L'inverse de " + i + " est " + 1/i);
}

```

Lorsque *i* vaut 0, on saute les instructions qui restent dans la boucle (ici, **alert()**), puis on reprend avec *i* = 1 (la valeur suivante).

On affiche donc le message pour *i* = -5, -4, -3, -2, -1, (on saute le 0), 1, 2, 3, 4 et 5.

Application

Bien, il est temps de mettre un peu en pratique ce que vous venez d'apprendre.

Exercice

Le script que vous allez créer est un script du SdZ... Oui, vous avez bien entendu ! C'est un script qui est utilisé sur ce site 😊 .

Le sujet

Vous avez tous utilisé au moins une fois le zCode...

Nous allons nous intéresser ici au bouton servant à insérer une liste à puces.

Notre script a pour rôle de demander le contenu de chaque puce à l'utilisateur, et de créer le zCode correspondant.



Rappel : zCode d'une liste à puces

Code : Zcode

```
<liste>
<puce>Contenu de la puce K/<puce>
<puce>Contenu de la puce X/<puce>
...
</liste>
```

Quelques consignes pour vous guider

On va demander le contenu de chaque puce grâce à **prompt()**.

C'est ici qu'on va devoir utiliser une boucle : on demande **tant que** l'utilisateur ne laisse pas le champ vide.

Au final, on va récupérer le zCode dans une variable.

On l'affichera grâce à **alert()** pour vérifier le fonctionnement du script.

À vous de jouer 😊 .

Correction

Je n'ai pas fait une, mais plusieurs corrections, plus ou moins bonnes.

Version 1 : "bof bof"

Voici ce qu'on peut obtenir après une petite réflexion :

Code : JavaScript

```
// initialisations
var zCode = '<liste>\n';
var saisie = '';

// boucle
do
{
    saisie = prompt('Contenu de la puce ?');
    if(saisie)
        zCode += "<puce>" + saisie + "</puce>\n";
} while(saisie);

// fin
zCode += "</liste>";
alert(zCode);
```

Plusieurs points ne sont pas très "propres".

Tout d'abord, si on ne remplit aucune puce, on se retrouve avec ceci : **Code : Zcode**

```
<liste>
</liste>
```

Certes, un "détail", mais si facile à corriger que c'est dommage de le laisser comme ça.

Autre remarque : à quelques lignes près, on voit deux fois le même test : **if(saisie)** et **while(saisie)**. Ce n'est jamais bien bon de se répéter (par exemple, si on doit modifier la condition, c'est un peu *relou* de le faire deux fois 😞).

Si on veut modifier ça, il va falloir enregistrer la saisie dans une variable, pour l'ajouter au zCode uniquement au tour de boucle suivant (une fois que le test de la boucle aura été fait)... Ce qui complique un peu l'affaire.

Version 1 améliorée

Voici le même code, après avoir modifié les problèmes soulignés ci-dessus.

Code : JavaScript

```
var zCode = '';
var saisie = '';
// variable "temporaire", pour le 2e point
var texte = '<liste>\n';

do
{
    // on ajoute le texte saisi au tour precedent
    zCode += texte;
    // on demande du texte et on enregistre dans la variable
    "temporaire"
    saisie = prompt("Contenu de la puce ?");
    texte = "<puce>" + saisie + "</puce>\n";
}while(saisie);

// le 1er "detail"
if(zCode == '<liste>\n')
    zCode = '';
else
    zCode += "</liste>";

alert(zCode);
```

Version 2 : tout-en-un

Une autre solution consisterait à *enregistrer* et à *tester* en même temps le texte saisi.

Comment ? Eh bien en plaçant l'affectation **dans** la condition.

En voici le principe :

Code : JavaScript

```
var saisie;
while(saisie = prompt("Texte"))
    // instructions
```

ce qui équivaut, je vous le rappelle, à ceci :

Code : JavaScript

```
var saisie;
while( (saisie=prompt("Texte")) == true )
    // instructions
```



Ne confondez pas le **=** (opérateur d'affectation) avec **==** (opérateur de comparaison) !

Il faut bien comprendre que cette condition (appelée à chaque tour de boucle) se compose de **deux** étapes :

- tout d'abord, l'instruction `saisie = prompt("Texte")`, que vous comprenez maintenant sans difficulté.
- Ensuite, le test, avec la nouvelle valeur de la variable modifiée ci-dessus.

Bref, pour revenir à notre exemple, ça nous donne ceci :

Code : JavaScript

```
// initialisation
var zCode = "<liste>\n";
var saisie;

// boucle
while(saisie = prompt("Contenu de la puce ?"))
    zCode += "<puce>" + saisie + "</puce>\n";

// finitions
if(zCode == '<liste>\n')
    zCode = '';
else
    zCode += "</liste>";

alert(zCode);
```

Pas mal, vous ne trouvez pas ? 😎

Un tableau, c'est quoi ?

Des variables numérotées

Quel est l'intérêt ?!

Pour reprendre l'exemple du jeu multijoueur, il serait bien pratique de pouvoir "numéroter" des variables. Ensuite, ça sera facile d'accéder au nom du *énième* joueur grâce aux numéros. C'est ce que nous permettent les tableaux : schématiquement, ils contiennent des "cases" portant des *numéros*, et on peut facilement accéder à une case (autrement dit, lire ou modifier le contenu) à partir de son numéro.

0	
1	
2	

Voici comment on peut "imaginer" un tableau



En JavaScript, comme dans de nombreux autres langages de programmation, on commence à compter à partir de 0 !

La première case portera donc le numéro 0, et la **énième case** le numéro **n-1**.

Le vocabulaire...

Plutôt que de parler de "cases" et de "numéros", employons dès maintenant les termes exacts.

- Le "numéro" s'appelle un **indice**. On entend également parler de **clé**, ou **key** en anglais. Comme nous le verrons par la suite, les indices peuvent aussi être des chaînes de caractères (un mot ou une expression) : on parle alors de **tableau associatif** (il associe une valeur, celle de la "case", à un mot).
- Le contenu d'une "case" s'appelle un **élément** du tableau.

Au lieu de parler de la "case portant le numéro *i*" (ça fait un peu tiercé 🤪), on va donc dire l'**élément d'indice *i***.

Comment ça marche ?

Maintenant qu'on peut se comprendre (entre geeks 😊), allons-y.

Un tableau, on l'enregistre... dans une variable.

Mais comme on a appris à déclarer (créer), initialiser et modifier une variable, on va maintenant apprendre à créer, initialiser et modifier le tableau que contient notre variable (en plus de s'occuper de la variable elle-même).

Créer un tableau

Pour créer un tableau, on utilise **new Array()** (qui signifie en anglais "**nouveau tableau**").



Il y a toujours des parenthèses après **Array**, même si elles sont vides. Nous allons voir, quelques lignes plus loin, ce qu'on peut y mettre.

C'est marrant ces parenthèses, c'est comme pour les fonctions... (on en reparle dans quelques chapitres, voulez-vous ? 😊)

Pour créer un tableau et l'enregistrer dans une variable (sinon ça ne sert pas à grand chose de l'avoir créé...), on fait comme on en a l'habitude : **Code : JavaScript**

```
var table = new Array();
```

Étudions de plus près ce qui se passe :

- l'ordinateur crée un tableau en mémoire (on lui en a donné l'ordre
- avec **new**). Il va également créer une variable dans sa mémoire (on
- lui a demandé avec **var**).

Mais schématiquement, voilà ce qui va se passer : en fait, on ne va pas "mettre" le tableau dans la variable (ça ne rentrerait pas, une variable, c'est trop petit). On va **dire à la variable où se situe notre tableau dans la mémoire de l'ordinateur** (c'est beaucoup plus facile comme ça).

Je vous explique ceci pour que vous puissiez comprendre ce qui se passe lorsque vous "copiez" un tableau...



Code : JavaScript

```
var toto = new Array();
var t = toto;
```

Ces deux variables contiennent donc "l'emplacement" du tableau créé : elle désignent donc le même tableau ! Si on le modifie avec **toto**, les changements seront valables aussi pour **t**.

Initialiser un tableau

Revenons à la création de tableaux : on a appris à créer un tableau **vide**.

Mais il est également possible de créer un tableau contenant certaines valeurs. Pour cela, on utilise justement les parenthèses après **Array**, en précisant dans l'ordre, et en les séparant par des virgules, les valeurs de notre tableau. **Code : JavaScript**

```
var noms = new Array("Pierre", "Paul", "Jacques");
var best_scores = new Array(2.5, 4.7, 3.14);
```

Lire ou modifier une valeur

Ce qui va vraiment nous servir par la suite, c'est pouvoir lire ou modifier les éléments d'un tableau.

Pour accéder à un élément, on utilise **tableau[indice]**, où **indice** est... l'indice de l'élément 🤖. Pour ajouter un nouvel élément, on modifie simplement sa valeur, fait comme s'il existait déjà.

Exemple :

Code : JavaScript

```
var table = new Array("Pierre", "Paul", "Jacques");
alert("La seconde case vaut : " + table[1]); // on lit l'element
d'indice 1
table[1] = "Toto"; // on le modifie
table[3] = "Dupont"; // on cree un nouvel element
```

Au final, ce tableau contiendra donc "Pierre" (indice 0), "Toto" (1), "Jacques" (2) et "Dupont" (3).

Pour un tableau associatif, on crée un tableau vide, et on associe "manuellement" (une par une) toutes les valeurs, en utilisant une chaîne de caractères en tant qu'indice, comme ceci : **Code : JavaScript**

```
var scores = new Array();
scores["Toto"] = 142;
scores["Pierre"] = 137;
```

Je ne parle pas de la lecture / modification des données, c'est exactement pareil qu'avec un tableau "numéroté".

Des tableaux sans queue ni tête...

En JavaScript, vous avez peut-être constaté qu'on a pas mal de "souplesse" (l'ordinateur ne vient pas vous embêter si vous oubliez un point-virgule, etc.).

Eh bien avec les tableaux, on peut faire à peu près ce qu'on veut :

- ajouter autant d'éléments qu'on veut faire un tableau numéroté ET
- associatif faire un tableau contenant à la fois des nombres, des
- chaînes de caractères, etc.

Un exemple :

Code : JavaScript

```
var t = new Array(5, "Bonjour", 290, 1.414, "Toto", false, 9, true);
t["txt"] = "Bienvenue à toi !";
t["est_majeur"] = true;
t["pi"] = 3.14;
```

Bref, ça devient un peu n'importe quoi...



C'est à vous de faire en sorte d'avoir des tableaux organisés...

Le but est bien sûr de pouvoir réutiliser les données stockées : dans l'exemple précédent, cela me paraît très très difficile.

Plus de contrôles

Maintenant que vous savez utiliser chaque élément de votre tableau, prenons un peu de recul pour s'occuper du tableau lui-même...

Je ne vais pas vous présenter toutes les fonctions qu'on peut appliquer à un tableau (nous y reviendrons plus tard si besoin est), mais simplement vous expliquer comment récupérer sa **longueur** (chose qui sera vraiment indispensable pour la suite), et vous présenter une fonction (déjà existante, rassurez-vous 😊) pour **trier** vos tableaux.

Longueur d'un tableau ?

Comme je l'ai dit, connaître la longueur d'un tableau (*grosso modo* le nombre d'éléments qu'il contient) est souvent fort utile (pour le parcourir notamment, comme nous le verrons dans la suite de ce chapitre). Pour un tableau nommé **monTableau**, on accède à sa longueur grâce à **monTableau.length**.



Cela ne fonctionne que si les indices de **monTableau** sont des nombres. Dans le cas de tableaux associatifs, les cases dont les indices sont des chaînes de caractères ne sont pas comptées.

L'écriture précédente est un peu nouvelle : **length** est en fait une variable qui "**appartient**" à **monTableau** (attention, je n'ai pas dit que c'était un *élément* du tableau), variable qui contient justement la longueur du tableau 😊.

Bref, reprenez surtout que **monTableau.length** est la longueur de "**monTableau**".

Un exemple, pour mieux comprendre :

Code : JavaScript

```
var table = new Array("Pierre", "Paul", "Jacques");  
alert(table.length);  
table[5] = "Toto";  
alert(table.length);
```

Si vous testez ce code, vous obtiendrez 3, puis 6. Dans le premier cas, pas de problème.

Mais dans le cas suivant, il faut également compter les cases vides situées avant la dernière valeur. Le tableau contient :

0. "Pierre"
1. "Paul"
2. "Jacques"
3. (*rien*)
4. (*rien*)
5. "Toto"

D'où une longueur de 6.

Vraiment facile, *isn't it* ? 😊

Trier un tableau

En restant dans une difficulté aussi élevée (😊), voici... euh... pour l'instant, appelons ça une fonction... qui permet de trier un tableau (par ordre croissant) grâce à `monTableau.sort()` (*to sort* signifie *trier* en anglais).



Le tri est **irréversible** !
Une fois trié, il est impossible de récupérer votre tableau dans l'ordre initial.

Reprenons l'exemple précédent (avec des chaînes de caractères, le tri se fait... par ordre alphabétique) :

Code : JavaScript

```
var table = new Array("Pierre", "Paul", "Jacques");  
table[5] = "Toto";  
table.sort();
```

Le tableau contient désormais :

0. "Jacques"
1. "Paul"
2. "Pierre"
3. "Toto"

Pfiou, quelle difficulté 😊 .

Exploiter un tableau

Eh bien je crois qu'on a tout vu... sauf l'essentiel 😊 .

On sait accéder à un élément particulier, et on sait comment connaître la longueur de notre tableau... Il ne nous reste plus qu'à accéder à **tous** les éléments de notre tableau, un par un !

Lire un tableau

Commençons par le plus simple : que peut-on faire en parcourant un tableau ? Tout d'abord, le lire !

On va donc créer une fonction, qui prend en argument un tableau, et qui va nous "lire" le tableau dans une chaîne de caractères.



On va devoir parcourir tout notre tableau... Mais comment faire ?

On doit en fait répéter une action sur chacun des éléments du tableau. On va donc utiliser... une *boucle*.

Boucle "for" classique

Il est facile de parcourir un tableau numéroté à l'aide d'une boucle **for** : en effet, on veut accéder à tous les `tableau[i]`, avec `i` allant de **0** à `tableau.length - 1` (ce qui nous fait bien nos `tableau.length` éléments).

Voici donc une fonction qui retourne, sous forme de chaîne de caractères, le contenu du tableau : **Code : JavaScript**

```
function lire1(tab)
{
    var chaine = "Le tableau contient : "
    for(var i=0; i<tab.length; i++)
        chaine += "\n" + i + " -> " + tab[i];
    return chaine;
}
```



Dans la pratique, on utilise généralement des boucles comme celle-ci (au moins, on est sûrs de n'avoir aucun problème de compatibilité, et c'est (souvent) aussi bien que la boucle que je vais vous présenter).

Une boucle "for" spéciale...

La boucle précédente est parfaite pour des tableaux numérotés, mais si vous avez fait l'essai avec un tableau associatif, vous avez pu vous rendre compte que ça n'affiche rien 😞.

Rassurez-vous, il existe une variante de la boucle **for** qui nous permet de parcourir un tel tableau 😊.

La syntaxe de la boucle est la suivante :

Code : JavaScript

```
for(var indice in tableau)
```

Cette boucle va parcourir un par un tous les indices du tableau. Et une fois qu'on a les indices, on a les valeurs qui vont avec 😊.

De plus, si on laisse des cases vides (dans le cas d'un tableau numéroté), elle ne seront pas parcourues par cette boucle.

Voici donc une nouvelle fonction pour lire un tableau :

Code : JavaScript

```
function lire2(tab)
{
    var chaine = "Le tableau contient :";
    for(var indice in tab)
        chaine += "\n" + indice + " -> " + tab[indice];
    return chaine;
}
```



Au risque de me répéter, on utilise le plus souvent des tableaux numérotés.
Il est donc inutile d'utiliser cette boucle dans ce cas-là.

Exploiter un tableau

Maintenant que vous savez lire un tableau, vous n'aurez pas de mal à le parcourir pour y effectuer les opérations que vous voulez (comme mettre toutes les cases à zéro).

Parcours d'un tableau classique

La structure est exactement la même : on utilise une boucle **for** (version 1 de préférence 😊) pour parcourir toute les cases du tableau, et on y fait ce que l'on veut.

Un petit exemple ?

On va créer une fonction qui calcule la moyenne d'un tableau contenant des nombres (et aucune case vide).



La moyenne de plusieurs nombres, c'est (la somme de ces nombres) divisée par (le nombre de nombres). Avec **deux** nombres x et y, c'est donc $(x+y) / 2$.

Code : JavaScript

```
function moyenne(tableau)
{
    var n = tableau.length; // nombre de valeurs
    var somme = 0;
    for(i=0; i<n; i++)
        somme += tableau[i];
    return somme/n; // somme divisee par le nombre de valeurs
}
```

Fonctions et variables

Enregistrer une fonction dans... une variable ?!

Nous avons, dans les chapitres précédents, étudié séparément les variables et les fonctions. Mais en fait, une fonction, ce n'est pas si différent que ça d'une variable...

Rappel

Histoire de vous rafraîchir un peu la mémoire, je vous rappelle qu'une fonction se déclare dans l'en-tête de la page, de cette manière :

Code : JavaScript

```
function toto(arg1, arg2)
{
    // code
    return valeur;
}
```

Quelques explications succinctes pour se remettre dans le bain :

- Le mot-clé **function** indique... qu'on *déclare* une fonction 😊 **toto** est le *nom* de notre fonction
- **arg1** et **arg2** sont les *arguments* de cette fonction, séparés par une **virgule**, et mis entre des
- **parenthèses** qui sont obligatoires (même s'il n'y a aucun argument).
On place le code de la fonction dans un *bloc d'instructions*, délimité par des accolades.
- On *renvoie* éventuellement une valeur (c'est la valeur que "prendra" la fonction quand on l'appellera).

Créer une fonction dans une variable

Eh bien figurez-vous qu'il existe une autre manière de créer une fonction, en utilisant une **variable**. Voyez plutôt :

Code : JavaScript

```
var toto = function(arg1, arg2)
{
    // code
}
```



Cette fonction est exactement la même que celle au dessus (à l'exception du **return valeur**).

Cette fois-ci :

- On a déclaré notre variable, à l'aide du mot-clé **var** (comme nous avons vu dans le chapitre à ce sujet).
- On a donné à notre variable le nom de **toto**.

On lui a *affecté* (on lui a donné pour valeur) une fonction, prenant deux paramètres nommés **arg1** et **arg2**, et exécutant le code entre les accolades.



Mais alors, si ce sont les mêmes fonctions, quel est l'intérêt d'utiliser cette nouvelle méthode plutôt que l'ancienne ?

C'est en fait un autre point de vue, assez différent : en déclarant une fonction de cette manière, on voit clairement qu'on enregistre notre fonction dans une variable.

On peut donc très facilement :

- créer une fonction locale ou globale
- re-déclarer une fonction (autrement dit, modifier son code), simplement en
- modifiant la variable créer des tableaux de fonctions etc.
-

Exemple : un tableau de fonctions

Pour vous montrer, créons un tableau de fonctions.

Pour cela, on commence par déclarer notre tableau, comme on l'a appris auparavant, puis on associe à chaque élément (chaque case) une fonction, comme nous venons de le voir :

Code : JavaScript

```
var operation = new Array();
operation["add"] = function(x,y){ return x+y; };
operation["soustr"] = function(x,y){ return x-y; };
operation["mult"] = function(x,y){ return x*y; };
operation["div"] = function(x,y){ return x/y; };
operation["moy"] = function(x,y){ return (x+y)/2; };
```

On fait un essai : on demande deux nombres à l'utilisateur, ainsi que le nom de la fonction à appliquer (add, soustr, mult, div ou moy).

Code : JavaScript

```
var a = parseFloat( prompt("Premier nombre ?") );
var b = parseFloat( prompt("Deuxieme nombre ?") );
var fct = prompt("Fonction a appliquer ?");

var resultat = operation[fct](a,b);
alert("Resultat : " + resultat);
```

C'est quand même plus joli que d'effectuer cinq if à la suite, pas vrai ? 😊

Rappel : portée des variables

Puisqu'on parle de variable, un petit rappel concernant la portée de celles-ci ne sera pas inutile...

En JavaScript, on distingue les variables **globales** (accessibles n'importe où dans votre code) des variables **locales**.

Les variables locales déclarées dans une fonction ne seront accessibles (et "visibles") qu'à l'intérieur de cette fonction. On peut donc sans problème utiliser le même nom pour des variables locales de deux fonctions différentes.



Comme je l'ai déjà dit, il faut éviter d'utiliser des variables globales.

Comment ça marche ?

- On déclare une variable *locale* à l'aide de **var**, dans le *bloc d'instructions* (qui est généralement "matérialisé" par des accolades) dans lequel elle doit être accessible.
- Il y a deux façons de créer une variable *globale* : soit en la déclarant en dehors de tout bloc d'instructions (tout au début du code JS), soit on l'utilise sans la déclarer, comme si elle l'avait déjà été (on parle alors de *déclaration implicite*).

Un exemple :

Code : JavaScript

```
var a; // on declare une variable globale
function test(argument)
{
    var resultat = 5*argument + 2;
    a = argument; // modification de la variable globale
    b = resultat; // declaration implicite d'une variable globale
    return resultat;
}
```

- Lorsque la fonction n'a pas été appelée, il y a une seule variable globale : **a**.
- Quand on appelle la fonction, celle-ci modifie cette variable globale, et on en déclare (implicitement) une autre : **b**.
- Une fois la fonction exécutée, on peut toujours accéder aux variables **a** et **b**, car elles sont globales. Cependant, on ne peut accéder à la variable **resultat** uniquement à l'intérieur de la fonction, car c'est une variable locale.

Voilà, ce n'était finalement pas compliqué 😊.

Les arguments facultatifs : nombre fixé d'arguments

Plutôt que de vous faire un long discours théorique, voyons ensemble le fonctionnement.

Comment ça se passe lorsqu'on appelle une fonction ?

Prenons une fonction comme celle-ci :

Code : JavaScript

```
function f(x,y)
{
    // code de la fonction
}
```

Elle a pour nom **f**, et prend deux paramètres, **x** et **y**.



Que se passe-t-il si on l'appelle en ne précisant qu'un seul argument, comme ci-dessous ?

Code : JavaScript

```
f(5);
```

Essayons de comprendre ce qui va se passer lors de cet appel :

- deux *variables locales*, **x** et **y**, vont être créées (ce sont les *arguments*)
- la variable **x** va être *initialisée* avec la valeur 5 (c'est la valeur qu'on a donnée lors de l'appel pour le premier argument) mais **y** ne sera pas initialisée, car on n'a pas précisé le second
- argument

Autrement dit, il va se passer quelque chose
comme ceci : **Code : JavaScript**

```
var x, y;
x = 5;
// code de la fonction
```

Une variable non initialisée !

Quels sont les symptômes ?

On se retrouve donc face à une variable *déclarée*, mais... qui n'a pas de valeur !

Poursuivons donc nos essais : que va-t-il se passer avec un tel code ? 🤖

Code : JavaScript

```
var y;
alert(y);
```

On voit alors s'afficher : **undefined**.



En effet, **undefined** est un mot-clé signifiant que la variable est déclarée, mais qu'on ne lui a jamais donné de



valeur. Si la variable n'avait pas été déclarée, on n'aurait pas eu de message du tout : le script aurait été interrompu (l'ordinateur ne peut pas deviner qu'il s'agit d'une variable, puisqu'on ne l'a pas déclarée).

Comment ça se soigne ?

On peut effectuer un test comme celui-ci pour vérifier si la variable **y** est définie :

Code : JavaScript

```
if (y == undefined)
```

Et on peut aussi créer une fonction qui renvoie **true** ("vrai", cf. le chapitre sur les conditions) si la variable est définie, **false** sinon.

Code : JavaScript

```
function isDefined(variable)
{
    return (variable != undefined);
}
```



Si l'écriture vous perturbe, sachez que c'est équivalent à ceci :

Code : JavaScript

```
function isDefined(variable)
{
    if (variable == undefined)
        return false;
    else
        return true;
}
```

Mais c'est "bête" d'écrire un tel code : "si *condition* est vrai, alors renvoie vrai, sinon renvoie faux"...



Notez que la valeur booléenne d'une variable non initialisée est **false**. Ainsi, en effectuant le test **if(y)**, si la variable **y** n'est pas initialisée, c'est comme si elle était initialisée avec **false**.

Exemple

Tout ça pour dire qu'on peut créer des fonctions pour lesquelles certains arguments sont facultatifs.

Un exemple : une fonction **dist(a,b)** qui calcule la distance entre **a** et **b** (autrement dit, la valeur absolue de **b-a**). Mais si on appelle **dist(a)**, on veut que la fonction calcule la valeur absolue de **a** (la distance entre **a** et **0**).

Comment faire ?

Pour le calcul de la distance, pas de problème, on teste si **a** est plus grand que **b**, et on calcule **a-b** ou **b-a** selon le cas.

En revanche, la nouveauté se trouve dans le second point : si on ne donne qu'un seul paramètre, il va falloir initialiser **b** à **0**.

Code : JavaScript

```
function dist(a,b)
{
    // on initialise b a 0 si besoin
    if(b == undefined)
        b = 0;
    // on continue ensuite normalement
    if(a > b)
        return a-b;
    else
        return b-a;
}
```

Au début de la fonction, il faut donc vérifier si les arguments facultatifs ont été précisés : si ce n'est pas le cas, il va falloir leur donner une valeur par défaut.

Les arguments facultatifs : nombre illimité d'arguments



C'est bien gentil tout ça, mais ça ne permet pas de réaliser la fonction d'addition dont on a parlé quelques chapitres plus tôt, qui additionnait **tous** les arguments qu'elle avait !

En effet, on avait parlé d'une fonction qui devait pouvoir s'utiliser comme ceci : **Code : JavaScript**

```
addition(12, 5); // nous donnerait 17
addition(21, 4, 15, 11, 6); // nous donnerait 57
// etc. avec autant de nombres qu'on veut
```

Où sont stockés tous ces arguments ?

Ce qui serait pratique, c'est que les arguments soient numérotés... un peu comme... un tableau !

Justement, il est possible de récupérer un tableau qui contient tous les arguments de la fonction. Du coup, le problème semble tout de suite plus simple 😊 .

Un tableau contenant tous les arguments

Partons de la fonction d'addition qu'on veut réaliser. On peut déjà commencer par la créer comme ceci : **Code : JavaScript**

```
function addition()  
{  
    // corps de la fonction  
};
```

Maintenant, il nous faut récupérer le tableau contenant les arguments.

Ce tableau est `addition.arguments` : ce sont les arguments de la fonction `addition`, stockés sous forme de tableau.



Ce tableau contient tous les arguments qu'on a donnés à la fonction lors de son appel. Ainsi, en appelant `addition(21, 4, 15, 11, 6)` on aurait (schématiquement) :

Code : JavaScript

```
addition.arguments = new Array(21, 4, 15, 11, 6);
```

Petite parenthèse : le point rouge `.` ne vous rappelle-t-il pas `monTableau.length`, qui est la longueur de `monTableau` ?

Le tour est joué !

Eh bien on a gagné !

Non ? Je vous sens sceptiques...

Commençons par créer une variable contenant ce tableau, qu'on va appeler `nb` (les arguments sont les nombres à additionner) :

Code

:

JavaScript

```
var nb = addition.arguments;
```



`nb` et `addition.arguments` sont un seul et même tableau ! Si on le modifie, il sera modifié sous les deux noms.

Et maintenant, il ne reste plus qu'à tout additionner (qui dit tableau, dit boucle...), ce qui nous donne cette fonction : **Code : JavaScript**

```
function addition()
{
    var nb = addition.arguments;
    var somme = 0;
    for(var i=0; i<nb.length; i++)
        somme += nb[i];
    return somme;
};
```

Il ne nous reste plus qu'à tester (le moment le plus stressant en programmation 😊) :

Code : JavaScript

```
alert( addition(12, 5) );
alert( addition(21, 4, 15, 11, 6) );
```

Qui nous affiche bel et bien 17, puis 57 😊.

Notez qu'on peut très bien appeler **addition()**, sans aucun paramètre : le résultat est 0.

Un autre exemple

Pour que ce soit bien clair pour vous, étudions un nouvel exemple : une fonction, prenant encore une fois autant de paramètres (des nombres) qu'on veut, et qui renvoie le plus grand de ces nombres.

Cette fois-ci, à la différence de l'addition, on ne sait pas trop quoi renvoyer s'il n'y a aucun paramètre... On va donc créer une fonction comme ceci :

Code : JavaScript

```
function maxi(m) { };
```



Dans le tableau d'arguments, la valeur de `m` sera également prise en compte, dans la première case (la case numéro 0).

On va ensuite parcourir le tableau, et enregistrer la plus grande valeur "lue" dans une variable (on peut utiliser la variable `m`) : cette variable contiendra donc le plus grand nombre parmi ceux qu'on a déjà parcourus. **Code : JavaScript**

```
function maxi(m)
{
  var nb = maxi.arguments;  // on a donc m = nb[0]
  for(var i=1; i<nb.length; i++)
    if(nb[i] > m)  // si l'argument est plus grand que le maximum...
      m = nb[i];  // ... alors c'est le maximum
  return m;
}
```

Et, encore une fois, on peut tester :

Code : JavaScript

```
var n = maxi(7, 3);
alert(n);

var p = maxi(2, 8, 4, 1, 9, 4);
alert(p);
```

Remarquez au passage que cette dernière fonction peut s'avérer utile 😊.
