

-
- Commençons par le commencement : la POO, c'est quoi ?
Derrière ce sigle qui signifie *Programmation Orientée Objet* se cache en fait une manière de programmer, qui
- va se baser sur... des objets ! 😊

Bon, reformulons la question :



En informatique, qu'est-ce qu'un objet ?

C'est un concept qu'on se représente très bien dès qu'on l'utilise, mais qui est difficile à expliquer clairement avec des mots. Pour cette raison, nous allons partir d'un exemple...

Sommaire du chapitre :



- [Qu'est-ce qu'un objet ?](#)
- [Comment ça s'utilise ?](#)
- [On récapitule le vocabulaire...](#)
- [Q.C.M.](#)

Qu'est-ce qu'un objet ?

Petite mise en situation

Prenons l'exemple d'un vendeur d'ordinateurs qui souhaiterait obtenir (en JS, même si ce n'est pas le langage le plus adapté) la liste des ordinateurs qu'il vend, ainsi que leurs caractéristiques. Le problème se pose de savoir comment enregistrer toutes ces données...

C'est là qu'interviennent les objets : ils vont en fait nous permettre de regrouper dans une même "structure" toutes les données communes à un même ordinateur.

Encore mieux : on va même pouvoir définir et appliquer des fonctions directement à cette structure, pour en récupérer certaines données (les caractéristiques de l'ordinateur notamment), pour les modifier, pour les comparer, etc.

Un objet, c'est quoi ?

D'abord, une classe...

En POO, on va définir des **classes d'objets**, qui regroupent :

- des variables, appelées **attributs**, qui caractériseront l'objet : ce sont les bases de notre "structure". Un attribut peut très bien être un objet : on dira alors que c'est un **sous-objet** de notre classe. des
- fonctions, appelées **méthodes**, qui permettront (entre autre) d'agir sur cet objet.



En JS, certains objets possèdent également des *événements*, que l'utilisateur pourra utiliser pour déclencher un script.

C'est un peu particulier, c'est pourquoi nous en parlons séparément.

Dans notre cas, on va définir notre classe "ordinateur" par les attributs :

- **cpu** : la fréquence du processeur **disqueDur** :
- la capacité du disque dur **ram** : la quantité de
- mémoire vive **carteGraphique** : le nom de la
- carte graphique

On pourra définir les méthodes suivantes :

- **description()** qui retourne (sous forme de phrase) les caractéristiques de l'ordinateur. Du style : *"Cet ordinateur est équipé d'un processeur cadencé à xx GHz, d'un disque dur d'une capacité de ... blabla..."*
- **caracteristiques()** : renvoie une liste en HTML des composants avec leur(s) caractéristique(s)
- **plusDeRamQue(x)** : renvoie **true** si l'ordi possède plus de x Mo de Ram, sinon **false** etc. selon les besoins / votre
- imagination 😊 .

Ensuite, des instances...

Bref : on a notre **classe d'objet**. C'est en quelque sorte un modèle, grâce auquel on va construire tous nos objets "ordinateur" : ces derniers sont appelés des **instances** (comprenez : des "exemplaires", des "répliques") de cette classe d'objets. Les instances s'enregistrent dans des variables.
Par exemple, on pourra créer l'instance "ordiDeJean", dont les attributs auront des valeurs qui correspondent à l'ordinateur de Jean.



Lorsqu'on parle d'**objet**, on désigne soit une **classe d'objet**, soit une **instance** : il faut bien en être conscient. Dans un premier temps, nous vous le préciserons, pour que vous ne mélangiez pas tout. Mais ensuite, ce sera à vous de distinguer les deux sens du mot "objet", selon son contexte...

Des exemples de classes d'objets

Voici quelques autres objets que l'on peut créer, qui auront des utilisations différentes...

On peut imaginer, comme l'a fait [M@teo21](#) dans son cours sur le C/C++, un objet "personnage" représentant un personnage dans un jeu de rôle. Parmi les attributs, on trouvera la vie, la mana, la force, etc. et parmi les méthodes, on aura attaquer, boire une potion, changer d'arme, etc.

Pour changer de domaine, on peut imaginer des objets mathématiques, tels que les fractions : on pourra les additionner, les soustraire, les multiplier, les diviser entre elles, simplifier une fraction, etc. L'intérêt, c'est de manipuler des nombres (y compris les résultats) sous forme de fraction.

Bref, suivant nos besoins, on va pouvoir créer toutes sortes d'objets. Vous verrez, c'est pratique 😊 .

Deux points de vue...

L'un des (nombreux ?) avantages de la POO, c'est qu'elle sépare le travail en deux : d'un côté, le concepteur de la classe d'objet, et de l'autre, l'utilisateur.

- Le concepteur va définir entièrement la classe d'objet en question (il va définir un certain nombre d'attributs et programmer des méthodes associées à cette classe).

Une fois la classe créée, il pourra l'améliorer (en rajoutant des méthodes ou en optimisant le code existant). • L'utilisateur va pouvoir utiliser cette classe d'objet dans ses programmes, en créant des instances de l'objet, et en utilisant les méthodes fournies par le concepteur.

Il n'a pas à se soucier de tout le code de la classe (c'est justement le boulot du concepteur) : c'est ça qui est génial 😊.

Dans un premier temps, nous allons adopter le second point de vue, pour découvrir les nombreux objets qui existent déjà en JS (ce qui n'est pas une mince affaire, croyez-moi 😊). Puis ensuite seulement nous allons nous attaquer à la création de nos propres objets, ce qui est un réel jeu de construction...



Avant d'attaquer la pratique, je vous conseille de lire la [première partie du chapitre d'introduction à la POO](#) (partie intitulée "Des objets... pour quoi faire ?") du [cours de C/C++](#) de [M@teo21](#).

Comment ça s'utilise ?

Assez parlé, voyons plutôt comment s'utilisent des objets. Ca vous aidera à comprendre ce que c'est, et à quoi ça sert. Mais en fait, vous connaissez déjà une classe d'objet : les tableaux, dont le nom est **Array** ! Son utilisation diffère un peu des objets "classiques", mais on y retrouve néanmoins les éléments typiques des objets.

Créer une instance

Pour créer une instance, on utilise simplement le mot-clé **new**.

Code : JavaScript

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

Dans cet exemple, nous avons donc créé une instance "monTableau" de la classe Array.

Dans les parenthèses, on spécifie des paramètres propres à cette instance de l'objet. Par exemple, pour l'objet "ordinateur", ça peut donner quelque chose comme ceci :

Code : JavaScript

```
var ordiDeJean = new Ordinateur("2 GHz", "120 Go", "512 Mo", "FX 2100");
```



L'objet (la classe) "ordinateur" n'a pas encore été défini, on ne sait donc pas trop s'il faut indiquer les caractéristiques sous forme de chaînes de caractères, de nombres, ...

Ce sera à nous de le choisir au moment de la création de cet objet, et il nous faudra l'indiquer au vendeur, pour qu'il l'utilise correctement.

Les attributs

Pour accéder à **un attribut** (une variable) **d'un objet** (d'une instance), on utilise le nom de l'objet, suivi d'un point, puis du nom de l'attribut, comme ceci :

objet.attribut

Un exemple avec l'attribut **length** de l'objet "Array" :

Code : JavaScript

```
monTableau.length
```

Que faire avec les attributs ? Et bien ce sont des variables qui "appartiennent" à l'objet, on peut donc les lire et/ou les modifier (bien que dans certains cas, comme ici, ça ne serve pas à grand chose de les modifier).

Ainsi, si Jean rajoute de la RAM à son ordinateur, on va pouvoir modifier l'attribut correspondant :

Code : JavaScript

```
ordiDeJean.ram = "1024 Mo";
```



Il est également possible d'accéder à un attribut de cette manière :

Code : JavaScript

```
ordiDeJean["ram"];
```

Eh oui, les éléments d'un tableau associatif ne sont en fait que des attributs qu'on rajoute à notre tableau...

Les méthodes

Pour exécuter une méthode d'un objet (d'une instance), on utilise la même syntaxe que pour accéder à un attribut, sans oublier les parenthèses (puisque une méthode est une fonction).

Un exemple avec la méthode **sort()** de l'objet **Array** :

Code : JavaScript

```
monTableau.sort();
```

Avec notre classe "ordinateur", on va pouvoir créer une méthode qui retournera une description sous forme de caractère. On l'utilisera ainsi :

Code : JavaScript

```
var message = ordiDeJean.description();  
alert(message);
```

Destruction d'un objet



Petit rappel : comment ça se passe dans l'ordinateur lorsqu'on crée un tableau, avec `var toto = new Array()` ?
Citation : chapitre sur les tableaux

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).

Bref, revenons à nos objets. Dans cet exemple, notre tableau se retrouve "perdu" dans la mémoire de l'ordinateur :
Code : JavaScript

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

Tout ça pour vous dire que JS possède un système nommé *ramasse-miettes*, ou encore *garbage collector* pour les anglophones, qui se charge de détruire automatiquement tout ce qui se retrouve ainsi "perdu". Ainsi, lorsqu'il n'y aura plus aucun "lien" vers un objet, ce dernier sera détruit.

Si on veut détruire un tableau, il nous suffit donc de modifier la valeur de la variable qui contient l'adresse du tableau. C'est ce qu'on a fait dans l'exemple ci-dessus : le tableau se retrouve "perdu" en mémoire, et le ramasse-miettes le détruira 🧑🔧.



Le mot-clé `null` est généralement utilisé pour désigner "rien".

On récapitule le vocabulaire...

Avant de terminer ce chapitre, récapitulons le vocabulaire utilisé à propos de la POO, pour qu'il soit clair pour vous.

- **Classe** : c'est un **modèle**, une **définition** d'une "structure", qui va posséder des *attributs* et des *méthodes*.
Exemple : une voiture, qui possède une couleur, une marque, un nom, etc.
- **Instance** : c'est un **exemplaire** d'une classe.
Exemple : la voiture de mon oncle, qui a la couleur "rouge", de marque "Pijo" et de nom "Pijo 404".
- **Objet** : selon le sens, il s'agit d'une **classe** ou d'une **instance**.
- **Attribut** : c'est une **variable** qui "appartient" à un objet.
Exemple : l'attribut "couleur" pour la voiture.
- **Méthode** : c'est une **fonction** propre à une classe. Elle va pouvoir agir sur l'instance à laquelle on l'applique.
Exemple : repeindre la voiture.

Q.C.M.

Parmi ces affirmations, l'une est fausse.
Laquelle ?

- ☐ La POO permet de regrouper des variables et fonctions dans ces fameux objets.
- ☐ La POO est un autre langage de programmation.
- ☐ La POO permet de séparer le travail en deux : le concepteur, et l'utilisateur des objets.
- ☐ POO signifie Programmation Orientée Objet. C'est une façon de programmer qui se base sur des objets.

De manière simple, on pourrait résumer comme ceci le fonctionnement d'un objet : ***on crée des exemplaires à partir d'un modèle.***

Avec les mots exacts, il faut dire :

- ☐ On crée des instances à partir d'une classe.
- ☐ On crée des classes à partir d'une instance.

Un objet peut posséder des [...], des [...], des [...] et des [...], mais pas :

- ☐
- ☐
- ☐
- ☐
- ☐ d'évènement
- ☐ d'attribut de
- ☐ méthode de partie
- ☐ de sous-objet

Quelle syntaxe utiliser pour accéder à **un élément d'un objet** ?

- ☐ element.objet
- ☐ objet.element
- ☐ element(objet)
- ☐ objet(element)
- ☐ objet->element
- ☐ element from objet

Correction !

[Statistiques de réponses au QCM](#)



L' objet "Array"

Sommaire

Les classiques

- **length**
- **join(separateur)** : retourne le contenu sous forme de chaîne de caractères **sort()** : trie le tableau.

Moins classiques

- **concat(t)** : concatène avec le tableau **t** **reverse()** :
- inverse l'ordre des éléments **slice(debut, fin)** : extrait
- une partie du tableau.

Les piles et les files

- **pop()** : retire et renvoie le dernier élément du tableau
- **push(x)** : ajoute l'élément **x** à la fin du tableau **shift()** :
- retire et renvoie le premier élément du tableau **unshift(x)**
- : ajoute l'élément **x** au début du tableau.

Les classiques **length**

Description

Cet attribut contient la longueur du tableau.

Exemple

Code : JavaScript

```
var t = new Array("toto", "bob");  
alert(t.length);  
t[4] = "homer";  
alert(t.length);
```

Ce qui affichera 2, puis 5.

En effet, dans le second cas, le tableau contiendra : "toto", "bob", [rien], [rien], "homer". Le dernier élément est donc le cinquième.

sort()

Description

Trie le tableau par ordre croissant (par ordre alphabétique pour des chaînes de caractères).
Ne renvoie rien : c'est le tableau avec lequel on appelle cette méthode qui est modifié.



En triant un tableau, l'ordre des éléments est modifié (normal 😊).
Une fois trié, il est impossible de retrouver l'ordre initial.

Exemple

Code : JavaScript

```
var t = new Array("toto", "bob", "homer");  
t.sort();
```

Au final, le tableau contient, dans cet ordre : "bob", "homer", "toto".

join(separateur)

Description

Revoie tous les éléments du tableaux séparés par **séparateur**, sous forme de chaîne de caractères. Si aucun séparateur n'est précisé, c'est la virgule qui est utilisée. Utile lorsqu'on teste un script.

Exemple

Code : JavaScript

```
var t = new Array();  
for(var i=0; i<9; i++)  
    t[i] = i+1;  
alert(t.join('-'));
```

Ceci affichera **1-2-3-4-5-6-7-8-9** .

Moins classiques

concat(t)

Description

Renvoie un tableau contenant les éléments du tableau, à la fin desquels sont ajoutés les éléments du tableau **t**. Il est possible de mettre plusieurs arguments : **monTableau.concat(t1, t2, t3, t4)**.

Exemple

Code : JavaScript

```
var t1 = new Array(1,3,5);  
var t2 = new Array(2, 4);  
var t = t1.concat(t2);
```

t contient **1,3,5,2,4**.

reverse()

Description

Inverse l'ordre des éléments (le tableau d'origine est modifié).

Exemple

Code : JavaScript


```
var t = new Array();  
for(var i=0; i<10; i++)  
    t[i] = i;  
  
t.reverse();  
t.reverse();
```

Au début (juste après la boucle), **t** contient 0,1,2,3,4,5,6,7,8,9.

Au milieu (entre les deux **reverse**), il contient 9,8,7,6,5,4,3,2,1,0. À la fin, il contient à nouveau 0,1,2,3,4,5,6,7,8,9.

L'objet "Math"

La visite guidée se poursuit avec un objet assez utile, dont le doux nom va en ravir certains, mais en effrayer d'autres.

Voici l'objet... **Math**. 

Il regroupe entre autres des fonctions mathématiques, plus ou moins utiles. Voyons tout cela sans plus tarder.



Selon votre niveau en maths, il est possible que certaines de ces fonctions vous soient inconnues. Mais ne vous inquiétez pas pour autant : ce sont généralement des fonctions qui sont peu utilisées en dehors d'un usage purement mathématique.

Sommaire

Fonctions basiques

Arrondir

Trigonométrie

Autres fonctions usuelles

Inclassables

Fonctions indépendantes de l'objet "Math"

Nos propres fonctions

Fonctions basiques **abs(x)**

Description

Renvoie la valeur absolue de **x**.

Exemple

La distance entre deux nombres (qui est simplement la valeur absolue de leur différence) :

Code : JavaScript

```
function distance(x,y)
{
    return Math.abs(x-y);
}
```

Si on veut tester : ce code affichera 4, puis 2.

Code : JavaScript

```
alert(distance(1,5));
alert(distance(4,2));
```

min(x, y)

Description

Renvoie le plus petit nombre parmi x et y.

Exemple

Ce code affichera 3 :

Code : JavaScript

```
alert (Math.min (3, 5) ) ;
```

max(x, y)

Description

Renvoie le plus grand nombre parmi x et y.

Exemple

Ce code affichera 5 :

Code : JavaScript

```
alert (Math.max (3, 5) ) ;
```

Arrondir



Ces fonctions arrondissent à l'unité.

À la fin de ce chapitre, nous allons créer une fonction pour arrondir avec une meilleure précision.

round(x)

Description

Renvoie x arrondi à l'entier le plus proche.

Exemple

Ce code affichera 3, puis -1 : **Code**
: JavaScript

```
alert(Math.round(3.14));  
alert(Math.round(-1.234));
```

floor(x)

Description

Renvoie **x** arrondi à l'entier inférieur ou égal.

Exemple

Ce code affichera 3, puis -2 : **Code**
: JavaScript

```
alert(Math.floor(3.14));  
alert(Math.floor(-1.234));
```

ceil(x)

Description

Renvoie **x** arrondi à l'entier supérieur ou égal.

Exemple

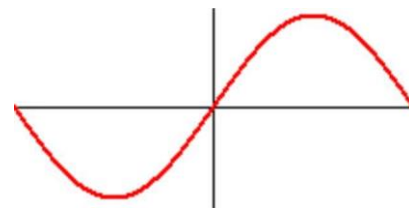
Ce code affichera 4, puis -1 : **Code**
: JavaScript

```
alert(Math.ceil(3.14));  
alert(Math.ceil(-1.234));
```

Trigonométrie **sin(x)**

Description

La fonction **sinus**.
L'angle est à indiquer en radians.



Exemple

Ce code affichera 0 :

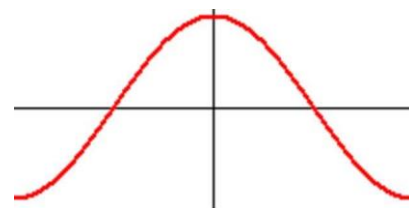
Code : JavaScript

```
alert(Math.sin(0));
```

cos(x)

Description

La fonction **cosinus**.
L'angle est à indiquer en radians.



Exemple

Ce code affichera 1 :

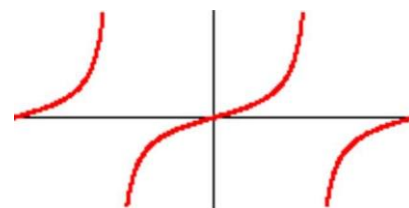
Code : JavaScript

```
alert(Math.cos(0));
```

tan(x)

Description

La fonction **tangente**.
L'angle est à indiquer en radians.



Exemple

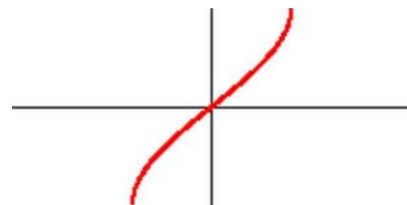
Ce code affichera 0 :

Code : JavaScript

```
alert (Math.tan(0));
```

asin(x)**Description**

La fonction **arc sinus**, ou Arcsin.

**Exemple**

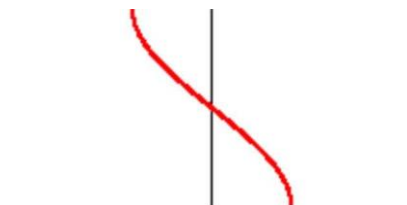
Ce code affichera la valeur de $\pi/2$:

Code : JavaScript

```
alert (Math.asin(1));
```

acos(x)**Description**

La fonction **arc cosinus**, ou Arccos.

**Exemple**

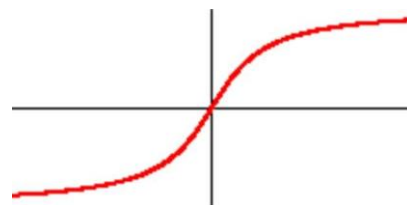
Ce code affichera la valeur de $\pi/2$:

Code : JavaScript

```
alert (Math.acos(0));
```

atan(x)**Description**

La fonction **arc tangente**, ou Arctan.



Exemple

Ce code affichera la valeur de $\pi/4$:

Code : JavaScript

```
alert(Math.atan(1));
```

Autres fonctions usuelles

exp(x)

Description

La fonction exponentielle (en base e).

Exemple

Code : JavaScript

```
alert("exp(-1) vaut " + Math.exp(-1));
```



log(x)

Description

La fonction logarithme népérien.

Remarque

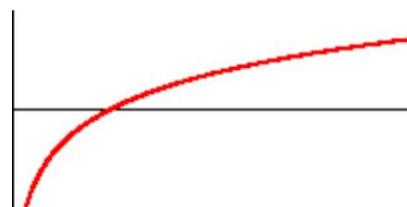
Notez que la fonction logarithme décimal n'est pas définie, ce sera à vous de le faire en cas de besoin. Il en est de même pour un logarithme en base b quelconque.

Vous pouvez pour cela utiliser les constantes **LN10**, **LN2**, **LN10E** (logarithme décimal de e) et **LN2E** (logarithme en base 2 de e) de notre objet **Math**.

Exemple

Code : JavaScript

```
var LN3 = Math.log(3);  
alert("ln(3) vaut " + LN3);
```



Si on veut définir notre logarithme décimal :

Code : JavaScript

```
function log10 (x)
{
    return Math.LOG10E * Math.log (x) ;
}
```

pow(x,y)

Description

Retourne x à la puissance y .

Exemple

Si on veut définir la fonction **cube** à partir de **pow** : **Code : JavaScript**

```
function cube (x)
{
    return Math.pow (x, 3) ;
}
```

sqrt(x)

Description

La fonction **radical** (parfois appelée *racine carrée* par abus de langage).



Exemple

Cet exemple affichera 3 :

Code : JavaScript

```
alert (Math.sqrt (9) ) ;
```

Voici la norme euclidienne d'un vecteur du plan :

Code : JavaScript

```
function norme(x,y)
{
    return Math.sqrt(x*x + y*y);
}
```

Inclassables

PI

Description

La fameuse constante **pi**.

Attention, le nom s'écrit bien en *majuscules* (comme c'est souvent le cas pour des constantes).

Exemple

Code : JavaScript

```
alert("Pi vaut environ " + Math.PI);
```

random()

Description

Renvoie un nombre aléatoire de l'intervalle **[0;1[** ; autrement dit, un nombre compris entre **0** (inclus) et **1** (exclus).



Cette fonction est difficilement utilisable ainsi.

À la fin de ce chapitre, nous allons créer une fonction pour générer un **entier** aléatoire compris entre les deux arguments de cette fonction.

Exemple

Code : JavaScript

```
alert("Un nombre aléatoire : " + Math.random());
```

Fonctions indépendantes de l'objet "Math"



Ces fonctions ne font pas partie de l'objet **Math** !
Cependant, comme elles se rapprochent du thème de ce chapitre, nous en parlons ici.

isNaN(x)

Description

Renvoie **true** si **x** n'est pas un nombre, **false** sinon.
Utile avant d'utiliser les fonctions mathématiques.

Exemple

Code : JavaScript

```
var x = prompt("Entrez un nombre");
if(isNaN(x))
    alert("Ce n'est pas un nombre");
else
    alert("Le carré de " + x + " est " + x*x);
```

isFinite(x)

Description

Renvoie **true** si **x** est un nombre que JS peut traiter, **false** sinon (par exemple, pour un nombre trop grand pour être supporté par JS).

Exemple

On va créer un nombre trèèès grand. 😊

Code : JavaScript

```
var x = Math.pow(111,222);
if(isFinite(x))
    alert("111 puissance 222 vaut " + x);
else
    alert("C'est trop grand !");
```

parseInt(str)

Description

Convertit la chaîne de caractères **str** en nombre entier.
Renvoie NaN (Not a Number) si la conversion échoue.

Exemple

Code : JavaScript

```
var a = parseInt("12h30");  
var b = parseInt("-3.14");  
var c = parseInt("Euh...");
```

Dans cet exemple, les variables vaudront respectivement **12** (la conversion s'arrête au "h"), **-3** (car on convertit en entier, donc on s'arrête au .) et **NaN**.

parseFloat(str)

Description

Convertit la chaîne de caractères **str** en nombre à virgule.
Renvoie NaN (Not a Number) si la conversion échoue.

Exemple

Code : JavaScript

```
var a = parseFloat("-3.14");  
var b = parseFloat("1.5m");  
var c = parseFloat("3,14");
```

Dans cet exemple, les variables vaudront respectivement **-3.14**, **1.5** (la conversion s'arrête au "m") et **3** (il faut utiliser le point, et non la virgule).

Nos propres fonctions

Comme annoncé dans ce chapitre, nous allons créer quelques fonctions supplémentaires (à partir de celles existantes).

arrondir(x, n)

Ce qu'on veut...

On veut arrondir **x**, mais en gardant **n** chiffres après la virgule.

Par exemple :

Code : Autre

```
arrondir(3.14, 1) = 3.1  
arrondir(2.718, 2) = 2.72  
arrondir(1.234, 0) = 1
```

Comment faire ?

La fonction dont nous disposons pour arrondir un nombre est **Math.round**.

Notre souci, c'est qu'elle arrondit à l'unité ; autrement dit, à l'endroit où se situe la virgule, comme ceci :

3.14159 -> **3**

Ce qu'on veut, c'est arrondir "plus loin que la virgule", comme dans cet exemple :

3.14159 -> **3.14**

En fait, ce qu'il faudrait, c'est **déplacer la virgule**, pour pouvoir arrondir avec **Math.round**, et ensuite remettre la virgule au bon endroit...



D'accord, mais comment on s'y prend pour déplacer une virgule ? 🤔

Citation : cours de primaire

En multipliant par 10, on déplace la virgule d'un cran vers la droite.

Et pour la déplacer de **n** crans, on multiplie par 10...00 (un "1" suivit de **n** "0"), autrement dit, par **10** puissance **n**.

Pour déplacer vers la gauche, on divise par ce même nombre.

Voilà, on a toutes les cartes en mains. 😊 *Allons-y*

On crée notre fonction sans problème :

Code : JavaScript

```
function arrondir(x, n)
```

Calculons maintenant **10** puissance **n**, qui nous servira à décaler la virgule. Code :

JavaScript

```
var decalage = Math.pow(10, n);
```

Pour notre exemple **arrondir(3.14159, 2)**, cette variable vaudra donc **100**.

On commence par décaler notre virgule à droite, en multipliant par ce nombre :

Code : JavaScript

```
x *= decalage;
```

Dans l'exemple, `x` vaudra donc **314.159**.

Ensuite, on arrondit :

Code : JavaScript

```
x = Math.round(x);
```

Désormais, `x` vaut **314**.

Et on remet notre virgule à sa place, en la décalant à gauche :

Code : JavaScript

```
x /= decalage;
```

On a bien `x = 3.14` 😊.

Il ne reste plus qu'à renvoyer ce résultat.

Code : JavaScript

```
return x;
```

Voici notre fonction

Notre fonction complète est donc la suivante :

Code : JavaScript

```
function arrondir(x, n)
{
    var decalage = Math.pow(10, n);
    x *= decalage;
    x = Math.round(x);
    x /= decalage;
    return x;
}
```

randomInt(mini, maxi)

Ce qu'on veut...

On veut un nombre entier aléatoire entre **mini** et **maxi**.

Comment faire ?

Pour l'instant, on ne sait récupérer des nombres aléatoires que dans l'intervalle **[0;1]**... on va donc en prendre un.

On le multiplie par : **maxi + 1 - mini** Il se retrouve donc dans **[0 ; maxi+1-mini[**.

On lui ajoute **mini**.

Il est alors maintenant dans **[mini ; maxi+1[**.

Et on termine en arrondissant à l'entier inférieur, qui est au moins **mini** et au plus **maxi** (car strictement inférieur à **maxi+1**).

Voici notre fonction

Code : JavaScript

```
function randomInt(mini, maxi)
{
    var nb = mini + (maxi+1-mini)*Math.random();
    return Math.floor(nb);
}
```

Et voilà, les mathématiques en JS n'ont maintenant plus de secrets pour vous. 😊



L'objet "Date"

L'objet **Date** va, comme son nom l'indique, nous permettre d'avoir accès à la date, mais également à l'heure. Il peut servir à l'afficher dans un coin de la page (et non pas une énorme horloge horrible qui suit la souris 😞), mais aussi à chronométrer le temps de visite d'une page ou bien le temps d'exécution d'un script JS.

Pour récupérer l'heure et la date actuelle, rien de compliqué : il suffit de créer une instance de cet objet, elle sera initialisée avec l'heure courante (qui est l'heure du PC du visiteur). **Code : JavaScript**

```
var date = new Date();
```

Maintenant qu'on a notre objet, on peut lire chaque propriété (l'heure, l'année, ...) via les fonctions présentées plus bas.

Il est aussi possible d'initialiser l'objet à une autre date, qu'on indiquera sous forme de paramètres, comme ceci... **Code : JavaScript**

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

Si les derniers paramètres ne sont pas précisés, ils sont mis à 0 :
Code : JavaScript

```
var uneAutreDate = new Date(annee, mois, jour);
```

Il est également possible de l'initialiser en donnant le nombre de millisecondes depuis le 01/01/1970, 0h00 : **Code : JavaScript**

```
var derniereDate = new Date(millisecondes);
```

Passons maintenant aux méthodes, pour savoir que faire de notre nouvel objet. Sommaire du chapitre :



- [Sommaire](#)
- [Méthodes diverses](#)
- [Récupérer et modifier l'heure](#)

Sommaire

Méthodes diverses

- **getTime()** : retourne le nombre de millisecondes écoulées depuis le 01/01/1970
- **setTime(x)** : modifie la date en précisant le nombre x de millisecondes écoulées depuis le 01/01/1970
- **getTimezoneOffset()** : retourne, en minutes, le décalage horaire avec le méridien de Greenwich.

Récupérer et modifier l'heure

- **getPropriété()** pour lire la propriété
- **setPropriété(x)** pour modifier la propriété
- **getUTCPropriété()** pour lire la propriété (au méridien de Greenwich)
- **setUTCPropriété(x)** pour modifier la propriété (au méridien de Greenwich).
-

Les propriétés sont...

- **FullYear** : l'année (à 4 chiffres)
- **Month** : le mois
- **Date** : le jour du mois
- **Day** : le jour de la semaine.
Il vaut 0 pour dimanche ; 1 pour lundi [...] ; 6 pour samedi.
- **Hours** : l'heure (de 0 à 23)
- **Minutes** : les minutes
- **Seconds** : les secondes
- **Milliseconds** : les millisecondes 🤖.

Méthodes diverses **getTime()**

Description

Cette méthode renvoie le nombre de millisecondes écoulées entre le 01/01/1970 et la date. Utile pour chronométrer un script, par exemple.

Exemple

Justement, chronométrons un petit script...

Code : JavaScript

```
var debut = new Date();

// le script ici
var i=0
while(i < 1234567)
    i++;
// fin du script

var fin = new Date();
tempsMs = fin.getTime() - debut.getTime();
alert("Le script a mis " + tempsMs/1000 + " secondes.");
```

Les **new Date()** servent à récupérer la date et l'heure à laquelle ils sont appelés. On en place donc un au début et un à la fin du script, et on fait la différence des **getTime()**.

setTime(x)

Description

Modifie la date, en indiquant le nombre de millisecondes écoulées entre le 01/01/1970 et la date. C'est un très bon format pour stocker une date (dans un cookie, par exemple) : on enregistre uniquement le nombre de millisecondes (cf. `getTime()`), qui permettra ensuite de retrouver la date ; quand besoin est, on demande à JS de re-crée notre date à partir de ce nombre.

Exemple

Code : JavaScript

```
var uneDate = new Date();
uneDate.setTime(1234567890);
```

On peut ensuite récupérer le jour, l'heure, etc. avec les méthodes présentées ci-après.

getTimezoneOffset()

Description

Retourne le décalage horaire, en minutes.

Exemple

Code : JavaScript

```
var date = new Date();
var decalage = date.getTimezoneOffset();
if(decalage > 0)
    alert("Vous avez " + decalage + " minutes d'avance sur les gens de Greenwich.");
```

Récupérer et modifier l'heure

On va enfin lire l'heure !

Il est possible de reconstituer l'heure grâce aux méthodes qui nous permettent d'accéder au nombre de minutes, de secondes, etc.

Comme il n'y a pas grand chose à dire de ces fonctions, nous vous proposons une fonction qui affiche l'heure.

Quelques petites remarques cependant...

Noms des jours et des mois

Comme il a été dit, les jours de la semaine et les mois sont retournés sous forme de nombres.



Mais si on veut leur nom, comment faire ?

Pour cela, il est judicieux d'utiliser un tableau, ce qui est d'autant plus simple que les nombres commencent justement à 0.

On aura donc :

Code : JavaScript

```
var jours = new Array("dimanche", "lundi", "mardi", "mercredi", "jeudi",  
"vendredi", "var mois = new Array("janvier", "fevrier", "mars", "avril", "mai",  
"juin", "juillet",
```

Attention aux minutes !

Un autre point "sensible" : lorsque les minutes sont inférieures à 10. En effet, afficher **1h1** n'est vraiment pas élégant... Ceci est aussi valable pour les jours et les mois : on n'affiche pas **1/1/1970**. Il faudra donc penser à rajouter un zéro devant (sous forme de chaîne de caractères).

Créons notre fonction

On va maintenant créer deux fonctions supplémentaires retournant la date pour l'une et l'heure pour l'autre, en français.

Pour la date, on va renvoyer une chaîne de caractères de la forme "mardi 5 avril 1988". **Code : JavaScript**

```
function dateFr()  
{  
    // les noms de jours / mois  
    var jours = new Array("dimanche", "lundi", "mardi", "mercredi", "jeudi",  
    "vendredi", "var mois = new Array("janvier", "fevrier", "mars", "avril", "mai",  
    "juin", "juil  
    // on recupere la date  
    var date = new Date();  
    // on construit le message  
    var message = jours[date.getDay()] + " "; // nom du  
    jour      message += date.getDate() + " "; // numero du  
    jour      message += mois[date.getMonth()] + " "; // mois  
    message += date.getFullYear();    return message;  
}
```

Pour l'heure, on se contentera d'un "1h01".

Code : JavaScript

```
function heure ()
{
    var date = new Date();
    var heure = date.getHours();
    var minutes = date.getMinutes();
    if(minutes < 10)
        minutes = "0" + minutes;
    return heure + "h" + minutes;
}
```

Maintenant, il ne vous reste plus qu'à créer un petit champ de formulaire pour afficher l'heure.



Pour qu'elle soit actualisée automatiquement, il faut utiliser `setInterval(fonction, delai)` qui appellera **fonction** toutes les **delai** millisecondes.

C'est une méthode de l'objet **window**, elle sera abordée dans le chapitre sur ce dernier.

Et un objet de plus, un ! 😊



Les chaînes de caractères, nommées **String** en anglais, sont elles-aussi des objets, qui sont de plus très souvent utilisés.

Découvrons sans plus tarder les nombreuses méthodes à utiliser, pour pouvoir en faire ce qu'on veut 😊.

Nous les avons classé en trois parties.

- Tout d'abord, celles permettant de traiter des caractères séparément (pour récupérer la position d'un caractère dans une chaîne, ou inversement, y lire le caractère qui a une position donnée).
- Puis celles qui utilisent des chaînes de caractères entières (extraire une sous-chaîne, en concaténer plusieurs, etc.).
- Enfin, des méthodes "rechercher / remplacer", qui utilisent les **regex**.
Ce thème et ces méthodes feront l'objet du chapitre suivant.

Sommaire du chapitre :

Hello
World !

- [Sommaire](#)
- [Opérations sur les caractères](#)
- [Opérations sur les chaînes](#)
- [Rechercher et remplacer](#)

Sommaire

Opérations sur les caractères

- **length** : longueur de la chaîne (nombre de caractères) **charAt(i)** :
- retourne le *i*^{ème} caractère
- **indexOf(str)** : retourne la position de **str** dans la chaîne (-1 si elle n'est pas trouvée)
- **lastIndexOf(str)** : idem, mais renvoie la position de la dernière occurrence de **str**
- **toLowerCase()** : retourne la chaîne en minuscules **toUpperCase()** : retourne la chaîne en majuscules

Opérations sur les chaînes

- **concat(str)** : retourne la chaîne concaténée avec **str** **split(str)** : retourne, sous forme de tableau, les
- portions de la chaînes délimitées par **str** **substring(debut,fin)** : extrait une sous-chaîne, depuis la
- position **debut** (inclusive) à **fin** (exclusive). **substr(debut,i)** : extrait une sous-chaîne, depuis la position
- **debut**, en prenant *i* caractères

Chaînes et expressions régulières

Ces fonctions sont détaillées dans le chapitre consacré aux expressions régulières.

- **match(regex)**
- **search(regex)**
- **replace(regex, str)**

Opérations sur les caractères

length, le retour

Description

Notre fameuse propriété **length** contient la longueur de la chaîne de caractère, autrement dit, le nombre de caractères dont cette chaîne est composée.

Exemple

Code : JavaScript

```
var chaine = "J'aime le JavaScript";
alert(chaine.length);
var longueur = "JavaScript".length;
alert(longueur);
```

Cet exemple affichera successivement **20** (nombre de caractères dans la phrase "J'aime le JavaScript", espaces compris) et **10** (il y a bien 10 lettres dans le mot "JavaScript").

charAt(i)

Description

La méthode **charAt(i)** renvoie le caractère qui se trouve à la position **i** de la chaîne.



Pour rappel, le premier caractère porte le numéro **0**.
Le dernier a donc le numéro **length-1** !

Exemple

Ceci affichera la lettre S :

Code : JavaScript

```
alert("JavaScript".charAt(4));
```

Voici une fonction qui sépare tous les caractères par un espace.

Pour cela, on parcourt la chaîne, et on recopie chaque caractère en ajoutant un espace.

Le test affichera : *V i v e - l e - J a v a S c r i p t*. **Code : JavaScript**

```
function separer(chaine)
{
    var sortie = '';
    for(var i=0; i<chaine.length; i++)
        sortie += chaine.charAt(i) + ' ';
    return sortie;
}
alert(separer('Vive-le-JavaScript'));
```

indexOf(str) et lastIndexOf(str)

Description

Ces deux méthodes sont assez utiles.

- **indexOf(str)** renvoie la position de **str** dans la chaîne de caractères.
Si **str** apparaît plusieurs fois, c'est la position de la première occurrence qui est renvoyée.
- **lastIndexOf(str)** renvoie de même la position de **str**.
La différence, c'est que si **str** apparaît plusieurs fois, c'est cette fois ci la position de la dernière occurrence qui est renvoyée.



Si ces méthodes ne trouvent pas **str**, elles retournent -1.

On peut donc les utiliser pour savoir si une chaîne de caractère contient une autre chaîne.



On peut préciser, en deuxième argument, la position à partir de laquelle doit commencer la recherche. Par défaut, il s'agit de **0** pour **indexOf** (le premier caractère), et de **length-1** pour **lastIndexOf** (le dernier caractère, car la recherche se fait en partant de la fin).

Exemple

On recherche *script* dans *JavaScript*.

Ceci affichera **Non**, en raison de la majuscule !

Code : JavaScript

```
var chaine = "JavaScript";
if(chaine.indexOf("script") == -1)
    alert("Non");
else
    alert("Oui");
```

Exemple : vérifier la vraisemblance d'une adresse e-mail

On s'occupe ici de la présence d'un @ et d'un point

On récupère, grâce à nos deux fonction, la position du premier @ et du dernier point.

La fonction doit renvoyer **true** quand le @ est après le premier caractère (le n°0), et quand le point est au moins deux caractères après le @.

L'adresse la plus courte qui est considérée comme correcte est donc : **x@x.x**

Code : JavaScript

```
function verifMail(email)
{
    var position_at = email.indexOf('@');
    var position_dot = email.lastIndexOf('.');
    return (position_at > 0 && position_dot >= position_at + 2);
}

if(verifMail("mon.adresse@bon.net"))
    alert("On dirait une adresse valide");
else
    alert("L'adresse e-mail n'existe pas");
```

toLowerCase() et toUpperCase()

Description

JavaScript met à disposition deux méthodes permettant d'en changer la casse d'une chaîne de caractères :

- **toLowerCase()** : renvoie toute la chaîne en minuscules
- **toUpperCase()** : renvoie toute la chaîne en majuscules



Ces méthodes sont très utiles dans le traitement des chaînes de caractères. Elle permettent par exemple de comparer des chaînes sans tenir compte de la casse.

Exemple

Cet exemple affichera *"salut tout le monde"*.

Code : JavaScript

```
var chaine = 'SaLut TouT LE mOnDe';
var chaineMin = chaine.toLowerCase();
alert(chaineMin);
```


Comparons maintenant deux chaînes : le message sera "*Ces chaînes sont les mêmes, aux majuscules près*". **Code : JavaScript**

```
var chaine1 = "JavaScript";
var chaine2 = "javascript";
if(chaine1 == chaine2)
    alert("Ces chaînes sont exactement les mêmes");
else if(chaine1.toLowerCase() == chaine2.toLowerCase())
    alert("Ces chaînes sont les mêmes, aux majuscules près");
else
    alert("Ces chaînes sont différentes");
```

Opérations sur les chaînes

concat(str), l'opérateur +

Description

L'opérateur + permet de concaténer deux chaînes : **chaine+str** est la chaîne constituée de **chaine** et de **str** mises bouts-à-bouts.



L'écriture précédente est strictement équivalente à **chaine.concat(str)**

Elle est bien plus lisible et intuitive, c'est pourquoi il est très rare d'utiliser **concat**.

Exemple

On affiche la chaîne "JavaScript", construite par concaténation (en utilisant les deux façons de faire).

Code : JavaScript

```
var chaine1 = 'Java';
var chaine2 = 'Script';
alert(chaine1 + chaine2);
alert(chaine1.concat(chaine2));
```

split(str)

Description

La méthode **split(str)** permet de découper une chaîne de caractères, en "coupant" aux endroits où la chaîne comporte **str**.

Les "morceaux" découpés sont renvoyés sous forme de tableau.

Cette méthode peut se révéler assez utile.



Cette méthode peut aussi s'utiliser avec une regex.
(abordé dans le chapitre suivant)

Exemple

Code : JavaScript

```
var chaine = 'www.siteduzero.com';  
var tableau = chaine.split('.');
```

Ici, la chaîne **chaine** est découpée selon les points.
Il y aura donc trois parties :

- **tableau[0]** qui contient **www**
- **tableau[1]** qui contient **siteduzero**
- **tableau[2]** qui contient **com**

substring(debut,fin)

Description

substring(debut,fin) renvoie la portion de chaîne délimitée par les positions **debut** (incluse) et **fin** (exclue).

Si **fin** n'est pas précisée, la portion est extraite de **debut** jusqu'à la fin de la chaîne.



La méthode nommée **slice** fait exactement la même chose que **substring**.

Exemple

Cet exemple affichera "Javascript".

Code : JavaScript

```
var chaine = 'Vive le Javascript !';  
var sousChaine = chaine.substring(8,18);  
alert(sousChaine);
```

substr(debut,nombre)

Description

substr(debut, i) renvoie la portion de chaîne commençant à la position **debut** et contenant **i** caractères.
Si **i** n'est pas précisée, ou s'il y a moins de **i** caractères, la portion est extraite jusqu'à la fin de la chaîne.



Ne confondez pas **substring** et **substr** !

Exemple

Cet exemple affichera "Javascript".

Code : JavaScript

```
var chaine = 'Vive le Javascript !';  
var sousChaine = chaine.substr(8,10);  
alert(sousChaine);
```

Rechercher et remplacer



Les méthodes présentées dans cette sous-partie utilisent les **regex**. Cette notion (assez complexe) est présentée dans le chapitre suivant.

match(regex)

match(regex) applique l'expression régulière **regex** à la chaîne, et renvoie le résultat sous forme de tableau.

replace(regex, str)

replace(regex, str) remplace les sous-chaînes satisfaisant l'expression régulière **regex** par le motif de remplacement **str**.

search(regex)

search(regex) renvoie la position de la première sous-chaîne satisfaisant l'expression régulière **regex**. Renvoie -1 s'il n'y a aucun résultat.



Pour rechercher une simple chaîne de caractère, cf. la méthode **indexOf**.

Dans le prochain chapitre, nous restons dans le même sujet, mais en abordant les **regex** : c'est un outil vraiment puissant pour travailler sur les chaînes de caractères.



Qu'est-ce qu'on apprendra à faire grâce à ces fameuses regex ?

On peut d'ores et déjà citer la vérification d'un numéro de téléphone ou d'une adresse e-mail (plus exactement, on vérifie s'il *semble* valide).

L'objet "String"

<http://www.siteduzero.com/tutoriel-3-8106-1-objet-string.html>

Ça permet également d'extraire des informations contenues dans une phrase qu'a écrit l'utilisateur (en particulier, récupérer des chiffres, tel qu'un âge, dans une phrase comme "*j'ai 99 ans*").

Et vous verrez, les regex sont extrêmement pratique dans ce genre de situations.



Syntaxe en JavaScript

Pour débiter avec les annexes, partons du commencement : la syntaxe du JavaScript.



Ce chapitre n'a pas pour but d'expliquer, mais de rappeler les points principaux.

- Si vous ne vous souvenez plus comment se déclare une fonction, c'est ici 😊 .
- En revanche, si vous ne savez plus ce qu'est un tableau et dans quels cas on l'utilise, le chapitre du cours traitant de ce sujet est là pour vous l'expliquer en détail, mais ce n'est certainement pas ici que vous trouverez les explications complètes.

On va donc passer en revue toutes les bases enseignées dans la première partie de ce tutoriel. En quelque sorte, on va rappeler les règles de la "grammaire" du JavaScript.

Au début de chaque sous-partie sera indiqué le nom du chapitre du cours traitant de ce sujet : c'est à lui que vous devrez vous référer si vous désirez plus d'explications.

Accrochez-vous, le chemin sera court, mais très garni 😊 .

Sommaire du chapitre :



- [Les boîtes de dialogues](#)
- [Où se place le JS ?](#)
- [Lisibilité du code](#)
- [Les variables](#)
- [Conditions, tests et boucles](#)
- [Les tableaux](#)
- [Les fonctions](#)

Les boîtes de dialogues

Introduites dans différents chapitres

Comme leur nom l'indique, elles servent... à **dialoguer** avec l'utilisateur. Elles sont particulièrement pratiques pour tester ses propres scripts 😊 , et c'est d'ailleurs pour cette raison qu'on commence par elles. Il en existe trois sortes.

Afficher du texte

La première, **alert(message)**, affiche un **message** (une chaîne de caractères) à l'utilisateur.



Code : JavaScript

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

Demander une chaîne de caractères

La boîte de dialogue **prompt(message, valeur)** demande à l'utilisateur d'entrer du texte. Si l'utilisateur clique sur "Annuler", la fonction renvoie alors **null**.

Le **message** sera affiché au-dessus du champ pour saisir le texte, champ qui aura initialement la **valeur** précisée.

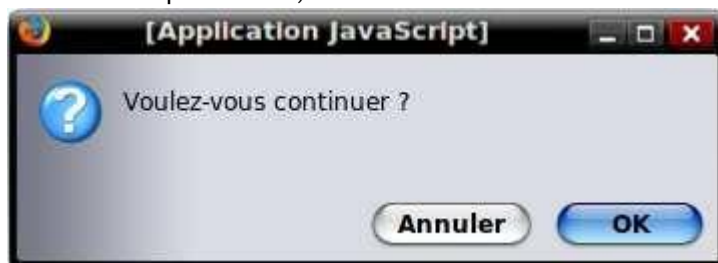


Code : JavaScript

```
var commentaire = prompt("Dites-nous ce que vous voulez," "Votre message  
ici");
```

Demander "oui" ou "non"

La dernière, **confirm(message)**, affiche le **message**, ainsi que deux boutons : Ok et Annuler. La valeur renvoyée est **true** si l'utilisateur clique sur "Ok", **false** sinon.



Code : JavaScript

```
var continuer = confirm("Voulez-vous continuer ?");
```

Où se place le JS ?

Le code JS peut se placer :

- soit directement dans les balises HTML, à l'aide d'un **gestionnaire d'événements** soit entre des
- balises propres au JS soit dans un fichier externe, portant l'extension .js, qui sera importé au
- chargement de la page.

Gestionnaires d'événements

Un gestionnaire d'événements est un attribut (une propriété) qui se place dans une balise HTML, qui s'écrit sous la forme **onEvenement**.

Il a pour rôle de lancer le script lorsque l'événement spécifié se produit.

Exemple :

Code : HTML

```
<a href="#" onclick="alert('Salut')">Cliquez</a>
```



Le gestionnaire d'événements est un **attribut** d'une balise HTML, et doit donc s'écrire en **minuscules** pour que la page soit valide.

Même si, lorsqu'on en parle, on l'écrit avec une majuscule pour faciliter la lecture.

Voici quelques événements :

- **click** : un clic de la souris sur l'élément concerné
- **dblclick** : un double-clic sur cet élément
- **mouseover** : la souris passe sur cet élément
- **mouseout** : la souris sort de cet élément
- (en quelque sorte le contraire de **mouseover**).

Balises propres au JS

Le code JS peut aussi se placer entre ces balises :

Code : HTML

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



Les "flèches" servent à masquer le JS aux navigateurs très anciens.

En leur absence, ces derniers afficheraient le code au milieu de la page... pas très esthétique 🙄.

Ces balises peuvent se placer :

- soit dans l'en-tête (**head**) de la page web : on y place du code qui doit être "mis en mémoire" pour être exécuté plus tard.
Un exemple typique : les déclarations de fonctions.
- Soit dans le corps (**<body></body>**) de la page, pour le code à exécuter au chargement de celle-ci. Par exemple, des fonctions pour initialiser ou lancer un script.

Importer un fichier externe

La dernière méthode (la plus "propre") consiste à écrire le code JS dans un fichier séparé, portant l'extension `.js`. On importe ensuite ce fichier à l'aide des balises précédentes, utilisées comme ceci (**fichier.js** est le nom du fichier à importer) :

Code : HTML

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

De même, cette ligne peut se placer soit dans l'en-tête, soit dans le corps de la page, selon les mêmes règles que pour le paragraphe précédent.

Lisibilité du code

Il est très important d'avoir un code lisible : ça permet de s'y retrouver, et donc d'éviter bon nombre d'erreurs. De plus, si vous commettez une erreur (chose qui n'est pas rare), il est beaucoup plus facile de la localiser dans un code bien organisé que dans un code illisible 🤪. C'est donc un point à ne pas négliger...

Les commentaires

Chapitre "Votre premier script"

Il existe deux manières d'écrire des commentaires en JS :

- en fin de ligne, après `//` n'importe où (peut s'étendre sur
- plusieurs lignes), entre `/*` et `*/`.

Code : JavaScript

```
// ceci est un commentaire en fin de ligne  
/* ceci est un commentaire qui peut prendre  
plusieurs lignes */
```



Leur rôle est de rendre la page plus facilement lisible.

Mais noyer le code entre des commentaires n'est pas forcément mieux que de ne pas le commenter.

Il faut donc qu'ils soient clairs, et utilisés là où ils sont nécessaires.

L'indentation

Chapitre "Créer ses propres fonctions"


L'indentation a elle aussi pour but de rendre le code plus lisible.
Elle consiste à "décaler" les instructions imbriquées.

Un exemple sera plus parlant :

Code : JavaScript

```
instruction1;
{
  instruction2;
  instruction3;
  {
    instruction4;
  }
}
instruction5;
```

Les variables

Chapitre "Les variables" 

Utilisation des variables

Les variables servent à stocker des données.

Déclaration

Pour créer (on dit *déclarer*) une variable, on utilise le mot-clé **var**, suivi du nom qu'on veut lui donner.

Code : JavaScript

```
var age;
```



Le nom ne peut être composé que des 26 lettres, majuscules et / ou minuscules (JS fait la différence entre les deux), des 10 chiffres et du *underscore* (trait de soulignement).

De plus, le nom ne doit pas commencer par un chiffre.



Certains mots-clé sont déjà utilisés en JS.

Ainsi, il y a certains mots, comme, **break**, **case**, **char**, **continue**, **delete**, **double**, **final**, **long**, **new**, **public** et **super**, que vous ne pouvez pas utiliser comme nom de variable (ou de fonction).

La déclaration se fait (sauf exception) en une seule fois.

Affectation

Pour *affecter* une valeur à une variable, on utilise le signe **=** comme ceci :

Code : JavaScript

```
age = 18;
```

Initialisation

Il est possible d'affecter une valeur à une variable lors de sa déclaration.

Code : JavaScript

```
var age = 18;
```

Variables locales

Chapitre "Créer ses propres fonctions"

Une variable déclarée à l'intérieur d'une fonction n'est accessible qu'à l'intérieur de cette même fonction. On parle de variable *locale*.



Il faut utiliser au plus possible des variables locales. Cela vous évitera déjà bon nombre de problèmes.

Types de données

Une variable peut contenir différents *types* de données. Présentation de deux types récurrents : les **nombre**s et les **chaînes de caractères**.

Les nombres

Exemple : on veut créer une variable **pi** contenant le nombre 3.14

Code : JavaScript

```
var pi = 3.14;
```



On utilise le **point** à la place de la virgule...

Il y a 5 opérations de bases : l'**addition** (+), la **soustraction** (-), la **multiplication** (*), la **division** (/) et le **modulo** (%).



Un petit mot sur la dernière opération : **a % b** (si **b** est positif) est le reste dans la division euclidienne de **a** par **b**.

Dans cet exemple, on verra s'afficher **3** (car **18 = 5 x 3 + 3**) :

Code : JavaScript

```
alert(18%5);
```

Il existe des opérateurs supplémentaires pour faciliter les calculs avec des variables :

- **+=** / **-=** / ***=** / **/=** pour : **augmenter de** / **diminuer de** / **multiplier par** / **diviser par** un nombre la valeur d'une variable.
De même, il existe aussi **%=**.

- **++** et **--** pour **incrémenter** / **décrémenter** de 1 la valeur d'une variable.

Un petit exemple ne sera pas superflu 😊.

Code : JavaScript

```
// Les 3 dernières lignes sont équivalentes  
// on augmente a chaque fois de 1 la valeur de n  
var n = 0;  
n = n+1;  
n += 1;  
n++;
```

Les chaînes de caractères

Une chaîne de caractère, c'est... du texte 😊.

Elle est délimitée par des guillemets " ou bien par des apostrophes '.

Code : JavaScript

```
var chaine = "Je suis une chaine de caracteres";  
var msg = 'Eh bien moi aussi !';
```

On utilise un **caractère d'échappement** pour certains caractères spéciaux, comme :

- **\n** pour un retour à la ligne
- **\'** et **\"** pour une apostrophe / un guillemet
- **** pour afficher un antislash
- **\uXXXX** pour insérer le caractère dont la valeur unicode est **XXXX**.

On utilise le signe **+** pour concaténer (mettre bout-à-bout) plusieurs chaînes de caractères.

Exemple :

Code : JavaScript

```
var age = 18;
alert("Vous avez " + age + " ans"); // on concatene les bouts de
phrases
```

(Notez que le nombre **18** est converti automatiquement en chaîne de caractères, pour pouvoir ensuite être concaténé.)

L'opérateur **+=** sert à ajouter une chaîne de caractères à la fin d'une variable.

Dans cet exemple, on ajoute un retour à la ligne à la fin :

Code : JavaScript

```
var msg = "Bonjour";
msg += "\n";
```

On peut utiliser les fonctions **parseInt(chaine)** et **parseFloat(chaine)** pour convertir une chaîne de caractères en nombre (**int** : entier ou **float** : décimal).

C'est utile pour additionner des nombres qui sont enregistrés sous forme de chaînes de caractères : le signe **+** les concatène si on l'utilise directement. **Code : JavaScript**

```
var a = "1";
var b = "8";
alert(a+b); // affiche 18
alert(parseInt(a)+parseInt(b)); // affiche 9 :)
```

Conditions, tests et boucles

Chapitres "Les conditions" et "Les boucles"

Les booléens

Chapitre "Les conditions"

Nous avons déjà parlé de deux types de variables : les *nombres* et les *chaînes de caractères*.

Mais nous allons maintenant avoir besoin d'un troisième type de variables : les **booléens**. Ces variables ont deux valeurs possibles : vrai (**true**), ou faux (**false**).

Voyons comment ça marche et à quoi ça sert...

Comparer des nombres

On peut obtenir des booléens en **comparant** des valeurs.

En effet, on dispose d'**opérateurs de comparaison**, qui sont au nombre de 6 :

- opérateur d'égalité, **==** (attention, il y a bien **deux** signes "égal")
- ... son inverse, l'opérateur "est différent de", qui se note **!=**
- l'opérateur "strictement plus grand que", **>** ... de même, on a
- "strictement plus petit que", **<** et pour terminer, "plus grand ou
- égal à", **>=** ... ainsi que "plus petit ou égal à", **<=**.
-

Petit exemple : une variable **est_majeur** qui contient **vrai** uniquement si l'âge saisi par l'utilisateur est plus grand ou égal à 18.

Code : JavaScript

```
var age = prompt("Quel age avez-vous ?");  
var est_majeur = (age >= 18);
```

Notez que les parenthèses autour de la condition servent à bien la distinguer (elles ne sont pas obligatoires, libre à vous de les mettre ou non).

Opérations sur les booléens

Nous avons vu qu'on peut additionner ou multiplier des nombres et concaténer des chaînes de caractères. Eh bien il existe aussi des opérations sur les booléens, au nombre de trois :

- la **négation**, notée **!** (si **a** est vrai, **!a** est faux, et inversement)
- le **ET** logique, **&&**. On a **a && b** qui est vrai si et seulement si **a** et **b** sont vrais le **OU** logique,
- **||** : **a || b** est vrai si et seulement si au moins l'une des deux valeurs est vraie.

Les conditions

Chapitre "Les conditions" (si si !!)

IF ... ELSE ...

On peut effectuer un test, pour exécuter des instructions différentes selon la valeur d'un booléen. Voici la syntaxe :

Code : JavaScript

```
if(boolean)  
    // instruction a executer si le boolean est vrai  
else  
    // instruction a effectuer s'il est faux
```



Avec cette écriture, on ne peut mettre qu'une instruction dans chaque "membre". Si on a plusieurs instructions, il suffit de les placer entre accolades, **{** et **}**.

Le **else** n'est d'ailleurs pas obligatoire, on peut n'écrire que :

Code : JavaScript

```
if(boolean)  
    // instruction a effectuer si le boolean est vrai
```

Comme d'habitude, un exemple :

Code : JavaScript

```
var age = prompt("Quel age avez-vous ?");
if (age >= 18)
{
    alert("Vous êtes majeur.");
    alert("Mais il n'est jamais trop tard pour apprendre à programmer :D");
}
else
    alert("Tu es mineur");
```

On peut bien sûr effectuer des tests les uns à l'intérieur des autres (imbriqués).

IF avec autre chose qu'un booléen



Et si on met autre chose qu'un booléen dans le **if** ?

Dans ce cas, JS va essayer de "convertir" la chose en booléen.
Le résultat sera **false** uniquement dans les cas suivants :

- 0 (zéro)
- "" ou " (chaîne de caractères vide) **undefined**
- (variable déclarée mais non définie) **null** (mot-clé utilisé pour des objets "vides")
- *S'il manque des cas, ce sont des cas similaires*

SWITCH

Mais il arrive qu'on ait besoin de tester plusieurs valeurs pour une même variable.
Dans ce cas, il est possible d'utiliser plusieurs tests : **if ... else if ... else if ... else ...**

Mais il existe une syntaxe moins lourde.

Voyez plutôt : on va distinguer plusieurs cas concernant le nombre d'enfants de l'utilisateur.

Code : JavaScript

```
var nb = prompt("Combien avez-vous d'enfants ?");
switch(nb)
{
    case 0: // si le nombre est 0...
        alert("Au moins, vous êtes tranquilles :p"); // ... on affiche
un message...
        break; // ... et on arrete le "switch" ici
    case 1: // si le nombre est 1
        alert("Oh il est tout seul...");
        break;
    case 2:
    case 3:
        // s'il y en a 2 ou 3
        alert("Il doit y avoir de l'ambiance chez vous ^");
        break;
    case 4:
        alert("Jolie famille !");
        break;
    default: // si c'est aucune des valeurs precedentes
        alert("Plus de 4 enfants ?! Waow...");
        break;
}
```

En rentrant dans le **switch**, l'ordinateur exécute le code à partir du **case** correspondant, et quitte le **switch** lorsqu'il rencontre **break**.

Les boucles

Chapitre "Les boucles"...

Les boucles permettent de répéter des instructions tant qu'une condition est vraie. On distingue trois types de boucles, adaptées à des situations différentes.

WHILE

La première est très classique :

Code : JavaScript

```
while(condition)
    // action
```

On répète l'action tant que la condition est vraie. Notez que si la condition est fausse dès le début, l'action n'est jamais effectuée.

Exemple :

Code : JavaScript

```
var i=0;
while(i<10)
{
    var j = i*i;
    alert("Le carré de " + i + " est " + j);
    i++;
}
```

FOR

La boucle **for** est une variante, plus complète, de la boucle **while**.

Elle s'utilise sous cette forme : **for(initialisation ; condition ; incrémentation) :**

- l'initialisation se fait une seule fois, au début de la boucle
- la boucle est exécutée tant que la condition est vraie à la fin de
- chaque tour de boucle, l'incrémentation est effectuée.



Cette boucle est très utilisée, car elle convient souvent au besoin.

Mais chaque boucle a une utilisation particulière : on choisit cette boucle avant tout parce qu'elle convient à notre problème, et non pas uniquement parce qu'elle est pratique.

Un exemple :

Code : JavaScript

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

Ce qui est équivalent à :

Code : JavaScript

```
var i=0;  
while(i<10)  
{  
    alert(i);  
    i++;  
}
```



Il faut éviter d'utiliser à l'intérieur de la boucle la variable servant de compteur (les parenthèses servent justement à regrouper tout ce qui la concerne).

Inversement, on évitera de mettre dans ces parenthèses des choses qui n'ont rien à voir avec ce compteur.

DO .. WHILE

Cette dernière boucle est un peu différente des précédentes.

En effet, la condition est évaluée **à la fin** de la boucle : les instructions seront toujours exécutées **au moins une fois**, même si la condition est fausse dès le départ.

Cette particularité fait qu'elle a une utilisation différente.

Sa syntaxe est la suivante :

Code : JavaScript


```
do
    // instruction a repeter
while(condition);
```



Notez la présence du point-virgule à la fin.
Ceci est dû au fait que la condition est à la fin de la boucle.

Voici un exemple : on demande une chaîne de caractères à l'utilisateur tant qu'il ne clique pas sur "Annuler".

Code : JavaScript

```
var msg;
do
    msg = prompt("Entrez ce que vous voulez, ou cliquez sur Annuler");
while(msg);
```



On effectue ici un test avec une valeur qui n'est pas un booléen, comme ça a été expliqué quelques paragraphes plus haut.

Réaliser une affectation dans une condition

Notez qu'on peut combiner une affectation et une condition.
Autrement dit, faire quelque chose comme ceci :

Code : JavaScript

```
var msg;
while( (msg = prompt("Entrez du texte")) != null)
    alert(msg);
```

Pour comprendre cette condition, on peut la séparer en deux, comme ceci :

Code : JavaScript

```
msg = prompt("Entrez du texte");    // c'est l'affectation, entre les
parentheses
msg != null;    // c'est la condition
```

Et en effectuant le test directement avec une valeur qui n'est pas booléenne, on peut écrire ceci :

Code : JavaScript

```
var msg;  
while(msg = prompt("Entrez du texte"))  
    alert(msg);
```



Il faut bien comprendre que le signe **=** est le signe d'affectation !

En décomposant la condition, ça nous donne :

Code : JavaScript

```
msg = prompt("Entrez du texte"); // affectation  
msg; // le test, avec une valeur qui n'est pas booléenne
```

Les tableaux

Chapitre "Les tableaux" 🤔

Les tableaux servent à avoir accès, de manière efficace, à un grand nombre de données. Ils permettent en effet de les **numéroter**, d'où leur côté pratique.



Le "numéro" s'appelle l'**indice**, ou la clé.

La valeur contenue dans une "case" est un **élément** de notre tableau.



En JS, on commence à compter à partir de **0** !

Créer et modifier un tableau

Ce qu'il faut savoir...

On crée un tableau de cette manière :

Code : JavaScript

```
var monTableau = new Array(valeur0, valeur1, valeur2);
```

Le tableau est initialisé avec les valeurs entre parenthèses (donc ici, les éléments d'indice **0**, **1** et **2** auront ces valeurs respectives).



La variable `monTableau` ne contient pas directement le tableau, mais son "adresse" dans la mémoire de l'ordinateur.

Dans l'exemple ci-dessous, `t` et `monTableau` concerneront donc le **même** tableau (les modifications effectuées sont valables pour les deux noms). **Code : JavaScript**

```
var t = monTableau;
```

Maintenant qu'on a notre tableau, on peut :

- accéder à l'élément d'indice **i** grâce à **monTableau[i]**, **monTableau** étant le nom du tableau. On manipule les éléments comme des variables. connaître sa longueur grâce à
- **monTableau.length** (c'est la **longueur de monTableau**).

Parcourir le tableau

Pour parcourir tout le tableau, on va utiliser une boucle pour accéder à chacun des éléments.

Dans cet exemple, on va afficher un par un tous les éléments de **monTableau**, et les remettre à zéro : **Code : JavaScript**

```
for(var i=0; i<monTableau.length; i++)
{
    alert(monTableau[i]); // on lit...
    monTableau[i] = 0;   // ... et on efface
}
```

Tableau à deux dimensions

On a parfois besoin de tableaux à deux dimensions : prenons l'exemple d'une grille de sudoku.

Pour créer une telle grille, on crée un "grand" tableau, qui va contenir les lignes de la grille. Et chaque ligne sera... un "petit" tableau !

Voici comment créer un tel tableau et le remplir de 0 :

Code : JavaScript

```
// on cree le "grand" tableau, contenant les lignes
var grille = new Array();

// on cree les lignes (les "petits" tableaux) les unes apres les autres
for(var i=0; i<9; i++)
    grille[i] = new Array();

// on parcourt les lignes...
for(var i=0; i<9; i++)
    // ... et dans chaque ligne, on parcourt les cellules
    for(var j=0; j<9; j++)
        grille[i][j] = 0;
```

Les fonctions

*Chapitres "Créer ses propres fonctions" (pour la partie "Généralités")
et "Retour sur les fonctions" (pour les deux autres parties)*

Généralités

Déclaration

Une fonction se déclare dans l'en-tête de la page.
Il y a deux manières différentes de déclarer une fonction.
La première utilise le mot-clé **function** :

Code : JavaScript

```
function f(x,y)
{
    // code
    return valeur;
};
```

La seconde utilise directement une variable :

Code : JavaScript

```
var f = function(x,y)
{
    // code
    return valeur;
};
```



Il n'y a pas de "meilleure" méthode.

La seconde a l'avantage d'utiliser directement une variable, notion que l'on connaît bien.

Arguments

Dans cet exemple, **x** et **y** sont les deux *arguments* (ou *paramètres*) de cette fonction.



Une fonction peut n'avoir aucun argument.

Dans ce cas, les parenthèses sont vides (mais **toujours présentes**).

Ce sont en fait des variables locales à la fonction, dont la valeur est fixée par l'utilisateur au moment où il appelle la fonction.

Par exemple, si on fait :

Code : JavaScript

```
f(1, "Toto");
```

La variable **x** sera initialisée à **1** et **y** avec la chaîne de caractères **"Toto"**.



Les arguments sont séparés par des **virgules**.

Valeur renvoyée

Une fonction peut renvoyer une valeur. Dans notre exemple, c'est la ligne **return valeur** (la valeur est par exemple contenue dans une variable).

C'est en quelque sorte la valeur que "prendra" la fonction lors de son appel.

On peut la récupérer de cette manière :

Code : JavaScript

```
var x = f(1, "Toto");
```

Ici, la variable x prendra la valeur renvoyée par la fonction.

Exemple

Une fonction qui calcule la moyenne de deux nombres.

- Elle prend donc deux arguments, qui sont les nombres dont on veut la moyenne.
- Elle renvoie la moyenne de ces deux nombres.

Code : JavaScript

```
function moyenne(x, y)
{
    var moy = (x+y) / 2;
    return moy;
};
```

Une autre façon d'écrire cette fonction :

Code : JavaScript

```
var moyenne = function(nb1, nb2)
{
    return (nb1+nb2) / 2;
};
```

Rendre un argument facultatif

On peut vouloir rendre un argument facultatif.

En reprenant la fonction **moyenne**, on peut décider que **moyenne(n)** renverra la moyenne de **n** et **0**.

Au début de la fonction, on va devoir vérifier si le second argument est défini (s'il n'est pas défini, sa valeur est **undefined**). Si ce n'est pas le cas, on lui donne la valeur **0** (comme convenu).

Code : JavaScript

```
function moyenne(x,y)
{
    if (y == undefined)
        y = 0;
    return (x+y) / 2;
};
```

Avoir un nombre illimité d'arguments

On accède au tableau contenant tous les arguments passés à une fonction `fct` à l'aide de `fct.arguments` (ce sont les arguments de `fct`).

Il ne reste plus qu'à travailler sur notre tableau d'arguments.

Exemple : une fonction renvoyant le plus grand des arguments.

Code : JavaScript

```
function maxi(m)
{
    var nb = maxi.arguments;
    for(var i=1; i<nb.length; i++)
        if (nb[i] > m)
            m = nb[i];
    return m;
}
```

Dans cet exemple, `m` est un argument "classique".

Cependant, on le retrouve aussi dans la première case du tableau d'arguments (c'est d'ailleurs pour ça que la boucle commence à 1).

On peut maintenant appeler cette fonction comme ceci :

Code : JavaScript

```
alert( maxi(1,4,9,2,5) );
alert( maxi(4,9) );
```

C'est parfois utile, car on a ainsi pas mal de souplesse lors de l'utilisation 😊.

Si vous arrivez jusqu'ici en ayant lu tout ce chapitre, c'est que :

- soit vous êtes persévérants, et vous avez voulu vous accorder une séance de révisions intenses (qui n'est, à mon avis, pas du temps perdu) : dans ce cas, j'espère que vous avez pris le temps de faire quelques "pauses" en cours de route 😊
- soit vous êtes paresseux, et vous avez voulu éviter toute la première partie de ce cours, auquel cas vous êtes passés à côté de nombreuses subtilités...
- soit vous êtes curieux, et vous voulez un aperçu général de la première partie de ce cours : si ce résumé vous a convaincus, direction le début du tuto pour approfondir tout cela 😊
- soit vous êtes totalement inconscients, et là je ne peux rien faire pour vous 😊

Toujours est-il que le but premier de ce chapitre est de mettre à votre disposition un résumé de ce qu'il faut savoir : si vous avez des trous de mémoire, il sera votre compagnon.