Introduction à Javascript

<http://www.coursweb.ch/javascript>

Cette introduction à javascript n'a pas la prétention de remplacer les nombreuses références sur le sujet que l'on trouve sur le web (voir dans la page [liens](http://www.coursweb.ch/javascript/liens.html))

Son but est de permettre au débutant de comprendre rapidement certains éléments du langage et de la programmation en général. Ceci par des articles aussi courts que possible.

Le lecteur complètera ses connaissances en se rendant sur des sites plus complets.

# Table des matières

[Table des matières 1](#_Toc433221868)

# Insertion de scripts

Pour écrire du code javascript utilisable dans une page HTML on utilise la balise **script**

On peut intégrer des scripts à différents emplacements dans la page. Soit le code est directement écrit dans la page, avec par exemple:

[<script](http://december.com/html/4/element/script.html) type='text/javascript'**>**  
*<!--  
  Le code javascript s'écrit ici  
//-->*  
**</script>**

Soit le code se trouve dans un fichier séparé (portant généralement l'extension .js)

[<script](http://december.com/html/4/element/script.html) type='text/javascript' src='url\_du\_script.js'**></script>**

A noter que dans le premier cas, on placera des commentaires HTML, en effet le code javascript contient souvent certains caractères qui sont conflictuels du points de vue du HTML. Par exemple le caractère **&** qui marque le début des entités HTML (&eacute; pour un é par exemple).

## Emplacement du script dans la page

L'emplacement du script a parfois de l'importance parfois pas. Cela dépend de ce qu'il contient. Certains scripts agissent directement sur des éléments de la page HTML. Dans ce cas, le script doit être placé après ceux-ci pour qu'ils soient "construits" au moment de l'exécution.

Si des scripts déclarent des fonctions qui manipulent des éléments de la page et que celles-ci seront appelées plus tard (une fois la page construite par exemple), alors la position du script n'a pas d'importance.

# Script de base

Pour tester les scripts présentés dans ce cours, il faut copier les codes dans une balise script tel que cela a été présenté dans la page [insertion de script](http://www.coursweb.ch/javascript/script-tag.html).

## Hello world

 alert("Hello world");

Pour commencer, voici le traditionnel **Hello world**. Ce script affiche le message **Hello world** dans une boite de dialogue.

**alert** est une méthode prédéfinie en javascript qui permet d'afficher une boite de dialogue simple (composée d'un message et d'un bouton ok pour la refermer). Le message est choisi par le programmeur et est passé en tant que paramètre de la méthode **alert**. Les paramètres des méthodes sont donnés à l'intérieur d'une paire de parenthèse qui suivent le nom de la méthode.

Dans cet exemple, il y a un seul paramètre, et il est de type texte. Les textes (ou chaines de caractères) sont délimité par des guillemets ou par des apostrophes (dit aussi guillemets simples).

On notera enfin la présence d'un point virgule qui marque la fin d'une instruction.

# Script simple

Voici un script simple qui contient divers éléments importants du langage javascript. Il faut s'habituer à distinguer ces éléments.

**var** pos;  
**var** speed = 1;  
**var** pos\_initial;  
  
**function** startAnim() {  
        **var** e = document.getElementById('newsbox');  
        pos\_initial = e.clientHeight + 10;  
        pos = pos\_initial;  
        setInterval(anim, 20);  
}  
  
**function** anim() {  
        **var** e = document.getElementById('newslist');  
        e.style.visibility = 'visible';  
        e.style.top = Math.floor(pos) + 'px';  
        pos = pos - speed;  
        **if**(pos < -e.clientHeight) pos = pos\_initial;  
}  
  
**function** setSpeed(n) {  
        speed = n;  
}

## Les fonctions

Le mot clé **function** marque la déclaration d'une fonction. Les accolades délimitent des blocs. Dans un premier temps, il faut être capable de discerner les instructions qui sont dans ou dehors des fonctions.

Les trois premières lignes de ce script sont trois instructions hors fonctions. (Elle appartiennent au contexte global). Les autres instructions de ce script appartiennent à leur fonctions respectives (elle sont a l'intérieur des accolades de fonctions).

Les fonctions portent un nom (ce n'est pas toujours le cas en javscript), celui-ci suit le mot clé **function**. Ce nom a été inventé par le programmeur qui a réalisé le script. Il permet de différencier les fonctions les unes des autres. Il a généralement un nom qui correspond à ce que fait la fonction. Dans ce script, il y a trois fonctions: **startAnim**, **anim**, **setSpeed**.

La fonction **setSpeed** attend un paramètre contrairement aux deux autres fonctions. Les paramètres attendus par les fonctions sont décrit par des noms séparés par des virgules (si il y en a plus d'un) et placés entre parenthèses après le nom de la fonction.

## Les points virgules

Les points virgules délimitent les instructions. Quand un code est bien écrit, on a généralement une instruction par ligne mais ce n'est pas obligatoire.

## Les opérateurs

Les caractères **+ - / \* % = < >** sont des opérateurs. Ils ont tous un rôle bien précis. Le plus fréquent d'entre-eux est l'opérateur égal **=**. Sont rôle est l'affectation. On copie toujours la valeur à droite du signe **=** dans la référence mémoire indiquée à sa gauche.

Certains opérateurs sont des mots clés du langage, comme le mot **var**. Celui-ci sert à déclarer une variable. Les variables sont des zones mémoire pouvant accueillir des valeurs (qui peuvent évoluer au fil du temps).

## Le point

Le symbole point est utilisé soit comme virgule pour les nombres soit comme séparateur pour la notation objet. C'est ce second cas qui est le plus fréquent. Par exemple **document.getElementById** signifie que **getElementById** est un élément appartenant à **document**, ou encore **e.style.top** signifie que **top** appartient à **style** qui appartient à **e**. Le point a un peu le rôle d'un slash **/** dans les urls. Il indique une forme hiérarchique.

## Mot réservés

En javascript il y a un certain nombre de [mots réservés](http://www.coursweb.ch/javascript/js-keywords.html). Dans cet exemples ils sont marqués en gras bleu foncé. Ce sont **function, var, if**. Les mots réservés sont peu nombreux en javascript. Ils ont un rôle bien précis défini par le langage javascript.

## Méthodes prédéfinies

Les méthodes sont des fonctions prédéfinies en javascript et attachées (appartenant) à des objets prédéfinis. Par exemple **Math.floor**, c'est la méthode **floor** qui appartient à l'objet **Math**. Ces méthodes et objets prédéfinis sont documentés dans les références javascript.

## Propriétés prédéfinies

Ce sont des références mémoire pouvant généralement être consultées ou modifiées (comme des variables), par exemple **e.style.visibility**. **visibility** est une propriété de **style** qui est une propriété de **e**.

## Méthode ou propriété ?

Ce qui permet de différentier une méthode (fonction) d'une propriété (variable), c'est la présence des parenthèses après le nom. **getElementById** est une méthode, alors que **clientHeight** est une propriété.

## Constantes

Dans ce script on rencontre encore des constantes qui sont soit numérique soit de type texte ici. Par exemple **newsbox** est une constante texte, parce qu'elle est entourée de guillemets (simple ou double). **1**, **10** et **20** sont des constantes numérique dans cet exemple (coloriées en rouge).

# Structures et types de données

On entend par **données** de l'information stockée en mémoire, cela peut être un nombre, un texte ou des éléments plus complexes. On entend par **structure** une manière d'organiser les données dans la mémoire.

Javascript est un langage où les données ne sont pas explicitement typées. Cela signifie que pour la personne qui lit du code source Javascript, il faudra comprendre l'usage qu'il est fait des données pour comprendre leur type.

## Structures élémentaires

Les constantes (ou valeurs en dur) sont les données les plus primitives. Elle figurent en clair dans un code. Ce sont souvent des nombres (entiers ou à virgule), des textes ou des valeurs booléennes (**true**, **false**).

Les textes sont reconnaissables par l'usage de guillemets qui les entourent. Les guillemets peuvent être simple ' ou doubles ".

alert("Bonjour je suis un texte");  
**var** a = 12;  *// 12 est un nombre entier*  
**var** b = a + 2.4 *// 2.4 est un nombre à virgule*  
**var** c = **true**; *// un booléen vrai*

On notera que c'est le symbole . (point) qui sert de virgule. Ces constantes (ou valeur en dur) sont omniprésentes dans un code javascript.

**Variables**

Les variables servent à identifier une structure de donnée (simple ou complexe) par un nom. On commence par déclarer une variable dans le programme pour la situer dans un contexte. Ensuite on lui affecte une valeur et enfin on utilise cette valeur.

Un exemple simple:

**var** a; *// Déclaration*  
a = 10; *// Affectation*  
alert(a); *// Utilisation*

La déclaration et l'affectation se font souvent en une seule instruction:

**var** a = 10;

Dans ces exemples, la variable est nommée **a**, ce nom est unique dans un même contexte et permet d'y faire référence lors de l'utilisation. On lui a affecté la constante **10** qui est un nombre entier. **alert(...)** est une fonction javascript prédéfinie qui permet d'afficher une valeur dans une fenêtre (boîte de dialogue) disposant d'un bouton OK qui permet de la fermer.

La valeur **10** est physiquement stockée quelque part dans la mémoire de l'ordinateur (cela n'a pas d'importance de savoir où dans un langage comme javascript).

Le mot variable signifie que la valeur peut varier. C'est à dire qu'au cours de l'exécution d'un programme on peut lui affecter une nouvelle valeur.

Voici un exemple:

**var** a = 10; *// Déclaration et affectation première*  
alert(a); *// première utilisation*  
a = 12; *// Nouvelle affectation*  
alert(a); *// Deuxième utilisation*

Le premier **alert** affichera **10**, le second affichera **12**. Lors de la nouvelle affectation, la valeur **10** est remplaçée par la nouvelle valeur **12**.

**Tableaux ou liste**

Les tableaux ou listes en javascript sont une manière d'organiser des données dans un ordre successif (les unes à la suites des autres).

Chaque donnée d'une liste est identifiable par le nom du tableau et sa position dans celui-ci (on parle d'index).

Exemple de déclaration d'un tableau:

**var** jours = **new** Array('Lun', 'Mar', 'Mer', 'Jeu', 'Ven', 'Sam', 'Dim');

Ici on défini une liste de 7 textes (javascript s'arrête là au niveau de sa compréhension). Pour l'humain qui lit ce code, il s'agit bien évidemment d'une énumération des abréviations des jours de la semaine dans l'ordre.

Pour accéder à un élément d'un tableau on utilise le nom du tableau et l'index de l'élément. **Attention, les indexes commencent à 0 et non à 1**

alert(jours[0]); *// affiche Lun*  
alert(jours[6]); *// affiche Dim*

Les cellules d'un tableau sont variables, il est donc tout à fait possible d'affecter à nouveau une d'entre elles, comme pour une variable.

jours[1] = 'Mardi';  
alert(jours[1]); *// Affiche Mardi*

On peut connaitre la longueur d'un tableau avec la propriété **length**

 alert(jours.length); *// Affiche 7*

Chaque cellule d'un tableau étant une variable, elle peut donc contenir une référence simple (comme un texte ou un nombre) ou une référence plus complexe tel qu'un tableau. Ceci permet de créer des tableaux à plusieurs dimensions, ou des structures de données très diverses.

**Tableaux associatifs**

Contrairement aux tableaux linéraires (listes) où les élément sont repérés par un index (leur position dans le tableau), Les tableaux associatifs permettent de repérer les éléments par une clé (qui est souvent un texte).

Exemple:

**var** tableau = **new** Array(); *// déclaration d'un tableau vide*  
tableau['nom'] = 'Moran';  
tableau['prenom'] = 'Bob';  
alert(tableau['nom']); *// affiche Moran*

**JSON Javascript Object Notation**

JSON est un format de structure de donnée pour écrire des tableaux associatifs. [JSON sur wikipedia](http://fr.wikipedia.org/wiki/JSON)

Ceci permet d'introduire des données structurée dans du code javascript de façon simple. On utilise aussi largement JSON comme format de transfert pour des requêtes AJAX.

Voici un exemple:

**var** data = {  
  'key1' : 'value1',  
  'key2' : 'value2',  
  'key3' : 'value3'  
};

Les paires clé valeurs sont séparées par une virgule. On peut accéder aux données de façon simple avec les notations suivantes:

**var** x = data.key2;  
**var** y = data['key3'];

# Structures de contrôle

En programmation, on appelle 'structures de contrôle' les éléments du langages qui permettent de gérer l'ordonnancement des instructions. Par défaut, les instructions sont exécutées dans l'ordre où elles sont écrites, les unes après les autres. C'est la **séquence**.

Parfois il est nécessaire d'exécuter un groupe d'instructions seulement si certaines conditions sont remplies. Ce sont les **tests**.

D'autres fois il faut répéter certaines séquences d'instruction plusieurs fois. Ce sont les **boucles**.

La séquence, les tests et les boucles sont les bases de l'ordonnancement.

# Tests

Un test permet d'exécuter ou non une partie de programme en fonction du résultat d'une expression. Il existe plusieurs instructions pour faire des tests, **if**, **switch** et l'opérateur ternaire **? :**. Plus d'info sur [wikibooks](http://fr.wikibooks.org/wiki/Programmation_JavaScript/Structures_de_contr%C3%B4le#Branchement_conditionnel)

**if**(expression)  
{  
  *// code exécutée si l'expression est vraie*  
}

L'expression est d'abord évaluée (calculée). Si son résultat est **true** (vrai), alors les instructions comprises dans le bloc d'accolades sont exécutées. Si le résultat de l'expression est **false** (faux) le programme continue normalement après l'accolade de fermeture.

Dans le cas où le code à exécuter est limité à une seule instruction, alors les accolades ne sont pas nécessaires.

**if**(expression) instruction;

## if et else

Dans certains cas on a besoin d'exécuter un code quand l'expression est vraie et un autre lorsque l'expression est fausse.

**if**(expression) {  
  *// ici si l'expression est vraie*  
}  
**else** {  
  *// ici si l'expression est fausse*  
}

On peut imbriquer les tests les uns dans les autres, ainsi:

**if**(expression1) {  
  *// ici si l'expression 1 est vraie*  
}  
**else** {  
  *// ici si l'expression 1 est fausse*  
  **if**(expression2) {  
    *// ici l'expression 2 est vraie*  
  }  
  **else** {  
    *// ici l'expression 2 est fausse*  
  }  
}

Le deuxième **if** est considéré comme un seule instruction (qui en comprend d'autres en interne). Ce qui fait que les accolades du **else** du premier **if** ne sont pas nécessaires. Ceci nous amène à l'écriture suivante:

**if**(expression1) {  
  *// ici si l'expression 1 est vraie*  
}  
**else** **if**(expression2) {  
  *// ici si l'expression 1 est fausse et l'expression 2 est vraie*  
}  
**else** {  
  *// ici si toutes les expressions sont fausses*  
}

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:if...else)

## switch

Cette structure de contrôle permet d'exécuter des codes différents pour des valeurs spécifiques d'une même expression.

**switch**(expression) {  
**case** valeur1:  
  *// ici si l'expression vaut valeur 1*  
  **break**;  
**case** valeur2:  
  *// ici si l'expression vaut valeur 2*  
  **break**;  
**case** valeur3:  
  *// ici si l'expression vaut valeur 3*  
  **break**;  
**default**:  
  *// ici pour toutes les autres valeurs*  
}

Suivant la valeur de l'expression, le programme "saute" dans le **case** correspondant et exécute les instructions qui s'y trouve dans l'ordre.

On notera la présence d'une instruction **break**, celle-ci provoque la continuation du programme après l'accolade de fermeture du **switch**. Si l'on omet cette instruction alors le programme continue à travers le **case** suivant. (Ceci est rarement pratiqué)

Si aucune valeur ne correspond, ce sont les instructions dans la section **default** qui sont exécutées. Cette section est optionnelle, si on l'omet, alors le programme continue après l'accolade fermante du **switch** si aucune valeur ne correspond à l'expression.

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:switch)

## l'opérateur ? :

Dans bien des cas on a besoin d'affecter une valeur à une variable en fonction d'une expression. Avec un **if** cela ressemblerait à ça:

**var** a;  
**if**(b>10) a=2;  
**else** a=4;

On peut utiliser l'opérateur ternaire **? :** pour alléger l'écriture.

**var** a = b>10 ? 2 : 4;

**a** prend la valeur 2 si **b** est plus grand que 10, sinon **a** prend la valeur 4.

# Boucles

Les boucles permettent d'exécuter plusieurs fois une série d'instructions. Plus de détails sur [wikibooks](http://fr.wikibooks.org/wiki/Programmation_JavaScript/Structures_de_contr%C3%B4le#Contr.C3.B4le_d.27it.C3.A9ration_.28boucles.29)

## while

**while**(expression)  
{  
  *// instructions*  
}

L'expression est d'abord évaluée (calculée). Si son résultat est **true** (vrai), alors les instructions comprises dans le bloc d'accolades sont exécutées puis on remonte au **while**. L'expression est à nouveau évaluée, si le résultat est **true** on entre à nouveau dans la boucle. Si le résultat de l'expression est **false** (faux) le programme continue normalement après l'accolade de fermeture.

Cela signifie que les instructions seront répétées tant que l'expression reste vraie. Il faut que l'expression aie une chance de devenir fausse si l'on ne veut pas avoir une boucle sans fin (plantée du prgramme). Si l'expression est fausse la première fois, alors aucune instruction de la boucle n'est exécutée et le programme continue après l'accolade de fin.

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:while)

## do while

**do** {  
  *// instructions*  
} **while**(expression);

La différence de cette boucle est que l'on entre dedans avant la première évaluation de l'expression. Donc les instructions sont au moins exécutées une fois. Après ce premier passage de boucle, on recommence seulement si l'expression est vraie.

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:do...while)

## for

**for**(initialisation ; expression ; modification)  
{  
  *// instructions*  
}

Cette boucle est une manière compacte d'écrire une boucle while. En effet, pour que l'expression aie une chance de passer de l'état vrai à faux, il faut souvent utiliser une variable. Cette variable doit avoir une valeur de départ et elle doit être modifiée à chaque tour de boucle. La boucle **for** permet d'écrire en une ligne: la valeur initiale de la variable, l'expression qui détermine si on rentre dans la boucle et le moyen de modifier la variable.

**for**(**var** i=0;i<5;i++) {  
  *// cette boucle tourne 5 fois, pour i valant 0, 1, 2, 3, 4*  
}

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:for)

## for in

Permet de parcourir toutes les propriétés d'un objet.

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:for...in)

## for each in

Permet de parcourir toutes les valeurs des propriétés d'un objet.

[Plus d'info](http://developer.mozilla.org/fr/docs/R%C3%A9f%C3%A9rence_de_JavaScript_1.5_Core:Instructions:for_each...in)

**Fonctions**

Les fonctions permettent de regrouper plusieurs instructions et de les exécuter dans un contexte propre.

On peut déclarer une fonction de différentes manières, voici des exemples:

*// fonction sans paramètre*  
**function** test1() {  
  *// insctructions*  
}  
  
*// fonction avec un paramètre*  
**function** test2(p) {  
  *// instructions*  
}  
  
*// fonction avec trois paramètres*  
**function** test3(p1, p2, p3) {  
  *// instructions*  
}

Dans ces exemples, les fonctions sont nommées **test1**, **test2** et **test3**, ces noms doivent respecter certaines règles de nommage.

Les fonctions **test2** et **test3** attendent un ou plusieurs paramètres qui doivent aussi respecter ces mêmes règles de nommage.

# Appels de fonctions

Une fonction peut être appelée (exécutée) depuis un endroit du programme avec la syntaxe suivante:

test(); *// fonction sans paramètre*  
truc(x, y); *// fonction avec des paramètres*

Dans le cas d'une utilisation normale, les paramètres attendus par la fonction sont renseignés lors de l'appel. (voir [paramètres de fonction](http://www.coursweb.ch/javascript/func-params.html))

## Valeur de retour

Une fonction javascript peut retourner une valeur à l'appelant. Voici un exemple:

**function** test() {  
  **return** 1;  
}  
  
**var** x = test(); *// x prend la valeur 1*

Ici la fonction **test** est appelée, elle retourne une valeur (grâce à **return**). Cette valeur est affecté à la variable **x**.

En javascript on ne peut pas savoir si une fonction retourne ou non une valeur. C'est la documentation (ou le code source) qui permet de le savoir. On peut très bien ignorer la valeur de retour d'une fonction qui en retournerai une.

**function** test() {  
  alert('Bonjour');  
  **return** 12;  
}  
  
test(); *// appel de fonction*

Ici on n'exploite pas la valeur de retour (**12**). Mais la fonction est tout de même exécutée et **Bonjour** est affiché.

On peut aussi tenter de traiter une hypothétique valeur de retour d'une fonction qui ne retourne rien:

**function** test() {  
  alert('Coucou');  
}  
  
**var** x = test(); *// x vaut undefined*

Dans ce cas, x prend la valeur **undefined** (non défini), mais la fonction est tout de même appelée sans provoquer d'erreur.

# Paramètres des fonctions

Les paramètres sont des informations transmises à une fonction lors d'un appel. Ces informations peuvent être différentes d'un appel à l'autre.

Exemple simple:

**function** calculeSurface(largeur, hauteur) {  
  **return** largeur \* hauteur; *// calcul la surface et la retourne*  
}

Cette fonction calcule la surface d'un rectangle. Pour ce faire, elle a besoin de connaitre la largeur et la hauteur du rectangle. Ces deux informations sont passées en paramètre.

Exemple d'utilisation:

**var** s = calculeSurface(8, 4); *// s prend 32 comme valeur*  
alert( calculeSurface(2, 3) ); *// affiche 6*

Comme on le voit ici, la fonction est appelée deux fois, mais avec des valeurs différentes. Les valeurs données lors d'un appel sont copiées dans les paramètres dans le même ordre. Lors du premier appel, largeur vaut 8 et hauteur vaut 4.

## Paramètres optionnels

En javascript il est possible de passer plus de paramètres qu'elle n'en attend dans sa signature. Il est aussi possible d'en donner moins, au même aucun. La fonction sera tout de même appelée. Cependant c'est la fonction qui doit gérer le surplus ou le manque de paramètres.

### Manque de paramètre

Si on reprend l'exemple ci-dessus pour calculer la surface d'un rectangle on pourrait admettre que si un seul paramètre est donné il s'agirait d'un carré (largeur identique à la hauteur). On peut gérer ça ainsi:

**function** calculeSurface(largeur, hauteur) {  
  **if**(**typeof** hauteur == 'undefined') hauteur = largeur;  
  **return** largeur \* hauteur; *// calcul la surface et la retourne*  
}  
  
**var** surface\_care = calculeSurface(12); *// retourne 144*

### Excédent de paramètre

Une fonction peut aussi gérer des paramètre excédentaires. Comme il n'y a pas de variable de paramètre déclarée dans la signature de la fonction elle doit utiliser le tableau **arguments** qui contient toutes les valeurs des paramètres dans l'ordre.

Voici un exemple d'une fonction qui calcule la somme de toutes les valeurs passées en paramètres. Comme le nombre de paramètres n'est pas connu d'avance, on doit utiliser une boucle.

**function** somme() {  
  **var** r = 0; *// résultat*  
  **for**(**var** i=0;i<arguments.length;i++) {  
    r += arguments[i]; *// ajoute au résultat*  
  }  
  **return** r;  
}  
  
alert( somme(1, 2, 3) ); *// affiche 6*  
alert( somme(4) ); *// affiche 4*  
alert( somme() ); *// affiche 0*

**Fonctions anonymes**

Les fonctions anonymes ne portent pas de nom. On les utilise beaucoup pour passer des fonctions de retour en paramètre, ou pour la déclarations des prototypes. Pour utiliser une fonction anonyme il faut une référence dessus.

Voici un exemple:

**var** r = **function**() { alert('Bonjour'); };

Ici un crée une fonction anonyme et on sauve sa référence dans une variable **r**.

Pour appeler la fonction on peut écrire:

r(); *// Affiche Bonjour*

**Fonctions de retour (callback functions)**

Une fonction de retour (callback) est une fonction comme les autres. Sa particularité est qu'elle est appelée par une autre qui l'a reçu en tant que paramètre. Voici une fonction qui reçoit une fonction en paramètre:

**function** test(fct\_retour) {  
  fct\_retour(); *// appel de la fonction*  
}

C'est la fonction **test** qui appelle la fonction de retour. Il est donc possible de donner différentes fonctions de retour entre deux appels successifs à la fonction test.

**function** retour1() {  
  alert('Retour 1');  
}  
  
**function** retour2() {  
  alert('Retour 2');  
}  
  
test(retour1); *// affiche 'Retour 1'*  
test(retour2); *// affiche 'Retour 2'*

On soulignera une première chose **importante**. Quand on passe une fonction en paramètre, on donne juste son nom, sans les parenthèses. La présence de parenthèses provoque un appel de fonction.

test(retour1()); *// Faux !*  
test(retour1); *// Juste*

Le premier appel ci-dessus ne passe pas la fonction retour1 en paramètre, mais le résultat retourné par celle-ci.

Dans l'exemple ci dessus on aurait pu passer une fonction anonyme en paramètre à la fonction **test** comme ceci:

test( **function**() { alert('Hello'); } );

Dans ce cas précis on préfère écrire le code ainsi:

test(  
  **function**() {  
    alert('Hello');  
  }  
);

**DHTML et DOM**

Le [DHTML](http://fr.wikipedia.org/wiki/DHTML) (Dynamique HTML) est une technologie mélangeant HTML, Javascript, CSS qui permet de rendre les pages web dynamiques. C'est à dire que leur contenu et apparence peuvent changer sans recharger la page en gérant des interactions avec l'utilisateur.

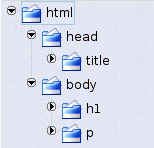
Le [DOM](http://fr.wikipedia.org/wiki/Document_Object_Model) (document object model) est un standard permettant la manipulation de documents depuis des langages informatiques. Dans le cas qui nous intéresse, le DOM représente une page XML, HTML ou XHTML. Et le langage qui la manipule est le javascript.

Il existe plusieurs versions du DOM (voir le lien ci-dessus), actuellement c'est la version [DOM-2](http://www.yoyodesign.org/doc/w3c/dom2-html/Overview.html) qui prédomine.

Le DOM est un moyen de représenter un document sous forme d'une arborescence (comparable à l'organisation des dossiers et fichiers sur un disque dur). Prenons l'exemple d'une page XHTML (épurée pour simplifier)

[**<html>**](http://december.com/html/4/element/html.html)  
  [**<head>**](http://december.com/html/4/element/head.html)  
    [**<title>**](http://december.com/html/4/element/title.html)Titre du document**</title>**  
  **</head>**  
  [**<body>**](http://december.com/html/4/element/body.html)  
    [**<h1>**](http://december.com/html/4/element/h1.html)Titre**</h1>**  
    [**<p>**](http://december.com/html/4/element/p.html)Un peu de texte**</p>**  
  **</body>**  
**</html>**

Ce qui peut être vu sous la forme de l'arborescence ci-dessous



Les éléments d'une page XHTML sont imbriqués les uns dans les autres. On dit qu'un élément imbriqué dans un autre est un enfant de celui-ci. L'élément principal (celui qui contient tous les autres) est appelé élément racine (root element). Ici c'est **html**.

On peut dire les affirmations suivantes concernant ce document:

* **html** est le parent direct de **head** et de **body**, ses deux enfants.
* **h1** est le premier enfant de **body**, **p** est le deuxième, il est frère de **h1**
* **title** est l'unique enfant de **head**.

Vu sous cette forme, il est plus simple de manipuler le document à l'aide de javascript. Il est possible d'ajouter, modifier ou supprimer des élements. On peut changer les attributs de chacun. On peut appliquer des styles CSS à tout ou partie des éléments.

Chaque modification du DOM est immédiatement visible dans le navigateur. Ceci sans devoir recharger une autre page web.

# Repérer les éléments

Avant de pouvoir manipuler un élément du document, il faut pouvoir le repérer (le référencer). Pour cela il faut pouvoir le trouver dans le document. Le DOM offre de nombreuses fonctionnalités pour ça.

## Le repérage par l'attribut ID

L'attribut ID d'un élément est (doit être) unique au sein d'un même document. Pour repérer un élément dont on connait l'ID on écrit:

**var** e = document.getElementById('id\_de\_l\_element');

Dans ce cas, la variable **e** prend la valeur d'une référence vers l'élément. Il est dés lors possible de manipuler cet élément en utilisant la variable e.

## Le repérage par le nom de l'élément

Chaque élément porte un nom, **h1**, **p**, **div**, etc... c'est ce que l'on appel le **tagName**. A ne pas confondre avec l'attribut **name** que peuvent avoir certains éléments.

le DOM offre une fonction **getElementsByTagName()** qui permet d'obtenir la liste de tous les éléments voulu. Par exemple:

**var** gros\_titres = document.getElementsByTagName('h1');

Cette fonction renvoie toujours un tableau d'éléments. Celui-ci peut être vide si aucun élément n'existe, il peut contenir un ou plusieurs éléments qui sont dans l'ordre dans lequel ils apparaissent dans le code source de la page.

**var** gros\_titres = document.getElementsByTagName('h1');  
**var** premier\_titre = gros\_titres[0];  
**var** second\_titre = gros\_titres[1];

## Repérage par l'attribut name

Il est aussi possible d'obtenir tous les éléments qui ont un attribut **name** de valeur spécifique. Contrairement à l'attribut **id** qui est unique, plusieurs éléments peuvent avoir le même attribut **name**. Cet attribut est généralement utilisé pour les champs de formulaires. Exemple typique, les boutons **radio**. Il y a plusieurs éléments **input** qui portent le même nom. Par exemple pour un choix de couleurs:

[<input](http://december.com/html/4/element/input.html) type='radio' name='couleur' value='rouge' /**>**  
[<input](http://december.com/html/4/element/input.html) type='radio' name='couleur' value='vert' /**>**  
[<input](http://december.com/html/4/element/input.html) type='radio' name='couleur' value='bleu' /**>**

Si l'on veut traiter ces champs on pourrait écrire:

**var** boutons = document.getElementsByName('couleur');

Ce qui dans cet exemple, nous retournerai un tableau de 3 éléments **input**

## Autres types de repérage

les fonctions du DOM pour repérer les éléments sont de bas niveau. Heureusement, de nombreuses fonctions existent dans les framework

Deux exemples qui sont souvent utiles et qui ne sont pas disponibles directement dans l'[API](http://fr.wikipedia.org/wiki/Interface_de_programmation) du DOM

* obtenir les éléments qui ont un attribut class spécifique
* obtenir les éléments qui correspondent à un selecteur CSS

# Modifier les styles CSS

Une fois que l'on a [repéré un élément](http://www.coursweb.ch/javascript/elements-matching.html) on peut agir sur les styles CSS de celui-ci. Ceci par deux moyens principaux:

## La propriété className

Cette propriété permet d'agir sur l'attribut **class** d'un élément. Pour rappel, cet attribut permet d'associer des styles défini dans des feuilles de style aux éléments. Pour autant qu'il y aie un sélecteur qui corresponde. On peut associer plusieurs classes CSS à un élément en les séparant par un espace.

**var** e = document.getElementById('logo');  
e.className = 'gros';

Ce code fixe l'attribut class à **gros** pour l'élément dont l'id est **logo**

Si l'élément en question disposait déjà d'un attribut class, celui-ci est remplacé. C'est pourquoi certains frameworks proposent des fonctions plus évoluées qui permettent d'ajouter ou de retirer un réglage à l'attribut class (en gérant les espaces entre les noms de classes des qu'il y a plus d'un réglage).

## La propriété style

La propriété style permet d'accéder finement aux styles qui concernent un élément particulier, de façon similaire à l'attribut style dans une balise.

**var** e = document.getElementById('logo');  
e.style.color = 'red';  
e.style.backgroundColor = 'yellow';  
e.style.fontSize = '12px';

L'usage de la propriété **style** est simple à deux exception près:

1) Les propriété CSS qui sont faites de mots composés, tel que **background-color** ne peuvent s'écrire ainsi en javascript, c'est pourquoi il faut supprimer le tiret (signe moins) et mettre en majuscule la lettre qui le suit, par exemple: **backgroundColor**.

2) Certains noms de styles sont en conflit avec des [mots réservés du langage javascript](http://www.coursweb.ch/javascript/js-keywords.html), par exemple **float** dans ce cas de figure il faudra utiliser le nom **styleFloat** pour IE et **cssFloat** pour les autres navigateurs. Cette différence entre navigateurs est peu pratique à gérer. Il est souvent plus simple d'utiliser la propriété **className** et de régler le **float** dans la classe correspondante.

# Les événements

Les éléments contenus dans une page web ont pour la plupart la possibilité de détecter certaines actions de l'utilisateur comme le click de la souris, les touches clavier et bien d'autres. Ces événements peuvent être rattaché à une fonction javascript qui est appelée lorsque l'événement se produit.

Voici quelques exemples d'événement

|  |  |
| --- | --- |
| onmousedown | Appui sur un bouton de la souris |
| onmouseup | Relâchement d'un bouton de la souris |
| onmousemove | Déplacement de la souris |
| onmouseover | Entrée sur un élément |
| onmouseout | Sortie d'un élément |
| onkeypress | Appui d'une touche qui produit un caractère |
| onkeydown | Touche enfoncée |
| onkeyup | Touche relâchée |
| onclick | Click souris ou action par la touche Entrée de l'élément qui a le focus |
| onload | Chargement de la page |
| onsubmit | Envoi de formulaire |
| onfocus | Réception du focus |
| onblur | Perte du focus |

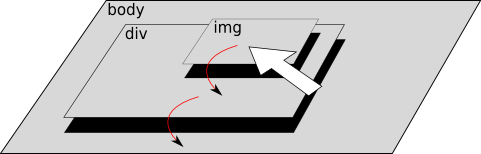
## Model de gestion des événements

La gestion des événements dans un navigateur est assez complexe. Pour en savoir d'avantage [suivre ce lien](http://www.yoyodesign.org/doc/w3c/dom2-events/Overview.html)

Il y a trois concepts importants à comprendre dans la gestion des événements. La capture, la remontée (bubbling) et le traitement par défaut. Imaginons une page html simple dont le code serait le suivant:

[<body>](http://december.com/html/4/element/body.html)  
  [<div>](http://december.com/html/4/element/div.html)  
    [<img](http://december.com/html/4/element/img.html) src='...' /**>**  
  **</div>**  
**</body>**

Et voici une représentation de cette page



Il faut voir les éléments comme disposés en couches. Les enfants étant plus en haut que les parents.

## La remontée des événements (bubbling)

Lorsque l'utilisateur clique sur l'image, l'événement va se propager vers les éléments parents. Il peut être traité par l'image, par le div ou par le body.

Si une fonction javascript est attachée au onclick de l'image, celle-ci sera appelée en priorité. Si une autre fonction est attachée au onclick du div, elle sera aussi exécutée, mais après celle de l'image.

Si une fonction décide de stopper la propagation, alors l'événement ne sera plus accessible aux éléments parents.

## La capture

Avant que l'image ne soit informée du click, les éléments sont consultés dans l'ordre de parenté (ici: body, div, img) pour savoir si l'un d'entre eux a demandé une capture de l'événement.

Si par exemple, le div a demandé la capture du onclick, alors l'image ne sera pas informée du click (la fonction javascript attachée ne sera donc pas exécutée). Par contre, la fonction attachée au div le sera. Et l'événement remontera vers le body. (pour autant que la fonction du div n'aie pas demander de stopper la propagation)

## Le comportement par défaut

Certains événement souris ou clavier ont un comportement par défaut. Deux exemples:

1) Si on clique droite sur une image, un menu contextuel apparaît pour permettre entre autre d'enregistrer l'image sur le disque.

2) Quand on utilise la combinaison de touches **CTRL + V** et qu'un champ de formulaire a le focus, on colle ce qui est dans le presse papier dans ce champ.

Il est tout a fait possible de déclencher une action personnalisée quand l'utilisateur clique droite sur une image, mais cette action sera suivie du traitement par défaut de l'événement, soit l'apparition du menu contextuel.

Pour palier à ce problème il est possible d'empêcher le comportement par défaut d'un événement.

**Attacher des évènements**

On parle ici d'attacher une fonction javascript qui doit être exécutée lorsqu'un évènement se produit. Il existe plusieurs façons de procéder. Voici l'ancienne méthode que l'on rencontre encore dans beaucoup de pages web existantes.

[**<h1**](http://december.com/html/4/element/h1.html) onclick='test();'**>**Titre**</h1>**

Ici on dit que la fonction **test** doit être appelée lorsque l'utilisateur clique sur l'élément **h1**.

Cette manière de faire n'est plus conseillée, il est préférable d'attacher l'évènement depuis javascript. On modifierait le code html ainsi:

[**<h1**](http://december.com/html/4/element/h1.html) id='toto'**>**Titre**</h1>**

Puis en javascript on peut écrire:

**var** e = document.getElementById('toto');  
e.onclick = test;

Dans cet exemple, on repère l'élément voulu (ici grâce à son id). Puis on affecte à la propriété **onclick** une référence vers la fonction **test**. Attention de ne pas écrire **test()**, ceci appellerai la fonction **test** immédiatement puis affecterai son résultat à la propriété **onclick**.

Le problème avec cette approche (tout comme la précédente), c'est que l'on est limité à une seule fonction pour réagir au click sur un élément.

La manière moderne c'est l'utilisation de la méthode [addEventListener](http://developer.mozilla.org/fr/docs/DOM:element.addEventListener) que l'on pourrait traduire par "ajouter un écouteur d'évènement". Cette méthode permet donc de lier plusieurs fonctions pour un même évènement.

Malheureusement, Internet Explorer ne respecte pas ce standard du w3c et propose une méthode alternative [attachEvent](http://developer.mozilla.org/fr/docs/DOM:element.addEventListener#Internet_Explorer) qui ne supporte pas la phase de capture dans la gestion des évènements.

Pour palier à ce problème d'incompatibilité, les frameworks proposent souvent une fonction de ce type:

**function** addEvent(e, evType, fn) {  
  **if**(e.addEventListener) {  
    e.addEventListener(evType, fn, **false**); *// sans capture*  
    **return** **true**;  
  } **else** **if** (e.attachEvent) {  
    **var** r = e.attachEvent("on"+evType, fn);  
    **return** r;  
  } **else** { **return** **false**; }  
}

Ici, on test si addEventListener est supportée si oui on l'utilise, sinon on utilise attachEvent. On notera aussi que le paramètre **evType** correspond au nom de l'évènement sans le **on** qui le précède. (onclick devient juste click).

Voici un exemple qui assigne deux fonctions au chargement de la page:

addEvent(window, 'load', fonction1);  
addEvent(window, 'load', fonction2);

C'est un cas très fréquent d'assigner plusieurs fonctions au chargement de la page (évènement onload de window), c'est moins fréquent d'assigner plusieurs fonctions quand on clique sur une image ou un lien.

Quoi qu'il en soit, il est recommandé d'utiliser cette dernière manière de faire car elle évite des problèmes lorsque plusieurs scripts (notamment des librairies de divers horizons) sont utilisées dans une même page.

# Traiter les évènements

Une fonction attachée à un évènement reçoit en paramètre un objet qui hérite de l'interface [Event](http://www.yoyodesign.org/doc/w3c/dom2-events/events.html#Events-interface). Cet objet permet d'obtenir des informations supplémentaires sur l'évènement.

**function** handleClick(event) {  
  *// traitement*  
}

Une fois encore, Internet Explorer diffère du standard. Si bien que l'objet n'est pas passé en tant que paramètre mais réside dans une variable de l'objet **window**. Pour récupérer l'objet **Event** dans tous les cas de figure on peut écrire:

**function** handleClick(event) {  
  **var** event = event || window.event;  
}

Cette notation crée une variable locale dont la valeur est **event** si celui-ci est défini ou **window.event** dans le cas contraire, ce qui résouts notre problème de manière assez élégante.

L'objet Event (qui en réalité est une interface) est la base pour les différents type d'évènements (souris, clavier, etc.)

Parmis les informations importantes qu'il comporte il y a la propriété **target** qui permet de savoir quel élément html est le premier concerné par l'évènement. Dans le cas où on attache onclick à un div qui contient une image, **target** représentera l'image si on clique dessus, et le div si on clique sur le div mais pas sur l'image.

Là aussi il faut traiter le cas d'Internet Explorer qui utilise la propriété srcElement.

**function** handleClick(event) {  
  **var** event = event || window.event;  
  **var** target = event.target || event.srcElement;  
}

## Traiter les évènements souris

Dans le cas d'un mousedown, mousemove etc... on peut obtenir des informations tel que la position de la souris, quel bouton a été cliqué etc... Ceci grâce à l'objet **Event** qui est de type [MouseEvent](http://www.yoyodesign.org/doc/w3c/dom2-events/events.html#Events-eventgroupings-mouseevents)

**function** handleMouseMove(event) {  
  **var** event = event || window.event;  
  **var** x = event.screenX; *// position de la souris sur l'écran*  
}

## Traiter les évènements clavier

Les évènements clavier souffrent d'un manque de standard pour le moment. On peut toujours essayer d'obtenir le code de la touche avec:

**function** handleKey(event) {  
  **var** event = event || window.event;  
  **var** touche = event.keyCode;  
}

Mais aussi **event.charCode** ou **event.which**, lire ces articles pour comprendre les problèmes de compatibilité:

<http://dev.webnaute.net/DOM/Events/keyboardEvent/> <http://www.brainjar.com/dhtml/events/default4.asp>

## Stopper la propagation (bubbling)

Pour empêcher le traitement de l'évènement par les parents on peut écrire:

**function** handleClick(event) {  
  **var** event = event || window.event;  
  **if** (event.stopPropagation) { *// Standard*  
    event.stopPropagation();  
  } **else** { *// Pour IE*  
    event.cancelBubble = **true**;  
  }  
}

## Empêcher le traitement par défaut

**function** handleClick(event) {  
  **var** event = event || window.event;  
  **if** (event.preventDefault) { *// Standard*  
    event.preventDefault();  
  } **else** { *// Pour IE*  
    event.returnValue = **false**;  
  }  
}

# Objets

Un objet décrit un ensemble de données (propriétés) et/ou de fonctions (méthodes) encapsulés ensembles. La programmation orientée objet permet de résoudre certains problèmes de manière plus élégante (plus rentable) que auparavant, mais elle ne remplace pas la programmation traditionnelle, c'est un complément.

Il faut faire la différence entre un objet et une instance. Par analogie on pourrait dire qu'un nombre entier est un objet et que les nombres 12, 423, etc.. sont des instances. Ou alors qu'un rectangle est un objet et qu'un rectangle de 13cm par 5cm est une instance.

L'objet est abstrait, généraliste. L'instance est réelle concrète.

L'objet décrit ce qui le caractérise, en terme de propriétés (variables membre) et de méthodes (fonction membres)

# Prototypes

En javascript on crée des objets d'une manière un peu différente des langages orientés objets purs (Java, PHP, C++). On dit que javascript n'est pas vraiment un langage orienté objet mais qu'il est un langage à prototypes. Ceci n'empêche pas pour autant de reproduire ce que l'on peut faire avec un langage orienté objet pur, et cela a aussi des avantages propre aux langages à prototypes qui permettent de faire des choses très intéressantes.

l'opérateur **this** permet d'accéder aux propriétés (ou fonctions membres) appartenant à l'instance d'un objet.

## Une fonction "est un objet"

Pour créer un objet en javascript on utilise une fonction.

*// Déclare une fonction constructeur*  
**function** Rectangle(largeur, hauteur) {  
  **this**.largeur = largeur; *// sauve la largeur dans l'instance*  
  **this**.hauteur = hauteur; *// sauve la hauteur dans l'instance*  
}  
  
**var** r1 = **new** Rectangle(10, 4); *// crée un instance*  
**var** r2 = **new** Rectangle(12, 8); *// crée une autre instance*

Les variables r1 et r2 sont des références vers des instances de **Rectangle**. On peut accéder au propriétés des instances de cette manière:

alert(r1.largeur); *// affiche la largeur du rectangle r1*  
alert(r2.hauteur); *// affiche la hauteur du rectangle r2*

On utilise l'opérateur **new** pour demander de créer une instance. Chaque instance occupe un peu de mémoire pour stocker les propriétés qui lui sont propre. Deux instances d'un même objet diffèrent par la valeur de leur propriétés. Il est possible qu'il existe des instances qui ont les mêmes valeurs, on parle alors de clones.

## Ajouter des méthodes

Pour ajouter des méthodes à un objet, on étend son prototype:

Rectangle.prototype.surface = **function**() {  
  **return** **this**.largeur \* **this**.hauteur;  
}

Ceci permet de calculer la surface de nos rectangle r1 et r2 ainsi:

alert( r1.surface() ); *// affiche 40*  
alert( r2.surface() ); *// affiche 96*

## Définir le prototype en une fois

Afin de coder toutes les méthodes de façon compact on peut utiliser [JSON](http://www.coursweb.ch/javascript/json.html) pour définir le prototype complètement

Rectangle.prototype = {  
  surface: **function**() {  
    **return** **this**.largeur \* **this**.hauteur;  
  },  
  
  perimetre: **function**() {  
    **return** (**this**.largeur + **this**.hauteur) \* 2;  
  }  
};

# Etendre des objets

En javascript il est possible d'ajouter des fonctionnalités à des objets (les vôtres ou ceux qui sont existants), Sur certains navigateurs il est aussi possible d'étendre les objets du DOM.

Pour étendre un objet on ajoute une fonction à son prototype, par exemple on peut étendre l'objet String avec une fonctionnalité qui permettrait de compter certains caractères.

String.prototype.countChars = **function**(c) {  
  **var** cpt = 0;  
  **for**(**var** i=0;i<this.length;i++) {  
    **if**(**this**.charAt(i)==c) cpt++;  
  }  
  **return** cpt;  
}  
  
**var** t = "Je suis un texte";  
alert(t.countChars(' ')); *// affiche 3*

## Particularité d'un langage à prototypes

Dans le code suivant on déclare un objet puis on crée deux instances **a** et **b**. Ensuite on ajoute une propriété au prototype de l'objet **Test**. Cette nouvelle caractéristique est disponible pour toutes instances (déjà crées ou pas encore crées).

**function** Test() { *// crée un objet Test*  
  **this**.abc = 444; *// initialise une propriété*  
}  
  
**var** a = **new** Test(); *// crée une instance*  
**var** b = **new** Test(); *// crée une instance*  
Test.prototype.toto = 12; *// ajoute un propriété au prototype*  
a.titi = 4; *// modifie une propriété de l'instance a*  
  
alert(a.toto); *// affiche 12*  
alert(a.titi); *// affiche 4*  
alert(b.toto); *// affiche 12*  
alert(b.titi); *// affiche undefined*

On observera que le fait d'affecter une propriété quelconque à une instance (ici **a.titi = 4**) n'affecte que l'instance **a** et non le prototype de l'objet **Test**.

**Apply et Call**

Lorsqu'une fonction est exécutée elle est liée à une instance d'un objet. Elle peut accéder aux propriétés et méthodes de cette instance avec l'opérateur **this**, par exemple:

**function** test() {  
  **this**.x = 12; *// modifie la propriété x de l'instance*  
  alert(**this**.y); *// affiche la valeur de la propriété y de l'instance*  
  **this**.toto(); *// appelle la fonction toto de l'instance*  
}

Contrairement aux langages de POO classiques, une fonction javascript peut "appartenir" à plusieurs objets. Et donc être appelée dans des contextes d'instances différents, on peu utiliser **apply** et **call** pour réaliser ça.

**function** getHtmlColor() {  
  **return** "#" + **this**.couleur;  
}  
  
**function** Voiture(code\_couleur) {  
  **this**.couleur = code\_couleur;  
}  
  
**function** Rectangle(larg, haut, rgb) {  
  **this**.couleur = rgb;  
  **this**.largeur = larg;  
  **this**.hauteur = haut;  
}  
  
**var** v = **new** Voiture('ccff33');  
**var** r = **new** Rectangle(10, 12, 'ff0033');  
  
alert( getHtmlColor.apply(v) ); *// affiche #ccff33*  
alert( getHtmlColor.apply(r) ); *// affiche #ff0033*

La fonction getHtmlColor fait usage de l'opérateur **this** qui lui permet d'accéder à la propriété **couleur** de l'instance. L'instance concernée correspond à la variable **v** ou **r** donnée comme paramètre de **apply**.

**apply** et **call** fonctionnent de la même manière à une différence. La manière de passer les paramètres à la fonction appelée. **apply** attend deux paramètres, le premier étant une référence à l'instance, le deuxième étant un tableau contenant les arguments. **call** attend au moins un paramètre (celui de l'instance) et propage les paramètres reçus en excédent à la fonction appelée.

une\_fonction.apply(instance, tableau);  
une\_fonction.call(instance, p1, p2, p3, ...);

**Closure (fermeture)**

Dans le cas typique des fonctions évènementielles, on a pas la maitrise des paramètres qui leurs sont envoyés. En effet, c'est le gestionnaire d'évènement qui donne les paramètres au moment de l'appel. Celui-ci se limite d'ailleurs à ne donner qu'un paramètre, une référence à un objet Event.

**function** handleClick(event) {  
  *// traiement de l'évènement*  
}  
  
addEvent(un\_element, 'click', handleClick);

Dans certains cas on souhaite utiliser une fonction qui traite un évènement et lui donner des paramètres différents.

**function** getClickHandler(x, y) {  
  **var** z = 12;  
  **return** **function**(event) {  
    alert(x + y + z);  
  }  
}  
  
addEvent(un\_element, 'click', getClickHandler(3,4));  
addEvent(un\_autre\_element, 'click', getClickHandler(5,3));

Ici nous sommes en présence d'une closure et d'une fonction anonyme imbriquée. Celle-ci a accès aux variables et paramètres de la fonction imbriquante **getClickHandler**.

Ainsi il est possible d'attacher une fonction évènementielle paramétrée. La fonction interne sera appelée au moment de l'évènement mais elle aura toujours accès au variables et paramètres de sa fonction imbriquante quand bien même celle-ci a terminé son exécution depuis.

**Evènements et instances**

Les fonctions liées à des éléments du DOM sont appelées lors du déclenchement d'un évènement avec un contexte d'instance qui représente l'élément du DOM. C'est souvent pratique, mais si l'on a besoin de gérer d'autre informations c'est parfois réducteur.

Voici un exemple d'objet qui sert à compter le nombres de clics sur un élément DOM et qui affiche un message après 10 clics.

**function** Compteur(element) {  
  **this**.counter =  10; *// variable pour compter*  
  addEvent(element, 'click', **this**.count); *// attache l'évènement*  
}  
  
Compteur.prototype.count = **function**() {  
  **this**.counter--; *// diminue counter de 1*  
  **if**(**this**.counter==0) alert('Message');  
}  
  
*// crée un compteur pour un élement*  
**new** Compteur( document.getElementById('element1') );  
*// crée un autre compteur pour un autre élément*  
**new** Compteur( document.getElementById('element2') );

Dans cet exemple il y a deux instances de l'objet **Compteur**. Le problème se situe au niveau de la fonction **count**, celle-ci est attachée au gestionnaire d'évènements et sera appelée quand on clic sur l'élément DOM lié.

**this** fait référence à l'élément du DOM et non à l'instance de l'objet **Compteur**. Ainsi **this.counter** fait référence à la propriété **counter** de l'élément du DOM. Cette propriété n'existe pas dans un élément DOM.

**apply** et **call** permettent de palier à ce problème. On attachera une fonction anonyme au gestionnaire d'évènement qui s'occupera de la liaison (binding).

**function** Compteur(element) {  
  **this**.counter =  10; *// variable pour compter*  
  **var** instance = **this**; *// crée une 'closure'*  
  addEvent(element, 'click', **function**() {  
    instance.count.apply(instance);  
  });  
}

**Binding**

Le binding est une technique permettant d'associer une instance d'un objet à une fonction par le biais d'une closure.

L'idée consiste à étendre le prototype de l'objet Function directement et de lui offire une méthode facilitant la liaison.

**Function**.prototype.bind = **function**(context) {  
  **var** m = **this**; *// référence l'instance de Function*  
  **return** **function**() {  
    **return** m.apply(context, arguments);  
  }  
}

Maintenant chaque fonction dispose d'une nouvelle méthode **bind**. Dans l'exemple de la page [précédente](http://www.coursweb.ch/javascript/event-instance.html) on avait utilisé **apply** pour faire l'appel. Dés maintenant il est plus simple d'écrire ceci:

**function** Compteur(element) {  
  **this**.counter =  10; *// variable pour compter*  
  addEvent(element, 'click', **this**.count.bind(**this**));  
}

**addEvent** ne reçoit plus une référence direct à **this.count** mais reçoit une référence à la fonction interne anonyme. Lorsque l'évènement se produit, c'est ce code qui sera d'abord exécuté:

**return** m.apply(instance, arguments);

Ce code invoque la méthode **apply** de l'instance **m**. **m** est connu de la fonction grâce à la [closure](http://www.coursweb.ch/javascript/closure.html). **m** correspond à l'instance de la fonction this.count dans ce cas. C'est donc this.count qui sera appelé.

Le premier paramètre de **apply** est l'instance souhaitée en tant que contexte. C'est le premier paramètre passé à **bind**. En l'occurrence c'est **this** lors de l'appel à **addEvent**. Dans ce contexte **this** référencie l'instance de **Compteur**.

# AJAX

AJAX est l'acronyme pour Asynchronous Javascript and XML, ce n'est pas un langage, c'est une technologie qui utilise différents langages et sous technologies.

Avant AJAX, une application web (ou site web) imposait à l'utilisateur de devoir recharger une page entière après chaque interaction. Ceci impliquait la nécessité d'attendre que le résultat soit revenu du serveur avant de pouvoir faire une autre interaction. Ce résultat représentant une page complète, il était souvent long à transférer. De ce fait les applications web ont souvent été considérées comme peu réactives en comparaison avec les applications locales.

Depuis AJAX il est possible pour le navigateur de dialoguer avec le serveur sans changer de page web. De plus il est possible de modifier des parties de page web après leur affichage et ceci sans les recharger complètement. Mais c'est n'est pas la seule révolution. Une amélioration significative de la gestion des évènements utilisateur en javascript, les nouvelles possibilités de [CSS](http://fr.wikipedia.org/wiki/Feuilles_de_style_en_cascade) et la nouvelle souplesse du [DOM](http://fr.wikipedia.org/wiki/Document_Object_Model) sont intimement liés à la naissance d'AJAX.

AJAX est la brique qui manquait pour pouvoir réaliser des applications web de qualité qui peuvent concurrencer leur équivalent locaux. On est encore loin de réaliser des outils comme Photoshop sous forme d'application Web. Par contre cela fait longtemps de les webmails n'ont plus rien à envier au clients mail locaux. Aujourd'hui apparaissent des applications web qui concurrence directement Word et Excel.

[AJAX sur Wikipedia](http://fr.wikipedia.org/wiki/AJAX)

**Requêtes serveur**

Pour envoyer des requêtes [HTTP](http://fr.wikipedia.org/wiki/HTTP) au serveur, on utilise le plus souvent l'objet [XMLHttpRequest](http://fr.wikipedia.org/wiki/XMLHttpRequest) bien qu'il existe d'autres [alternatives](http://ajaxpatterns.org/XMLHttpRequest_Call#Alternatives).

Il y a malheureusement une différence entre les navigateurs pour obtenir une instance de l'objet. C'est pourquoi on crée généralement une fonction permettant de l'obtenir et traitant les différences. Voici un exemple d'une telle fonction:

**function** getXhr() {  
  **try** { **return** **new** XMLHttpRequest(); } **catch** (e) { } *// Standard*  
  **try** { **return** **new** ActiveXObject("Msxml2.XMLHTTP"); } **catch** (e) { }  
  **try** { **return** **new** ActiveXObject("Microsoft.XMLHTTP"); } **catch** (e) { }  
  **return** **null**;  
}

Une fois l'instance créée on peut faire des requêtes HTTP pour obtenir des fichiers depuis le serveur. Attention, seules les requêtes adressées au même domaine d'où provient le script sont autorisées (pour des raisons de sécurité).

Lors de l'envoi d'une requête, il faut donner une référence à une fonction qui gèrera la réponse. Cette fonction sera appelée pour les différents états de progression de la requête. A savoir:

|  |  |
| --- | --- |
| 0 | non initialisé |
| 1 | ouverture. La méthode open() a été appelée avec succès |
| 2 | envoyé. La méthode send() a été appelée avec succès |
| 3 | en train de recevoir. Des données sont en train d'être transférées, mais le transfert n'est pas terminé |
| 4 | terminé. Les données sont chargées |

Généralement c'est l'étape 4 qui nous intéresse. C'est à cet instant que l'on peut traiter les données. Pour autant que la réponse soit du type 200 (code HTTP qui signifie que tout c'est bien passé).

Voici un exemple de fonction qui traite la réponse:

**function** response(xhr) {  
  **if** (xhr.readyState==4) {  
    **if** (xhr.status==200) {  
      *// Traitement de la réponse*  
      alert(xhr.responseText);  
    }  
  }

Cette fonction est en réalité appelée pour chaque étape de la requête. C'est pourquoi il y a un premier **if** qui test que c'est bien l'étape 4 (terminé). A l'intérieur on test le code HTTP de réponse. Si les deux tests passent, alors on peut traiter les données reçues.

Voici un exemple qui utilise les deux fonctions précédentes pour charger un fichier texte.

**var** xhr = getXhr();  
xhr.onreadystatechange = response; *// fonction de traitement*  
xhr.open("GET", "/fichier.txt", **true**); *// prépare une requête asynchrone*  
xhr.send(**null**); *// envoie la requête*

[Plus d'info sur XMLHttpRequest](http://www.openweb.eu.org/articles/objet_xmlhttprequest/)

# Ajax et le référencement

Par nature, une requête Ajax fait appel à Javascript pour obtenir une ressource. Donc les moteurs de recherche ne sont pas capable aujourd'hui de suivre ces liens.

Prenons l'exemple d'un site Web de trois pages utilisant Ajax pour charger le contenu des pages sans recharger la totalité de la page affichée (habillage et navigation sont conservés).

La navigation pourrait ressembler à Ceci:

[<ul](http://december.com/html/4/element/ul.html) id='nav'**>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='javascript:load("page1.html")'**>**Page 1**</a></li>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='javascript:load("page2.html")'**>**Page 2**</a></li>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='javascript:load("page3.html")'**>**Page 3**</a></li>**  
**</ul>**

Dans un site web traditionnel la navigation ressemblerai plutôt à ceci:

[<ul](http://december.com/html/4/element/ul.html) id='nav'**>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page1.html'**>**Page 1**</a></li>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page2.html'**>**Page 2**</a></li>**  
  [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page3.html'**>**Page 3**</a></li>**  
**</ul>**

Pour palier au problème du référencement, les moteurs de recherche devraient avoir accès à cette deuxième version.

Les deux ne sont pas incompatibles techniquement. Il suffit de produire un code html pour les moteurs de recherche et de court-circuiter les liens avec Javascript pour faire des chargement via Ajax.

**function** initNav() {  
  **var** nav = document.getElementById('nav');  
  **var** liens = nav.getElementsByTagName('A');  
  **for**(**var** i=0;i<liens.length;i++) liens[i].onclick = load;  
}

La fonction initNav doit être exécutée au chargement de la page et la fonction load doit interrompre le traitement par défaut du click et nous avons résolu notre premier problème.

## Fonctionnement sans Javascript

Si javascript est désactivé alors les urls définis dans le href vont être utilisés quand on clique sur les liens. Le navigateur devra recevoir un contenu de page complet (de même pour les moteurs de recherche).

Dans le cadre d'une application Ajax les contenus que l'on souhaite charger sont partiels (pas de DOCTYPE, ni de body), juste le contenu, c'est à dire des fragments html.

Pour palier à ce problème il faut pouvoir produire les deux types de contenu depuis le serveur. Le plus simple étant de séparer les fragments html dans des fichiers distincts et de travailler avec un model html contenant l'habillage du site et sa navigation mais sans contenu.

Exemple de structure du site:

model.html  
fragments/page1.html  
fragments/page2.html  
fragments/page3.html  
index.php

La page model.html pourrait ressembler à ceci

<!DOCTYPE ...>  
[<html](http://december.com/html/4/element/html.html) ...**>**  
  [<head>](http://december.com/html/4/element/head.html)...**</head>**  
  [<body>](http://december.com/html/4/element/body.html)  
    [<ul](http://december.com/html/4/element/ul.html) id='nav'**>**  
      [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page1.html'**>**Page 1**</a></li>**  
      [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page2.html'**>**Page 2**</a></li>**  
      [<li>](http://december.com/html/4/element/li.html)[<a](http://december.com/html/4/element/a.html) href='page3.html'**>**Page 3**</a></li>**  
    **</ul>**  
    [<div](http://december.com/html/4/element/div.html) id='contenu'**>**[[CONTENU]]**</div>**  
  **</body>**  
**</html>**

On peut utiliser php pour produire les page avec un code du genre:

$page = $\_GET['page'];  
$fragment = [file\_get\_contents](http://www.php.net/file_get_contents)("fragments/$page");  
$model = [file\_get\_contents](http://www.php.net/file_get_contents)('model.html');  
[echo](http://www.php.net/echo) [str\_replace](http://www.php.net/str_replace)('[[CONTENU]]', $fragment, $model);

Les urls produisants des pages complètes sont de la forme: **index.php?page=page1.html** et ceux permettant d'obtenir des fragments sont ainsi: **fragments/page1.html** (ceux-ci ne seront jamais indexés)

En utilisant l'[url rewriting](http://fr.wikipedia.org/wiki/.htaccess) on peut facilement transformer les urls comme **page1.html** en **index.php?page=page1.html**

Nous avons maintenant une application totalement fonctionnelle sans javascript et qui peut être référencée par les moteurs de recherche.

## Gestion de l'url avec Ajax

Pour éviter le rechargement de la page avec Ajax on ne peut pas se permettre de changer l'url de la page, ce qui rentre en conflit avec ce qui a été écrit plus haut et avec une règle qui dit qu'à chaque contenu doit correspondre un url et réciproquement.

Heureusement il est possible de modifier l'ancre d'un url sans recharger la page. (les ancres servant initialement à se déplacer dans une même page).

On peut modifier l'url de la barre d'adresse du navigateur en javascript sans recharger la page en appliquant une ancre avec **document.location.replace('#page1.html')** Ceci doit être fait après chaque requête Ajax qui charge un nouveau contenu.

Les urls de notre application Ajax ressemblent donc à

/#page1.html  
/#page2.html  
/#page3.html

Il est important que l'application Ajax examine l'url du navigateur lors du premier chargement de la page de manière à charger le bon contenu. Ces urls comportant des ancres peuvent être utilisés suite à la mise en favoris d'une page par exemple.

## Conflit lors de l'arrivée sur le site par une page référencée.

Si on arrive sur le site par l'url page2.html référencé par un moteur de recherche il n'y a pas d'ancre, l'application Ajax pourrait traiter ce cas et charger le contenu de la page2 initialement. Mais ensuite quand on navigue dans le site sur la page3 on aurait un url **page2.html#page3.html** ce qui n'est pas très cohérent d'une part, et qui contredit la règle d'unicité entre le contenu et les urls d'autre part.

L'application Ajax devrait donc faire une redirection en javascript vers l'url **/#page2.html** aussi vite que possible, avant l'affichage du corps de la page (en plaçant un script dans l'entête).

## Exemple fonctionnel

Voici un petit exemple fonctionnel à placer sur votre serveur (qui doit supporter le .htaccess pour l'url rewriting) [Télécharger](http://www.coursweb.ch/sections/javascript/files/ajax-referencement.zip) (2.58 ko)

**Validation de formulaire**

Exemple de formulaire html

[**<form**](http://december.com/html/4/element/form.html) id='f1' action='test.php' method="get"**>**  
[**<label**](http://december.com/html/4/element/label.html) class='req' for='f1\_nom'**>**Nom**</label>**  
[**<input**](http://december.com/html/4/element/input.html) id='f1\_nom' type="text" name="nom" /**>**  
[**<label**](http://december.com/html/4/element/label.html) for='f1\_prenom'**>**Prénom**</label>**  
[**<input**](http://december.com/html/4/element/input.html) id='f1\_prenom' type="text" name="prenom" /**>**  
[**<label**](http://december.com/html/4/element/label.html) class='req' for='f1\_ec'**>**Etat civil**</label>**  
[**<select**](http://december.com/html/4/element/select.html) name='etat\_civil' id='f1\_ec'**>**  
  [**<option**](http://december.com/html/4/element/option.html) value=''**>**--Choisir--**</option>**  
  [**<option**](http://december.com/html/4/element/option.html) value='Cel'**>**Célibataire**</option>**  
  [**<option**](http://december.com/html/4/element/option.html) value='Mar'**>**Marié**</option>**  
  [**<option**](http://december.com/html/4/element/option.html) value='Div'**>**Divorcé**</option>**  
**</select>**  
[**<input**](http://december.com/html/4/element/input.html) type="submit" value="Envoyer" /**>**  
**</form>**

Exemple de script de validation de formulaire

**var** TXT\_ERRMSG\_REQUIRED = "Le champ '{0}' est requis";  
  
*// affiche un message en substituant les parametre à un message type*  
**function** \_message(message) {  
        **for**(**var** i=1;i<\_message.arguments.length;i++) {  
                message = message.replace('{'+(i-1)+'}', \_message.arguments[i]);  
        }  
        window.alert(message);  
}  
  
**function** addEvent(e, evType, fn) {  
  **if**(e.addEventListener) {  
    e.addEventListener(evType, fn, **false**); *// sans capture*  
    **return** **true**;  
  } **else** **if** (e.attachEvent) {  
    **var** r = e.attachEvent("on"+evType, fn);  
    **return** r;  
  } **else** { **return** **false**; }  
}  
  
addEvent(window, 'load', initialisation);  
  
**function** initialisation(event) {  
        **var** forms = document.getElementsByTagName('form');  
        **for**(**var** i=0;i<forms.length;i++) {  
                addEvent(forms[i], 'submit', validation);  
        }  
}  
  
**function** estVide(id, n) {  
  **var** e = document.getElementById(id);  
  **var** type = e.type ? e.type.toLowerCase() : (e.length ? e[0].type.toLowerCase() : '' );  
  **if**(type=='select') {  
    **var** v = e.options[e.selectedIndex].value;  
    **if**(v.length!=0) **return** **false**;  
  }  
  **else** **if**(type=='radio') {  
    **var** form = e.parentNode;  
    **while**(form.tagName.toLowerCase()!='form') form = form.parentNode;  
    **var** inputs = form.getElementsByTagName('input');  
    **for**(**var** i=0;i<inputs.length;i++) **if**(inputs[i].name==e.name && inputs[i].checked) **return** **false**;  
  }  
  **else** {  
    **var** v = e.value;  
    **if**(v.length!=0) **return** **false**;  
  }  
  alert("Le champ " + n + " est obligatoire");  
  e.focus();  
  **return** **true**;  
}  
  
**function** testChamps(champs) {  
  **for**(**var** i=0;i<champs.length;i++) {  
    **var** label = champs[i];  
        **if**(label.className!='req') **continue**;  
        **if**( estVide(label.getAttribute('for'), label.innerHTML) ) **return** **true**;  
  }  
  **return** **false**;  
}  
  
**function** validation(event) {  
  **if**( testChamps(**this**.getElementsByTagName('label')) ) {  
          **var** event = event || window.event;  
          **if** (event.preventDefault) { *// Standard*  
                event.preventDefault();  
          } **else** { *// Pour IE*  
                event.returnValue = **false**;  
          }  
  }  
}

# Annexes

# Mots réservés en Javascript

Les mots réservés en javascript sont des mots que l'on ne peut pas utiliser comme noms de variables, de propriétés ou noms de fonctions.

Ils ont un usage précis dans un contexte donné.

Lorsque vous travaillez avec un éditeur javascript, ils sont automatiquement coloriés ce qui permet de les éviter facilement.

## Mot réservés en JavaScript

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| break | continue | do | for | import | new | this | void |
| case | default | else | function | in | return | typeof | while |
| comment | delete | export | if | label | switch | var | with |

## Mots réservé en Java (Reservés par JavaScript)

|  |  |  |
| --- | --- | --- |
| abstract | implements | protected |
| boolean | instanceOf | public |
| byte | int | short |
| char | interface | static |
| double | long | synchronized |
| false | native | throws |
| final | null | transient |
| float | package | true |
| goto | private |  |

## Mot réservé ECMAScipt

|  |  |  |
| --- | --- | --- |
| catch | enum | throw |
| class | extends | try |
| const | finally |  |
| debugger | super |  |

## Autres mots réservés

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| alert | eval | Link | outerHeight | scrollTo |
| Anchor | FileUpload | location | outerWidth | Select |
| Area | find | Location | Packages | self |
| arguments | focus | locationbar | pageXoffset | setInterval |
| Array | Form | Math | pageYoffset | setTimeout |
| assign | Frame | menubar | parent | status |
| blur | frames | MimeType | parseFloat | statusbar |
| Boolean | Function | moveBy | parseInt | stop |
| Button | getClass | moveTo | Password | String |
| callee | Hidden | name | personalbar | Submit |
| caller | history | NaN | Plugin | sun |
| captureEvents | History | navigate | print | taint |
| Checkbox | home | navigator | prompt | Text |
| clearInterval | Image | Navigator | prototype | Textarea |
| clearTimeout | Infinity | netscape | Radio | toolbar |
| close | innerHeight | Number | ref | top |
| closed | innerWidth | Object | RegExp | toString |
| confirm | isFinite | onBlur | releaseEvents | unescape |
| constructor | isNan | onError | Reset | untaint |
| Date | java | onFocus | resizeBy | unwatch |
| defaultStatus | JavaArray | onLoad | resizeTo | valueOf |
| document | JavaClass | onUnload | routeEvent | watch |
| Document | JavaObject | open | scroll | window |
| Element | JavaPackage | opener | scrollbars | Window |
| escape | length | Option | scrollBy |  |

**Opérateurs**

Voici une listes des principaux opérateurs en javascript. Plus de détail [ici](http://fr.wikibooks.org/wiki/Programmation_JavaScript/Op%C3%A9rateurs)

|  |  |
| --- | --- |
| = | affectation |
| + | addition |
| - | soustraction (ou moins unaire) |
| \* | multiplication |
| / | division |
| % | modulo |
| ++ | pré et post incrémentation |
| -- | pré et post décrémentation |
| ! | non (boolean) |
| && | et (boolean) |
| || | ou (boolean) |
| == | égal |
| != | différent |
| >= | plus grand ou égal |
| < | plus petit que |
| <= | plus petit ou égal |
| > | plus grand que |
| var | déclaration de variable |
| new | instanciation d'objets |
| this | référence à l'instance courante |

# Liens

## Exemples de scripts

[DHTMLGoodies](http://www.dhtmlgoodies.com/)   
Pour découvrir les possibilités du DHTML

[Cours Web Ludomedia](http://coursweb.ludomedia.ch)   
Quelques exemples simples en DHTML

[Listamatic](http://css.maxdesign.com.au/listamatic/)   
Exemples de liste (menu) en CSS

[Scriptaculous](http://script.aculo.us)   
Librairie drag drop et plus

## Tutoriaux, cours

[Référence Javascript](http://developer.mozilla.org/fr/docs/JavaScript)   
Sur le site Mozilla Developer Center

[Self HTML](http://fr.selfhtml.org/javascript/)   
Introduction au javascript et au DOM

[Developpez.com](http://www.developpez.com)   
Cours, tutoriaux, et forum concernant de multiples langages et technologies

## Frameworks et librairies

[Documentation de Prototype](http://www.prototypejs.org/api)

[Documentation de Scriptaculous](http://wiki.script.aculo.us/scriptaculous/)

[Yahoo UI](http://developer.yahoo.com/yui/)

[Open Rico](http://openrico.org/)

[Google Maps API](http://www.google.com/apis/maps/)

[Affichage d'image](http://www.huddletogether.com/projects/lightbox2/)

## Flash

[Tutoriels flash pour les vidéos](http://www.gotoandlearn.com/index.php)

# Exercices

Cliquez dans la navigation pour accèder aux exercices.