Dynamisez vos sites web avec JavaScript

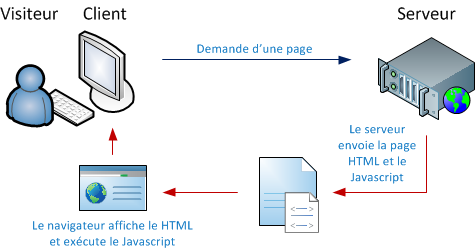
## Introduction au JavaScript

Le JavaScript est un langage de programmation de scripts orienté objet. Un **langage de programmation** est un langage qui permet aux développeurs d'écrire du **code source** qui sera analysé par l'ordinateur. C'est un ensemble d'actions, appelées **instructions**, qui vont permettre de donner des ordres à l'ordinateur afin de faire fonctionner le programme. Il existe trois manières d'utiliser du code source :

* **Langage compilé** : le code source est donné à un programme appelé **compilateur** qui va lire le code source et le convertir dans un langage que l'ordinateur sera capable d'interpréter : c'est le langage binaire, fait de 0 et de 1. Les langages comme le C ou le C++ sont des langages dits compilés.
* **Langage précompilé** : ici, le code source est compilé partiellement, généralement dans un code plus simple à lire pour l'ordinateur, mais qui n'est pas encore du binaire. Ce code intermédiaire devra être lu par ce que l'on appelle une « machine virtuelle », qui exécutera ce code. Les langages comme le C# ou le Java sont dits précompilés.
* **Langage interprété** : dans ce cas, il n'y a pas de compilation. Le code source reste tel quel, et si on veut exécuter ce code, on doit le fournir à un interpréteur qui se chargera de le lire et de réaliser les actions demandées. Éventuellement, pour obtenir de significatifs gains de performances, le code peut-être compilé à la volée pendant son exécution, c'est aujourd'hui ce que font la plupart des interpréteurs JavaScript.

Chaque navigateur possède un interpréteur JavaScript, qui diffère selon le navigateur. Le JavaScript est majoritairement utilisé sur Internet, conjointement avec les pages Web HTML. Le JavaScript s'inclut directement dans la page Web (ou dans un fichier externe) et permet de dynamiser une page HTML.

Le JavaScript est un langage dit ***client-side***, c'est-à-dire que les scripts sont exécutés par le navigateur chez le **client**.



Si le JavaScript a été conçu pour être utilisé conjointement avec le HTML, le langage. Il est régulièrement utilisé pour réaliser des extensions pour différents programmes, Chrome et Firefox possèdent tous deux un panel gigantesque d'extensions en partie codées en JavaScript.

Mais là où le JavaScript a su grandement évoluer ces dernières années c'est dans la possibilité d'être exécuté sur n'importe quelle machine. Quelques projets permettent d'utiliser le JavaScript en dehors de votre navigateur, le projet le plus connu est sans nul doute Node.js.

**Les versions du JavaScript**

Les versions du JavaScript sont basées sur celles de l'ECMAScript (que nous abrégeons ES). Ce cours portera sur la version 5 de l'ECMAScript. L’ES 6 a été finalisé en décembre 2014.

## Premiers pas en JavaScript

**Afficher une boîte de dialogue**

Voici un code HTML simple contenant une instruction JavaScript placée au sein d'un élément <script>:

<!DOCTYPE html>

<html>

<head>

<title>Hello World!</title>

</head>

<body>

<script>

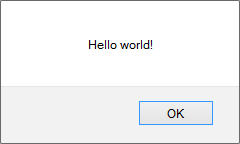
alert('Hello world!');

</script>

</body>

</html>

alert() est une instruction simple, appelée **fonction**, qui permet d'afficher une boîte de dialogue contenant un message :



**La syntaxe du JavaScript**

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

instruction\_1;

instruction\_2;

En réalité le point-virgule n'est pas obligatoire si l'instruction qui suit se trouve sur la ligne suivante.

**Les espaces**

Le JavaScript n'est pas sensible aux espaces. Cela veut dire que nous pouvons aligner des instructions comme nous le voulons, sans que cela ne gêne l'exécution du script.

**Les commentaires**

Ils servent à commenter une instruction. Le texte placé dans un commentaire est ignoré lors de l'exécution du script. Un commentaire mono-ligne commence par deux slashs :

// Ceci est un commentaire

Un commentaire multi-ligne commence par /\* et se termine par \*/ :

/\* Ceci est un commentaire :

- Blabla 1

- Blabla 2 qui fait autre chose

\*/

**Les fonctions**

Une fonction se compose de deux choses : son nom, suivi d'un couple de parenthèses :

myFunction();

Entre les parenthèses se trouvent les **arguments**, que l'on appelle aussi **paramètres**. Ceux-ci contiennent des valeurs qui sont transmises à la fonction.

**Où placer le code dans la page**

Les codes JavaScript sont insérés au moyen de l'élément <script>. Cet élément possède un attribut type qui sert à indiquer le type de langage que l'on va utiliser. En HTML 4 et XHTML 1.x, l'attribut type est obligatoire. En revanche, en HTML5, il ne l'est pas. C'est pourquoi les exemples de ce cours, en HTML5, ne comporteront pas cet attribut. Si nous n'utilisons pas le HTML5, l'attribut type prend comme valeur text/javascript, qui est le type MIME d'un code JavaScript.

**Le JavaScript « dans la page »**

Pour placer du code JavaScript directement dans une page Web, on place le code au sein de l'élément <script> :

<script>

alert('Hello world!');

</script>

**Le JavaScript externe**

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

<script src="hello.js"></script> // Appelle le contenu du fichier “hello.js »

On suppose ici que le fichier hello.js se trouve dans le même répertoire que la page Web. Il vaut mieux privilégier un fichier externe plutôt que d'inclure le code JavaScript directement dans la page, pour la simple et bonne raison que le fichier externe est mis en cache par le navigateur et n'est donc pas rechargé à chaque chargement de page, ce qui accélère l’affichage de la page.

**Positionner l'élément <script>**

Une page Web est lue par le navigateur de façon linéaire, c'est-à-dire qu'il lit d'abord le <head>, puis les éléments de <body> les uns à la suite des autres. Si nous appelons un fichier JavaScript dès le début du chargement de la page, le navigateur va donc charger ce fichier, et si ce dernier est volumineux, le chargement de la page s'en trouvera ralenti. C'est normal puisque le navigateur va charger le fichier avant de commencer à afficher le contenu de la page. Pour pallier ce problème, il est conseillé de placer les éléments <script> juste avant la fermeture de l’élément <body>.

## Les variables

Une variable est un espace de stockage sur votre ordinateur permettant d'enregistrer tout type de données, que ce soit une chaîne de caractères, une valeur numérique ou bien des structures un peu plus particulières.

**Déclarer une variable**

Le nom d'une variable ne peut contenir que des caractères alphanumériques. L'underscore **(\_)** et le dollar **($)** sont aussi acceptés. Le nom de la variable ne peut pas commencer par un chiffre et ne peut pas être constitué uniquement de mots-clés utilisés par le JavaScript.

Pour déclarer une variable, il suffit d'écrire var myVariable; . Il est à noter que le JavaScript est un langage sensible à la casse (donc myVariable != myvariable).

Pour y stocker une données, il suffit d’écrire myVariable = 2; . Quand on donne une valeur à une variable, on dit que l'on fait une affectation, car on affecte une valeur à la variable.

Il est possible de simplifier ce code en une seule ligne : var myVariable = 5.5; .

De même, il est possible de déclarer et assigner des variables sur une seule et même ligne :

var myVariable1, myVariable2 = 4, myVariable3;

Quand nous utilisons une seule fois l'instruction var pour déclarer plusieurs variables, nous devons placer une virgule après chaque variable et ne placer le point-virgule qu'à la fin de la déclaration de toutes les variables.

Enfin une dernière chose qui pourra être utile de temps en temps :

var myVariable1, myVariable2;

myVariable1 = myVariable2 = 2;

**Les types de variables**

Le JavaScript est un langage typé dynamiquement. Cela veut dire, généralement, que toute déclaration de variable se fait avec le mot-clé var sans distinction du contenu. Cela veut aussi dire que l'on peut y mettre du texte en premier lieu puis l'effacer et y mettre un nombre quel qu'il soit, et ce, sans contraintes. Voici les trois types principaux en JavaScript :

* **Le type numérique (alias number)** : il représente tout nombre, que ce soit un entier, un nombre négatif, scientifique, etc. Le JavaScript reconnaît plusieurs écritures pour les nombres, comme l'écriture décimale var number = 5.5; , l'écriture scientifique var number = 3.65e+5; ou encore l'écriture hexadécimale var number = 0x391;.
* **Les chaînes de caractères (alias string)** : ce type représente n'importe quel texte. On peut l'assigner de deux façons différentes :

var text1 = "Mon premier texte";

var text2 = 'Mon deuxième texte';

Si nous utilisons les apostrophes pour « encadrer » le texte, il faudra « échapper » ces apostrophes pour les utiliser à l’intérieur du texte :

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

* **Les booléens (alias boolean)**

**Tester l'existence de variables avec typeof**

Il se peut que nous ayons besoin de tester l'existence d'une variable ou d'en vérifier son type. Dans ce genre de situations, l'instruction typeof est très utile :

var number = 2;

alert(typeof number); // Affiche : « number »

var text = 'Mon texte';

alert(typeof text); // Affiche : « string »

var aBoolean = false;

alert(typeof aBoolean); // Affiche : « boolean »

Voici comment tester l'existence d'une variable :

alert(typeof nothing); // Affiche : « undefined »

Si l'instruction typeof renvoie undefined, c'est soit que la variable est inexistante, soit qu'elle est déclarée mais ne contient rien.

**Les opérateurs arithmétiques**

Ces derniers sont à la base de tout calcul et sont au nombre de cinq :

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

Le modulo est le reste d'une division. Par exemple, si nous divisons 5 par 2 alors il reste 1, c'est le modulo !

Pour effectuer des calculs, comme en Java number = number + 5; équivaut à number += 5;. Ceci fonctionne avec tous les autres opérateurs arithmétiques (\*=, %=, /= etc..).

**Initiation à la concaténation et à la conversion des types**

L'opérateur + permet également de faire ce que l'on appelle des **concaténations** entre des chaînes de caractères. Une concaténation consiste à ajouter une chaîne de caractères à la fin d'une autre :

var hi = 'Bonjour', name = 'toi', result;

result = hi + name;

alert(result); // Affiche : « Bonjourtoi »

Autre exemple :

var text = 'Bonjour ';

text += 'toi';

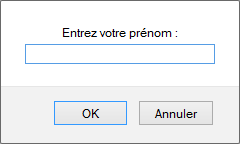
alert(text); // Affiche « Bonjour toi ».

**Interagir avec l'utilisateur**

La fonction prompt() s'utilise comme alert() mais elle renvoie ce que l'utilisateur a écrit sous forme d'une chaîne de caractères. Voilà pourquoi on écrit de cette manière : var text = prompt('Tapez quelque chose :');. Par exemple :

var userName = prompt('Entrez votre prénom :');

Cela affichera :



**Convertir une chaîne de caractères en nombre**

Cela se fait via la fonction parseInt() qui s'utilise de cette manière :

var text = '1337', number;

number = parseInt(text);

alert(typeof number); // Affiche : « number »

alert(number); // Affiche : « 1337 »

**Convertir un nombre en chaîne de caractères**

Il est possible de concaténer un nombre et une chaîne sans conversion, mais pas deux nombres, car ceux-ci s'ajouteraient à cause de l'emploi du +. D'où le besoin de convertir un nombre en chaîne :

var text, number1 = 4, number2 = 2;

text = number1 + '' + number2;

alert(text); // Affiche : « 42 »

Il existe une solution un peu moins archaïque que de rajouter une chaîne vide que nous découvrirons plus tard.

## Les conditions

Les opérateurs de comparaison vont permettre de comparer diverses valeurs entre elles :

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

On les utilise de la manière suivante :

result = number1 == number2;

result = number1 < number3;

Cela renvoie true si condition est vérifiée, sinon renvoie false.

Les opérateurs === et !== peuvent être difficiles à comprendre. Voici un exemple explicitant leur utilisation :

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

result = number == text;

alert(result); // Affiche « true » alors que « number » est un nombre et « text » une chaîne de caractères

result = number === text;

alert(result); // Affiche « false » car cet opérateur compare aussi les types des variables en plus de leurs valeurs

Les conditions « normales » font des conversions de type pour vérifier les égalités, ce qui fait que si nous voulons différencier le nombre 4 d'une chaîne de caractères contenant le chiffre 4 il faudra alors utiliser le triple égal ===.

**Les opérateurs logiques**

Ils fonctionnent sur le même principe qu'une table de vérité en électronique :

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

**La condition « if else »**

Exemple :

if (true) {

alert("Ce message s'est bien affiché.");

}

if (false) {

alert("Pas la peine d'insister, ce message ne s'affichera pas.");

}

Une condition est constituée :

* De la structure conditionnelle if.
* De parenthèses qui contiennent la condition à analyser.
* D'accolades qui permettent de définir la portion de code qui sera exécutée si la condition se vérifie.

**Petit intermède : la fonction confirm()**

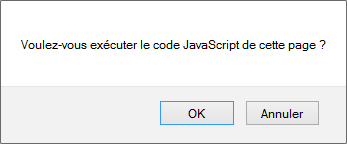
On lui passe en paramètre une chaîne de caractères qui sera affichée à l'écran et elle retourne un booléen en fonction de l'action de l'utilisateur :

if (confirm('Voulez-vous exécuter le code JavaScript de cette page ?')) {

alert('Le code a bien été exécuté !');

}

La fonction confirm() affichera :



Le fait que cliquer sur le bouton « Ok » renverra true. Sinon, confirm() renverra false.

**La structure else pour dire « sinon »**

Exemple :

if (confirm('Pour accéder à ce site vous devez avoir 18 ans ou plus, cliquez sur "OK" si c\'est le cas.')) {

alert('Vous allez être redirigé vers le site.');

} else {

alert("Désolé, vous n'avez pas accès à ce site.");

}

La structure else permet d'exécuter un certain code si la condition n'a pas été vérifiée.

**La structure else if pour dire « sinon si »**

La structure else if permet de fonctionner de la façon suivante :

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

var floor = parseInt(prompt("Entrez l'étage où l'ascenseur doit se rendre (de -2 à 30) :"));

if (floor == 0) {

alert('Vous vous trouvez déjà au rez-de-chaussée.');

} else if (-2 <= floor && floor <= 30) {

alert("Direction l'étage n°" + floor + ' !');

} else {

alert("L'étage spécifié n'existe pas.");

}

**La condition « switch »**

La function switch est très pratique pour faire du cas par cas :

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

switch (drawer) {

case 1:

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

break;

case 2:

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

break;

case 3:

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

break;

case 4:

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

break;

default:

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

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

Un switch permet de faire une action en fonction d'une valeur mais aussi en fonction du type de la valeur (comme l'opérateur ===). Par exemple :

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

// N’affichera pas bravo

switch (drawer) {

case 1:

alert('Bravo !');

break;

}

// N’affichera pas bravo);

switch (drawer) {

case ‘1’:

alert('Bravo !');

break;

}

**Les ternaires**

Avec les ternaires nous pouvons simplifier notre code de façon substantielle :

var startMessage = 'Votre catégorie : ',

endMessage,

adult = confirm('Êtes-vous majeur ?');

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

alert(startMessage + endMessage);

Comment fonctionnent les ternaires ? Pour le comprendre il faut regarder la ligne suivante du code précédent : endMessage = adult ? '18+' : '-18';. Si l'on décompose cette ligne on peut voir :

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

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

**Les conditions sur les variables**

Il est possible de tester si une variable possède une valeur sans même utiliser l'instruction typeof. Tout se joue au niveau de la conversion des types. Les variables peuvent être de plusieurs types : les nombres, les chaînes de caractères, etc. Eh bien le type d'une variable, quel qu'il soit, peut être converti en booléen même si à la base on possède un nombre ou une chaîne de caractères :

var conditionTest = 'Fonctionnera ? Fonctionnera pas ?';

if (conditionTest) {

alert('Fonctionne !');

} else {

alert('Ne fonctionne pas !');

}

Le code nous affiche le texte « Fonctionne ! » parce que la variable conditionTest a été convertie en booléen et que son contenu est évalué comme étant vrai (true). False est renvoyé uniquement pour un nombre qui vaut zéro ou une chaîne de caractères vide. La valeur undefined est aussi évaluée à false.

**Le cas de l'opérateur OU**

L'opérateur OU, en plus de sa fonction principale, permet de renvoyer la première variable possédant une valeur évaluée à true :

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

alert(conditionTest1 || conditionTest2);

Ce code nous retournera la valeur « Une chaîne de caractères ».

Remarque : La fonction parseInt() renverra la valeur NaN (évaluée à false) qui signifie Not a Number si l’argument à parser n’est pas un nombre.

## Les boucles

L'incrémentation permet d'ajouter une unité à un nombre au moyen d'une syntaxe courte. À l'inverse, la décrémentation permet de soustraire une unité.

var number = 0;

number++;

number--;

**L'ordre des opérateurs**

Il existe deux manières d'utiliser l'incrémentation en fonction de la position de l'opérateur :

number\_1++;

++number\_2;

La différence réside en fait dans la priorité de l'opération, et ça a de l'importance si vous voulez récupérer le résultat de l'incrémentation. Dans l'exemple suivant, ++number retourne la valeur de number incrémentée :

var number = 0;

var output = ++number;

alert(number); // Affiche : « 1 »

alert(output); // Affiche : « 1 »

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

var number = 0;

var output = number++;

alert(number); // Affiche : « 1 »

alert(output); // Affiche : « 0 »

**La boucle while**

À chaque fois que la boucle se répète on parle d'**itération**. Pour faire fonctionner une boucle, il est nécessaire de définir une condition. Tant que celle-ci est vraie (true), la boucle se répète. Dès que la condition est fausse (false), la boucle s'arrête. Voici un exemple de la syntaxe d'une boucle while :

while (condition) {

instruction\_1;

}

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

Une **variable témoin**, ou bien **variable de boucle**, est une variable qui n'intervient pas directement dans les instructions de la boucle mais qui sert juste pour tester la condition. Pour arrêter une boucle, on peut également utiliser le mot clé break dans une des conditions de la boucle.

**Utilisation de continue**

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

**La boucle do while**

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

do {

instruction\_1;

} while (condition);

**La boucle for**

Le schéma d'une boucle for est le suivant :

for (initialisation; condition; incrémentation) {

instruction\_1;

}

Dans les parenthèses de la boucle trouvent trois blocs : **initialisation**, **condition**, et **incrémentation**. Ces trois blocs sont séparés par un point-virgule ;. Par exemple :

for (var iter = 0; iter < 5; iter++) {

alert('Itération n°' + iter);

}

Une boucle for peut également être utilisée sans incrémentation :

for (var nicks = '', nick; true;) {

nick = prompt('Entrez un prénom :');

if (nick) {

nicks += nick + ' ';

} else {

break;

}

}

alert(nicks);

On sortira donc de cette boucle dès que l’on atteindra l’instruction break. Cependant, le bloc d’itération étant requis, on doit quand même mettre le point-virgule après le deuxième bloc (la condition).

**Portée des variables de boucle**

En JavaScript, il est déconseillé de déclarer des variables au sein d'une boucle (entre les accolades), pour une raison logique : il n'y a en effet pas besoin de déclarer une même variable à chaque passage dans la boucle ! Il est conseillé de déclarer les variables directement dans le bloc d'initialisation, comme montré dans les exemples précédents. Une fois que la boucle est exécutée, la variable existe toujours. Ce comportement est différent de celui de nombreux autres langages, dans lesquels une variable déclarée dans une boucle est « détruite » une fois la boucle exécutée.

**Priorité d'exécution**

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

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

## Les fonctions

**Concevoir des fonctions**

Quand on parle de fonction ou variable native, il s'agit d'un élément déjà intégré au langage utilisé. Voici comment écrire ses propres fonctions :

function myFunction(arguments) {

// Le code que la fonction va devoir exécuter

}

* Le mot-clé function est présent à chaque déclaration de fonction.
* Vient ensuite le nom de la fonction, ici myFunction.
* S'ensuit un couple de parenthèses contenant ce que l'on appelle des **arguments**. Ces arguments servent à fournir des informations à la fonction lors de son exécution.
* Vient enfin un couple d'accolades contenant le code que la fonction devra exécuter.

Tout comme les variables, les noms de fonctions sont limités aux caractères alphanumériques (et aux deux caractères **\_** et **$**. L’exemple suivant montre comment déclarer puis utiliser une fonction :

function byTwo() {

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

alert(result \* 2);

}

byTwo();

**La portée des variables**

Il est important de faire attention à la portée des variables. Prenons par exemple le code suivant :

function sayHello() {

var ohai = 'Hello world !';

}

sayHello();

alert(ohai);

Cela ne donne aucun résultat car il produit ce que l'on appelle une erreur. L'erreur en question (nous allons revenir sur l'affichage de cette erreur dans un futur chapitre) nous indique que la variable ohai n'existe pas au moment de son affichage avec la fonction alert(). Cela est dû au fait que **toute variable déclarée dans une fonction n'est utilisable que dans cette même fonction.** Ces variables spécifiques à une seule fonction ont un nom : les **variables locales**.

**Les variables globales**

Celles déclarées en-dehors d'une fonction sont nommées les **variables globales** car elles sont accessibles partout dans notre code, y compris à l'intérieur de nos fonctions. Si nous créons une variable globale nommée message et une variable locale du même nom au sein d’une fonction, la variable locale prend le dessus sur la variable globale de même nom pendant tout le temps de l'exécution de la fonction. Mais une fois la fonction terminée, c'est la variable globale qui reprend ses droits. Il est plutôt déconseillé de créer des variables globales et locales de même nom.

**Les arguments et les valeurs de retour**

Les arguments permettent d'envoyer des informations à notre fonction tandis que les valeurs de retour représentent tout ce qui est retourné par la fonction une fois que celle-ci a fini de travailler.

**Créer et utiliser un argument**

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

function myFunction(arg) {

alert('Votre argument : ' + arg);

}

**La portée des arguments**

Lorsqu'une fonction reçoit un argument, celui-ci est stocké dans une variable dont nous avons choisi le nom lors de la déclaration de la fonction. Ce fonctionnement est exactement le même que lorsque nous créons nous-mêmes une variable dans la fonction : elle ne sera accessible que dans cette fonction et nulle part ailleurs. Les arguments sont propres à leur fonction, ils ne serviront à aucune autre fonction.

**Les arguments multiples**

Si une fonction a besoin de plusieurs arguments pour fonctionner il faudra les écrire de la façon suivante : function moar(first, second) {…}. Les différents arguments sont séparés par une virgule.

**Les arguments facultatifs**

Admettons que nous créions une fonction basique pouvant accueillir un argument mais que l'on ne le spécifie pas à l'appel de la fonction :

function optional(arg) {

alert(arg); // On affiche l'argument non spécifié pour voir ce qu'il contient

}

optional();

Nous obtiendrons undefined. La variable arg a été déclarée par la fonction mais pas initialisée car nous ne lui avons pas passé d'argument. Le contenu de cette variable est donc indéfini.

Pour l’exemple, imaginons que l'on décide de créer une fonction qui affiche à l'écran une fenêtre et autorise ou non l'utilisateur à quitter la fenêtre sans entrer de texte. Dans un cas comme celui-ci, l’argument facultatif peut être utilisé pour permettre au non cela :

function prompt2(text, allowCancel) {

if (typeof allowCancel === 'undefined') { // Souvenez-vous de typeof, pour vérifier le type d'une variable

allowCancel = false;

}

// Le code qui effectuera une action en function de la Valeur de l’argument facultative allowCancel.

}

prompt2('Entrez quelque chose :'); // On exécute la fonction seulement avec le premier argument, pas besoin du deuxième

**Les valeurs de retour**

Les fonctions ne peuvent retourner qu'une seule et unique valeur chacune, qui peut être un tableau ou un objet. Pour faire retourner une valeur à notre fonction, il suffit d'utiliser l'instruction return suivie de la valeur à retourner :

function sayHello() {

return 'Bonjour !'; // Renvoi de la valeur ‘Bonjour’.

}

La présence du return met fin à la fonction, puis retourne la valeur. Toute instruction écrite après le return ne sera donc pas effectuée.

**Les fonctions anonymes**

Ces fonctions spéciales sont anonymes car elles ne possèdent pas de nom. C’est la seule et unique différence avec une fonction traditionnelle. Pour déclarer une fonction anonyme, il suffit de faire comme pour une fonction classique mais sans indiquer de nom :

function (arguments) {

// Le code de notre fonction anonyme

}

Il existe plusieurs solutions que nous verrons au fur et à mesure pour exécuter de telle fonctions. L’une d’entre elles est de l‘assigner à une variable :

var sayHello = function() {

alert('Bonjour !');

};

sayHello(); // Affiche : « Bonjour ! »

Nous noterons ici la présence d’un « ; » après les accolades. Le point-virgule est ici nécessaire car nous assignons une fonction à une variable. Il est donc nécessaire de mettre un point-virgule comme pour une initialisation « classique » de variable.

Si nous voulons immédiatement exécuter une fonction anonyme , nous pouvons utiliser la syntaxe suivante :

(function() {

// Code de notre function.

})();

Ces fonctions immédiatement exécutées se nomment des Immediately-Invoked Function Expression, abrégées en IIFE. L’intérêt de ce fonctionnement réside dans la gestion de la portée des variables : nous pouvons créer autant de variables que nous le souhaitons dans cette fonction avec les noms que nous souhaitons, tout sera détruit une fois que notre fonction aura fini de s'exécuter. Par exemple :

var test = 'noir'; // On crée une variable « test » contenant le mot « noir ».

(function() { // Début de la zone isolée.

var test = 'blanc'; // On crée une variable du même nom avec le contenu « blanc » dans la zone isolée.

alert('Dans la zone isolée, la couleur est : ' + test);

})(); // Fin de la zone isolée. Les variables créées dans cette zone sont détruites.

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

Il est également possible d’enregistrer dans le code global une des valeurs générées dans une zone isolée :

var sayHello = (function() {

return 'Yop !';

})();

alert(sayHello); // Affiche : « Yop ! »

## Les objets et les tableaux

Les objets contiennent trois choses distinctes :

* un constructeur.
* des propriétés.
* des méthodes.

**Le constructeur**

Le constructeur est un code qui est exécuté quand on utilise un nouvel objet. Il permet d’effectuer des actions comme définir diverses variables au sein même de l'objet (comme le nombre de caractères d'une chaîne de caractères).

**Les propriétés**

Toute valeur va être placée dans une variable au sein de l'objet : c'est ce que l'on appelle une **propriété**. Une propriété est une variable contenue dans l'objet, elle contient des informations nécessaires au fonctionnement de l'objet.

**Les méthodes**

Il est possible de modifier l'objet. Cela se fait par l'intermédiaire des **méthodes**. Les méthodes sont des fonctions contenues dans l'objet, et qui permettent de réaliser des opérations sur le contenu de l'objet.

Par exemple :

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

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

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

**Objets natifs**

Nous en avons déjà rencontré trois :

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

**Les tableaux**

Un tableau, ou array, est une variable qui contient plusieurs valeurs, appelées **items**. Chaque item est accessible au moyen d'un **indice** (index) et dont la numérotation commence à partir de 0.

**Déclarer un tableau**

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

var myArray\_b = [42, 'Sébastien', 12, 'Laurence'];

Il existe également une syntaxe plus longue qui est vouée à disparaître :

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

Il est important de noter que cette syntaxe est dépréciée.

**Récupérer et modifier des valeurs**

myArray[1] = 'Clarisse';

alert(myArray[1]);

**Opérations sur les tableaux**

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

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

La méthode unshift() fonctionne comme push(), excepté que les items sont ajoutés au début du tableau. Les méthodes shift() et pop() retirent respectivement le premier et le dernier élément du tableau :

myArray.shift();

myArray.pop();

**Chaînes de caractères et tableaux**

Les chaînes de caractères possèdent une méthode split() qui permet de les découper en un tableau, en fonction d'un séparateur :

var cousinsString = 'Pauline Guillaume Clarisse',

cousinsArray = cousinsString.split(' ');

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

var cousinsString\_2 = cousinsArray.join('-');

Si nous ne spécifions rien comme séparateur, les chaînes de caractères seront collées les unes aux autres.

**Parcourir un tableau**

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

alert(myArray[i]);

}

Il est à noter que le bloc condition de la boucle for est exécuté à chaque tour de boucle. Afin d’éviter d’avoir recourt à la méthode length() à chaque passage, nous pouvons optimiser notre code de la manière suivante :

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

alert(myArray[i]);

}

**Les objets littéraux**

S'il est possible d'accéder aux items d'un tableau via leur indice, il peut être pratique d'y accéder au moyen d'un identifiant. Pour ce faire, nous allons créer nous-mêmes un tableau sous la forme d'un objet littéral :

var family = {

self: 'Sébastien',

sister: 'Laurence',

brother: 'Ludovic',

cousin\_1: 'Pauline',

cousin\_2: 'Guillaume'

};

**La syntaxe d'un objet**

Nous avons vu que pour créer un array vide il suffisait d'écrire : var myArray = [];.Pour les objets c'est à peu près similaire : var myObject = {};.

**Accès aux items**

Pour récupérer le contenu d’un champ de notre objet, par exemple le champ sister, il suffira d’écrire : family.sister;.

Il existe une autre manière de faire, semblable à celle qui permet d'accéder aux items d'un tableau en connaissant l'indice : family['sister'];.

**Ajouter des items**

Il est possible d'ajouter un item en spécifiant un identifiant qui n'est pas encore présent. Par exemple : family['uncle'] = 'Didier'; .

Ou bien sous cette forme : family.uncle = 'Didier';.

**Parcourir un objet avec for in**

Il n'est pas possible de parcourir un objet littéral avec une boucle for puisqu'une boucle for est surtout capable d'incrémenter une variable numérique, ce qui n’est d'aucune utilité dans le cas d'un objet littéral puisque nous devons posséder un identifiant.

La boucle for in ne sert qu'à une seule chose : parcourir un objet. Le fonctionnement est quasiment le même que pour un tableau, excepté qu'il suffit de fournir une « variable clé » qui reçoit un identifiant (au lieu d'un index) et de spécifier l'objet à parcourir :

for (var id in family) { // On stocke l'identifiant dans « id » pour parcourir l'objet « family »

alert(family[id]);

}

**Utilisation des objets littéraux**

Les objets littéraux peuvent être utiles dans les fonctions. Les return ne savent retourner qu'une seule variable. Si on veut retourner plusieurs variables, il faut les placer dans un tableau et retourner ce dernier. Mais il est plus commode d'utiliser un objet littéral. Par exemple :

function getCoords() {

return {

x: 12,

y: 21

};

}

var coords = getCoords();

alert(coords.x); // 12

alert(coords.y); // 21

## Débuguer son code

La méthode console.log()permet d'afficher la valeur d'une variable ou d’un objet dans la console du mode debug du navigateur sans bloquer l'exécution de notre code. Par exemple :

// On crée un objet basique.

var helloObject = {

english: 'Hello',

french: 'Bonjour',

spanish: 'Hola'

};

// Et on l'affiche.

console.log(helloObject);

Nous pouvons également émettre des alertes ou des erreurs avec les méthodes warn() et error().

La fonction native isNaN() (is Not a Number) permet de tester si une variable est un nombre ou non, et renvoie true ou false.

## Manipuler le code HTML

Le **Document Object Model** (abrégé **DOM**) est une interface de programmation pour les documents XML et HTML. Une interface de programmation, qu'on appelle aussi une **API** (pour **Application Programming Interface**), est un ensemble d'outils qui permettent de faire communiquer entre eux plusieurs programmes ou, dans le cas présent, différents langages.

Le DOM est donc une API qui s'utilise avec les documents XML et HTML, et qui va nous permettre, via le JavaScript, d'accéder au code XML et/ou HTML d'un document.

**L'objet window**

L'objet window est ce qu'on appelle un objet global qui représente la fenêtre du navigateur. C'est à partir de cet objet que le JavaScript est exécuté. L'objet window est dit implicite, c'est-à-dire qu'il n'y a généralement pas besoin de le spécifier. Par exemple, alert() n'est pas vraiment une fonction mais une méthode appartenant à l'objet window. Ainsi, ces deux instructions produisent le même effet :

window.alert('Hello world!');

alert('Hello world!');

De même, lorsque nous déclarons une variable dans le contexte global de notre script, celle-ci deviendra en réalité une propriété de l'objet window.

Une dernière chose importante: toute variable non déclarée (donc utilisée sans jamais écrire le mot-clé var) deviendra immédiatement une propriété de l'objet window, et ce quel que soit l'endroit où elle est utilisée. Par exemple :

(function() { // On utilise une IIFE pour « isoler » du code

text = 'Variable accessible !'; // Cette variable n'a jamais été déclarée et pourtant on lui attribue une valeur

})();

alert(text); // Affiche : « Variable accessible ! »

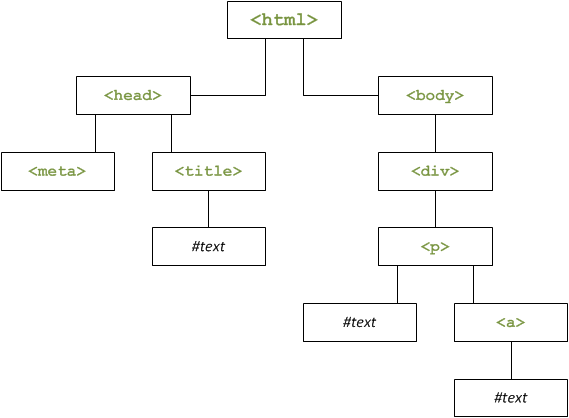
Notre variable a été utilisée pour la première fois dans une IIFE et pourtant nous y avons accès depuis l'espace global. Tout simplement parce que le JavaScript va chercher à résoudre le problème que nous lui avons donné : on lui demande d'attribuer une valeur à la variable text, il va donc chercher cette variable mais ne la trouve pas, la seule solution pour résoudre le problème qui lui est donné est alors d'utiliser l'objet window. Ce qui veut dire qu'en écrivant : text = 'Variable accessible !'; le code sera alors interprété de cette manière si aucune variable accessible n'existe avec ce nom : window.text = 'Variable accessible !';. Cependant cette pratique est déconseillée. Si nous souhaitons déclarer une variable dans l'espace global alors que nous nous trouvons actuellement dans un autre espace (une IIFE, par exemple), il vaut mieux spécifier explicitement l'objet window.

**Le document**

L'objet document est un sous-objet de window, l'un des plus utilisés. Et pour cause, il représente la page Web et plus précisément la balise <html>. C'est grâce à cet élément-là que nous allons pouvoir accéder aux éléments HTML et les modifier.

**Naviguer dans le document**

Le DOM pose comme concept que la page Web est vue comme un arbre, comme une hiérarchie d'éléments. On peut donc schématiser une page Web simple comme ceci :



Le texte présent dans une page Web est vu par le DOM comme un nœud de type #text.

**Accéder aux éléments**

L'accès aux éléments HTML via le DOM est assez simple mais demeure actuellement plutôt limité. L'objet document possède trois méthodes principales : getElementById(), getElementsByTagName() et getElementsByName().

**getElementById()**

Cette méthode permet d'accéder à un élément en connaissant son ID :

<div id="myDiv">

<p>Un peu de texte <a>et un lien</a></p>

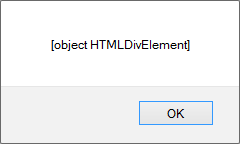
</div>

<script>

var div = document.getElementById('myDiv');

alert(div);

</script>



Il nous dit que div est un objet de type HTMLDivElement. En clair, c'est un élément HTML qui se trouve être un <div>, ce qui nous montre que le script fonctionne correctement.

**getElementsByTagName()**

Cette méthode permet de récupérer, sous la forme d'un tableau, tous les éléments de la famille. Si, dans une page, on veut récupérer tous les <div>, il suffit de faire comme ceci :

var divs = document.getElementsByTagName('div');

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

alert('Element n° ' + (i + 1) + ' : ' + divs[i]);

}

La méthode retourne une collection d'éléments (utilisable de la même manière qu'un tableau). Pour accéder à chaque élément, il est nécessaire de parcourir le tableau avec une boucle.

* Cette méthode est accessible sur n'importe quel élément HTML et pas seulement sur l'objet document.
* En paramètre de cette méthode nous pouvons mettre une chaîne de caractères contenant un astérisque \* qui récupérera tous les éléments HTML contenus dans l'élément ciblé.

**getElementsByName()**

Cette méthode est semblable à getElementsByTagName() et permet de ne récupérer que les éléments qui possèdent un attribut name spécifié. L'attribut name n'est utilisé qu'au sein des formulaires, et est déprécié depuis la spécification HTML5 dans tout autre élément que celui d'un formulaire. Cette méthode est dépréciée en XHTML mais est standardisée en HTML5.

**Accéder aux éléments grâce aux technologies récentes**

querySelector() et querySelectorAll() ont pour particularité de grandement simplifier la sélection d'éléments dans l'arbre DOM grâce à leur mode de fonctionnement. Ces deux méthodes prennent pour paramètre un seul argument : une chaîne de caractères. Cette chaîne de caractères doit être un sélecteur CSS. Par exemple : #menu .item span. Ce sélecteur CSS stipule que l'on souhaite sélectionner les balises de type <span> contenues dans les classes .item elles-mêmes contenues dans un élément dont l'identifiant est #menu.

querySelector() renvoie le premier élément trouvé correspondant au sélecteur CSS, tandis que querySelectorAll() va renvoyer tous les éléments correspondant au sélecteur CSS fourni sous forme de tableau.

**L'héritage des propriétés et des méthodes**

Le JavaScript voit les éléments HTML comme étant des objets, cela veut donc dire que chaque élément HTML possède des propriétés et des méthodes. Tous ne possèdent pas les mêmes propriétés et méthodes. Certaines sont néanmoins communes à tous les éléments HTML, car tous les éléments HTML sont d'un même type : le type Node.

**Notion d'héritage**

Nous avons vu qu'un élément <div> est un objet HTMLDivElement, mais un objet, en JavaScript, peut appartenir à différents groupes. Ainsi, notre <div> est un HTMLDivElement, qui est un sous-objet d'HTMLElement qui est lui-même un sous-objet d'Element. Element est enfin un sous-objet de Node :



L'objet Node apporte un certain nombre de propriétés et de méthodes qui pourront être utilisées depuis un de ses sous-objets. En clair, les sous-objets héritent des propriétés et méthodes de leurs objets parents.

**Éditer les éléments HTML**

Pour interagir avec les attributs, l'objet Element nous fournit deux méthodes, getAttribute() et setAttribute() permettant respectivement de récupérer et d'éditer un attribut. Le premier paramètre est le nom de l'attribut, et le deuxième, dans le cas de setAttribute() uniquement, est la nouvelle valeur à donner à l'attribut :

<body>

<a id="myLink" href="http://www.un\_lien\_quelconque.com">Un lien modifié dynamiquement</a>

<script>

var link = document.getElementById('myLink');

var href = link.getAttribute('href'); // On récupère l'attribut « href »

alert(href);

link.setAttribute('href', 'http://www.siteduzero.com'); // On édite l'attribut « href »

</script>

</body>

On commence par récupérer l'élément myLink, et on lit son attribut href via getAttribute(). Ensuite on modifie la valeur de l'attribut href avec setAttribute().

**Les attributs accessibles**

pour la plupart des éléments courants comme <a>, il est possible d'accéder à un attribut via une propriété. Ainsi, si on veut modifier la destination d'un lien, on peut utiliser la propriété href :

<body>

<a id="myLink" href="http://www.un\_lien\_quelconque.com">Un lien modifié dynamiquement</a>

<script>

var link = document.getElementById('myLink');

var href = link.href;

alert(href);

link.href = 'http://www.siteduzero.com';

</script>

</body>

Attention : un attribut auquel on accède par le biais de la méthode getAttribute() renverra la valeur exacte de ce qui est écrit dans le code HTML tandis que l'accès par le biais de sa propriété peut entraîner quelques changements. Par exemple pour <a href="/">Retour à l'accueil du site</a>, l'accès à l'attribut href  avec la méthode getAttribute() retournera bien un simple slash tandis que l'accès à la propriété retournera une URL absolue. Si notre nom de domaine est « mon\_site.com » nous obtiendrons alors « http://mon\_site.com/ ».

**La classe**

On peut penser que pour modifier l'attribut class d'un élément HTML, il suffit d'utiliser element.class. Ce n'est pas possible, car le mot-clé class est réservé en JavaScript, bien qu'il n'ait aucune utilité. À la place de class, il faudra utiliser className.

<!doctype html>

<html>

<head>

<meta charset="utf-8" />

<title>Le titre de la page</title>

<style>

.blue {

background: blue;

color: white;

}

</style>

</head>

<body>

<div id="myColoredDiv">

<p>Un peu de texte <a>et un lien</a></p>

</div>

<script>

document.getElementById('myColoredDiv').className = 'blue';

</script>

</body>

</html>

Dans cet exemple, on définit la classe CSS .blue à l'élément myColoredDiv, ce qui fait que cet élément sera affiché avec un arrière-plan bleu et un texte blanc.

Toujours dans le même cas, le nom for est réservé lui aussi en JavaScript (pour les boucles). Vous ne pouvez donc pas modifier l'attribut HTML for d'un <label> en écrivant element.for, il faudra utiliser element.htmlFor à la place.

Attention : si notre élément comporte plusieurs classes (exemple : <a class="external red u">) et que nous récupérons la classe avec className, cette propriété ne retournera pas un tableau avec les différentes classes, mais bien la chaîne « external red u », ce qui n'est pas vraiment le comportement souhaité. Il nous faudra alors couper cette chaîne avec la méthode split() (.split(' ')) pour obtenir un tableau.

La propriété classList permet de consulter les classes sous forme d'un tableau et de les manipuler aisément :

var div = document.querySelector('div');

// Ajoute une nouvelle classe

div.classList.add('new-class');

// Retire une classe

div.classList.remove('new-class');

// Retire une classe si elle est présente ou bien l'ajoute si elle est absente

div.classList.toggle('toggled-class');

// Indique si une classe est présente ou non

if (div.classList.contains('old-class')) {

alert('La classe .old-class est présente !');

}

// Parcourt et affiche les classes CSS

var result = '';

for (var i = 0; i < div.classList.length; i++) {

result += '.' + div.classList[i] + '\n';

}

alert(result);

**Le contenu : innerHTML**

innerHTML permet de récupérer le code HTML enfant d'un élément sous forme de texte. Ainsi, si des balises sont présentes, innerHTML les retournera sous forme de texte :

<body>

<div id="myDiv">

<p>Un peu de texte <a>et un lien</a></p>

</div>

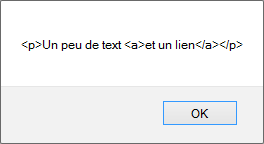
<script>

var div = document.getElementById('myDiv');

alert(div.innerHTML);

</script>

</body>



**Ajouter ou éditer du HTML**

Pour éditer ou ajouter du contenu HTML, il suffit de faire l'inverse, c'est-à-dire de définir un nouveau contenu : document.getElementById('myDiv').innerHTML = '<blockquote> Mets une citation à la place du paragraphe</blockquote>';

Pour ajouter du contenu, et ne pas modifier le contenu déjà en place, il suffit d’utiliser += à la place de l'opérateur d'affectation. Cependant, il ne faut pas utiliser le **+=** dans une boucle ! En effet, innerHTML ralentit considérablement l'exécution du code si l'on opère de cette manière, il vaut donc mieux concaténer son texte dans une variable pour ensuite ajouter le tout via innerHTML. De plus, ajouter une balise <script> à une page par le biais de la propriété innerHTML ne fonctionne pas !

**innerText**

La propriété innerText a aussi été introduite dans Internet Explorer et n'a jamais été standardisée et n'est pas supportée par tous les navigateurs. Internet Explorer (pour toute version antérieure à la neuvième) ne supporte que cette propriété et non pas la version standardisée que nous verrons par la suite. Le fonctionnement d'innerText est le même qu'innerHTML excepté le fait que seul le texte est récupéré, et non les balises.

**textContent**

textContent est la version standardisée d'innerText. Elle est reconnue par tous les navigateurs à l'exception des versions d'Internet Explorer antérieures à la 9.

**Tester le navigateur**

Comment faire un script qui fonctionne à la fois pour Internet Explorer et les autres navigateurs ? Il est possible via une simple condition de tester si le navigateur prend en charge telle ou telle méthode ou propriété :

<body>

<div id="myDiv">

<p>Un peu de texte <a>et un lien</a></p>

</div>

<script>

var div = document.getElementById('myDiv');

var txt = '';

if (div.textContent) { // « textContent » existe ? Alors on s'en sert !

txt = div.textContent;

} else if (div.innerText) { // « innerText » existe ? Alors on doit être sous IE.

txt = div.innerText + ' [via Internet Explorer]';

} else { // Si aucun des deux n'existe, cela est sûrement dû au fait qu'il n'y a pas de texte

txt = ''; // On met une chaîne de caractères vide

}

alert(txt);

</script>

</body>

Cela dit, ce code est quand même très long et redondant. Il est possible de le raccourcir de manière considérable : txt = div.textContent || div.innerText || '';.

## Manipuler le code HTML partie 2

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

Les méthodes getElementById() et getElementsByTagName() sont utilisables pour accéder aux éléments HTML. Une fois que l'on a atteint un élément, il est possible de se déplacer de façon un peu plus précise, avec toute une série de méthodes et de propriétés disponibles.

**La propriété parentNode**

La propriété parentNode permet d'accéder à l'élément parent d'un élément :

<blockquote>

<p id="myP">Ceci est un paragraphe !</p>

</blockquote>

Admettons qu'on doive accéder à myP, et que pour une autre raison on doive accéder à l'élément <blockquote>, qui est le parent de myP. Il suffit d'accéder à myP puis à son parent, avec parentNode :

var paragraph = document.getElementById('myP');

var blockquote = paragraph.parentNode;

**nodeType et nodeName**

nodeType et nodeName servent respectivement à vérifier le type d'un nœud et le nom d'un nœud. nodeType retourne un nombre, qui correspond à un type de nœud. Le tableau suivant liste les types possibles, ainsi que leurs numéros (les types courants sont mis en gras) :

|  |  |
| --- | --- |
| **Numéro** | **Type de nœud** |
| **1** | **Nœud élément** |
| **2** | **Nœud attribut** |
| **3** | **Nœud texte** |
| 4 | Nœud pour passage CDATA (relatif au XML) |
| 5 | Nœud pour référence d'entité |
| 6 | Nœud pour entité |
| 7 | Nœud pour instruction de traitement |
| **8** | **Nœud pour commentaire** |
| 9 | Nœud document |
| 10 | Nœud type de document |
| 11 | Nœud de fragment de document |
| 12 | Nœud pour notation |

nodeName, quant à lui, retourne simplement le nom de l'élément, en majuscule. Il est toutefois conseillé d'utiliser toLowerCase() (ou toUpperCase()) pour forcer un format de casse et ainsi éviter les mauvaises surprises.

var paragraph = document.getElementById('myP');

alert(paragraph.nodeType + '\n\n' + paragraph.nodeName.toLowerCase());

**Lister et parcourir des nœuds enfants**

firstChild et lastChild servent respectivement à accéder au premier et au dernier enfant d'un nœud.

<p id="myP">Un peu de texte, <a>un lien</a> et <strong>une portion en emphase</strong></p>

</div>

<script>

var paragraph = document.getElementById('myP');

var first = paragraph.firstChild;

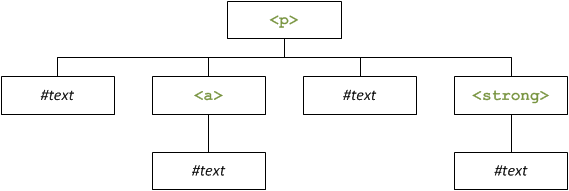
var last = paragraph.lastChild;

alert(first.nodeName.toLowerCase());

alert(last.nodeName.toLowerCase());

</script>

En schématisant l'élément myP précédent, on obtient ceci :



Le premier enfant de <p> est un nœud textuel, alors que le dernier enfant est un élément <strong>. Dans le cas où nous ne souhaiterions récupérer que les enfants qui sont considérés comme des éléments HTML (et donc éviter les nœuds **#text** par exemple), il existe les propriétés firstElementChild et lastElementChild. Ainsi, dans l'exemple précédent, la propriété firstElementChild renverrait l'élément <a>.

**nodeValue et data**

Il faut maintenant récupérer le texte du premier enfant, et le texte contenu dans l'élément <strong>. Il faut soit utiliser la propriété nodeValue soit la propriété data. Si on recode le script précédent, nous obtenons ceci :

var paragraph = document.getElementById('myP');

var first = paragraph.firstChild;

var last = paragraph.lastChild;

alert(first.nodeValue);

alert(last.firstChild.data);

first contient le premier nœud, un nœud textuel. Il suffit de lui appliquer la propriété nodeValue (ou data) pour récupérer son contenu ; pas de difficulté ici. En revanche, il y a une petite différence avec notre élément <strong> : vu que les propriétés nodeValue et data ne s'appliquent que sur des nœuds textuels, il nous faut d'abord accéder au nœud textuel que contient notre élément, c'est-à-dire son nœud enfant. Pour cela, on utilise firstChild (et non pas firstElementChild), et ensuite on récupère le contenu avec nodeValue ou data.

**childNodes**

La propriété childNodes retourne un tableau contenant la liste des enfants d'un élément. L'exemple suivant illustre le fonctionnement de cette propriété, de manière à récupérer le contenu des éléments enfants :

<body>

<div>

<p id="myP">Un peu de texte <a>et un lien</a></p>

</div>

<script>

var paragraph = document.getElementById('myP');

var children = paragraph.childNodes;

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

if (children[i].nodeType === Node.ELEMENT\_NODE) { // C'est un élément HTML

alert(children[i].firstChild.data);

} else { // C'est certainement un nœud textuel

alert(children[i].data);

}

}

</script>

</body>

Ici, nous n'avons pas comparé la propriété nodeType à la valeur 1 mais à Node.ELEMENT\_NODE, il s'agit en fait d'une constante qui contient la valeur 1, ce qui est plus facile à lire que le chiffre seul. Il existe une constante pour chaque type de nœud.

**nextSibling et previousSibling**

nextSibling et previousSibling sont deux propriétés qui permettent d'accéder respectivement au nœud suivant et au nœud précédent :

<div>

<p id="myP">Un peu de texte, <a>un lien</a> et <strong>une portion en emphase</strong></p>

</div>

<script>

var paragraph = document.getElementById('myP');

var first = paragraph.firstChild;

var next = first.nextSibling;

alert(next.firstChild.data); // Affiche « un lien »

</script>

Dans cet exemple, on récupère le premier enfant de myP, et sur ce premier enfant on utilise nextSibling, qui permet de récupérer l’élément <a>. Avec ça, il est même possible de parcourir les enfants d'un élément, en utilisant une boucle while :

<div>

<p id="myP">Un peu de texte <a>et un lien</a></p>

</div>

<script>

var paragraph = document.getElementById('myP');

var child = paragraph.lastChild; // On prend le dernier enfant

while (child) {

if (child.nodeType === Node.ELEMENT\_NODE) { // C'est un élément HTML

alert(child.firstChild.data);

} else { // C'est certainement un nœud textuel

alert(child.data);

}

child = child.previousSibling; // À chaque tour de boucle, on prend l'enfant précédent

}

</script>

Tout comme pour firstChild et lastChild, sachez qu'il existe les propriétés nextElementSibling et previousElementSibling qui permettent, elles aussi, de ne récupérer que les éléments HTML.

**Attention aux nœuds vides**

En considérant le code HTML suivant, on peut penser que l'élément <div> ne contient que trois enfants <p> :

<div>

<p>Paragraphe 1</p>

<p>Paragraphe 2</p>

<p>Paragraphe 3</p>

</div>

Mais attention, car ce code est radicalement différent de celui-ci :

<div><p>Paragraphe 1</p><p>Paragraphe 2</p><p>Paragraphe 3</p></div>

En fait, les espaces entre les éléments tout comme les retours à la ligne sont considérés comme des nœuds textuels (enfin, cela dépend des navigateurs) ! Ainsi donc, si l'on schématise le premier code, on obtient ceci :



Alors que le deuxième code peut être schématisé comme ça :



Il existe une solution à ce problème : Les attributs firstElementChild, lastElementChild, nextElementSibling et previousElementSibling ne retournent que des éléments HTML et permettent donc d'ignorer les nœuds textuels. Ils s'utilisent exactement de la même manière que les attributs de base (firstChild, lastChild, etc.).

**Créer et insérer des éléments**

Avec le DOM, l'ajout d'un élément HTML se fait en trois temps :

* On crée l'élément.
* On lui affecte des attributs.
* On l'insère dans le document, et ce n'est qu'à ce moment-là qu'il sera « ajouté ».

**Création de l'élément**

La création d'un élément se fait avec la méthode createElement() d’un sous-objet de l'objet racine, c'est-à-dire document dans la majorité des cas : var newLink = document.createElement('a');.

On crée ici un nouvel élément <a>. Cet élément est créé, mais n'est pas inséré dans le document, il n'est donc pas visible.

**Affectation des attributs**

Ici, on définit les attributs soit avec setAttribute() soit directement avec les propriétés adéquates :

newLink.id = 'sdz\_link';

newLink.href = 'http://www.siteduzero.com';

newLink.title = 'Découvrez le Site du Zéro !';

newLink.setAttribute('tabindex', '10');

**Insertion de l'élément**

On utilise la méthode appendChild() pour insérer l'élément, ce qui signifie qu'il nous faut connaître l'élément auquel on va ajouter l'élément créé. Pour ajouter notre élément <a> dans l'élément <p> portant l'ID myP il suffit de récupérer cet élément et d'ajouter notre élément <a> via appendChild() : document.getElementById('myP').appendChild(newLink);.

appendChild() insérera toujours l'élément en tant que dernier enfant.

**Ajouter des nœuds textuels**

L'élément a été inséré mais il manque le contenu textuel. La méthode createTextNode() sert à créer un nœud textuel :

var newLinkText = document.createTextNode("Le Site du Zéro");

newLink.appendChild(newLinkText);

Enfin, appendChild() retourne une référence pointant sur l'objet qui vient d'être inséré, référence qui peut être récupérée :

var span = document.body.appendChild(document.createElement('span'));

span.innerHTML = 'Du texte en plus !';//Pointe vers le span créé par document.createElement('span')

**Notions sur les références**

En JavaScript et comme dans beaucoup de langages, le contenu des variables est « passé par valeur ». Cela veut donc dire que si une variable nick1 contient le prénom « Clarisse » et qu'on affecte cette valeur à une autre variable, la valeur est copiée dans la nouvelle. On obtient alors deux variables distinctes, contenant la même valeur. Si on modifie la valeur de nick2, la valeur de nick1 reste inchangée : normal, les deux variables sont bien distinctes.

**Les références**

Outre le « passage par valeur », le JavaScript possède un « passage par référence ». En fait, quand une variable est créée, sa valeur est mise en mémoire par l'ordinateur. Pour pouvoir retrouver cette valeur, elle est associée à une adresse que seul l'ordinateur connaît et manipule.

Quand on passe une valeur par référence, on transmet l'adresse de la valeur, ce qui va permettre d'avoir deux variables qui pointent sur une même valeur. Quand on manipule une page Web avec le DOM, on est confronté à des références.

**Les références avec le DOM**

Schématiser le concept de référence avec le DOM est assez simple : deux variables peuvent accéder au même élément :

var newLink = document.createElement('a');

var newLinkText = document.createTextNode('Le Site du Zéro');

newLink.id = 'sdz\_link';

newLink.href = 'http://www.siteduzero.com';

newLink.appendChild(newLinkText);

document.getElementById('myP').appendChild(newLink);

// On récupère, via son ID, l'élément fraîchement inséré

var sdzLink = document.getElementById('sdz\_link');

sdzLink.href = 'http://www.siteduzero.com/forum.html';

// newLink.href affiche bien la nouvelle URL :

alert(newLink.href);

La variable newLink contient en réalité une référence vers l'élément <a> qui a été créé. newLink ne contient pas l'élément, il contient une adresse qui pointe vers ce fameux <a>. Une fois que l'élément HTML est inséré dans la page, on peut y accéder de nombreuses autres façons, comme avec getElementById(). Quand on accède à un élément via getElementById(), on le fait aussi au moyen d'une référence.

Ce qu'il faut retenir, c'est que les objets du DOM sont toujours accessibles par référence, et c'est la raison pour laquelle ce code ne fonctionne pas :

var newDiv1 = document.createElement('div');

var newDiv2 = newDiv1; // On tente de copier le <div>

En effet, newDiv2 contient une référence qui pointe vers le même <div> que newDiv1. Pour dupliquer un élément, il faut le cloner.

**Cloner, remplacer, supprimer…**

Pour cloner un élément, il suffit d’utiliser : cloneNode(). Cette méthode requiert un paramètre booléen ( true ou false) : si nous désirons cloner le nœud avec (true) ou sans (false) ses enfants et ses différents attributs :

var hr1 = document.createElement('hr');

var hr2 = hr1.cloneNode(false);

Remarque : L'élément est cloné, mais pas « inséré » tant que l'on n'a pas appelé appendChild(). De plus, la méthode cloneNode() ne copie pas les événements associés et créés avec le DOM (avec addEventListener()).

**Remplacer un élément par un autre**

Pour remplacer un élément par un autre, il suffit d’utiliser replaceChild(). Cette méthode accepte deux paramètres : le premier est le nouvel élément, et le deuxième est l'élément à remplacer. Tout comme cloneNode(), cette méthode s'utilise sur tous les types de nœuds (éléments, nœuds textuels, etc.) :

var link = document.querySelector('a');

var newLabel = document.createTextNode('et un hyperlien');

**Supprimer un élément**

Pour insérer un élément, on utilise appendChild(), et pour en supprimer un, on utilise removeChild(). Cette méthode prend en paramètre le nœud enfant à retirer :

var link = document.querySelector('a');

link.parentNode.removeChild(link);

À noter que la méthode removeChild() retourne l'élément supprimé, ce qui veut dire qu'il est parfaitement possible de supprimer un élément HTML pour ensuite le réintégrer où on le souhaite dans le DOM.

**Vérifier la présence d'éléments enfants**

Pour vérifier la présence d'éléments enfants il faut utiliser hasChildNodes(). Si l’élément étudié possède au moins un enfant, la méthode renverra true :

var paragraph = document.querySelector('p');

alert(paragraph.hasChildNodes());

**Insérer à la bonne place : insertBefore()**

La méthode insertBefore() permet d'insérer un élément avant un autre. Elle reçoit deux paramètres : le premier est l'élément à insérer, tandis que le deuxième est l'élément avant lequel l'élément va être inséré :

<p id="myP">Un peu de texte <a>et un lien</a></p>

<script>

var paragraph = document.querySelector('p');

var emphasis = document.createElement('em'),

emphasisText = document.createTextNode(' en emphase légère ');

emphasis.appendChild(emphasisText);

paragraph.insertBefore(emphasis, paragraph.lastChild);

</script>

**Une bonne astuce : insertAfter()**

Le JavaScript met à disposition insertBefore(), mais pas insertAfter().Pour insérer après un élément, on va d'abord récupérer l'élément parent. C'est logique, puisque l'insertion de l'élément va se faire soit via appendChild(), soit via insertBefore() : si on veut ajouter notre élément après le dernier enfant, c'est simple, il suffit d'appliquer appendChild(). Par contre, si l'élément après lequel on veut insérer notre élément n'est pas le dernier, on va utiliser insertBefore() en ciblant l'enfant suivant, avec nextSibling :

function insertAfter(newElement, afterElement) {

var parent = afterElement.parentNode;

if (parent.lastChild === afterElement) { // Si le dernier élément est le même que l'élément après lequel on veut insérer, il suffit de faire appendChild()

parent.appendChild(newElement);

} else { // Dans le cas contraire, on fait un insertBefore() sur l'élément suivant

parent.insertBefore(newElement, afterElement.nextSibling);

}

}

## Les événements

Voici la liste des événements principaux, ainsi que les actions à effectuer pour qu'ils se déclenchent :

|  |  |
| --- | --- |
| **Nom de l'événement** | **Action pour le déclencher** |
| click | Cliquer (appuyer puis relâcher) sur l'élément |
| dblclick | Double-cliquer sur l'élément |
| mouseover | Faire entrer le curseur sur l'élément |
| mouseout | Faire sortir le curseur de l'élément |
| mousedown | Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément |
| mouseup | Relâcher le bouton gauche de la souris sur l'élément |
| mousemove | Faire déplacer le curseur sur l'élément |
| keydown | Appuyer (sans relâcher) sur une touche de clavier sur l'élément |
| keyup | Relâcher une touche de clavier sur l'élément |
| keypress | Frapper (appuyer puis relâcher) une touche de clavier sur l'élément |
| focus | « Cibler » l'élément |
| blur | Annuler le « ciblage » de l'élément |
| change | Changer la valeur d'un élément spécifique aux formulaires (input, checkbox, etc.) |
| ﻿input | Taper un caractère dans un champ de texte |
| select | Sélectionner le contenu d'un champ de texte (input, textarea, etc.) |

Il existe aussi deux événements spécifiques à l'élément <form> :

|  |  |
| --- | --- |
| **Nom de l'événement** | **Action pour le déclencher** |
| submit | Envoyer le formulaire |
| reset | Réinitialiser le formulaire |

**Utiliser les événements**

Nous allons donc voir comment les utiliser sans le DOM :

<span onclick="alert('Hello !');">Cliquez-moi !</span>

Il suffit de cliquer sur le texte pour que la boîte de dialogue s'affiche. Afin d'obtenir ce résultat nous avons ajouté à notre <span> un attribut contenant les deux lettres « on » et le nom de notre événement « click ». Nous obtenons donc « onclick ».

Cet attribut possède une valeur qui est un code JavaScript. Nous pouvons y écrire quasiment tout ce que nous souhaitons, mais tout doit tenir entre les guillemets de l'attribut.

**Le mot-clé this**

Il s'agit d'une propriété pointant sur l'objet actuellement en cours d'utilisation. Donc, si nous faisons appel à ce mot-clé lorsqu'un événement est déclenché, l'objet pointé sera l'élément qui a déclenché l'événement :

<span onclick="alert('Voici le contenu de l\'élément que vous avez cliqué :\n\n' + this.innerHTML);">Cliquez-moi !</span>

**Le focus**

Le focus définit ce qui peut être appelé le « ciblage » d'un élément. Lorsqu'un élément est ciblé, il va recevoir tous les événements du clavier. Un exemple simple est d'utiliser un <input> de type text : si nous cliquons dessus alors l'input possède le focus.

Le focus peut s'appliquer à de nombreux éléments, ainsi, si nous tapons sur la touche Tabulation de notre clavier alors que nous sommes sur une page Web, nous allons avoir un élément de ciblé ou de sélectionné qui recevra alors tout ce que vous tapez sur le clavier.

**Bloquer l'action par défaut de certains événements**

Cela peut par exemple permettre de bloquer l’action de redirection d’un lien. Pour bloquer cela, il suffit juste d'ajouter le code return false; dans notre événement click :

<a href="http://www.siteduzero.com" onclick="alert('Vous avez cliqué !'); return false;">Cliquez-moi !</a>

L'utilisation de return true; permet de faire fonctionner l'événement comme si de rien n'était. En clair, comme si on n'utilisait pas de return false. Cela peut avoir son utilité si nous utilisons, par exemple, la fonction confirm() dans notre événement.

**L'utilisation de « javascript: » dans les liens : une technique prohibée**

Dans certains cas, nous allons devoir créer des liens juste pour leur attribuer un événement click et non pas pour leur fournir un lien vers lequel rediriger. Dans ce genre de cas, il est courant de voir ce type de code : <a href="javascript: alert('Vous avez cliqué !');">Cliquez-moi !</a>. Il s'agit d'une vieille méthode qui permet d'insérer du JavaScript directement dans l'attribut href de notre lien juste en ajoutant javascript: au début de l'attribut. Cette technique est maintenant obsolète. Il est conseillé d’utiliser une méthode alternative : <a href="#" onclick="alert('Vous avez cliqué !'); return false;">Cliquez-moi !</a>. On utilise return false; car le dièse redirige tout en haut de la page Web.

**Les événements au travers du DOM-0**

Cette interface est vieille mais reste très pratique pour créer des événements et peut parfois être préférée au DOM-2. Commençons par créer un simple code avec un événement click :

<span id="clickme">Cliquez-moi !</span>

<script>

var element = document.getElementById('clickme');

element.onclick = function() {

alert("Vous m'avez cliqué !");

};

</script>

* On récupère tout d'abord l'élément HTML dont l'ID est clickme.
* On accède ensuite à la propriété onclick à laquelle on assigne une fonction anonyme.
* Dans cette même fonction, on fait un appel à la fonction alert() avec un texte en paramètre.

On définit maintenant les événements non plus dans le code HTML mais directement en JavaScript. Chaque événement standard possède donc une propriété dont le nom est, à nouveau, précédé par les lettres « on ». Cette propriété ne prend plus pour valeur un code JavaScript brut, mais soit le nom d'une fonction, soit une fonction anonyme.

Concernant la suppression d'un événement avec le DOM-0, il suffit tout simplement de lui attribuer une fonction anonyme vide : element.onclick = function() {};.

**Le DOM-2**

En ce qui concerne le DOM-0, il a deux problèmes majeurs : il est vieux, et il ne permet pas de créer plusieurs fois le même événement. Le DOM-2, lui, permet la création multiple d'un même événement et gère aussi l'objet Event. Voici un exemple avec l'événement click :

<span id="clickme">Cliquez-moi !</span>

<script>

var element = document.getElementById('clickme');

element.addEventListener('click', function() {

alert("Vous m'avez cliqué !");

});

</script>

Concrètement, voici qui change par rapport au DOM-0 : tout d'abord nous n'utilisons plus une propriété mais une méthode nommée addEventListener(), et qui prend trois paramètres (bien que nous n'en ayons spécifié que deux) :

* Le nom de l'événement (sans les lettres « on »).
* La fonction à exécuter.
* Un booléen **optionnel** pour spécifier si l'on souhaite utiliser la phase de capture ou bien celle de bouillonnement. Nous expliquerons ce concept un peu plus tard.

Le DOM-2 permet la création multiple d'événements identiques pour un même élément, ainsi nous pouvons faire :

// Premier événement click

element.addEventListener('click', function() {

alert("Et de un !");

});

// Deuxième événement click

element.addEventListener('click', function() {

alert("Et de deux !");

});

Venons-en maintenant à la suppression des événements. Celle-ci s'opère avec la méthode removeEventListener() et se fait de manière très simple :

element.addEventListener('click', myFunction); // On crée l'événement

element.removeEventListener('click', myFunction); // On supprime l'événement en lui repassant les mêmes paramètres

Toute suppression d'événement avec le DOM-2 se fait avec les mêmes paramètres que ceux utilisés lors de sa création. Cependant, cela ne fonctionne pas aussi facilement avec les fonctions anonymes. Tout événement DOM-2 créé avec une fonction anonyme est particulièrement complexe à supprimer, car il faut posséder une référence vers la fonction concernée, ce qui n'est généralement pas le cas avec une fonction anonyme.

**Les phases de capture et de bouillonnement**

Ces deux phases sont deux étapes distinctes de l'exécution d'un événement. La première, la **capture**, s'exécute avant le déclenchement de l'événement, tandis que la deuxième, le **bouillonnement** (bubbling), s'exécute après que l'événement a été déclenché. Toutes deux permettent de définir le sens de propagation des événements.

Mais qu'est-ce que la propagation d'un événement ? Pour expliquer cela, prenons un exemple avec ces deux éléments HTML :

<div>

<span>Du texte !</span>

</div>

Si nous attribuons une fonction à l'événement click de chacun de ces deux éléments et que l'on clique sur le texte, quel événement va se déclencher en premier ?

La réponse se trouve dans les phases de capture et de bouillonnement. Si nous décidons d'utiliser la capture, alors l'événement du <div> se déclenchera en premier puis viendra ensuite l'événement du <span>. En revanche, si nous utilisons le bouillonnement, ce sera d'abord l'événement du <span> qui se déclenchera, puis viendra par la suite celui du <div>.

La phase de bouillonnement est celle définie par défaut.

**L'objet Event**

Cet objet sert à fournir une multitude d'informations sur l'événement actuellement déclenché. Par exemple quelles sont les touches actuellement enfoncées, les coordonnées du curseur, l'élément qui a déclenché l'événement…

Cet objet est particulier dans le sens où il n'est accessible que lorsqu'un événement est déclenché. Son accès ne peut se faire que dans une fonction exécutée par un événement. Cela se fait de la manière suivante avec le DOM-0 :

element.onclick = function(e) { // L'argument « e » va récupérer une référence vers l'objet « Event »

alert(e.type); // Ceci affiche le type de l'événement (click, mouseover, etc.)

};

Et de cette façon-là avec le DOM-2 :

element.addEventListener('click', function(e) { // L'argument « e » va récupérer une référence vers l'objet « Event »

alert(e.type); // Ceci affiche le type de l'événement (click, mouseover, etc.)

});

**Les fonctionnalités de l'objet Event**

Nous connaissons déjà la propriété type qui permet de savoir quel type d'événement s'est déclenché.

**Récupérer l'élément de l'événement actuellement déclenché**

Une des plus importantes propriétés de notre objet se nomme target. Celle-ci permet de récupérer une référence vers l'élément dont l'événement a été déclenché (exactement comme le this pour les événements sans le DOM ou avec DOM-0). Ainsi, nous pouvons très bien modifier le contenu d'un élément qui a été cliqué :

<span id="clickme">Cliquez-moi !</span>

<script>

var clickme = document.getElementById('clickme');

clickme.addEventListener('click', function(e) {

e.target.innerHTML = 'Vous avez cliqué !';

});

</script>

**Récupérer l'élément à l'origine du déclenchement de l'événement**

Ceci n’est pas la même chose que ce que nous venons juste de voir. Pour expliquer cela de façon simple, certains événements appliqués à un élément parent peuvent se propager d'eux-mêmes aux éléments enfants ; c'est le cas des événements mouseover, mouseout, mousemove, click… ainsi que d'autres événements moins utilisés. Voici un exemple :

<div id="parent1">

Parent

<div id="child1">Enfant N°1</div>

<div id="child2">Enfant N°2</div>

</div>

<script>

var parent1 = document.getElementById('parent1'),

result = document.getElementById('result');

parent1.addEventListener('mouseover', function(e) {

result.innerHTML = "L'élément déclencheur de l'événement \"mouseover\" possède l'ID : " + e.target.id;

});

</script>

Ici, le mouseover pourra avoir lieu sur le parent, mais aussi sur child1 et child2. Ainsi, result pourra contenir « parent1 », « child1 » ou « child2 » en fonction de ce que la souris survolera.

Avec cet exemple, target renverra toujours l'élément déclencheur de l'événement, or nous souhaitons obtenir l'élément sur lequel a été appliqué l'événement. Autrement dit, on veut connaître l'élément à l'origine de cet événement, et non pas ses enfants.

La solution est simple : il faut utiliser la propriété currentTarget au lieu de target. Après modification de cette seule ligne, l'ID affiché ne changera jamais et vaudra toujours « parent1 » :

result.innerHTML = "L'élément déclencheur de l'événement \"mouseover\" possède l'ID : " + e.currentTarget.id;

**Récupérer la position du curseur**

Généralement, on récupère la position du curseur par rapport au coin supérieur gauche de la page Web, cela dit il est aussi possible de récupérer sa position par rapport au coin supérieur gauche de l'écran. Dans ce tutoriel, nous allons nous limiter à la page Web.

Pour récupérer la position de notre curseur, il existe deux propriétés : clientX pour la position horizontale et clientY pour la position verticale. Étant donné que la position du curseur change à chaque déplacement de la souris, il est donc logique de dire que l'événement le plus adapté à la majorité des cas est mousemove :

<div id="position"></div>

<script>

var position = document.getElementById('position');

document.addEventListener('mousemove', function(e) {

position.innerHTML = 'Position X : ' + e.clientX + 'px<br />Position Y : ' + e.clientY + 'px';

});

</script>

**Récupérer l'élément en relation avec un événement de souris**

Cette fois nous allons étudier une propriété un peu plus « exotique » assez peu utilisée mais qui peut pourtant se révéler très utile ! Il s'agit de relatedTarget qui ne s'utilise qu'avec les événements mouseover et mouseout.

Cette propriété remplit deux fonctions différentes selon l'événement utilisé. Avec l'événement mouseout, elle vous fournira l'objet de l'élément sur lequel le curseur vient d'entrer. Avec l'événement mouseover, elle vous fournira l'objet de l'élément dont le curseur vient de sortir.

**Récupérer les touches frappées par l'utilisateur**

La récupération des touches frappées se fait par le biais de trois événements différents.

Les événements keyup et keydown sont conçus pour capter toutes les frappes de touches. Ainsi, il est parfaitement possible de détecter l'appui sur la touche A voire même sur la touche Ctrl. Avec ces deux événements, toutes les touches retournant un caractère retourneront un caractère majuscule, que la touche Maj soit pressée ou non.

L'événement keypress, lui, est d'une toute autre utilité : il sert uniquement à capter les touches qui écrivent un caractère (donc pas les touches Ctrl, Alt et autres qui n'affichent pas de caractère). Don avantage réside dans sa capacité à détecter les combinaisons de touches ! Ainsi, si nous faisons la combinaison Maj + A, l'événement keypress détectera bien un A majuscule là où les événements keyup et keydown se déclencheront deux fois, une fois pour la touche Maj et une deuxième fois pour la touche A.

Si nous devions énumérer toutes les propriétés capables de fournir une valeur permettant d’identifier la touche appuyée, il y en aurait trois : keyCode, charCode et which. Ces propriétés renvoient chacune un code ASCII correspondant à la touche pressée. Cependant, la propriété keyCode est amplement suffisante dans tous les cas.

Pour récupérer un caractère et non un code, il n'existe qu'une seule solution : la méthode fromCharCode(). Elle prend en paramètre une infinité d'arguments. Cependant, pour des raisons un peu particulières qui ne seront abordées que plus tard dans ce cours, que cette méthode s'utilise avec le préfixe String. : String.fromCharCode(/\* valeur \*/);. Cette méthode est donc conçue pour convertir les valeurs ASCII vers des caractères lisibles. Il faut donc faire attention à n'utiliser cette méthode qu'avec un événement keypress afin d'éviter d'afficher, par exemple, le caractère d'un code correspondant à la touche Ctrl (ce qui ne fonctionnerait pas).

**Bloquer l'action par défaut de certains événements**

Nous avons vu qu'il est possible de bloquer l'action par défaut de certains événements, comme la redirection d'un lien vers une page Web. Sans le DOM-2, cette opération était très simple vu qu'il suffisait d'écrire return false;. Avec l'objet Event, c'est quasiment tout aussi simple puisqu'il suffit juste d'appeler la méthode preventDefault(). Par exemple :

<a id="link" href="http://www.siteduzero.com">Cliquez-moi !</a>

<script>

var link = document.getElementById('link');

link.addEventListener('click', function(e) {

e.preventDefault(); // On bloque l'action par défaut de cet événement

alert('Vous avez cliqué !');

}); </script>

**Résoudre les problèmes d'héritage des événements**

Prenez donc les codes HTML et CSS suivants :

<div id="myDiv">

<div>Texte 1</div>

<div>Texte 2</div>

<div>Texte 3</div>

<div>Texte 4</div>

</div>

<div id="results"></div>

#myDiv, #results {

margin: 50px;

}

#myDiv {

padding: 10px;

width: 200px;

text-align: center;

background-color: #000;

}

#myDiv div {

margin: 10px;

background-color: #555;

}

Notre but objectif est de faire en sorte de détecter quand le curseur entre sur notre élément #myDiv et quand il en ressort. Nous pouvons donc penser qu’il suffirait d’écrire :

var myDiv = document.getElementById('myDiv'),

results = document.getElementById('results');

myDiv.addEventListener('mouseover', function() {

results.innerHTML += "Le curseur vient d'entrer.<br />";

});

myDiv.addEventListener('mouseout', function() {

results.innerHTML += "Le curseur vient de sortir.<br />";

});

Lorsque l’on passe le curseur sur toute la surface du <div>#myDiv, il y a quelques lignes en trop qui s'affichent dans nos résultats. Cela vient du fait que comme nous l’avons dit précédemment, les enfants héritent des propriétés de certains événements appliqués aux éléments parents. Ainsi, lorsque nous déplaçons notre curseur depuis le <div>#myDiv jusqu'à un <div> enfant, nous allons déclencher l'événement mouseout sur #myDiv et l'événement mouseover sur le <div> enfant.

**La solution**

Afin de pallier ce problème, il existe une solution assez tordue. La propriété relatedTarget abordée a pour but de détecter quel est l'élément vers lequel le curseur se dirige ou de quel élément il provient.

Ainsi, nous avons deux cas de figure :

* Dans le cas de l'événement mouseover, nous devons détecter la provenance du curseur. Si le curseur vient d'un enfant de #myDiv alors le code de l'événement ne devra pas être exécuté. S'il provient d'un élément extérieur à #myDiv alors l'exécution du code peut s'effectuer.
* Dans le cas de mouseout, le principe est similaire, si ce n'est que là nous devons détecter la destination du curseur. Dans le cas où la destination du curseur est un enfant de #myDiv alors le code de l'événement n'est pas exécuté, sinon il s'exécutera sans problème.

La solution consiste à remonter tout le long des éléments parents du div jusqu'à tomber soit sur myDiv, soit sur l'élément <body> qui désigne l'élément HTML le plus haut dans notre document. Il va donc nous falloir une boucle while. Il nous suffit alors d'insérer une condition qui exécutera le code de notre événement uniquement dans le cas où la variable relatedTarget ne pointe pas sur l'élément myDiv.

Cependant, il reste encore un cas de figure qui n'a pas été géré et qui peut être source de problèmes. La balise <body> ne couvre pas forcément la page Web complète de notre navigateur, ce qui fait que notre curseur peut provenir d'un élément situé encore plus haut que la balise <body>. Cet élément correspond à la balise <html> — soit l'élément document en JavaScript. Il nous faut donc faire une petite modification afin de bien préciser que si le curseur provient de document il ne peut forcément pas provenir de myDiv :

myDiv.addEventListener('mouseover', function(e) {

var relatedTarget = e.relatedTarget;

while (relatedTarget != myDiv && relatedTarget.nodeName != 'BODY' && relatedTarget != document) {

relatedTarget = relatedTarget.parentNode;

}

if (relatedTarget != myDiv) {

results.innerHTML += "Le curseur vient d'entrer.";

}

});

Il suffit alors de faire la même chose pour le deuxième événement en remplaçant mousover par mouseout.

## Les formulaires

Il est possible d'accéder à n'importe quelle propriété d'un élément HTML juste en tapant son nom. Il en va donc de même pour des propriétés spécifiques aux éléments d'un formulaire comme value, disabled, checked, etc.

**Une propriété classique : value**

Cette propriété permet de définir une valeur pour différents éléments d'un formulaire comme les <input>, les <button> etc. On lui assigne une valeur (une chaîne de caractères ou un nombre qui sera alors converti implicitement) et elle est immédiatement affichée sur notre élément HTML :

<input id="text" type="text" size="60" value="Vous n'avez pas le focus !" />

<script>

var text = document.getElementById('text');

text.addEventListener('focus', function(e) {

e.target.value = "Vous avez le focus !";

});

text.addEventListener('blur', function(e) {

e.target.value = "Vous n'avez pas le focus !";

});

</script>

Petite précision : cette propriété s'utilise aussi avec un élément <textarea>. En effet, en HTML, on prend souvent l'habitude de mettre du texte dans un <textarea> en écrivant : <textarea>Et voilà du texte !</textarea>. En JavaScript, on est donc souvent tenté d'utiliser innerHTML pour récupérer le contenu de notre <textarea>, cependant cela ne fonctionne pas : il faut bien utiliser value à la place !

**Les booléens avec disabled, checked et readonly**

Contrairement à la propriété value, les trois propriétés disabled, checked et readonly ne s'utilisent pas de la même manière qu'en HTML où il suffit d'écrire, par exemple, <input type="text" disabled="disabled" /> pour désactiver un champ de texte. En JavaScript, ces trois propriétés deviennent booléennes :

<input id="text" type="text" />

<script>

var text = document.getElementById('text');

text.disabled = true;

</script>

La propriété checked avec une checkbox fonctionne de la même manière qu'avec la propriété disabled. En revanche, les boutons de type radio, chaque bouton radio coché se verra attribuer la valeur true à sa propriété checked. Il va donc nous falloir utiliser une boucle for pour vérifier quel bouton radio a été sélectionné :

<label><input type="radio" name="check" value="1" /> Case n°1</label><br />

<label><input type="radio" name="check" value="2" /> Case n°2</label><br />

<input type="button" value="Afficher la case cochée" onclick="check();" />

<script>

function check() {

var inputs = document.getElementsByTagName('input'),

inputsLength = inputs.length;

for (var i = 0; i < inputsLength; i++) {

if (inputs[i].type === 'radio' && inputs[i].checked) {

alert('La case cochée est la n°' + inputs[i].value);

}

}

}

</script>

Il est possible de simplifier ce code grâce à la méthode querySelectorAll()  :

function check() {

var inputs = document.querySelectorAll('input[type=radio]:checked'),

inputsLength = inputs.length;

for (var i = 0; i < inputsLength; i++) {

alert('La case cochée est la n°' + inputs[i].value);

}

}

Toutes les vérifications concernant le type du champ et le fait qu'il soit coché ou non sont faites au niveau de querySelectorAll(). On peut ainsi supprimer l'ancienne condition.

**Les listes déroulantes avec selectedIndex et options**

Les listes déroulantes possèdent elles aussi leurs propres propriétés. Nous allons en retenir seulement deux parmi toutes celles qui existent : selectedIndex, qui nous donne l'index (l'identifiant) de la valeur sélectionnée, et options qui liste dans un tableau les éléments <option> de notre liste déroulante :

<select id="list">

<option>Sélectionnez votre sexe</option>

<option>Homme</option>

<option>Femme</option>

</select>

<script>

var list = document.getElementById('list');

list.addEventListener('change', function() {

// On affiche le contenu de l'élément <option> ciblé par la propriété selectedIndex

alert(list.options[list.selectedIndex].innerHTML);

});

</script>

Dans le cadre d'un <select> multiple, la propriété selectedIndex retourne l'index du premier élément sélectionné.

**Les méthodes et un retour sur quelques événements**

Les formulaires ne possèdent pas uniquement des propriétés, ils possèdent également des méthodes dont certaines sont bien pratiques.

**Les méthodes spécifiques à l'élément <form>**

Un formulaire, ou plus exactement l'élément <form>, possède deux méthodes intéressantes. La première, submit(), permet d'effectuer l'envoi d'un formulaire sans l'intervention de l'utilisateur. La deuxième, reset(), permet de réinitialiser tous les champs d'un formulaire. Ces deux méthodes ont le même rôle que les éléments <input> de type submit ou reset.

L'utilisation de ces deux méthodes est très simple, il suffit juste de les appeler sans aucun paramètre (elles n'en ont pas) :

var element = document.getElementById('un\_id\_de\_formulaire');

element.submit(); // Le formulaire est expédié

element.reset(); // Le formulaire est réinitialisé

Il existe également deux événements des mêmes noms : submit et reset. Il est important de préciser une chose : envoyer un formulaire avec la méthode submit() du JavaScript ne déclenchera jamais l'événement submit !

**La gestion du focus et de la sélection**

Il existe deux méthodes focus() et blur() permettant respectivement de donner et retirer le focus à un élément. Leur utilisation est très simple :

<input id="text" type="text" value="Entrez un texte" />

<br /><br />

<input type="button" value="Donner le focus" onclick="document.getElementById('text').focus();" /><br />

<input type="button" value="Retirer le focus" onclick="document.getElementById('text').blur();" />

Il existe aussi la méthode select() qui, en plus de donner le focus à l'élément, sélectionne le texte de celui-ci si cela est possible. Cette méthode ne fonctionne que sur des champs de texte comme un <input> de type text ou bien un <textarea>.

**Explications sur l'événement change**

il est bon de savoir que cet événement attend que l'élément auquel il est attaché perde le focus avant de se déclencher (s'il y a eu modification du contenu de l'élément). Donc, si nous souhaitons vérifier l'état d'un input à chacune de ses modifications sans attendre la perte de focus, il vous faudra plutôt utiliser d'autres événements du style keyup (et ses variantes) ou click, cela dépend du type d'élément vérifié.

Deuxièmement, cet événement est bien entendu utilisable sur n'importe quel input dont l'état peut changer, par exemple une checkbox ou un <input type="file" />.

## Manipuler le CSS

Comme nous venons de le voir, il y a deux manières de modifier le CSS d'un élément HTML, nous allons ici aborder la méthode la plus simple et la plus utilisée : l'utilisation de la propriété style. Cela se fait de la manière suivante :

element.style; // On accède à la propriété « style » de l'élément « element »

Une fois que l'on a accédé à notre propriété, comment modifier les styles CSS ? Eh bien tout simplement en écrivant leur nom et en leur attribuant une valeur, par exemple :

element.style.width = '150px'; // On modifie la largeur de notre élément à 150px

Comment accède-t-on à une propriété CSS qui possède un nom composé (par exemple background-color ) ? En JavaScript, les tirets sont interdits dans les noms des propriétés. La solution est simple : supprimer les tirets et chaque mot suivant normalement un tiret voit sa première lettre devenir une majuscule : element.style.backgroundColor = 'blue' ;.

L'édition du CSS d'un élément n'est pas bien compliquée. Cependant, il y a une limitation de taille : la lecture des propriétés CSS ! Par exemple :

<style type="text/css">

#myDiv {

background-color: orange;

}

</style>

<div id="myDiv">Je possède un fond orange.</div>

<script>

var myDiv = document.getElementById('myDiv');

alert('Selon le JavaScript, la couleur de fond de ce <div> est : ' + myDiv.style.backgroundColor); // On affiche la couleur de fond

</script>

En écrivant ce code, nous n’obtiendrons rien. En effet, notre code va lire uniquement les valeurs contenues dans la propriété style. C'est-à-dire, rien du tout dans notre exemple, car nous avons modifié les styles CSS depuis une feuille de style, et non pas depuis l'attribut style. En revanche, en modifiant le CSS avec l'attribut style, on retrouve sans problème la couleur de notre fond :

<div id="myDiv" style="background-color: orange">Je possède un fond orange.</div>

<script>

var myDiv = document.getElementById('myDiv');

alert('Selon le JavaScript, la couleur de fond de ce DIV est : ' + myDiv.style.backgroundColor); // On affiche la couleur de fond

</script>

**Récupérer les propriétés CSS venant d’une feuille de style**

La fonction getComputedStyle() va se charger de récupérer, à notre place, la valeur de n'importe quel style CSS. Qu'il soit déclaré dans la propriété style, une feuille de style ou bien même encore calculé automatiquement, cela importe peu : getComputedStyle() la récupérera sans problème. A noter que toutes les valeurs obtenues par le biais de getComputedStyle() ou currentStyle sont en lecture seule !

<style>

#text {

color: red;

}

</style>

<span id="text"></span>

<script>

var text = document.getElementById('text'),

color = getComputedStyle(text).color;

alert(color);

</script>

**Les propriétés de type offset**

Certaines valeurs de positionnement ou de taille des éléments ne pourront pas être obtenues de façon simple avec getComputedStyle(). Pour pallier ce problème il existe les propriétés offset qui sont, dans notre cas, au nombre de cinq :

|  |  |
| --- | --- |
| **Nom de l'attribut** | **Contient…** |
| offsetWidth | Contient la largeur complète (width + padding + border) de l'élément. |
| offsetHeight | Contient la hauteur complète (height + padding + border) de l'élément. |
| offsetLeft | Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord gauche de son élément parent. |
| offsetTop | Surtout utile pour les éléments en position absolue. Contient la position de l'élément par rapport au bord supérieur de son élément parent. |
| offsetParent | Utile que pour un élément en position absolue ou relative ! Contient l'objet de l'élément parent par rapport auquel est positionné l'élément actuel. |

Leur utilisation ne se fait pas de la même manière que n'importe quel style CSS, tout d'abord parce que ce ne sont pas des styles CSS ! Ce sont juste des propriétés (en lecture seule) mises à jour dynamiquement qui concernent certains états physiques d'un élément. Pour les utiliser, on les lit directement sur l'objet de notre élément HTML : alert(el.offsetHeight);.

Attention : les valeurs contenues dans ces propriétés (à part offsetParent) sont exprimées en pixels et sont donc de type Number, pas comme les styles CSS qui sont de type String et pour lesquelles les unités sont explicitement spécifiées (px, cm, em, etc.).

**La propriété offsetParent**

Lorsque nous décidons de mettre un de nos éléments HTML en positionnement absolu, celui-ci est sorti du positionnement par défaut des éléments HTML et va aller se placer tout en haut à gauche de notre page Web, par-dessus tous les autres éléments. Seulement, ce principe n'est applicable que lorsque notre élément n'est pas déjà lui-même placé dans un élément en positionnement absolu. Si cela arrive, alors notre élément se positionnera non plus par rapport au coin supérieur gauche de la page Web, mais par rapport au coin supérieur gauche du précédent élément placé en positionnement absolu, relatif ou fixe.

Si offsetParent existe, c'est parce que les propriétés offsetTop et offsetLeft contiennent le positionnement de notre élément par rapport à son précédent élément parent et non pas par rapport à la page ! Si nous voulons obtenir son positionnement par rapport à la page, il faudra alors aussi ajouter les valeurs de positionnement de son (ses) élément(s) parent(s). Pour calculer la position de notre élément dans la page, il suffit d’écrire une fonction JavaScript :

function getOffset(element) { // Notre fonction qui calcule le positionnement complet

var top = 0,

left = 0;

do {

top += element.offsetTop;

left += element.offsetLeft;

} while (element = element.offsetParent); // Tant que « element » reçoit un « offsetParent »

valide alors on additionne les valeurs des offsets

return { // On retourne un objet, cela nous permet de retourner les deux valeurs calculées

top: top,

left: left

};

}

**Application : système de drag & drop sur des div**

Partie HTML :

<div class="draggableBox">1</div>

<div class="draggableBox">2</div>

<div class="draggableBox">3</div>

Styles CSS :

.draggableBox {

position: absolute;

width: 80px; height: 60px;

padding-top: 10px;

text-align: center;

font-size: 40px;

background-color: #222;

color: #CCC;

cursor: move;

}

Script :

(function() { // On utilise une IIFE pour ne pas polluer l'espace global

var storage = {}; // Contient l'objet de la div en cours de déplacement

function init() { // La fonction d'initialisation

var elements = document.querySelectorAll('.draggableBox'),

elementsLength = elements.length;

for (var i = 0; i < elementsLength; i++) {

elements[i].addEventListener('mousedown', function(e) { // Initialise le drag & drop

var s = storage;

s.target = e.target;

s.offsetX = e.clientX - s.target.offsetLeft;

s.offsetY = e.clientY - s.target.offsetTop;

});

elements[i].addEventListener('mouseup', function() { // Termine le drag & drop

storage = {};

});

}

document.addEventListener('mousemove', function(e) { // Permet le suivi du drag & drop

var target = storage.target;

if (target) {

target.style.top = e.clientY - storage.offsetY + 'px';

target.style.left = e.clientX - storage.offsetX + 'px';

}

});

}

init(); // On initialise le code avec notre fonction toute prête.

})();

On notera ici la ligne : document.addEventListener('mousemove', function(e)). On applique l’événement mousmove au document car si nous appliquions cet événement à l'élément ciblé, que se passerait-il ? Dès que l'on bougerait la souris, l'événement se déclencherait et tout se passerait comme on le souhaite. Mais si nous bougions la souris trop rapidement, le curseur sortirait de notre élément avant que celui-ci n'ait eu le temps de se déplacer, ce qui fait que l'événement ne se déclenchera plus tant que l'on ne replacerait pas notre curseur sur l'élément.

Un autre problème pourrait aussi surgir : dans notre code actuel, nous ne gérons pas le style CSS z-index, ce qui fait que lorsqu'on déplace le premier élément et que l'on place notre curseur sur un des deux autres éléments, le premier élément se retrouve alors en dessous d'eux. Si nous avions appliqué le mousemove sur notre élément au lieu du document alors cet événement ne se déclencherait pas vu que l'on bouge notre curseur sur un des deux autres éléments et non pas sur notre élément en cours de déplacement.

La solution est donc de mettre l'événement mousemove sur notre document. Vu que cet événement se propage aux enfants, nous sommes sûrs qu'il se déclenchera à n'importe quel déplacement du curseur sur la page.

Enfin, il nous a fallu enregistrer la position du curseur par rapport au coin supérieur gauche de notre élément dès l'initialisation du drag & drop :



Si nous ne le faisions pas, à chaque fois que nous déplacerions notre élément, celui-ci placerait son bord supérieur gauche sous notre curseur et ce n'est clairement pas ce que l'on souhaite.

## Les objets

L'utilisation des objets se fait en deux temps :

* On définit l'objet via un constructeur, cette étape permet de définir un objet qui pourra être réutilisé par la suite. Cet objet ne sera pas directement utilisé car nous en utiliserons une « copie » : on parle alors d'**instance**.
* À chaque fois que l'on a besoin d'utiliser notre objet, on crée une instance de celui-ci.

**Définition via un constructeur**

Le constructeur va contenir la structure de base de notre objet. En programmation orientée objet dans un langage tel que le Java ce constructeur ressemble, sur le principe, à une classe. La syntaxe d'un constructeur est la même que celle d'une fonction :

function Person() {

// Code du constructeur

}

De manière générale on met une majuscule à la première lettre d'un constructeur. Cela permet de mieux le différencier d'une fonction « normale ».

Le code du constructeur va contenir une petite particularité : le mot-clé this. Si on utilise this au sein du constructeur Person, this pointe vers Person. Grâce à this, nous allons pouvoir définir les propriétés de l'objet Person :

function Person(nick) {

this.nick = nick;

}

Les paramètres de notre constructeur vont être détruits à la fin de l'exécution de ce dernier, alors que les propriétés définies par le biais de this vont rester présentes.

**Utilisation de l'objet**

Pour pouvoir utiliser un objet, on définit une variable qui va contenir une instance cet objet. Pour indiquer au JavaScript qu'il faut utiliser une instance, on utilise le mot-clé new.

// Définition de l'objet Person via un constructeur

function Person(nick) {

this.nick = nick;

}

// On crée des variables qui vont contenir une instance de l'objet Person :

var lau = new Person('Laurence');

alert(lau.nick); // Affiche : « Laurence »

Il est possible de faire un test pour savoir si la variable lau est une instance de Person :

alert(lau instanceof Person); // Affiche true

**Modifier les données**

Une fois une variable définie, on peut modifier les propriétés exactement comme s'il s'agissait d'un objet littéral : lau.nick = 'Bastien'; // On change le prénom.

**Ajouter des méthodes**

Il y a deux manières de définir une méthode pour un objet : dans le constructeur ou via prototype. Définir les méthodes directement dans le constructeur est facile puisque c'est nous qui créons le constructeur. La définition de méthodes via prototype est utile surtout si on n'a pas créé le constructeur : ce sera alors utile pour ajouter des méthodes à des objets natifs, comme String ou Array.

**Définir une méthode dans le constructeur**

On procède comme pour les propriétés, sauf qu'il s'agit d'une fonction :

function Person(friends) {

this.friends = friends;

this.addFriend = function(nick, age, sex, parent, work, friends) {

this.friends.push(new Person(nick, age, sex, parent, work, friends));

};

}

**Ajouter une méthode via prototype**

Lorsque l’on définit un objet, il possède automatiquement un sous-objet appelé prototype. Cet objet prototype va nous permettre d'ajouter des méthodes à un objet :

Person.prototype.addFriend = function(nick, age, sex, parent, work, friends) {

this.friends.push(new Person(nick, age, sex, parent, work, friends));

}

Le this fait ici aussi référence à l'objet dans lequel il s'exécute, c'est-à-dire l'objet Person. L'ajout de méthodes par prototype est particulier, car les méthodes ajoutées ne seront pas copiées dans les instances de notre objet. Autrement dit, en ajoutant la méthode addFriend() par prototype, une instance comme lau ne possèdera pas la méthode directement dans son propre objet, elle sera obligée d'aller la chercher au sein de son objet constructeur, ici Person. Cela veut dire que si nous faisons une modification sur une méthode contenue dans un prototype alors nous affecterons toutes les instances de notre objet (y compris celles qui sont déjà créées), cette solution est donc à privilégier.

**Ajouter des méthodes aux objets natifs**

Les objets natifs possèdent eux aussi un objet prototype autorisant donc la modification de leurs méthodes. Par exemple, créons une méthode qui afficherait le contenu d'un objet littéral via alert(), mais de façon plus élégante :

var family = {

self: 'Sébastien',

sister: 'Laurence',

};

Comme il s'agit d'un objet, le type natif est Object. Nous allons utiliser son sous-objet prototype pour lui ajouter la méthode voulue :

// Testons si cette méthode n'existe pas déjà !

if (!Object.prototype.debug) {

// Créons la méthode

Object.prototype.debug = function() {

var text = 'Object {\n';

for (var i in this) {

if (i !== 'debug') {

text += ' [' + i + '] => ' + this[i] + '\n';

}

}

alert(text + '}');

}

}

On test si i est différent de 'debug' parce qu'en ajoutant la méthode debug() aux objets, elle s'ajoute même aux objets littéraux. Autrement dit, debug() va se lister elle-même ce qui n'a pas beaucoup d'intérêt.

Nous avons ajouté ici une méthode à Object pour l'exemple, mais ceci ne doit jamais être reproduit dans un vrai script. En effet, après ajout d'une méthode ou d'une propriété à Object, celle-ci sera listée à chaque fois que nous utiliserons un for in.

**Remplacer des méthodes**

Quand nous utilisons prototype, nous affectons une fonction. Cela veut donc dire qu'il est possible de modifier les méthodes natives des objets en leur affectant une nouvelle méthode. Cela peut se révéler très utile dans certains cas.

**Limitations**

En théorie, chaque objet peut se voir attribuer des méthodes via prototype. Si cela fonctionne avec les objets natifs génériques comme String, Date, Array, Object, Number, Boolean et de nombreux autres, cela fonctionne moins bien avec les objets natifs liés au DOM comme Node, Element ou encore HTMLElement, en particulier dans Internet Explorer.

**Les namespaces**

En informatique, un **namespace** est un ensemble fictif qui contient des informations, généralement des propriétés et des méthodes, ainsi que des sous-namespaces. Le but d'un namespace est de s'assurer de l'unicité des informations qu'il contient.

Par exemple, Sébastien et Johann habitent tous deux au numéro 42. Sébastien dans la rue de Belgique et Johann dans la rue de France. Les numéros de leurs maisons peuvent être confondus, puisqu'il s'agit du même. Ainsi, si Johann dit « J'habite au numéro 42 », c'est ambigu, car Sébastien aussi. Alors, pour différencier les deux numéros, nous allons toujours donner le nom de la rue. Ce nom de rue peut être vu comme un namespace : il permet de différencier deux données identiques.

En programmation, quelque chose d'analogue peut se produire : imaginons que nous développions une fonction myBestFunction(), et que nous trouvions un script tout fait pour réaliser un effet quelconque. Nous ajoutons alors ce script au nôtre. Problème : dans ce script tout fait, une fonction myBestFunction() est aussi présente. Notre fonction se retrouve écrasée par l'autre, et notre script ne fonctionnera plus correctement.

Pour éviter ce genre de désagrément, nous allons utiliser un namespace ! Le JavaScript, au contraire de langages comme le C# ou le Java, ne propose pas de vrai système de namespace. Ce que l'on étudie ici est un système pour reproduire plus ou moins correctement un tel système.

**Définir un namespace**

On commence par créer un objet littéral appelé myNamespace. Ensuite on définit une méthode : myBestFunction() :

var myNamespace = {

myBestFunction: function() {

alert('Ma meilleure fonction !');

}

};

// On exécute la fonction :

myNamespace.myBestFunction();

Pour appeler myBestFunction(), il faut obligatoirement passer par l'objet myNamespace, ce qui limite très fortement la probabilité de voir notre fonction écrasée par une autre. Notre namespace doit être original pour être certain qu'un autre développeur n'utilise pas le même. Cette technique n'est donc pas infaillible, mais réduit considérablement les problèmes.

**Un style de code**

Utiliser un namespace est aussi élégant, car cela permet d'avoir un code visuellement propre et structuré. Une grande majorité des « gros » scripts sont organisés via un namespace, notamment car il est possible de décomposer le script en catégories. Par exemple pour un script qui gère un webmail:

var thundersebWebMail = {

// Propriétés

version: 1.42,

lang: 'english',

// Initialisation

init: function() { /\* initialisation \*/ },

// Gestion des mails

mails: {

list: function() { /\* affiche la liste des mails \*/ },

show: function() { /\* affiche un mail \*/ },

trash: function() { /\* supprime un mail \*/ },

// et cætera…

},

};

Structurer son code de cette manière est propre et lisible, ce qui n'est pas toujours le cas d'une succession de fonctions. Voici l'exemple que nous venons de voir mais cette fois-ci sans namespaces :

var webmailVersion = 1.42,

webmailLang = 'english';

function webmailInit() { /\* initialisation \*/ }

function webmailMailsList() { /\* affiche la liste des mails \*/ }

function webmailMailsShow() { /\* affiche un mail \*/ }

function webmailMailsTrash() { /\* supprime un mail \*/ }

**L'emploi de this**

Le mot-clé this s'utilise ici exactement comme dans les objets vus précédemment. Mais attention, si nous utilisons this dans un sous-namespace, celui-ci pointera vers ce sous-namespace, et non vers le namespace parent.

**Vérifier l'unicité du namespace**

Une sécurité supplémentaire est de vérifier l'existence du namespace : s'il n'existe pas, on le définit et dans le cas contraire, on ne fait rien pour ne pas risquer d'écraser une version déjà existante, tout en retournant un message d'erreur :

// On vérifie l'existence de l'objet myNamespace

if (typeof myNamespace === 'undefined') {

var myNamespace = {

// Tout le code

};

} else {

alert('myNamespace existe déjà !');

}

**Modifier le contexte d'une méthode**

Commençons par étudier la différence entre une fonction et une méthode. Une fonction est indépendante et ne fait partie d'aucun objet (ou presque, n'oublions pas window). La fonction alert() est dans cette catégorie, car nous pouvons l'appeler sans la faire précéder du nom d'un objet : alert('Test !');. Une méthode, en revanche, est dépendante d'un objet. C'est le cas par exemple de la méthode push() qui est dépendante de l'objet Array. Le fait qu'elle soit dépendante est à la fois un avantage et un inconvénient :

* Un avantage car nous n'avons pas à spécifier quel objet la méthode doit modifier.
* Un inconvénient car cette méthode ne pourra fonctionner que sur l'objet dont elle est dépendante.

Cet inconvénient peut être résolu grâce à deux méthodes nommées apply() et call(). Une méthode utilise généralement le mot-clé this pour savoir à quel objet elle appartient, c'est ce qui fait qu'elle est dépendante. Les deux méthodes apply()et call()existent pour permettre de rediriger la référence du mot-clé this vers un autre objet.

La méthode toString() a pour but de fournir une représentation d'un objet sous forme de chaîne de caractères, c'est elle qui est appelée par la fonction alert() lorsque nous lui passons un objet en paramètre. Elle possède cependant un fonctionnement différent selon l'objet sur lequel elle est utilisée :

alert(['test']); // Affiche : « test »

alert({0:'test'}); // Affiche : « [object Object] »

Comme vous avez pu le constater, la méthode toString() renvoie un résultat radicalement différent selon l'objet. Dans le cas d'un tableau, elle retourne son contenu, mais quand il s'agit d'un objet, elle retourne son type converti en chaîne de caractères. Notre objectif va être de faire en sorte d'appliquer la méthode toString() de l'objet Object sur un objet Array, et ce afin d'obtenir sous forme de chaîne de caractères le type de notre tableau au lieu d'obtenir son contenu.

C'est là qu'entrent en jeu les méthodes apply() et call(). Elles vont nous permettre de redéfinir le mot-clé this de la méthode toString(). Ces deux méthodes fonctionnent quasiment de la même manière, elles prennent toutes les deux en paramètre un premier argument obligatoire qui est l'objet vers lequel va pointer le mot-clé this. Les deux méthodes se différencient sur les arguments facultatifs.

Dans notre exemple actuel, nous voulons modifier la méthode toString() de l'objet Object. Il faut alors écrire :

var result = Object.prototype.toString.call(['test']);

alert(result); // Affiche : « [object Array] »

La méthode toString() de Object a bien été appliquée à notre tableau, nous obtenons donc son type et non pas son contenu.

Revenons maintenant sur les arguments facultatifs de nos deux méthodes apply() et call(). La première prend en paramètre facultatif un tableau de valeurs, tandis que la deuxième prend une infinité de valeurs en paramètres. Ces arguments facultatifs servent à la même chose : ils seront passés en paramètres à la méthode souhaitée. Ainsi, si nous écrivons :

var myArray = [];

myArray.push.apply(myArray, [1, 2, 3]);

Cela revient au même que si nous avions écrit :

var myArray = [];

myArray.push.call(myArray, 1, 2, 3);

Le tout correspondant à :

var myArray = [];

myArray.push(1, 2, 3);

**L'héritage**

Pour appliquer le principe d’héritage au JavaScript, créons par exemple un constructeur Vehicle:

function Vehicle(licensePlate, tankSize) {

this.engineStarted = false; // Notre véhicule est-il démarré ?

this.licensePlate = licensePlate; // La plaque d'immatriculation de notre véhicule.

this.tankSize = tankSize; // La taille de notre réservoir en litres.

}

// Permet de démarrer notre véhicule.

Vehicle.prototype.start = function() {

this.engineStarted = true;

};

// Permet d'arrêter notre véhicule.

Vehicle.prototype.stop = function() {

this.engineStarted = false;

};

Maintenant que notre objet Vehicle  est prêt, nous pouvons l'exploiter. L'héritage va ici consister à créer des objets constructeurs qui vont tous hériter de Vehicle. Par exemple pour une voiture :

function Car(licensePlate, tankSize, trunkSize) {

// On appelle le constructeur de « Vehicle » par le biais de la méthode

// call() afin qu'il affecte de nouvelles propriétés à « Car ».

Vehicle.call(this, licensePlate, tankSize);

// Une fois le constructeur parent appelé, l'initialisation de notre objet peut continuer.

this.trunkOpened = false; // Notre coffre est-il ouvert ?

this.trunkSize = trunkSize; // La taille de notre coffre en mètres cube.

}

// L'objet prototype de « Vehicle » doit être copié au sein du prototype

// de « Car » afin que ce dernier puisse bénéficier des mêmes méthodes.

Car.prototype = Object.create(Vehicle.prototype, {

// Le prototype copié possède une référence vers son constructeur, actuellement

// défini à « Vehicle », nous devons changer sa référence pour « Car »

// tout en conservant sa particularité d'être une propriété non-énumerable.

constructor: {

value: Car,

enumerable: false,

writable: true,

configurable: true

}

});

// Il est bien évidemment possible d'ajouter de nouvelles méthodes.

Car.prototype.openTrunk = function() {

this.trunkOpened = true;

};

La méthode create() permet de créer un nouvel objet avec l’objet prototype et les propriétés spécifiés.

Il est maintenant possible d'instancier une nouvelle voiture de manière tout à fait classique :

var myCar = new Car('AA-555-AA', 70, 2.5);

## Les chaînes de caractères

Pour créer une chaîne de caractères, on utilise généralement cette syntaxe :

var myString = "Chaîne de caractères primitive";

Cet exemple crée ce que l'on appelle une chaîne de caractères primitive, qui n'est pas un objet String. Pour instancier un objet String, il faut faire comme ceci :

var myRealString = new String("Chaîne");

Cela est valable pour les autres objets : Array, Object, Boolean, Number. La différence entre type primitif et instance est minime pour nous, développeurs. Prenons l'exemple de la chaîne de caractères : à chaque fois que nous allons faire une opération sur une chaîne primitive, le JavaScript va automatiquement convertir cette chaîne en une instance temporaire de String, de manière à pouvoir utiliser les propriétés et méthodes fournies par l'objet String. Une fois les opérations terminées, l'instance temporaire est détruite. Au final, utiliser un type primitif ou une instance revient au même du point de vue de l'utilisation. Mais il subsiste de légères différences avec l'opérateur instanceof qui peut retourner de drôles de résultats.

Pour une raison ou une autre, imaginons que l'on veuille savoir de quelle instance est issu un objet :

var myString = 'Chaîne de caractères';

if (myString instanceof String) {

// Faire quelque chose

}

Ici, la condition sera fausse. C’est normal puisque myString est une chaîne primitive et non une instance de String. Pour tester le type primitif, il convient d'utiliser l'opérateur typeof :

if (typeof myString === 'string') {

// Faire quelque chose

}

typeof permet de vérifier le type primitif (en anglais on parle de datatype). Mais il faut ici aussi faire attention à un piège, car la forme primitive d'une instance de String est object. Il est même d'ailleurs déconseillé de faire ce genre de tests vu le nombre de problèmes que cela peut causer. La seule valeur retournée par typeof dont on peut être sûr est "undefined".

**L'objet String**

String ne possède qu'une seule propriété, length, qui retourne le nombre de caractères contenus dans une chaîne. Les espaces, les signes de ponctuation, les chiffres sont considérés comme des caractères.

Il n'est pas toujours obligatoire de déclarer une variable pour utiliser les propriétés et les méthodes d'un objet. En effet, nous pouvons écrire directement le contenu de notre variable et utiliser une de ses propriétés ou de ses méthodes, comme c'est le cas ici : alert('Ceci est une chaîne !'.length);. Cependant cela ne fonctionne pas avec les nombres sous forme primitive car le point est le caractère permettant d'ajouter une ou plusieurs décimales. Ainsi, ce code générera une erreur : 0.toString();.

**La casse et les caractères**

toLowerCase() et toUpperCase() permettent respectivement de convertir une chaîne en minuscules et en majuscules : myString = myString.toUpperCase();.

La méthode charAt() permet de récupérer un caractère en fonction de sa position dans la chaîne de caractères. La méthode reçoit en paramètre la position du caractère : var first = myString.charAt(0);.

La méthode charCodeAt() fonctionne comme charAt() à la différence que ce n'est pas le caractère qui est retourné mais le code ASCII du caractère.

fromCharCode() permet de faire plus ou moins l'inverse de charCodeAt() : instancier une nouvelle chaîne de caractères à partir d'une chaîne ASCII, dont chaque code est séparé par une virgule. Son fonctionnement est particulier puisqu'il est nécessaire d'utiliser l'objet String lui-même : var myString = String.fromCharCode(74, 97, 118, 97, 83, 99, 114, 105, 112, 116); // le mot JavaScript en ASCII.

trim() sert à supprimer les espaces avant et après une chaîne de caractères. C'est particulièrement utile quand on récupère des données saisies dans une zone de texte, car l'utilisateur est susceptible d'avoir laissé des espaces avant et après son texte.

**Rechercher, couper et extraire**

La méthode indexOf() est utile dans deux cas de figure :

* Savoir si une chaîne de caractères contient un caractère ou un morceau de chaîne.
* Savoir à quelle position se trouve le premier caractère de la chaîne recherchée.

indexOf() retourne la position du premier caractère/chaine de caractères trouvé, et s'il n'y en a pas la valeur -1 est retournée. lastIndexOf() retourne la position de la dernière occurrence trouvée. Ces deux fonctions possèdent chacune un deuxième argument qui permet de spécifier à partir de quel index la recherche doit commencer.

**Utiliser le tilde avec indexOf() et lastIndexOf()**

Une particularité intéressante et relativement méconnue du JavaScript est son caractère tilde ~. Il s'agit de l'opérateur binaire « NOT » qui a pour rôle d'inverser tous les bits d'une valeur. Comme le binaire ne nous sert pas à grand chose dans le cadre de notre apprentissage, il est plus logique d'expliquer son influence sur les nombres en base décimale : le tilde incrémente la valeur qui le suit et y ajouter une négation : alert(~2); // Affiche : « -3 ».

**Extraire une chaîne avec substring(), substr() et slice()**

substring(a, b) permet d'extraire une chaîne à partir de la position a (incluse) jusqu'à la position b (exclue).

Une autre manière de procéder serait d'utiliser substr(), la méthode sœur de substring(). substr(a, n) accepte deux paramètres : le premier est la position de début, et le deuxième le nombre de caractères à extraire. Cela suppose donc de connaître le nombre de caractères à extraire.

Une dernière méthode d'extraction existe : slice(). slice() ressemble très fortement à substring(), mais avec une option en plus. Une valeur négative est transmise pour la position de fin. slice() va extraire la chaîne jusqu'à la fin, en décomptant le nombre de caractères indiqué. Par exemple, si on ne veut récupérer que « Thunder », on peut faire comme ceci : var nick\_1 = 'Thunderseb'.slice(0, -3);.

**Couper une chaîne en un tableau avec split()**

La méthode split() permet de couper une chaîne de caractères à chaque fois qu'une sous-chaîne est rencontrée. Les « morceaux » résultant de la coupe de la chaîne sont placés dans un tableau :

var splitted = myString.split(','); // On coupe à chaque fois qu'une virgule est rencontrée

**Tester l'existence d'une chaîne de caractères**

La méthode valueOf(), héritée de Object, renvoie la valeur primitive de n'importe quel objet. Ainsi, si on crée une instance de String :

var string\_1 = new String('Test');

et que l'on récupère le résultat de sa méthode valueOf() dans la variable string\_2 :

var string\_2 = string\_1.valueOf();

alors l'instruction typeof montre bien que string\_1 est une instance de String et que string\_2 est une valeur primitive :

alert(typeof string\_1); // Affiche : « object »

alert(typeof string\_2); // Affiche : « string »

Grâce à cette méthode, il devient simple de vérifier si une variable contient une chaîne de caractères :

function isString(variable) {

return typeof variable.valueOf() === 'string'; // Si le type de la valeur primitive est « string » alors on retourne « true »

}

Si on passe une valeur primitive à notre fonction, celle-ci renverra la même valeur. valueOf() retourne la valeur primitive d'un objet, mais si cette méthode est utilisée sur une valeur qui est déjà de type primitif, alors elle va retourner la même valeur primitive.

Enfin, il est aussi possible d'obtenir une instanciation d'objet à partir d'un type primitif. Il suffit de procéder de cette manière : var myString = Object('Mon texte');. En créant une instance de Object avec un type primitif en paramètre, l'objet instancié sera de même type que la valeur primitive. Si nous passons en paramètre un type primitif string alors nous obtiendrons une instance de l'objet String avec la même valeur passée en paramètre.

## Les expressions régulières

Les regex ne s'utilisent pas seules, et il y a deux manières de s'en servir : soit par le biais de RegExp qui est l'objet qui gère les expressions régulières, soit par le biais de certaines méthodes de l'objet String :

* match() : retourne un tableau contenant toutes les occurrences recherchées.
* search() : retourne la position d'une portion de texte (semblable à indexOf() mais avec une regex).
* split() : la méthode split()avec une regex en paramètre.
* replace() : effectue un rechercher/remplacer.

L'instanciation d'un objet RegExp se fait comme ceci : var myRegex = /contenu\_à\_rechercher/;. Si notre regex contient elle-même des slashs, il faudra les échapper en utilisant un anti-slash.

La méthode test() de l’objet RegExp renvoie true en cas de réussite du test, sinon false :

var myRegex = /contenu\_à\_rechercher/;

if (myRegex.test('Chaîne de caractères dans laquelle effectuer la recherche')) {

// Retourne true si le test est réussi

} else {

// Retourne false dans le cas contraire

}

On peut aussi écrire if (/contenu\_à\_rechercher/.test('Chaîne de caractères bla bla bla')).

**Recherches de mots**

Pour chercher si dans une phrase le mot « raclette » apparaît on écrit :

if (/raclette/.test('Je mangerais bien une raclette savoyarde !')) {

alert('Ça semble parler de raclette');

} else {

alert('Pas de raclette à l\'horizon');

}

Dans ce cas précis, la casse est importante. Il est possible, grâce aux options, de dire à la regex d'ignorer la casse. Cette option s'appelle i, et comme chaque option, elle se place juste après le slash de fermeture de la regex : /Raclette/i.

Pour effectuer une recherche sur des mots différents, nous disposons de l'opérateur OU, représenté par la barre verticale pipe | : /Raclette|Tartiflette|Fondue|Croziflette/.

**Début et fin de chaîne**

Les symboles ^ et $ permettent respectivement de représenter le début et la fin d'une chaîne de caractères. Si un de ces symboles est présent, il indique à la regex que ce qui est recherché commence ou termine la chaîne : /^raclette savoyarde$/. Voici un tableau avec divers tests qui sont effectués pour montrer l'utilisation de ces deux symboles :

|  |  |  |
| --- | --- | --- |
| **Chaîne** | **Regex** | **Résultat** |
| Raclette savoyarde | /^Raclette savoyarde$/ | true |
| Une raclette savoyarde | /^Raclette/ | false |
| Une raclette savoyarde | /savoyarde$/ | true |
| Une raclette savoyarde ! | /raclette savoyarde$/ | false |

**Les caractères et leurs classes**

Une classe de caractères est écrite entre crochets et sa signification est simple : une des lettres qui se trouve entre les crochets peut convenir. Cela veut donc dire que l'exemple suivant va trouver les mots « gras » et « gros », car la classe, à la place de la voyelle, contient aux choix les lettres a et o : /gr[ao]s/.

**Les intervalles de caractères**

Toujours au sein des classes de caractères, il est possible de spécifier un intervalle de caractères. Si nous voulons trouver les lettres allant de a à o, on écrira [a-o]. Si n'importe quelle lettre de l'alphabet peut convenir, il est donc inutile de les écrire toutes, écrire [a-z] suffit.

Plusieurs intervalles peuvent être écrits au sein d'une même classe. Ainsi, la classe [a-zA-Z] va rechercher toutes les lettres de l'alphabet, qu'elles soient minuscules ou majuscules. Si un intervalle fonctionne avec des lettres, il fonctionne aussi avec des chiffres. La classe [0-9] trouvera donc un chiffre allant de 0 à 9. Il est bien évidemment possible de combiner des chiffres et des lettres : [a-z0-9] trouvera une lettre minuscule ou un chiffre.

**Exclure des caractères**

Si au sein d'une classe on peut inclure des caractères, on peut aussi en exclure. Pour cela, il suffit de faire figurer un accent circonflexe au début de la classe, juste après le premier crochet. Ainsi cette classe ignorera les voyelles : [^aeiouy]. L'exclusion d'un intervalle est possible aussi : [^b-y].

Il faut prendre en compte le fait que la recherche n'ignore pas les caractères accentués. Ainsi, [a-z] trouvera a, b, i, o… mais ne trouvera pas â, ï ou encore ê. S'il s'agit de trouver un caractère accentué, il faut l'indiquer explicitement : [a-zâäàéèùêëîïôöçñ]. Il n'y a toutefois pas besoin d'écrire les variantes en majuscules si l'option i est utilisée : /[a-zâäàéèùêëîïôöçñ]/i.

Enfin, Le point permet de trouver n'importe quel caractère, à l'exception des sauts de ligne (les retours à la ligne) : /gr.s/.

**Les quantificateurs**

Les quantificateurs permettent de dire combien de fois un caractère doit être recherché. Il est possible de dire qu'un caractère peut apparaître 0 ou 1 fois, 1 fois ou une infinité de fois :

* ? : ce symbole indique que le caractère qui le précède peut apparaître 0 ou 1 fois.
* + : ce symbole indique que le caractère qui le précède peut apparaître 1 ou plusieurs fois ;
* \* : ce symbole indique que le caractère qui le précède peut apparaître 0, 1 ou plusieurs fois.

**Les accolades**

À la place des trois symboles vus précédemment, on peut utiliser des accolades pour définir explicitement combien de fois un caractère peut être répété :

* {n} : le caractère est répété n fois.
* {n,m} : le caractère est répété de n à m fois. Par exemple, si on a {0, 5}, le caractère peut être présent de 0 à 5 fois.
* {n,} : le caractère est répété de n fois à l'infini.

Par exemple :

|  |  |  |
| --- | --- | --- |
| **Chaîne** | **Regex** | **Résultat** |
| Hellowwwwwwwww | /Hellow+/ | true |
| Goooooogle | /Go{2,}gle/ | true |
| Le 1er septembre | /Le [1-9][a-z]{2,3} septembre/ | true |
| Le 1er septembre | /Le [1-9][a-z]{2,3}[a-z]+/ | false |

La dernière regex est fausse à cause de l'espace. En effet, la classe [a-z] ne trouvera pas l'espace.

**Les métacaractères**

Les caractères suivants sont ce que l'on appelle des **métacaractères**, et en voici la liste complète :

! ^ $ ( ) [ ] { } ? + \* . / \ |

Si on veut chercher la présence d'un **métacaractères** dans une chaîne, il suffit de l'échapper au moyen d'un anti-slash. Il faut même penser à échapper l'anti-slash avec… un anti-slash. Par exemple : /\/ et un anti-slash \\/ renverra true pour « Un slash / et un anti-slash \ ».

Au sein d'une classe de caractères, il n'y a pas besoin d'échapper les métacaractères, à l'exception des crochets (qui délimitent le début et la fin d'une classe), du tiret (qui est utilisé pour définir un intervalle) et de l'anti-slash (qui sert à échapper). Concernant le tiret, il existe une petite exception : il n'a pas besoin d'être échappé s'il est placé en début ou en fin de classe. Par exemple : /[a-z!?\/\\]/.

**Types génériques et assertions**

Les types génériques s'écrivent tous de la manière suivante : \x, où x représente une lettre. Voici la liste de tous les types génériques :

|  |  |
| --- | --- |
| **Type** | **Description** |
| \d | Trouve un caractère décimal (un chiffre) |
| \D | Trouve un caractère qui n'est pas décimal (donc pas un chiffre) |
| \s | Trouve un caractère blanc |
| \S | Trouve un caractère qui n'est pas un caractère blanc |
| \w | Trouve un caractère « de mot » : une lettre, accentuée ou non, ainsi que l'underscore |
| \W | Trouve un caractère qui n'est pas un caractère « de mot » |

En plus de cela existent les caractères de type « espace blanc » :

|  |  |
| --- | --- |
| **Type** | **Description** |
| \n | Trouve un retour à la ligne |
| \t | Trouve une tabulation |

Ces deux caractères sont reconnus par le type générique \s.

**Les assertions**

Les assertions s'écrivent comme les types génériques mais ne fonctionnent pas tout à fait de la même façon. Un type générique recherche un caractère, tandis qu'une assertion recherche entre deux caractères :

|  |  |
| --- | --- |
| **Type** | **Description** |
| \b | Trouve une limite de mot |
| \B | Ne trouve pas de limite de mot |

Il faut faire attention avec l'utilisation de \b, car cette assertion reconnaît les caractères accentués comme des « limites de mots ».

**Construire une regex**

Pour construire une regex, il suffit de procéder par étapes. Par exemple, vérifierions si une chaîne de caractères correspond à une adresse e-mail. On écrit tout d'abord la partie locale, qui n'est composée que de lettres, de chiffres et éventuellement d'un tiret, un trait de soulignement et un point. Tous ces caractères peuvent être répétés plus d'une fois, il faut donc utiliser le quantificateur + : /^[a-z0-9.\_-]+$/. On ajoute l'arobase. Ce n'est pas un métacaractère, donc pas besoin de l'échapper : /^[a-z0-9.\_-]+@$/. Après vient le label du nom de domaine, lui aussi composé de lettres, de chiffres, de tirets et de traits de soulignement. Ne pas oublier le point, car il peut s'agir d'un sous-domaine (par exemple @cours.openclassrooms.com) : /^[a-z0-9.\_-]+@[a-z0-9.\_-]+$/. Puis vient le point de l'extension du domaine : attention à ne pas oublier de l'échapper, car il s'agit d'un métacaractère : /^[a-z0-9.\_-]+@[a-z0-9.\_-]+\.$/. Et pour finir, l'extension ! Une extension de nom de domaine ne contient que des lettres, au minimum 2, au maximum 6. Ce qui nous fait : /^[a-z0-9.\_-]+@[a-z0-9.\_-]+\.[a-z]{2,6}$/.

**L'objet RegExp**

L'objet RegExp est l'objet qui gère les expressions régulières. Il y a donc deux façons de déclarer une regex : via RegExp ou via son type primitif que nous avons utilisé jusqu'à présent :

var myRegex1 = /^Raclette$/i;

var myRegex2 = new RegExp("^Raclette$", "i");

Le constructeur RegExp reçoit deux paramètres : le premier est l'expression régulière sous la forme d'une chaîne de caractères, et le deuxième est l'option de recherche, ici i. L'intérêt d'utiliser RegExp est qu'il est possible d'inclure des variables dans la regex, chose impossible en passant par le type primitif :

var nickname = "Laura";

var myRegex = new RegExp("Mon prénom est " + nickname, "i");

Il est conseillé d'utiliser la notation littérale (le type primitif) quand l'utilisation du constructeur RegExp n'est pas nécessaire.

**Méthodes**

RegExp ne possède que deux méthodes : test() et exec(). La méthode test() a déjà été utilisée et permet de tester une expression régulière. Elle renvoie true si le test est réussi ou false si le test échoue. De son côté, exec() applique également une expression régulière, mais renvoie un tableau dont le premier élément contient la portion de texte trouvée dans la chaîne de caractères. Si rien n'est trouvé, null est renvoyé.

var sentence = "Si ton tonton";

var result = /\bton\b/.exec(sentence); // On cherche à récupérer le mot « ton »

if (result) { // On vérifie que ce n'est pas null

alert(result); // Affiche « ton »

}

**Propriétés**

L'objet RegExp contient neuf propriétés, appelées $1, $2, $3… jusqu'à $9. Il est possible d'utiliser une regex pour extraire des portions de texte, et ces portions sont accessibles via les propriétés $1 à $9.

**Les parenthèses**

Pour définir les informations à extraire, on utilise des parenthèses, que l'on appelle **parenthèses capturantes**. Par exemple :

var birth = 'Je suis né en mars';

/^Je suis né en (\S+)$/.exec(birth);

alert(RegExp.$1); // Affiche : « mars »

Dans un premier temps, on crée la regex avec les fameuses parenthèses. Comme les mois sont faits de caractères qui peuvent être accentués, on peut directement utiliser le type générique \S. \S+ indique qu'on recherche une série de caractères, jusqu'à la fin de la chaîne (délimitée, pour rappel, par $) : ce sera le mois. On englobe ce « mois » dans des parenthèses pour faire comprendre à l'interpréteur JavaScript que leur contenu devra être extrait. Le premier couple de parenthèses sera enregistré dans la propriété $1, le deuxième dans $2 et ainsi de suite, jusqu'au neuvième, dans $9. Et pour accéder aux propriétés, il suffit de faire RegExp.$1, RegExp.$2, etc.

**Les parenthèses non capturantes**

Il se peut que dans des regex assez longues et complexes, il y ait besoin d'utiliser beaucoup de parenthèses. Toutes ces parenthèses n'ont peut-être pas besoin de capturer quelque chose, elles peuvent juste être là pour proposer un choix. Par exemple, si on vérifie une URL, on peut commencer la regex comme ceci : /(https|http|ftp|steam):\/\//. Mais on n'a pas besoin que ce soit une parenthèse capturante. Pour la rendre non capturante il suffit d’ajouter ?: au début de la parenthèse : /(?:https|http|ftp|steam):\/\//.

**Les recherches non-greedy**

En JavaScript, les regex sont généralement gourmandes, ce qui veut dire que lorsqu'on utilise un quantificateur comme le +, le maximum de caractères est recherché, alors que ce n'est pas toujours le comportement espéré. Nous allons construire une regex qui va extraire l'adresse Web à partir de cette portion HTML sous forme de chaîne de caractères :

var html = '<a href="www.mon-adresse.be">Mon site</a>';

Voici la regex qui peut être construite : /<a href="(.+)">/. Et ça marche :

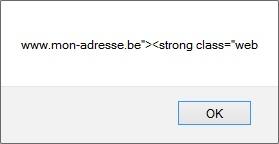
/<a href="(.+)">/.exec(html);

alert(RegExp.$1); // www.mon-adresse.be

Maintenant, supposons que la chaîne de caractères ressemble à ceci :

var html = '<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>';

Ceci affichera :



En spécifiant .+ comme quantificateur, on demande de rechercher le plus possible de caractères jusqu'à rencontrer les caractères « "> », et c'est ce que le JavaScript fait :

JavaScript s'arrête à la dernière occurrence souhaitée

Le JavaScript va trouver la partie surlignée : il cherche jusqu'à ce qu'il tombe sur la dernière apparition des caractères « "> ». Pour pallier ce problème, nous allons écrire le quantificateur directement suivi du point d'interrogation :

var html = '<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>';

/<a href="(.+?)">/.exec(html);

alert(RegExp.$1);

Le point d'interrogation va faire en sorte que la recherche soit moins gourmande et s'arrête une fois que le minimum requis est trouvé , d'où l’appellation non-greedy (« non gourmande »).

**Rechercher et remplacer**

Une fonctionnalité intéressante des regex est de pouvoir effectuer des « rechercher-remplacer ». Rechercher-remplacer signifie qu'on recherche des portions de texte dans une chaîne de caractères et qu'on remplace ces portions par d'autres. Un rechercher-remplacer se fait au moyen de la méthode replace() de l'objet String. Elle reçoit deux arguments : la regex et une chaîne de caractères qui sera le texte de remplacement :

var sentence = 'Je m\'appelle Laura’;

var result = sentence.replace(/Laura/, 'Johann');

alert(result); // Affiche : « Je m'appelle Johann »

**Utilisation de replace() sans regex**

À la place d'une regex, il est aussi possible de fournir une simple chaîne de caractères :

var result = 'Je m\'appelle Laura'.replace('Laura', 'Johann');

alert(result); // Affiche : « Je m'appelle Johann »

**L'option g**

L’option, g signifie « rechercher plusieurs fois ». Par défaut, la regex de l’exemple précédent ne sera exécutée qu'une fois : dès que « Laura » sera trouvé, il sera remplacé, et c'est tout. Donc si le prénom « Sébastien » est présent deux fois, seul le premier sera remplacé. Pour éviter ça, on utilisera l'option g qui va dire de continuer la recherche jusqu'à ce que plus rien ne soit trouvé :

var sentence = 'Elle s\'appelle Laura. Laura écrit un tutoriel.';

var result = sentence.replace(/Laura/g, 'Johann');

alert(result); // Elle s'appelle Johann. Johann écrit un tutoriel.

Ainsi, toutes les occurrences de « Laura » sont correctement remplacées par « Johann ».

**Rechercher et capturer**

Il est possible d'utiliser les parenthèses capturantes pour extraire des informations et les réutiliser au sein de la chaîne de remplacement. Par exemple, nous avons une date au format américain : 05/26/2011, et nous souhaitons la convertir au format jour/mois/année. Rien de plus simple :

var date = '05/26/2011';

date = date.replace(/^(\d{2})\/(\d{2})\/(\d{4})$/, 'Le $2/$1/$3');

alert(date); // Le 26/05/2011

Pour placer un simple caractère $ dans la chaîne de remplacement, il suffit de le doubler :

var total = 'J\'ai 25 dollars en liquide.';

alert(total.replace(/dollars?/, '$$'); // J'ai 25 $ en liquide

Un point d'interrogation a été placé après le « s » pour pouvoir trouver soit « dollars » soit « dollar ».

Voici un autre exemple illustrant ce principe. L'idée ici est de convertir une balise BBCode de mise en gras ([b]un peu de texte[/b]) en un formatage HTML de ce type : <strong>un peu de texte</strong> :

var text = 'bla bla [b]un peu de texte[/b] bla [b]bla bla en gras[/b] bla bla';

text = text.replace(/\[b\]([\s\S]\*?)\[\/b\]/g, '<strong>$1</strong>');

Pourquoi avoir utilisé [\s\S] et non pas juste le point ? Parce qu’il s'agit ici de trouver tous les caractères qui se trouvent entre les balises. Or, le point ne trouve que des caractères et des espaces blanc hormis le retour à la ligne. C'est la raison pour laquelle on utilisera souvent la classe comprenant \s et \S quand il s'agira de trouver du texte comportant des retours à la ligne.

Cette petite regex de remplacement est la base d'un système de prévisualisation du BBCode :

<script>

function preview() {

var value = document.getElementById("text").value;

value = value.replace(/\[b\]([\s\S]\*?)\[\/b\]/g, '<strong>$1</strong>'); // Gras

value = value.replace(/\[i\]([\s\S]\*?)\[\/i\]/g, '<em>$1</em>'); // Italique

value = value.replace(/\[s\]([\s\S]\*?)\[\/s\]/g, '<del>$1</del>'); // Barré

value = value.replace(/\[u\]([\s\S]\*?)\[\/u\]/g, '<span style="text-decoration: underline">$1</span>'); // Souligné

document.getElementById("output").innerHTML = value;

}

</script>

<form>

<textarea id="text"></textarea><br />

<button type="button" onclick="preview()">Prévisualiser</button>

<div id="output"></div>

</form>

**Utiliser une fonction pour le remplacement**

À la place d'une chaîne de caractères, il est possible d'utiliser une fonction pour gérer le ou les remplacements. Cela permet, par exemple, de réaliser des opérations sur les portions capturées. Les paramètres de la fonction sont soumis à une petite règle, car ils doivent respecter un certain ordre (mais leurs noms peuvent bien évidemment varier) : function(str, p1, p2, p3 /\* ... \*/, offset, s). Les paramètres p1, p2, p3… représentent les $1, $2, $3. S'il n'y a que trois parenthèses capturantes, il n'y aura que trois « p ». S'il y en a cinq, il y aura cinq « p » :

* Le paramètre str contient la portion de texte trouvée par la regex.
* Les paramètres p\* contiennent les portions capturées par les parenthèses.
* Le paramètre offset contient la position de la portion de texte trouvée.
* Le paramètre s contient la totalité de la chaîne.

Si on n'a besoin que de p1 et de p2 et pas des deux derniers paramètres, ces deux derniers peuvent être omis.

Pour illustrer cela, nous allons réaliser un script qui recherchera des nombres dans une chaîne et les transformera en toutes lettres. La transformation se fera au moyen de de la fonction num2Letters() (non explicitée ici) :

var sentence = 'Dans 22 jours, j\'aurai 24 ans';

var result = sentence.replace(/(\d+)/g, function(str, p1) {

p1 = parseInt(p1);

if (!isNaN(p1)) {

return num2Letters(p1);

}

});

alert(result); // Affiche : « Dans vingt-deux jours, j'aurai vingt-quatre ans »

L'exemple utilise une fonction anonyme, mais il est bien évidemment possible de déclarer une fonction :

function convertNumbers(str) {

str = parseInt(str);

if (!isNaN(str)) {

return num2Letters(str);

}

}

var sentence = 'Dans 22 jours, j\'aurai 24 ans';

var result = sentence.replace(/(\d+)/g, convertNumbers);

**Autres recherches**

Il reste deux méthodes de String à voir, search() et match().

**Rechercher la position d'une occurrence**

La méthode search(), toujours de l'objet String, ressemble à indexOf() mis à part le fait que le paramètre est une expression régulière. search() retourne la position de la première occurrence trouvée. Si aucune occurrence n'est trouvée, -1 est retourné.

**Récupérer toutes les occurrences**

La méthode match() de l'objet String fonctionne comme search(), à la différence qu'elle retourne un tableau de toutes les occurrences trouvées.

**Couper avec une regex**

Nous avions vu que la méthode split() recevait une chaîne de caractères en paramètre. Mais il est également possible de transmettre une regex. C'est très pratique pour découper une chaîne à l'aide de plusieurs caractères distincts : /[,:;]/.

## Les données numériques

La gestion des données numériques en JavaScript se fait essentiellement par le biais des objets Number et Math.

**L'objet Number**

Cet objet possède des fonctions intéressantes comme, par exemple, celle permettant de faire des conversions depuis une chaîne de caractères jusqu'à un nombre en instanciant un nouvel objet Number : var myNumber = new Number('3'). Cependant, cette conversion est imprécise dans le sens où on ne sait pas si on obtiendra un nombre entier ou flottant en retour. On lui préfère donc les fonctions parseInt() et parseFloat().De plus, parseInt() utilise un second argument permettant de spécifier la base (2 pour le système binaire, 10 pour le système décimal...) du nombre écrit dans la chaîne de caractères.

Cet objet possède des propriétés accessibles directement sans aucune instanciation (on appelle cela des propriétés propres à l'objet constructeur). Elles sont au nombre de cinq :

* NaN : Not A Number permet, généralement, d'identifier l'échec d'une conversion de chaîne de caractères en un nombre. À noter que cette propriété est aussi disponible dans l'espace global. Passer par l'objet Number pour y accéder n'a donc que peu d'intérêt, surtout qu'il est rare d'utiliser cette propriété car on lui préfère la fonction isNaN(), plus fiable.
* MAX\_VALUE : cette propriété représente le nombre maximum pouvant être stocké dans une variable en JavaScript. Cette constante peut changer selon la version du JavaScript utilisée.
* MIN\_VALUE : identique à la constante MAX\_VALUE sauf que qu’il s'agit de la valeur minimale.
* POSITIVE\_INFINITY : il s'agit d'une constante représentant l'infini positif. Il est possible de l'obtenir en résultat d'un calcul si nous divisons une valeur positive par 0. Cependant, son utilisation est rare car on lui préfère la fonction isFinite(), plus fiable.
* NEGATIVE\_INFINITY : identique à POSITIVE\_INFINITY sauf qu’il s'agit de l'infini négatif. Il est possible d’obtenir cette constante en résultat d'un calcul si nous divisons une valeur négative par 0.

Exemple :

var max = Number.MAX\_VALUE, // Nous n'instancions pas d'objet, comme pour un accès au « prototype »

inf = Number.POSITIVE\_INFINITY;

if (max < inf) {

alert("La valeur maximum est inférieure à l'infini ! Surprenant, n'est-ce pas ?");

}

L'objet Number n'est pas très intéressant concernant les méthodes qu’il possède, celles-ci sont déjà supportées par l'objet Math.

**L'objet Math**

Toutes les propriétés et méthodes de cet objet sont accessibles directement sans aucune instanciation, on appelle cela des méthodes et propriétés statiques.

**Les propriétés**

Les propriétés de l'objet Math sont des constantes qui définissent certaines valeurs mathématiques comme le nombre pi (π) ou le nombre d'Euler (e) :

alert(Math.PI); // Affiche la valeur du nombre pi

alert(Math.E); // Affiche la valeur du nombre d'Euler

**Les méthodes**

La méthode floor() retourne le plus grand entier inférieur ou égal à la valeur passée en paramètre : Math.floor(33.95); // Retourne : 33.

La méthode ceil()retourne le plus petit entier supérieur ou égal à la valeur passée en paramètre :

Math.ceil(33.15); // Retourne : 34.

La méthode round()fait un arrondi classique : Math.round(33.15); // Retourne : 33.

La méthode pow() permet de calculer une puissance : Math.pow(3, 2); // Le premier paramètre est la base, le deuxième est l'exposant.

La méthode sqrt()permet de calculer une racine carré : Math.sqrt(9); // Retourne : 3.

Il existe des méthodes permettant de travailler sur des angles : cos(), sin(), acos() et asin().es résultats obtenus sont exprimés en radians.

Les méthodes max() et min() permettent de retrouver les valeurs maximum et minimum dans une liste de nombres : Math.max(42, 4, 38, 1337, 105); // Retourne : 1337.

La méthode random() permet d’obtenir un nombre décimal aléatoire entre 0 (inclus) et 1 (exclu) :

alert(Math.random());. Il suffit donc de créer un script pour obtenir un nombre aléatoire contenu dans un intervalle donné :

function rand(min, max, integer) {

if (!integer) {

return Math.random() \* (max - min) + min;

} else {

return Math.floor(Math.random() \* (max - min + 1) + min);

}

}

Le troisième paramètre sert à définir si l'on souhaite obtenir un nombre entier ou non.

**Les inclassables**

En JavaScript, certaines fonctions globales permettent de faire quelques conversions et contrôles de données poussés.

Les fonctions parseInt() et parseFloat()permettent de convertir une chaîne de caractères en un nombre. La première possède deux arguments tandis que la seconde en possède un seul :

* Le premier argument est obligatoire, il s'agit de la chaîne de caractère à convertir en nombre.
* Le deuxième argument est facultatif. Il permet de spécifier la base que le nombre utilise dans la chaîne de caractères.

L'importance du deuxième argument est simple à démontrer avec un exemple :

var myString = '08';

alert(parseInt(myString)); // Affiche : 0

alert(parseInt(myString, 10)); // Affiche : 8

En l'absence d'un second argument, les fonctions parseInt() et parseFloat() vont tenter de deviner la base utilisée par le nombre écrit dans la chaîne de caractères. Ici, la chaîne de caractères commence par un 0, ce qui est caractéristique du système octal, on obtient donc 0 en retour.

La fonction isNaN() permet de savoir si notre variable contient un nombre. Renvoie true si la variable ne contient pas de nombre.

La fonction isFinite() permet de déterminer si le nombre est fini. Renvoie true si le nombre ne tend pas vers l'infini.

## La gestion du temps

La gestion du temps est importante en JavaScript. Elle permet de temporiser un code et donc de créer des animations, des compteurs à rebours etc…

**Le système de datation**

L'objet Date fournit un grand nombre de méthodes pour lire ou écrire une date.

Le constructeur de Date prend en paramètre de nombreux arguments et s'utilise de différentes manières. Voici les quatre manières de l'utiliser :

new Date();

new Date(timestamp);

new Date(dateString);

new Date(année, mois, jour [, heure, minutes, secondes, millisecondes ]);

À chaque instanciation de l'objet Date, ce dernier enregistre soit la date actuelle si aucun paramètre n'est spécifié, soit une date que nous avons choisie.

Détaillons l'utilisation de notre constructeur :

* La première ligne instancie un objet Date dont la date est fixée à celle de l'instant même de l'instanciation.
* La deuxième ligne permet de spécifier le timestamp à utiliser pour notre instanciation.
* La troisième ligne prend en paramètre une chaîne de caractères qui doit être interprétable par la méthode parse().
* Enfin, la dernière ligne permet de spécifier manuellement chaque composant qui constitue une date. Nous retrouvons en paramètres obligatoires l'année, le mois et le jour. Les quatre derniers paramètres sont facultatifs (c'est pour cela qu’il y a des crochets, ils signifient que les paramètres sont facultatifs). Ils seront initialisés à 0 s'ils ne sont pas spécifiés.

L'objet Date possède une méthode statique appelée parse(). Celle-ci prend en unique paramètre une chaîne de caractères représentant une date et renvoie le timestamp associé. Cependant, il faut respecter une certaine syntaxe qui se présente de la manière suivante :

Sat, 04 May 1991 20:00:00 GMT+02:0. Ainsi nous aurons :

var timestamp = Date.parse('Sat, 04 May 1991 20:00:00 GMT+02:00');

alert(timestamp); // Affiche : 673380000000

**Les méthodes des instances de l'objet Date**

* getFullYear() : renvoie l'année sur 4 chiffres.
* getMonth() : renvoie le mois (0 à 11).
* getDate() : renvoie le jour du mois (1 à 31).
* getDay() : renvoie le jour de la semaine (0 à 6, la semaine commence le dimanche).
* getHours() : renvoie l'heure (0 à 23).
* getMinutes() : renvoie les minutes (0 à 59).
* getSeconds() : renvoie les secondes (0 à 59).
* getMilliseconds() : renvoie les millisecondes (0 à 999).

Ces huit méthodes possèdent chacune une fonction homologue de type « set ».

Chacune de ces méthodes s'utilise après instanciation d’un objet Date :

var myDate = new Date('Sat, 04 May 1991 20:00:00 GMT+02:00');

alert(myDate.getMonth()); // Affiche : 4

alert(myDate.getDay()); // Affiche : 6

Deux autres méthodes pourront aussi être utiles :

* La méthode getTime() renvoie le timestamp de la date de notre objet.
* La méthode setTime()permet de modifier la date de notre objet en lui passant en unique paramètre un timestamp.

**Mise en pratique : calculer le temps d'exécution d'un script**

Si nous avons une fonction slow() que nous soupçonnons d'être assez lente, nous pouvons vérifier sa vitesse d'exécution. Pour cela, nous allons récupérer le timestamp avant l'exécution de la fonction puis après son exécution :

var firstTimestamp = new Date().getTime(); // On obtient le timestamp avant l'exécution

slow(); // La fonction travaille…

var secondTimestamp = new Date().getTime(), // On récupère le timestamp après l'exécution

result = secondTimestamp - firstTimestamp; // On fait la soustraction

alert("Le temps d'exécution est de : " + result + " millisecondes.");

**Les fonctions temporelles**

Les fonctions setTimeout() et setInterval permettent respectivement de déclencher un code au bout d'un temps donné et de déclencher un code à un intervalle régulier que nous aurons spécifié.

Ces deux fonctions ont exactement les mêmes paramètres : le premier est la fonction à exécuter, le deuxième est le temps en millisecondes.

**Avec un simple appel de fonction**

Concernant le premier paramètre, il y a deux manières de le spécifier :

* En passant la fonction en référence : setTimeout(myFunction, 2000); // myFunction sera exécutée au bout de 2 secondes.
* En définissant une fonction anonyme :

setTimeout(function() {

// Code…

}, 2000);

En ce qui concerne le deuxième paramètre, il s'agit du temps à spécifier (en millisecondes).

**Avec une fonction nécessitant des paramètres**

Nos deux fonctions temporelles possèdent toutes les deux deux paramètres spécifiques, mais il est possible d'en attribuer autant que l'on veut. Les paramètres supplémentaires seront alors passés à la fonction appelée par la fonction temporelle : setTimeout(myFunction, 2000, param1, param2);. Ainsi, au terme du temps passé en deuxième paramètre, notre fonction myFunction() sera appelée de la manière suivante : myFunction(param1, param2);.

Cependant, cette technique ne fonctionne pas sur les versions d'Internet Explorer antérieures à la 10, il faut donc ruser :

setTimeout(function() {

myFunction(param1, param2);

}, 2000);

**Annuler une action temporelle**

Il se peut que nous ayons parfois besoin d'annuler une action temporelle. Par exemple, nous avons utilisé la fonction setTimeout() pour qu'elle déclenche une alerte si l'utilisateur n'a pas cliqué sur une image dans les dix secondes qui suivent. Si l'utilisateur clique sur l'image il va alors falloir annuler notre action temporelle avant son déclenchement. C'est là qu'entrent en jeu les fonctions clearTimeout() et clearInterval(). La première s'utilise pour la fonction setTimeout() et la deuxième pour setInterval().

Ces deux fonctions prennent toutes les deux un seul argument : l'identifiant de l'action temporelle à annuler. Cet identifiant (qui est un simple nombre entier) est retourné par les fonctions setTimeout() et setInterval() :

<button id="myButton">Annuler le compte à rebours</button>

<script>

(function() {

var button = document.getElementById('myButton');

var timerID = setTimeout(function() { // On crée notre compte à rebours

alert("Vous n'êtes pas très réactif vous !");

}, 5000);

button.addEventListener('click', function() {

clearTimeout(timerID); // Le compte à rebours est annulé

alert("Le compte à rebours a bien été annulé."); // Et on prévient l'utilisateur

});

})();

</script>

**Mise en pratique : les animations !**

Une animation, c'est la modification progressive de l'état d'un objet. Prenons l'exemple de la transparence. On veut que notre image passe d'une opacité de 1 à 0,2. Nous voulons que cela soit fait de manière progressive afin de générer une animation. Pour cela, nous devons faire :

var myImg = document.getElementById('myImg');

for (var i = 0.9; i >= 0.2; i -= 0.1) {

myImg.style.opacity = i;

}

Tout ceci se passe en une fraction de seconde. C'est là que les actions temporelles vont entrer en action et afin de temporiser notre code et de lui laisser le temps d'afficher la progression à l'utilisateur. Dans notre cas, nous allons utiliser la fonction setTimeout() :

var myImg = document.getElementById('myImg');

function anim() {

var s = myImg.style,

result = s.opacity = parseFloat(s.opacity) - 0.1;

if (result > 0.2) {

setTimeout(anim, 50); // La fonction anim() fait appel à elle-même si elle n'a pas terminé son travail

}

}

anim(); // Et on lance la première phase de l'animation

Nous avons utilisé la fonction setTimeout() au lieu de setInterval() pour réaliser notre animation. setTimeout() est en fait bien plus « stable » que setInterval(), ce qui fait que nous obtiendrons des animations bien plus fluides. Dans l'ensemble, mieux vaut se passer de setInterval() et utiliser setTimeout() en boucle, quel que soit le cas d'application.

## Les tableaux

L'objet Array est à la base de tout tableau. Il possède toutes les méthodes et les propriétés nécessaires à leur utilisation et à leur modification. Cet objet ne concerne que les tableaux itératifs, les objets littéraux ne sont pas des tableaux, ce sont des objets.

**Le constructeur**

Cet objet peut être instancié de trois manières différentes. Cependant, l'utilisation de son type primitif est préférable à l'instanciation de son objet :

* Instanciation sans arguments : var myArray = new Array();.
* Instanciation en spécifiant chaque valeur à attribuer : var myArray = new Array('valeur1', 'valeur2', …, 'valeurX');. Ce code revient à créer un tableau de cette manière : var myArray = ['valeur1', 'valeur2', …, 'valeurX'];.
* Instanciation en spécifiant la longueur du tableau : var myArray = new Array(longueur\_du\_tableau);.

**Les propriétés**

Les tableaux ne possèdent qu'une seule propriété (accessible uniquement après instanciation) : length. Cette propriété est en lecture seule et indique combien d'éléments existent dans notre tableau.

**Les méthodes**

Le JavaScript ne permet pas l'utilisation de l'opérateur + pour concaténer plusieurs tableaux entre eux. Si on tente de s'en servir, on obtient en sortie une chaîne de caractères contenant tous les éléments des tableaux. Les tableaux possèdent une méthode nommée concat() qui permet d'obtenir le résultat souhaité : var myArray = ['test1', 'test2'].concat(['test3', 'test4']);.

**Parcourir un tableau**

La méthode forEach() prend pour paramètre deux arguments. Le premier reçoit la fonction à exécuter pour chaque index existant et le deuxième (qui est facultatif) reçoit un objet qui sera pointé par le mot-clé this dans la fonction que vous spécifiée pour le premier argument. La méthode forEach() n'est pas supportée par les versions d'Internet Explorer antérieures à la 9.

La fonction passée en paramètre sera exécutée pour chaque index existant (dans l'ordre) et recevra en paramètres trois arguments :

* Le premier contient la valeur contenue à l'index actuel.
* Le deuxième contient l'index actuel.
* Le troisième est une référence au tableau actuellement parcouru.

var myArray = ["C'est", "un", "test"];

myArray.forEach(function(value, index, array) {

alert(

'Index : ' + index + '\n' +

'Valeur : ' + value

);

});

Cette méthode ne fonctionne qu'avec des tableaux, elle n'existe pas pour les collections d'éléments retournées par le DOM. Ainsi cela ne fonctionne pas sur la propriété classList ou encore les tableaux retournés par les méthodes de sélection telles querySelectorAll()  ou getElementsByTagName().

**Rechercher un élément dans un tableau**

Les tableaux possèdent aussi les fonctions indexOf() et lastIndexOf().Elles fonctionnent de la même manière que pour les chaines de caractères, sauf qu'au lieu de ne chercher qu'une chaîne de caractères nous pouvons faire une recherche pour n'importe quel type de valeur, que ce soit une chaîne de caractères, un nombre ou un objet. La valeur retournée par la fonction est l'index du tableau dans lequel se trouve l’élément recherché. En cas d'échec la fonction retourne toujours la valeur -1.

Ces deux fonctions possèdent aussi un second paramètre permettant de spécifier à partir de quel index nous souhaitons faire débuter la recherche.

**La méthode reverse()**

Cette méthode ne prend aucun argument en paramètre et ne retourne aucune valeur, son seul rôle est d'inverser l'ordre des valeurs de notre tableau :

var myArray = [1, 2, 3, 4, 5];

myArray.reverse();

alert(myArray); // Affiche : 5,4,3,2,1

**La méthode sort()**

Par défaut cette méthode trie un tableau par ordre alphabétique uniquement. Mais elle possède aussi un argument facultatif permettant de spécifier l'ordre à définir. Prenons un exemple :

var myArray = [3, 1, 5, 10, 4, 2];

myArray.sort();

alert(myArray); // Affiche : 1,10,2,3,4,5

Cette méthode commence par convertir toutes les données du tableau en chaînes de caractères et ce n'est qu'après ça qu'elle applique son tri alphabétique. Ainsi dans l’exemple précédent, si nous essayons de remplacer nos chiffres par des caractères cela devient plus logique : 0 = a ; 1 = b ; 2 = c.

L'argument facultatif de sort() a pour but de réaliser un tri personnalisé. Il doit contenir une référence vers une fonction que nous avons créée, cette dernière devant posséder deux arguments qui seront spécifiés par la méthode sort(). La fonction devra alors dire si les valeurs transmises en paramètres sont de même valeur, ou bien si l'une des deux est supérieure à l'autre. La méthode sort() ne convertit pas les données du tableau en chaînes de caractères lorsque nous avons défini l'argument facultatif, ce qui fait que les valeurs que nous recevrons de l’exemple précédent seront bien de type Number et non pas de type String.

Pour lui indiquer à la méthode sort() qu'une valeur est inférieure, supérieure ou égale à l'autre, il faut lui renvoyer :

* -1 lorsque a est inférieur à b.
* 1 lorsque a est supérieur à b.
* 0 quand les valeurs sont égales.

Pour trier notre tableau de chiffres, nous devons donc écrire :

var myArray = [3, 1, 5, 10, 4, 2];

myArray.sort(function(a, b) {

if (a < b) {

return -1;

} else if (a > b) {

return 1;

} else {

return 0;

}

});

alert(myArray); // Affiche : 1,2,3,4,5,10

**Extraire une partie d'un tableau**

La méthode slice() est faite pour extraire une partie d'un tableau. Elle prend en paramètre deux arguments, dont le deuxième est facultatif. Le premier est l'index (inclus) à partir duquel nous souhaitons commencer l'extraction du tableau, le deuxième est l'index (exclu) auquel l'extraction doit se terminer. S'il n'est pas spécifié, l'extraction continue jusqu'à la fin du tableau.

var myArray = [1, 2, 3, 4, 5];

alert(myArray.slice(1, 3)); // Affiche : 2,3

alert(myArray.slice(2)); // Affiche : 3,4,5

**Remplacer une partie d'un tableau**

splice() reçoit deux arguments obligatoires, puis une infinité d'arguments facultatifs. Le premier argument est l'index à partir duquel nous souhaitons effectuer nos opérations, le deuxième est le nombre d'éléments que nous souhaitons supprimer à partir de cet index :

var myArray = [1, 2, 3, 4, 5];

var result = myArray.splice(1, 2); // On retire 2 éléments à partir de l'index 1

alert(myArray); // Affiche : 1,4,5

alert(result); // Affiche : 2,3

Les arguments qui suivent les deux premiers contiennent les éléments qui doivent être ajoutés en remplacement de ceux effacés :

var myArray = [1, null, 4, 5];

myArray.splice(1, 1, 2, 3);

alert(myArray); // Affiche : 1,2,3,4,5

Si nous ajoutons des éléments dans le tableau, nous pouvons mettre le deuxième argument à 0, ce qui aura pour effet d'ajouter des éléments sans être obligé d'en supprimer d'autres. Cette méthode splice() peut donc être utilisée comme une méthode d'insertion de données.

**Tester l'existence d'un tableau**

Les tableaux possèdent une méthode propre à l'objet constructeur nommée isArray(). Elle permet de tester si la variable passée en paramètre contient un tableau : alert(Array.isArray(['test']));.

**Les piles et les files**

Les piles et les files sont deux manières de manipuler nos tableaux. Ceux-ci peuvent être vus comme une pile de livres où le dernier posé sera au final le premier récupéré, ou bien comme une file d'attente, où le dernier entré sera le dernier sorti.

**Retour sur les méthodes étudiées**

Quatre méthodes ont été étudiées au cours des premiers chapitres de ce cours :

* push() : ajoute un ou plusieurs éléments à la fin du tableau (un argument par élément ajouté) et retourne la nouvelle taille de ce dernier.
* pop() : retire et retourne le dernier élément d'un tableau.
* unshift() : ajoute un ou plusieurs éléments au début du tableau (un argument par élément ajouté) et retourne la nouvelle taille de ce dernier.
* shift() : retire et retourne le premier élément d'un tableau.

**Les piles**

Les piles partent du principe que le premier élément ajouté sera le dernier retiré, comme une pile de livres. Elles sont utilisables de deux manières différentes : soit avec les deux méthodes push() et pop(), soit avec les deux restantes unshift() et shift(). Dans le premier cas, la pile sera empilée et dépilée à la fin du tableau, dans le deuxième cas, les opérations se feront au début du tableau.

**Les files**

Le premier élément ajouté est le premier sorti, comme une file d'attente. Elles sont, elles aussi, utilisables de deux manières différentes : soit avec le couple push()/shift(), soit avec le couple unshift()/pop().

En JavaScript, les files sont bien moins utilisées que les piles, car elles sont dépendantes des méthodes unshift() et shift(), qui souffrent d'un manque de performance.

**Quand les performances sont absentes : unshift() et shift()**

Les deux méthodes unshift() et shift() utilisent chacune un algorithme qui fait qu'en retirant ou en ajoutant un élément en début de tableau, elles vont devoir réécrire tous les index des éléments qui suivent.

## Les images

L’objet Image permet de faire des manipulations assez sommaires sur une image de savoir si elle a été entièrement téléchargée.

**Le constructeur**

Le constructeur de l'objet Image ne prend aucun argument en paramètre : var myImg = new Image();.

**Propriétés**

|  |  |
| --- | --- |
| **Nom de la propriété** | **Contient** |
| width | Contient la largeur originale de l'image. Nous pouvons redéfinir cette propriété pour modifier la taille de l'image. |
| height | Contient la hauteur originale de l'image. Nous pouvons redéfinir cette propriété pour modifier la taille de l'image. |
| src | Cette propriété sert à spécifier l'adresse (absolue ou relative) de l'image. Une fois que cette propriété est spécifiée, l'image commence immédiatement à être chargée. |

**Événements**

L'objet Image ne possède qu'un seul événement nommé load. Il est très utile, notamment lorsque l'on souhaite créer un script de type Lightbox, car il permet de savoir quand une image est chargée. Son utilisation se fait comme tout événement :

var myImg = new Image();

myImg.src = 'adresse\_de\_mon\_image';

myImg.addEventListener('load', function() {

// Etc.

});

Dans ce cas, notre événement pourrait ne jamais se déclencher. Nous avons spécifié l'adresse de notre image avant même d'avoir spécifié notre événement, ce qui fait que si l'image a été trop rapidement chargée, l'événement load se sera déclenché avant même que nous n'ayons eu le temps de le modifier. Il suffit de spécifier l'adresse de notre image après avoir modifié notre événement :

var myImg = new Image();

myImg.addEventListener('load', function() { // Étape 1 : on modifie notre événement

// Etc.

});

myImg.src = 'adresse\_de\_mon\_image'; // Étape 2 : on spécifie l'adresse de notre image

**Particularités**

L'objet Image est un peu spécial dans le sens où nous pouvons l'ajouter à notre arbre DOM comme nous le ferions avec la valeur retournée par la méthode document.createElement(). Ce comportement est spécial et ne peut se révéler utile que dans de très rares cas d'application :

var myImg = new Image();

myImg.src = 'adresse\_de\_mon\_image';

document.body.appendChild(myImg); // L'image est ajoutée au DOM

**Mise en pratique**

Nous allons voir ici le prince d’une Lightbox. Le principe d'une Lightbox est de permettre l'affichage d'une image en taille réelle directement dans la page Web où se trouvent les miniatures de toutes nos images.

Pour cela il nous faut tout d'abord mettre en place un overlay. En développement Web, il s'agit généralement d'une surcouche sur la page Web, permettant de différencier deux couches de contenu. Pour l'overlay, nous allons avoir besoin d'une balise supplémentaire dans notre code HTML : <div id="overlay"></div>.

Puis nous lui donnons un style CSS afin qu'il puisse couvrir toute la page Web :

#overlay {

display : none; /\* Par défaut, on cache l'overlay \*/

position: absolute;

top: 0; left: 0;

width: 100%; height: 100%;

text-align: center; /\* Pour centrer l'image que l'overlay contiendra \*/

/\* Ci-dessous, nous appliquons un background de couleur noire et d'opacité 0.6. Il s'agit d'une propriété CSS3. \*/

background-color: rgba(0,0,0,0.6);

}

Maintenant, le principe va être d'afficher l'overlay quand on cliquera sur une miniature. Il faudra mettre un message pour faire patienter le visiteur pendant le chargement de l'image. Une fois l'image chargée, il ne restera plus qu'à supprimer le texte et à ajouter l'image originale à la place :

function displayImg(link) {

var img = new Image(),

overlay = document.getElementById('overlay');

img.addEventListener('load', function() {

overlay.innerHTML = '';

overlay.appendChild(img);

});

img.src = link.href;

overlay.style.display = 'block';

overlay.innerHTML = '<span>Chargement en cours...</span>';

}

Nous allons maintenant permettre de fermer l'overlay pour choisir une autre image. La solution est simple, il suffit de quitter l'overlay lorsque l'on clique quelque part dessus :

document.getElementById('overlay').addEventListener('click', function(e) {

// currentTarget est utilisé pour cibler l'overlay et non l'image

e.currentTarget.style.display = 'none';

});

Reste enfin le code html des images miniatures :

<p>

<a href="imgs/1.jpg" title="Afficher l'image originale"><img src="imgs/1\_min.jpg" alt="Miniature" /></a>

<a href="imgs/2.jpg" title="Afficher l'image originale"><img src="imgs/2\_min.jpg" alt="Miniature" /></a>

</p>

Pour finir, nous devons parcourir les liens et bloquer leurs redirections :

var links = document.getElementsByTagName('a'),

linksLen = links.length;

for (var i = 0; i < linksLen; i++) {

links[i].addEventListener('click', function(e) {

e.preventDefault(); // On bloque la redirection

// On appelle notre fonction pour afficher les images

// currentTarget est utilisé pour cibler le lien et non l'image

displayImg(e.currentTarget);

});

}

Pour finir, nous noterons que ce script est actuellement inutilisable sur un site en production. Cependant, si nous souhaitons l’améliorer afin de le publier, il est conseillé d'étudier ces quelques points :

* Éviter d'appliquer l'événement click à tous les liens de la page. Ajouter un élément différenciateur aux liens de la Lightbox, tel qu'une classe ou un attribut name.
* Redimensionner dynamiquement les images originales afin d'éviter qu'elles ne débordent de la page. Utiliser soit un redimensionnement fixe (très simple à faire), soit un redimensionnement variant selon la taille de l'écran.
* Implémenter l'utilisation des touches fléchées : flèche droite pour l'image suivante, flèche gauche pour la précédente.

## Les polyfills et les wrappers

**Introduction aux polyfills**

Certaines technologies récentes sont plus ou moins bien supportées par certains navigateurs, voire même pas du tout. Pour réaliser nos projets, il nous faut utiliser des conditions permettant de tester si le navigateur actuel supporte telle ou telle technologie. Dans le cas contraire il nous faut alors déployer des solutions dont certaines sont peu pratiques.

Il existe un moyen de se faciliter plus ou moins la tâche, cela s'appelle les **polyfills**. Un polyfill est un script qui a pour but de fournir une technologie à tous les navigateurs existants. Une fois implémenté dans notre code, un polyfill a deux manières de réagir :

* Le navigateur est récent et supporte la technologie souhaitée, le polyfill ne va alors strictement rien faire et va nous laisser utiliser cette technologie comme elle devrait l'être nativement.
* Le navigateur est trop vieux et ne supporte pas la technologie souhaitée, le polyfill va alors « imiter » cette technologie grâce à diverses astuces et nous permettra de l'utiliser comme si elle était disponible nativement.

Par exemple, le script suivant marchera avec un navigateur récent comme sur un vieux navigateur ne supportant pas la méthode isArray() :

if (!Array.isArray) { // Si isArray() n'existe pas, alors on crée notre méthode alternative :

Array.isArray = function(element) {

return Object.prototype.toString.call(element) == '[object Array]';

};

}

alert(Array.isArray([])); // Affiche : « true »

alert(Array.isArray({})); // Affiche : « false »

La méthode isArray() fonctionne maintenant sur tous les navigateurs. Pas besoin de vérifier à chaque fois si elle existe, il suffit juste de s'en servir comme à notre habitude et notre polyfill s'occupera de tout.

La plupart d'entre nous n'aura pratiquement jamais à réaliser nos propres polyfills, car ils sont déjà nombreux à avoir été créés par d'autres développeurs JavaScript. Pour trouver des polyfills adaptés, il suffira d’effectuer une recherche sur le net.

**Introduction aux wrappers**

Il se peut que, par moments, nous ayons besoin de créer une méthode supplémentaire pour certains objets natifs du JavaScript. Nous avons vu que cela fonctionnait bien avec les objets natifs génériques (String, Date), mais moins bien avec les objets natifs liés au DOM. De plus, la modification d'un objet natif est déconseillée car cela risque de modifier une méthode déjà existante.

La solution apportée est nommée « **wrapper** ». Un wrapper est un code qui a pour but d'encadrer l'utilisation de certains éléments du JavaScript. Il peut ainsi contrôler la manière dont ils sont employés et peut réagir en conséquence pour fournir des fonctionnalités supplémentaires aux développeurs. Il s'agit en fait d'une surcouche par laquelle nous allons passer pour pouvoir contrôler nos éléments. Dans l'idéal, un wrapper doit permettre au développeur de se passer de l'élément original, ainsi le travail ne s'effectuera que par le biais de la surcouche que constitue le wrapper.

Pour l’exemple, nous allons tenter d’effectuer un wrapper sur la propriété complete de l’objet Image. Celle-ci ayant un comportement hasardeux, nous allons tenter de permettre son support. Puisque notre wrapper doit servir de surcouche de A à Z, celui-ci va se présenter sous forme d'objet qui sera instancié à la place de l'objet Image. Notre but étant de permettre le support de la propriété complete, nous allons devoir créer par défaut un événement load qui se chargera de modifier la propriété complete lors de son exécution. Il nous faut aussi assurer le support de l'événement load pour les développeurs :

function Img() {

var obj = this; // Nous faisons une petite référence vers notre objet Img. Cela nous facilitera la tâche.

this.originalImg = new Image(); // On instancie l'objet original, le wrapper servira alors d'intermédiaire

this.complete = false;

this.onload = function() {}; // Voici l'événement que les développeurs pourront modifier

this.originalImg.onload = function() {

obj.complete = true; // L'image est chargée !

obj.onload(); // On exécute l'événement éventuellement spécifié par le développeur

};

}

Actuellement, notre wrapper fait ce qu'on voulait qu'il fasse : assurer un support de la propriété complete. Cependant, il nous est actuellement impossible de spécifier les propriétés standards de l'objet original sans passer par notre propriété originalImg, or ce n'est pas ce que l'on souhaite car le développeur pourrait compromettre le fonctionnement de notre wrapper, par exemple en modifiant la propriété onload de l'objet original. Il va donc nous falloir créer une méthode permettant l'accès à ces propriétés sans passer par l'objet original. Nous allons pour cela ajouter deux méthodes set() et get() assurant le support des propriétés d'origine. Afin que notre wrapper soit une surcouche « transparent », nous allons faire passer les modifications/lectures des propriétés par les méthodes set() et get() dans tous les cas, y compris lorsqu'il s'agit de propriétés appartenant au wrapper.

Img.prototype.set = function(name, value) {

var allowed = ['width', 'height', 'src'], // On spécifie les propriétés dont on autorise la modification

wrapperProperties = ['complete', 'onload'];

if (allowed.indexOf(name) != -1) {

this.originalImg[name] = value; // Si la propriété est autorisée alors on la modifie

} else if (wrapperProperties.indexOf(name) != -1) {

this[name] = value; // Ici, la propriété appartient au wrapper et non pas à l'objet original

}

};

Img.prototype.get = function(name) {

// Si la propriété n'existe pas sur le wrapper, on essaye alors sur l'objet original :

return typeof this[name] != 'undefined' ? this[name] : this.originalImg[name];

};

Il nous reste maintenant une dernière chose à mettre en place qui peut se révéler pratique : pouvoir spécifier l'adresse de l'image dès l'instanciation de l'objet. On modifie donc notre wrapper de la manière suivante :

function Img(src) { // On ajoute un paramètre « src »

var obj = this; // Nous faisons une petite référence vers notre objet Img. Cela nous facilitera la tâche.

this.originalImg = new Image(); // On instancie l'objet original, le wrapper servira alors d'intermédiaire

this.complete = false;

this.onload = function() {}; // Voici l'événement que les développeurs pourront modifier

this.originalImg.onload = function() {

obj.complete = true; // L'image est chargée !

obj.onload(); // On exécute l'événement éventuellement spécifié par le développeur

};

if (src) {

this.originalImg.src = src; // Si elle est spécifiée, on défini alors la propriété src

}

}

Il suffit d’ajouter dans ce wrapper les fonctions get() et set() écrites précédemment pour le rendre fonctionnel.

## Les closures

Commençons par étudier un peu plus en profondeur de quelle manière sont gérées les variables par le JavaScript.

function area() {

var myVar = 1;

function show() {

alert(myVar);

}

setTimeout(show, 1000);

}

area();

Ce code fonctionnera même si la méthode show() est exécutée une seconde après la méthode area(). Donc la méthode show() a été capable d’accéder à la variable myVar même après la disparition de l’espace où elle a été définie. Ceci est dû au fait que le JavaScript « passe une variable par référence ». Ainsi, une variable peut posséder plusieurs références. Dans notre fonction area(), nous avons une première référence vers notre variable, car elle y est déclarée sous le nom myVar. Dans la fonction show(), nous avons une deuxième référence du même nom, myVar.

Quand une fonction termine son exécution, la référence vers la variable est détruite, rendant son accès impossible. C’est ce qui se produit avec notre fonction area(). La variable en elle-même continue à exister tant qu'il reste encore une référence qui est susceptible d'être utilisée. C'est aussi ce qui se produit avec la fonction show(). Puisque celle-ci possède une référence vers notre variable, cette dernière n'est pas détruite.

**Comprendre le problème**

Les problèmes que les closures sont supposées résoudre ne sont pas simples à comprendre. Prenons l’exemple suivant :

var number = 1;

setTimeout(function() {

alert(number);

}, 100);

number++;

La fonction alert() affichera la valeur 2, alors que nous avons fait appel à setTimeout() avant le changement de valeur. Cela vient du fait que ce n'est que la fonction setTimeout() qui a été exécutée avant le changement de valeur. La fonction anonyme, elle, n'est exécutée que 100 millisecondes après l'exécution de setTimeout(), laisse le temps à la valeur de number de changer.

Prenons un cas concret en exemple. Le code suivant serait sensé ajouter 10 balises <div> à notre page :

var divs = document.getElementsByTagName('div'),

divsLen = divs.length;

for (var i = 0; i < divsLen; i++) {

setTimeout(function() {

divs[i].style.display = 'block';

}, 200 \* i); // Le temps augmentera de 200 ms à chaque élément

}

La console d'erreurs, signalera que la variable divs[i] est indéfinie, et ce dix fois de suite, ce qui correspond à nos dix itérations de boucle. Si nous regardons d'un peu plus près le problème, nous constatons alors que la variable i vaut toujours 10 à chaque fois qu'elle est utilisée dans les fonctions anonymes, ce qui correspond à sa valeur finale une fois que la boucle a terminé son exécution. Notre fonction anonyme ne prend en compte que la valeur finale de notre variable. Heureusement les closures peuvent contourner ce désagrément.

**Explorer les solutions**

Une **closure**, en JavaScript, est une fonction ayant pour but de capter des données susceptibles de changer au cours du temps, de les enregistrer dans son espace fonctionnel et de les fournir en cas de besoin.

Pour que l’exemple précédent fonctionne il faut que nous créions notre variable currentI déclarée en dehors de l'espace global de notre code. Il suffirait de le faire dans une IIFE, rendant impossible sa réécriture depuis l'extérieur. Cette variable étant inaccessible depuis l’extérieure, il faudra utiliser la méthode setTimeout()dans la fonction contenant la variable :

var divs = document.getElementsByTagName('div'),

divsLen = divs.length;

for (var i = 0; i < divsLen; i++) {

(function() {

var currentI = i;

setTimeout(function() {

divs[currentI].style.display = 'block';

}, 200 \* i);

})();

}

Il est fréquent de tomber sur des closures écrites de cette manière :

var divs = document.getElementsByTagName('div'),

divsLen = divs.length;

for (var i = 0; i < divsLen; i++) {

(function(currentI) {

setTimeout(function() {

divs[currentI].style.display = 'block';

}, 200 \* i);

})(i);

}

Ici, nous avons tout créé un argument currentI pour notre IIFE et nous lui passons en paramètre la valeur de i. Cette modification fait gagner un peu d'espace et permet de mieux organiser le code, on distingue plus facilement ce qui constitue la closure ou non.

Nous pouvons également remplacer le nom de currentI par i. Cela est bien plus pratique à gérer et prête moins à confusion pour peu que l'on ait compris que dans la closure nous utilisons une variable i différente de celle située en-dehors de la closure.

**Une autre utilité, les variables statiques**

Les variables statiques sont déclarées à la première exécution de la fonction, mais ne sont pas supprimées à la fin des exécutions. Elles sont conservées pour les prochaines utilisations de la fonction.

C'est comme si nous déclarions une variable globale en JavaScript et que nous l'utilisions dans notre fonction : la variable et sa valeur ne seront jamais détruites. En revanche, la variable globale est accessible par toutes les fonctions, tandis qu'une variable statique n'est accessible que pour la fonction qui a fait sa déclaration (il est à noter que le mot-clé static est réservé par le JavaScript, mais qu’il ne sert à rien).

Pour mimer le comportement des variables static, nous pouvons utiliser les closures. Pour accéder à notre fonction anonyme, il suffira de la retourner avec le mot-clé return et passer sa référence à une variable. Voici une fonction avec une variable statique nommée myVar :

var display = (function() {

var myVar = 0; // Déclaration de la variable pseudo-statique

return function(value) {

if (typeof value != 'undefined') {

myVar = value;

}

alert(myVar);

};

})();

display(); // Affiche : 0

display(42); // Affiche : 42

display(); // Affiche : 42

## Qu'est-ce que l'AJAX ?

**AJAX** est l'acronyme d'Asynchronous JavaScript and XML. Derrière ce nom se cache un ensemble de technologies destinées à réaliser de rapides mises à jour du contenu d'une page Web, sans qu'elles nécessitent le moindre rechargement visible par l'utilisateur de la page Web. Les technologies employées sont diverses et dépendent du type de requêtes que l'on souhaite utiliser, mais d'une manière générale le JavaScript est constamment présent. Le transfert de données est géré exclusivement par le JavaScript, et utilise certaines technologies de formatage de données, comme le XML ou le JSON.

**Les formats de données**

L'AJAX est un ensemble de technologies visant à effectuer des transferts de données. Dans ce cas, il faut savoir structurer nos données. Il existe de nombreux formats pour transférer des données, nous verrons ici les quatre principaux :

* Le format texte (le plus simple) : il ne possède aucune structure prédéfinie. Il sert essentiellement à transmettre une phrase à afficher à l'utilisateur, comme un message d'erreur ou autre.
* Le HTML : c’est aussi une manière de transférer facilement des données. Généralement, il a pour but d'acheminer des données qui sont déjà formatées par le serveur puis affichées directement dans la page sans aucun traitement préalable de la part du JavaScript.
* Le XML : il permet de stocker les données dans un langage de balisage semblable au HTML. Il est très pratique pour stocker de nombreuses données ayant besoin d'être formatées, tout en fournissant un moyen simple d'y accéder.
* Le plus courant est le JSON : acronyme de JavaScript Object Notation. Il a pour particularité de segmenter les données dans un objet JavaScript. Il est très avantageux pour de petits transferts de données segmentées et est de plus en plus utilisé dans de très nombreux langages.

**Les formats classiques**

Nous désignons ici le texte et le HTML. Ces deux formats n'ont rien de bien particulier, nous récupérons leur contenu et l'affichons là où il faut. Ils ne nécessitent aucun traitement.

**Le XML**

Le XML permet de structurer des données de la même manière qu'en HTML, mais avec des balises personnalisées. Là où l'utilisation du XML est intéressante, c'est que, en utilisant la requête appropriée, nous pouvons parcourir ce code XML avec les mêmes méthodes que celle qui nous utilisons pour le DOM HTML, comme getElementsByTagName() par exemple. Cela est dû au fait que suite à notre requête, le JavaScript va recevoir une chaîne de caractères contenant notre code XML. Une fois la requête terminée et toutes les données reçues, un parseur (ou analyseur syntaxique) va se mettre en route pour analyser le code reçu, le décomposer, et enfin le reconstituer sous forme d'arbre DOM qu'il sera possible de parcourir.

**Le JSON**

Le JSON est le format le plus utilisé et le plus pratique pour nous. Il s'agit d'une représentation des données sous forme d'objet JavaScript. Essayons, par exemple, de représenter une liste de membres ainsi que leurs informations :

{

Membre1: {

posts: 6230,

inscription: '22/08/2003'

},

Membre2: {

posts: 200,

inscription: '04/06/2011'

}

}

Tout comme avec le XML, nous recevrons ce code sous forme de chaîne de caractères. Cependant, le parseur ne se déclenche pas automatiquement pour ce format. Il faut utiliser l'objet nommé JSON, qui possède deux méthodes très pratiques :

* parse() : prend en paramètre la chaîne de caractères à analyser et retourne le résultat sous forme d'objet JSON.
* stringify() : permet de faire l'inverse. Elle prend en paramètre un objet JSON et retourne son équivalent sous forme de chaîne de caractères.

Voici un exemple d'utilisation de ces deux méthodes :

var obj = {

index: 'contenu'

},

string;

string = JSON.stringify(obj);

alert(typeof string + ' : ' + string); // Affiche : « string : {"index":"contenu"} »

obj = JSON.parse(string);

alert(typeof obj + ' : ' + obj); // Affiche : « object : [object Object] »

## XMLHttpRequest

Nous allons maintenant mettre le principe de l'AJAX en pratique avec l'objet XMLHttpRequest. Cette technique AJAX est la plus courante et est définitivement incontournable.

Le principe de l’objet XMLHttpRequest est classique : une requête HTTP est envoyée à l'adresse spécifiée, une réponse est alors attendue en retour de la part du serveur. Une fois la réponse obtenue, la requête s'arrête et peut éventuellement être relancée. L'objet que nous allons étudier dans ce chapitre possède deux versions majeures. La première version est celle issue de la standardisation de l'objet d'origine et son support est assuré par tous les navigateurs. La deuxième version est supportée par la plupart des navigateurs, sauf IE version 9 et antérieures.

**Première version : les bases**

L'utilisation de l'objet XHR se fait en deux étapes bien distinctes :

* Préparation et envoi de la requête.
* Réception des données.

**Préparation et envoi de la requête**

Pour commencer à préparer notre requête, il nous faut tout d'abord instancier un objet XHR :

var xhr = new XMLHttpRequest();

La préparation de la requête se fait par le biais de la méthode open(), qui prend en paramètres cinq arguments différents, dont trois facultatifs :

* Le premier argument contient la méthode d'envoi des données, les trois méthodes principales sont GET, POST et HEAD.
* Le deuxième argument est l'URL à laquelle nous souhaitons soumettre notre requête, par exemple : 'http://mon\_site\_web.com'.
* Le troisième argument est un booléen facultatif dont la valeur par défaut est true. À true, la requête sera de type asynchrone, à false elle sera synchrone (la différence est expliquée plus tard).
* Les deux derniers arguments sont à spécifier en cas d'identification nécessaire sur le site Web (à cause d'un .htaccess par exemple). Le premier contient le nom de l'utilisateur, tandis que le deuxième contient le mot de passe.

Voici une utilisation basique et courante de la méthode open() :

xhr.open('GET', 'http://mon\_site\_web.com/ajax.php');

Cette ligne de code prépare une requête afin que cette dernière contacte la page ajax.php sur le nom de domaine mon\_site\_web.com par le biais du protocole http. Tout paramètre spécifié à la requête sera transmis par le biais de la méthode GET. Après préparation de la requête, il ne reste plus qu'à l'envoyer avec la méthode send(). Cette dernière prend en paramètre un argument obligatoire que nous étudierons plus tard. Dans l'immédiat, nous lui spécifions la valeur null : xhr.send(null);.

Si nous travaillons avec des requêtes asynchrones, il existe une méthode abort() permettant de stopper toute activité. La connexion au serveur est alors interrompue et notre instance de l'objet XHR est remise à zéro. Son utilisation est très rare, mais elle peut servir si nous avons des requêtes qui prennent trop de temps.

**Synchrone ou asynchrone ?**

Une requête synchrone va bloquer notre script tant que la réponse n'aura pas été obtenue, tandis qu'une requête asynchrone laissera continuer l'exécution de notre script et nous préviendra de l'obtention de la réponse par le biais d'un événement. La requête asynchrone permet de gérer notre interface pendant que nous attendons la réponse du serveur. Nous pouvons donc indiquer au client de patienter ou nous occuper d'autres tâches en attendant.

**Transmettre des paramètres**

Les méthodes HEAD est une méthode de réception : en spécifiant cette méthode, nous ne recevrons pas le contenu du fichier dont nous avons spécifié l'URL, mais juste son en-tête (son header, d'où le HEAD). Cette utilisation est pratique quand nous souhaitons simplement vérifier, par exemple, l'existence d'un fichier sur un serveur.

Revenons maintenant aux deux autres méthodes qui sont, elles, conçues pour l'envoi de données. Il est possible de transmettre des paramètres par le biais de la méthode GET. La transmission de ces paramètres se fait de la même manière qu'avec une URL classique, il faut les spécifier avec les caractères ? et & dans l'URL que nous passons à la méthode open() :

xhr.open('GET', 'http://mon\_site\_web.com/ajax.php?param1=valeur1&param2=valeur2');

Il est cependant conseillé, quelle que soit la méthode utilisée (GET ou POST), d'encoder toutes les valeurs que passées en paramètre grâce à la fonction encodeURIComponent(), afin d'éviter d'écrire d'éventuels caractères interdits dans une URL :

var value1 = encodeURIComponent(value1),

value2 = encodeURIComponent(value2);

xhr.open('GET', 'http://mon\_site\_web.com/ajax.php?param1=' + value1 + '&param2=' + value2);

En ce qui concerne la méthode POST, les paramètres ne sont pas à spécifier avec la méthode open() mais avec la méthode send() :

xhr.open('POST', 'http://mon\_site\_web.com/ajax.php');

xhr.send('param1=' + value1 + '&param2=' + value2);

Cependant, la méthode POST consiste généralement à envoyer des valeurs contenues dans un formulaire, il faut donc modifier les en-têtes d'envoi des données afin de préciser qu'il s'agit de données provenant d'un formulaire (même si, à la base, ce n'est pas le cas) :

xhr.open('POST', 'http://mon\_site\_web.com/ajax.php');

xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xhr.send('param1=' + value1 + '&param2=' + value2);

La méthode setRequestHeader() permet l'ajout ou la modification d'un en-tête, elle prend en paramètres deux arguments : le premier est l'en-tête concerné et le deuxième est la valeur à lui attribuer.

**Réception des données**

La réception des données d'une requête se fait par le biais de nombreuses propriétés. Cependant, les propriétés à utiliser diffèrent selon que la requête est synchrone ou non.

**Requête asynchrone : spécifier la fonction de callback**

Dans le cas d'une requête asynchrone, il nous faut spécifier une fonction de callback afin de savoir quand la requête s'est terminée. Pour cela, l'objet XHR possède un événement nommé readystatechange auquel il suffit d'attribuer une fonction :

xhr.addEventListener('readystatechange', function() {

// Notre code…

});

Cet événement ne se déclenche pas seulement lorsque la requête est terminée, mais plutôt, comme son nom l'indique, à chaque changement d'état. Il existe cinq états différents représentés par des constantes spécifiques à l'objet XMLHttpRequest :

|  |  |  |
| --- | --- | --- |
| **Constante** | **Valeur** | **Description** |
| UNSENT | 0 | L'objet XHR a été créé, mais pas initialisé (la méthode open() n'a pas encore été appelée). |
| OPENED | 1 | La méthode open() a été appelée, mais la requête n'a pas encore été envoyée par la méthode send(). |
| HEADERS\_RECEIVED | 2 | La méthode send() a été appelée et toutes les informations ont été envoyées au serveur. |
| LOADING | 3 | Le serveur traite les informations et a commencé à renvoyer les données. Tous les en-têtes des fichiers ont été reçus. |
| DONE | 4 | Toutes les données ont été réceptionnées. |

L'utilisation de la propriété readyState est nécessaire pour connaître l'état de la requête. L'état qui nous intéresse est le cinquième (la constante DONE), car nous voulons simplement savoir quand notre requête est terminée. Il existe deux manières pour vérifier que la propriété readyState contient bien une valeur indiquant que la requête est terminée, la première consiste à utiliser la constante elle-même :

xhr.addEventListener('readystatechange', function() {

if (xhr.readyState === XMLHttpRequest.DONE) { // La constante DONE appartient à l'objet XMLHttpRequest, elle n'est pas globale

// Notre code…

}

});

Tandis que la deuxième manière de faire consiste à utiliser directement la valeur de la constante, soit 4 pour la constante DONE :

xhr.addEventListener('readystatechange', function() {

if (xhr.readyState === 4) {

// Notre code…

}

});

De cette manière, notre code ne s'exécutera que lorsque la requête aura terminé son travail. Toutefois, même si la requête a terminé son travail, cela ne veut pas forcément dire qu'elle l'a mené à bien, pour cela nous allons devoir consulter le statut de la requête grâce à la propriété status. Cette dernière renvoie le code correspondant à son statut, comme le fameux 404 pour les fichiers non trouvés. Le statut qui nous intéresse est le 200, qui signifie que tout s'est bien passé :

xhr.addEventListener('readystatechange', function() {

if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {

// Votre code…

}

});

iIl existe aussi une propriété nommée statusText contenant une version au format texte du statut de la requête en anglais. Par exemple, un statut 404 donnera « Not Found ».

Nous avons traité ici le cas d'une requête asynchrone. Pour une requête synchrone il n'y a qu'à vérifier le statut de notre requête.

**Traitement des données**

Une fois la requête terminée, il faut récupérer les données obtenues. Ici, deux possibilités s'offrent à nous :

* Les données sont au format XML, nous pouvons alors utiliser la propriété responseXML qui permet de parcourir l'arbre DOM des données reçues.
* Les données sont dans un format autre que le XML, il faut alors utiliser la propriété responseText, qui fournit toutes les données sous forme d'une chaîne de caractères. C'est à nous qu'incombe la tâche de faire d'éventuelles conversions, par exemple avec un objet JSON :

var response = JSON.parse(xhr.responseText);.

Les deux propriétés nécessaires à l'obtention des données sont responseText et responseXML. Cette dernière est particulière, dans le sens où elle contient un arbre DOM que nous pouvons facilement parcourir. Par exemple, si nous recevons l'arbre DOM suivant :

<?xml version="1.0" encoding="utf-8"?>

<table>

<line>

<cel>Ligne 1 - Colonne 1</cel>

<cel>Ligne 1 - Colonne 2</cel>

</line>

<line>

<cel>Ligne 2 - Colonne 1</cel>

<cel>Ligne 2 - Colonne 2</cel>

</line>

</table>

vous pouvez récupérer toutes les balises <cel> de la manière suivante :

var cels = xhr.responseXML.getElementsByTagName('cel');

**Récupération des en-têtes de la réponse**

Il se peut que nous ayons besoin de récupérer les valeurs des en-têtes fournis avec la réponse de notre requête. Pour cela, nous pouvons utiliser deux méthodes. La première se nomme getAllResponseHeaders() et retourne tous les en-têtes de la réponse en vrac. Voici ce que cela peut donner :

Date: Sat, 17 Sep 2011 20:09:46 GMT

Server: Apache

Vary: Accept-Encoding

Content-Encoding: gzip

Content-Length: 20

Keep-Alive: timeout=2, max=100

Connection: Keep-Alive

Content-Type: text/html; charset=utf-8

La deuxième méthode, getResponseHeader(), permet la récupération d'un seul en-tête. Il suffit d'en spécifier le nom en paramètre et la méthode retournera sa valeur :

var xhr = new XMLHttpRequest();

xhr.open('HEAD', 'http://mon\_site\_web.com/', false);

xhr.send(null);

alert(xhr.getResponseHeader('Content-type')); // Affiche : « text/html; charset=utf-8 »

**Mise en pratique**

Nous allons pour l’exemple créer une page qui va s'occuper de charger le contenu de deux autres fichiers selon le choix de l'utilisateur.

Voici la page HTML correspondante :

<p>

Veuillez choisir quel est le fichier dont vous souhaitez voir le contenu :

</p>

<p>

<input type="button" value="file1.txt" />

<input type="button" value="file2.txt" />

</p>

<p id="fileContent">

<span>Aucun fichier chargé</span>

</p>

Puis le code JavaScript qui sera appelée lors d'un clic sur un des deux boutons, elle sera chargée de s'occuper du téléchargement et de l'affichage du fichier passé en paramètre :

function loadFile(file) {

var xhr = new XMLHttpRequest();

// On souhaite juste récupérer le contenu du fichier, la méthode GET suffit amplement :

xhr.open('GET', file);

xhr.addEventListener('readystatechange', function() { // On gère ici une requête asynchrone

if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) { // Si le fichier est chargé sans erreur

document.getElementById('fileContent').innerHTML = '<span>' + xhr.responseText + '</span>'; // Et on affiche !

}

});

xhr.send(null); // La requête est prête, on envoie tout !

}

Une fois les événements de type click sur les boutons gérés, le fait de cliquer sur l’un ou l’autre bouton permettra d’afficher le contenu du fichier correspondant dans la balise <span> du HTML.

**XHR et les tests locaux**

L'exemple précédent ne fonctionnera pas en local. En effet, nous utilisons la propriété status  pour savoir si la requête HTTP a abouti. Cependant en local, il n'y a pas de requête http. Et donc, status vaudra 0 . Pour qu'un code XHR fonctionne en local, il faut donc gérer le cas où status  peut valoir 0. Il faudra penser à retirer cette condition lorsque notre script sera sur notre serveur, car la valeur 0  est une valeur d'erreur.

**Gestion des erreurs**

Le code de l'exercice que nous venons de réaliser ne sait pas prévenir en cas d'erreur. La gestion d’une erreur se fera dans via l’événement readystatechange :

xhr.addEventListener('readystatechange', function() { // On gère ici une requête asynchrone

if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) { // Si le fichier est chargé sans erreur

document.getElementById('fileContent').innerHTML = '<span>' + xhr.responseText + '</span>'; // On l'affiche !

} else if (xhr.readyState === XMLHttpRequest.DONE && xhr.status != 200) { // En cas d'erreur !

alert('Une erreur est survenue !\n\nCode :' + xhr.status + '\nTexte : ' + xhr.statusText);

}

});

**Résoudre les problèmes d'encodage**

L'encodage des caractères est une manière de représenter les caractères en informatique. Lorsque nous tapons un caractère sur notre ordinateur, il est enregistré au format binaire dans la mémoire de l'ordinateur. Ce format binaire est un code qui représente notre caractère. Ce code ne représente qu'un seul caractère, mais peut très bien désigner des caractères très différents selon les normes utilisées.

La norme UTF-8 stocke les caractères sur un nombre variable de bits, ce qui permet par exemple d’afficher deux langues très différentes (français et japonais) dans un même fichier. Un caractère classique, comme la lettre A, sera stocké sur 8 bits (1 octet donc), mais un caractère plus exotique comme le A en japonais (あ) est stocké sur 24 bits (3 octets), le maximum de bits utilisables par l'UTF-8 étant 32, soit 4 octets. En clair, l'UTF-8 est une norme qui sait s'adapter aux différentes langues et est probablement la norme d'encodage la plus aboutie de notre époque.

**L'encodage et le développement Web**

Nous allons étudier quelles sont les étapes pour bien définir son encodage des caractères sur le Web. Dans le monde du Web, il faut spécifier quel est l'encodage que nous souhaitons utiliser pour nos fichiers, alors que les navigateurs pourraient le détecter d'eux-mêmes. Prenons l'exemple d'un fichier PHP contenant du HTML et listons les différentes manières pour définir le bon encodage sur la machine du client :

* Une étape toujours nécessaire est de bien encoder ses fichiers. Cela se fait dans les paramétrages de l'éditeur de texte que nous utilisons.
* Le serveur HTTP (généralement Apache) peut indiquer quel est l'encodage utilisé par les fichiers du serveur. Cela est généralement paramétré de base, mais nous pouvons redéfinir ce paramétrage avec un fichier .htaccess (fichier de configuration Apache) contenant la ligne : AddDefaultCharset UTF-8.
* Le langage serveur peut aussi définir l'encodage utilisé dans les en-têtes du fichier. Si un encodage est spécifié par le PHP, alors il va remplacer celui indiqué par Apache. Cela se fait grâce à la ligne suivante : <?php header('Content-Type: text/html; charset=utf-8'); ?>.
* Le HTML permet de spécifier l'encodage de notre fichier, mais cela n'est généralement que peu nécessaire, car les encodages spécifiés par Apache ou le PHP font que le navigateur ignore ce qui est spécifié par le document HTML. Cela dit, mieux vaut le spécifier pour le support des très vieux navigateurs. Cela se fait dans la balise <head> avec la ligne suivante : <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />.

Remarque : Dans l’éditeur de texte, il se peut qu’il y ait deux types d'encodage UTF-8 proposés : un nommé « UTF-8 avec BOM », et l'autre nommé « UTF-8 sans BOM ». Il faut utiliser en permanence l'encodage sans BOM. Le BOM est une indication de l'ordre des octets qui est ajoutée au tout début du fichier, ce qui fait que, si nous souhaitons appeler la fonction header() en PHP, nous ne pourrons pas le faire, car des caractères auront déjà été envoyés, en l’occurrence les caractères concernant le BOM.

**L'AJAX et l'encodage des caractères**

Lorsque nous faisons une requête AJAX, toutes les données sont envoyées avec un encodage UTF-8, quel que soit l'encodage du fichier HTML qui contient le script pour la requête AJAX ! Cela pose problème si nous travaillons autrement qu'en UTF-8 côté serveur. Si le fichier PHP appelé par la requête AJAX est encodé, par exemple, en ISO 8859-1, alors il se doit de travailler avec des données ayant le même encodage, ce que ne fournira pas une requête AJAX. Ainsi, nous allons

nous retrouver avec des caractères étranges en lieu et place de certains caractères situés dans le texte d'origine, particulièrement pour les caractères accentués.

**Comprendre la démarche de l'AJAX**

Voici les étapes d'encodage d'une requête avec des fichiers en ISO 8859-1 (que nous allons abréger ISO) :

* La requête est envoyée, les données sont alors converties proprement de l'ISO à l'UTF-8. Ainsi, le ê en ISO est toujours un ê en UTF-8, l'AJAX sait faire la conversion d'encodage sans problème.
* Les données arrivent sur le serveur, c'est là que se pose le problème : elles arrivent en UTF-8, alors que le serveur attend des données ISO, cette erreur d'encodage n'étant pas détectée, le caractère ê n'est plus du tout le même vis-à-vis du serveur, il s'agit alors des deux caractères Ãª (car ê est encodé sur 2 octets en UTF-8, alors que les caractères en ISO 8859-1 ne sont encodés que sur un seul octet).
* Le serveur renvoie des données au format ISO, mais celles-ci ne subissent aucune modification d'encodage lors du retour de la requête. Les données renvoyées par le serveur en ISO seront bien réceptionnées en ISO.

Ainsi, une requête AJAX n'opère en UTF-8 que lors de l'envoi des données, le problème d'encodage ne survient donc que lorsque les données sont réceptionnées par le serveur, et non pas quand le client reçoit les données renvoyées par le serveur.

**Deux solutions**

Il existe deux solutions pour éviter ce problème d'encodage sur nos requêtes AJAX.

La première, qui est de loin la plus simple et la plus pérenne, consiste à ce que notre site soit entièrement encodé en UTF-8. Ainsi, les requêtes AJAX envoient des données en UTF-8 qui seront reçues par un serveur demandant à traiter de l'UTF-8, donc sans aucun problème. Un site en UTF-8 implique que tous nos fichiers textes soient encodés en UTF-8, que le serveur indique au client le bon encodage, et que nos ressources externes, comme les bases de données, soient aussi en UTF-8.  
Cette solution est vraiment la meilleure, mais est difficile à mettre en place sur un projet Web déjà bien entamé.

La deuxième solution, encore souvent rencontrée, est plus adaptée si notre projet est déjà bien entamé et que nous ne pouvons pas nous permettre de faire une conversion complète de son encodage. Il s'agit de décoder les caractères reçus par le biais d'une requête AJAX avec la fonction PHP utf8\_decode().

Admettons que nous envoyions une requête AJAX à la page suivante :

<?php

header('Content-Type: text/plain; charset=iso-8859-1'); // On précise bien qu'il s'agit d'une page en ISO 8859-1

echo $\_GET['parameter'];

?>

Si la requête AJAX envoie en paramètre la chaîne de caractères « Drôle de tête », le serveur va alors vous renvoyer ceci : DrÃ´le de tÃªte. La solution consiste donc à décoder l'UTF-8 reçu pour le convertir en ISO 8859-1, la fonction utf8\_decode() intervient donc ici :

<?php

header('Content-Type: text/plain; charset=iso-8859-1'); // On précise bien qu'il s'agit d'une page en ISO 8859-1

echo utf8\_decode($\_GET['parameter']);

?>

Si nous travaillons dans un encodage autre que l'ISO 8859-1, il faudra utiliser la fonction mb\_convert\_encoding().

**Deuxième version : usage avancé**

La deuxième version du XHR ajoute de nombreuses fonctionnalités intéressantes. Le XHR2 ne fait pas partie de la spécification du HTML5 mais utilise de nombreuses technologies liées au HTML5.

* L'objet utilisé pour la deuxième version est le même que celui utilisé pour la première, à savoir XMLHttpRequest.
* Toutes les fonctionnalités présentes dans la première version sont présentes dans la deuxième.

**Les requêtes cross-domain**

Les requêtes **cross-domain** sont des requêtes effectuées depuis un nom de domaine A vers un nom de domaine B. Elles sont pratiques, mais absolument inutilisables avec la première version du XHR en raison de la présence d'une sécurité basée sur le principe de la same origin policy. Cette sécurité étant appliquée aux différents langages utilisables dans un navigateur Web, le JavaScript est donc concerné. Il est important de comprendre en quoi elle consiste et comment elle peut-être « contournée », car les requêtes cross-domain sont au cœur du XHR2.

Bien que la same origin policy soit une sécurité contre de nombreuses failles, elle est un véritable frein pour le développement Web, car elle a pour principe de n'autoriser les requêtes XHR qu'entre les pages Web possédant le même nom de domaine. Si, par exemple, nous nous trouvons sur notre site personnel dont le nom de domaine est mon\_site\_perso.com et que nous tentons de faire une requête XHR vers google.com, nous allons alors rencontrer une erreur et la requête ne sera pas exécutée, car les deux noms de domaine sont différents.

Cette sécurité s'applique aussi dans d'autres cas, comme deux sous-domaines différents. Afin de présenter rapidement et facilement les différents cas concernés ou non par cette sécurité, voici un tableau largement réutilisé sur le Web. Il illustre différents cas où les requêtes XHR sont possibles ou non. Les requêtes sont exécutées depuis la page http://www.example.com/dir/page.html :

|  |  |  |
| --- | --- | --- |
| **URL appelée** | **Résultat** | **Raison** |
| http://www.example.com/dir/page.html | Succès | Même protocole et même nom de domaine0 |
| http://www.example.com/dir2/other.html | Succès | Même protocole et même nom de domaine, seul le dossier diffère0 |
| http://www.example.com:81/dir/other.html | Échec | Même protocole et même nom de domaine, mais le port est différent (80 par défaut) |
| https://www.example.com/dir/other.html | Échec | Protocole différent (HTTPS au lieu de HTTP)0 |
| http://en.example.com/dir/other.html | Échec | Sous-domaine différent0 |
| http://example.com/dir/other.html | Échec | Si l'appel est fait depuis un nom de domaine dont les « www » sont spécifiés, alors il faut faire de même pour la page appelée. |

Cette sécurité est impérative, mais il se peut que nous possédions deux sites Web dont les noms de domaine soient différents, mais dont la connexion doit se faire par le biais des requêtes XHR. La deuxième version du XHR introduit donc un système simple et efficace permettant l'autorisation des requêtes cross-domain.

**Autoriser les requêtes cross-domain**

Il existe une solution implémentée dans la deuxième version du XHR, qui consiste à ajouter un simple en-tête dans la page appelée par la requête pour autoriser le cross-domain. Cet en-tête se nomme Access-Control-Allow-Origin et permet de spécifier un ou plusieurs domaines autorisés à accéder à la page par le biais d'une requête XHR.

Pour spécifier un nom de domaine, il suffit d'écrire :

Access-Control-Allow-Origin: http://example.com. Ainsi, le domaine example.com aura accès à la page qui retourne cet en-tête. Il est impossible de spécifier plusieurs noms de domaine mais il est possible d'autoriser tous les noms de domaine à accéder à votre page en utilisant \*.

Il ne reste ensuite plus qu'à ajouter cet en-tête aux autres en-têtes de notre page Web, comme ici en PHP :

<?php

header('Access-Control-Allow-Origin: \*');

?>

Cependant, il faut prendre garde à l'utilisation de cet astérisque. Il ne faut l’utiliser que si nous n'avons pas le choix, car, lorsque nous autorisons un nom de domaine à faire des requêtes cross-domain sur notre page, c'est comme si nous désactivions une sécurité contre le piratage vis-à-vis de ce domaine.

**Éviter les requêtes trop longues**

Il se peut que, de temps en temps, certaines requêtes soient excessivement longues. Afin d'éviter ce problème, il est parfaitement possible d'utiliser la méthode abort() couplée à setTimeout(). Cependant, le XHR2 fournit une solution bien plus simple à mettre en place. Il s'agit de la propriété timeout, qui prend pour valeur un temps en millisecondes. Une fois ce temps écoulé, la requête se terminera :

xhr.timeout = 10000; // La requête se terminera si elle n'a pas abouti au bout de 10 secondes

**Forcer le type de contenu**

Vous souvenez-vous lorsque nous avions abordé le fait qu'il fallait bien spécifier le type MIME de vos documents afin d'éviter que vos fichiers XML ne soient pas parsés ?

Si nous n'avons pas la possibilité de spécifier le type MIME de nos documents afin d'éviter que nos fichiers XML ne soient pas parsés (par exemple, si nous n'avons pas accès au code de la page que nous appelons), nous pouvons réécrire le type MIME reçu afin de parser correctement le fichier. Ceci se réalise avec la nouvelle méthode overrideMimeType(), qui prend en paramètre un seul argument contenant le type MIME exigé :

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://example.com');

xhr.overrideMimeType('text/xml'); // L'envoi de la requête puis le traitement des données reçues peuvent se faire

Cette méthode ne peut être utilisée que lorsque la propriété readyState possède les valeurs 1 ou 2. Autrement dit, lorsque la méthode open() vient d'être appelée ou bien lorsque les en-têtes viennent d'être reçus, ni avant, ni après.

**Accéder aux cookies et aux sessions avec une requête cross-domain**

Il est possible pour une page appelée par le biais d'une requête XHR (versions 1 et 2) d'accéder aux cookies ou aux sessions du navigateur. Cela se fait sans contrainte. Nous pouvons, par exemple, accéder aux cookies comme nous le faisons d'habitude :

<?php

echo $\_COOKIE['cookie1']; // Aucun problème !

?>

Cependant, cette facilité d'utilisation est loin d'être présente lorsque nous souhaitons accéder à ces ressources avec une requête cross-domain, car aucune valeur ne sera retournée par les tableaux $\_COOKIE et $\_SESSION. Il s’agit ici d’une sécurité, car vous allez devoir autoriser le navigateur et le serveur à gérer ces données. Quand nous parlons du serveur, nous voulons surtout parler de la page appelée par la requête. Nous allons devoir y spécifier l'en-tête suivant pour autoriser l'envoi des cookies et des sessions : Access-Control-Allow-Credentials: true.

Côté serveur, cela ne suffira pas si nous avons spécifié l'astérisque \* pour l'en-tête Access-Control-Allow-Origin. Il faut absolument spécifier un seul nom de domaine. Nous devrions donc avoir une page PHP commençant par un code de ce genre :

<?php

header('Access-Control-Allow-Origin: http://example.com');

header('Access-Control-Allow-Credentials: true');

?>

Cependant, nous obtiendrons des valeurs nulles pour les cookies ou les sessions. La raison est simple : le serveur est configuré pour permettre l'accès à ces données, mais le navigateur ne les envoie pas. Pour pallier ce problème, il suffit d'indiquer à notre requête que l'envoi de ces données est nécessaire. Cela se fait après initialisation de la requête et avant son envoi (autrement dit, entre l'utilisation des méthodes open() et send()) avec la propriété withCredentials :

xhr.open( … );

xhr.withCredentials = true; // Avec « true », l'envoi des cookies et des sessions est bien effectué

xhr.send( … );

Ici, si nous avons une page Web nommée client.php située sur un nom de domaine A, et que depuis cette page, nous appelons la page server.php située sur le domaine B grâce à une requête cross-domain, les cookies et les sessions reçus par la page server.php seront ceux du domaine B. Si nous faisons une requête cross-domain, les cookies et les sessions envoyés seront constamment ceux qui concernent le domaine de la page appelée. Cela s'applique aussi si nous utilisons la fonction PHP setcookie() dans la page appelée : les cookies modifiés seront ceux du domaine de cette page, et non pas ceux du domaine d'où provient la requête.

Remarque : ce qui a été étudié ne nous concerne que lorsque nous faisons une requête cross-domain. Dans le cas d'une requête dite « classique », nous n'avons pas à faire ces manipulations, tout fonctionne sans cela, même pour une requête XHR1.

**Quand les événements s'affolent**

La première version du XHR ne comportait qu'un seul événement, la deuxième en comporte maintenant huit si on compte l'événement readystatechange. Ces ajouts ont été faits parce que le XHR1 ne permettait clairement pas de faire un suivi correct de l'état d'une requête.

**Les événements classiques**

Commençons par trois événements simples : loadstart, load et loadend. Le premier se déclenche lorsque la requête démarre (lorsque nous appelons la méthode send()). Les deux derniers se déclenchent lorsque la requête se termine, mais avec une petite différence : si la requête s'est correctement terminée (pas d'erreur 404 ou autre), alors load se déclenche, tandis que loadend se déclenche dans tous les cas. L'avantage de l'utilisation de load et loadend, c'est que nous pouvons alors nous affranchir de la vérification de l'état de la requête avec la propriété readyState, comme nous le ferions pour l'événement readystatechange.

Les deux événements suivants sont error et abort. Le premier se déclenche en cas de non-aboutissement de la requête (quand readyState n'atteint même pas la valeur finale : 4), tandis que le deuxième s'exécutera en cas d'abandon de la requête avec la méthode abort() ou bien avec le bouton « Arrêt » de l'interface du navigateur Web.

Concernant la propriété timeout, il existe un événement du même nom qui se déclenche quand la durée maximale spécifiée dans la propriété associée est atteinte.

**Le cas de l'événement progress**

Son rôle est de se déclencher à intervalles réguliers pendant le rapatriement du contenu exigé par notre requête. Son utilisation n'est nécessaire que dans les cas où le fichier rapatrié est assez volumineux. Cet événement a pour particularité de fournir un objet en paramètre à la fonction associée. Cet objet contient deux propriétés nommées loaded et total. Elles indiquent, respectivement, le nombre d'octets actuellement téléchargés et le nombre d'octets total à télécharger. Leur utilisation se fait de cette manière :

xhr.addEventListener('progress', function(e) {

element.innerHTML = e.loaded + ' / ' + e.total;

});

**L'objet FormData**

Cet objet consiste à faciliter l'envoi des données par le biais de la méthode POST des requêtes XHR. Comme nous l'avons dit plus tôt dans ce chapitre, l'envoi des données par le biais de POST est une chose assez fastidieuse, car il faut spécifier un en-tête dont on ne se souvient que très rarement de tête.

Au-delà de son côté pratique en terme de rapidité d'utilisation, l'objet FormData est aussi un formidable outil permettant de faire un envoi de données binaires au serveur. Ce qui, concrètement, veut dire qu'il est possible de faire de l'upload de fichiers par le biais des requêtes XHR. Cependant, l'upload de fichiers nécessite des connaissances approfondies sur le HTML5, cela sera donc traité plus tard. Nous allons tout d'abord nous contenter d'une utilisation relativement simple. Tout d'abord, l'objet FormData doit être instancié : var form = new FormData();.

Une fois instancié, Nous pouvons nous servir de son unique méthode : append(). Celle-ci ne retourne aucune valeur et prend en paramètres deux arguments obligatoires : le nom d'un champ (qui correspond à l'attribut name des éléments d'un formulaire) et sa valeur. Son utilisation est donc très simple :

form.append('champ1', 'valeur1');

form.append('champ2', 'valeur2');

C'est là que cet objet est intéressant : pas besoin de spécifier un en-tête particulier pour dire que l'on envoie des données sous forme de formulaire. Il suffit juste de passer notre objet de type FormData à la méthode send(), ce qui donne ceci sur un code complet :

var xhr = new XMLHttpRequest();

xhr.open('POST', 'script.php');

var form = new FormData();

form.append('champ1', 'valeur1');

form.append('champ2', 'valeur2');

xhr.send(form);

Et côté serveur, nous pouvons récupérer les données tout aussi simplement que d'habitude :

<?php

echo $\_POST['champ1'] . ' - ' . $\_POST['champ2']; // Affiche : « valeur1 - valeur2 »

?>

Revenons rapidement sur le constructeur de cet objet, car celui-ci possède un argument bien pratique. Si nous passer en paramètre un élément de formulaire, notre objet FormData sera alors prérempli avec toutes les valeurs de ce formulaire. Voici un exemple simple :

<form id="myForm">

<input id="myText" name="myText" type="text" value="Test ! Un, deux, un, deux !" />

</form>

<script>

var xhr = new XMLHttpRequest();

xhr.open('POST', 'script.php');

var myForm = document.getElementById('myForm'),

form = new FormData(myForm);

xhr.send(form);

</script>

Ce qui, côté serveur, donne ceci :

<?php

echo $\_POST['myText']; // Affiche : « Test ! Un, deux, un, deux ! »

?>

## Upload via une iframe

L'élément HTML <iframe> permet d'insérer une page Web dans une autre. Voici un petit rappel de la syntaxe d'une iframe :

<iframe src="file.html" name="myFrame" id="myFrame"></iframe>

**Accéder au contenu**

Pour accéder au contenu de l'iframe, il faut d'abord accéder à l'iframe elle-même et ensuite passer par la propriété contentDocument :

var frame = document.getElementById('myFrame').contentDocument;

Une fois que l'on a accédé au contenu de l'iframe, c'est-à-dire à son document, on peut naviguer dans le DOM comme s'il s'agissait d'un document « normal » :

var frame\_links = frame.getElementsByTagName('a').length;

Concernant la règle de sécurité same origin policy, celle-ci s'applique aussi aux iframes ! Cela veut dire que si nous sommes sur une page d'un domaine A et que nous appelons une page d'un domaine B par le biais d'une iframe, alors nous ne pourrons pas accéder au contenu de la page B depuis la page A.

**Chargement de contenu**

Il y a deux techniques pour charger une page dans une iframe. La première est de tout simplement changer l’attribut src de l'iframe via le JavaScript, la deuxième est d'ouvrir un lien dans l'iframe. Cette action est rendue possible via l'attribut target (standardisé en HTML5) que l'on peut utiliser sur un lien ou sur un formulaire. C'est cette dernière technique que nous utiliserons pour la réalisation du système d'upload.

**Charger une iframe en changeant l'URL**

On change simplement l'URL de l'iframe en changeant sa propriété src. Cette technique est simple et permet de transmettre des paramètres directement dans l'URL :

document.getElementById('myFrame').src = 'request.php?nick=Thunderseb';

**Avec target et un formulaire**

L'intérêt d'utiliser un formulaire est que nous allons pouvoir envoyer des données via la méthode POST. L'utilisation de POST va nous permettre d'envoyer des fichiers, ce qui nous sera utile pour un upload de fichiers. En fait, pour cette technique, il n'y a pas vraiment besoin du JavaScript, c'est du HTML pur :

<form id="myForm" method="post" action="request.php" target="myFrame">

<div>

<!-- formulaire -->

<input type="submit" value="Envoyer" />

</div>

</form>

<iframe src="#" name="myFrame" id="myFrame"></iframe>

L'attribut target indique au formulaire que son contenu doit être envoyé au sein de l'iframe dont l'attribut name est myFrame (l'attribut name est donc obligatoire ici). De cette manière le contenu du formulaire y sera envoyé, et la page courante ne sera pas rechargée.

Le JavaScript pourra être utilisé comme méthode alternative pour envoyer le formulaire. Pour rappel, pour envoyer un formulaire, il faut utiliser la méthode submit() :

document.getElementById('myForm').submit();

**Détecter le chargement avec l'événement load**

Les iframes possèdent un événement load, déclenché une fois que le contenu de l'iframe est chargé. À chaque contenu chargé, load est déclenché. C'est un moyen efficace pour savoir si le document est chargé, et ainsi pouvoir le récupérer :

<iframe src="file.html" name="myFrame" id="myFrame" onload="trigger()"></iframe>

<script>

function trigger() {

var frame = document.getElementById('myFrame').contentDocument;

alert(frame.body.textContent);

}

</script>

**Avec une fonction de callback**

Quand une page Web est chargée dans l'iframe, son contenu est affiché et les scripts sont exécutés. Il est également possible, depuis l'iframe, d'appeler une fonction présente dans la page « mère », c'est-à-dire la page qui contient l'iframe. Pour appeler une fonction depuis l'iframe, il suffit d'utiliser :

window.top.window.nomDeLaFonction();

L'objet window.top pointe vers la fenêtre « mère », ce qui nous permet ici d'atteindre la page qui contient l'iframe. Par exemple, pour la page mère suivante :

<iframe src="file.html" name="myFrame" id="myFrame"></iframe>

<script>

function trigger() {

var frame = document.getElementById('myFrame').contentDocument;

alert('Page chargée !');

}

</script>

La page « fille » suivate, intégrée via la balise « iframe », pourra utiliser la fonctionner trigger() de la page mère pour indiquant la fin de son chargement :

<script>

window.top.window.trigger(); // On appelle ici notre fonction de callback

</script>

<p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Suspendisse molestie suscipit arcu.</p>

**Récupérer du contenu**

Le chargement de données via une iframe a un gros avantage : il est possible de charger n'importe quoi comme données. Ça peut être une page Web complète, du texte brut ou même du JavaScript, comme le format JSON.

**Récupérer des données JavaScript**

Si on reprend l'exemple vu précédemment, avec le callback, il est possible de récupérer facilement des données JavaScript, comme un objet. Dans ce cas, il suffit d'utiliser du PHP pour construire un objet qui sera transmis en paramètre de la fonction de callback, comme ceci :

<?php

$fakeArray = array('Sébastien', 'Laurence', 'Ludovic');

?>

<script>

window.top.window.trigger(['<?php echo implode("','", $fakeArray) ?>']);

</script>

Ici, un tableau JavaScript est construit via le PHP et envoyé à la fonction trigger() en tant que paramètre.

**Le système d'upload**

Par le biais d'un formulaire et d'une iframe, créer un système d'upload n'est pas compliqué. Les éléments <form> possèdent un attribut enctype qui doit absolument contenir la valeur multipart/form-data. Pour faire simple, cette valeur indique que le formulaire est prévu pour envoyer de grandes quantités de données (les fichiers sont des données volumineuses). Notre formulaire d'upload peut donc être écrit comme ceci :

<form id="uploadForm" enctype="multipart/form-data" action="upload.php" target="uploadFrame" method="post">

<label for="uploadFile">Image :</label>

<input id="uploadFile" name="uploadFile" type="file" />

<br /><br />

<input id="uploadSubmit" type="submit" value="Upload !" />

</form>

Ensuite, on place l'iframe, ainsi qu'un autre <div> que nous utiliserons pour afficher le résultat de l'upload :

<div id="uploadInfos">

<div id="uploadStatus">Aucun upload en cours</div>

<iframe id="uploadFrame" name="uploadFrame"></iframe>

</div>

Et pour finir, notre JavaScript :

function uploadEnd(error, path) {

if (error === 'OK') {

document.getElementById('uploadStatus').innerHTML = '<a href="' + path + '">Upload done !</a><br /><br /><a href="' + path + '"><img src="' + path + '" /></a>';

} else {

document.getElementById('uploadStatus').innerHTML = error;

}

}

document.getElementById('uploadForm').addEventListener('submit', function() {

document.getElementById('uploadStatus').innerHTML = 'Loading...';

});

Dès que le formulaire est envoyé, la fonction anonyme de l'événement submit est exécutée. Celle-ci va remplacer le texte du <div>#uploadStatus pour indiquer que le chargement est en cours. Car, en fonction de la taille du fichier à envoyer, l'attente peut être longue. L'argument error contiendra soit « OK », soit une explication sur une erreur éventuelle. L'argument path contiendra l'URL du fichier venant d'être uploadé. L'appel vers la fonction uploadEnd() sera fait via l'iframe, comme nous le verrons plus loin.

**Le code côté serveur : upload.php**

Le JavaScript étant mis en place, il ne reste plus qu'à nous occuper de la page upload.php qui va réceptionner le fichier uploadé :

<?php

$error = NULL;

$filename = NULL;

if (isset($\_FILES['uploadFile']) && $\_FILES['uploadFile']['error'] === 0) {

$filename = $\_FILES['uploadFile']['name'];

$targetpath = getcwd() . '/' . $filename; // On stocke le chemin où enregistrer le fichier

// On déplace le fichier depuis le répertoire temporaire vers $targetpath

if (@move\_uploaded\_file($\_FILES['uploadFile']['tmp\_name'], $targetpath)) { // Si ça fonctionne

$error = 'OK';

} else { // Si ça ne fonctionne pas

$error = "Échec de l'enregistrement !";

}

} else {

$error = 'Aucun fichier réceptionné !';

}

// Et pour finir, on écrit l'appel vers la fonction uploadEnd :

?>

<script>

window.top.window.uploadEnd("<?php echo $error; ?>", "<?php echo $filename; ?>");

</script>

## Dynamic Script Loading

Au cours de ce chapitre vous allez découvrir une manière astucieuse de dialoguer avec un serveur. Elle possède un avantage considérable face à l'objet XMLHttpRequest : elle n'est en aucun cas limitée par le principe de la same origin policy.

**Un concept simple**

Avec le DOM, il est possible d'insérer n'importe quel élément HTML au sein d'une page Web, et cela vaut également pour un élément <script>. Il est donc possible de lier et d'exécuter un fichier JavaScript après que la page a été chargée :

window.addEventListener('load', function() {

var scriptElement = document.createElement('script');

scriptElement.src = 'url/du/fichier.js';

document.body.appendChild(scriptElement);

});

Avec ce code, un nouvel élément <script> sera inséré dans la page une fois que cette dernière aura été chargée.

**Un premier exemple**

Nous allons commencer par quelque chose de très simple : dans une page HTML, on va charger un fichier JavaScript qui exécutera une fonction. Cette fonction se trouve dans la page HTML :

<script>

function sendDSL() {

var scriptElement = document.createElement('script');

scriptElement.src = 'dsl\_script.js';

document.body.appendChild(scriptElement);

}

function receiveMessage(message) {

alert(message);

}

</script>

<p><button type="button" onclick="sendDSL()">Exécuter le script</button></p>

Voici maintenant le contenu du fichier dsl\_script.js :

receiveMessage('Ce message est envoyé par le serveur !');

Dès qu'on clique sur le bouton, la fonction sendDSL() va charger le fichier JavaScript qui contient un appel vers la fonction receiveMessage(), tout en prenant soin de lui passer un message en paramètre. Ainsi, via la fonction receiveMessage(), on est en mesure de récupérer du contenu. Évidemment, cet exemple n'est pas très intéressant puisque l'on sait à l'avance ce que le fichier JavaScript va renvoyer. Ce que nous allons faire, c'est créer le fichier JavaScript via du PHP !

**Avec des variables et du PHP**

Maintenant, au lieu d'appeler un fichier JavaScript, nous allons appeler une page PHP. Si on reprend le code donné précédemment, on peut modifier l'URL du fichier JavaScript :

scriptElement.src = 'dsl\_script.php?nick=' + prompt('Quel est votre pseudo ?');

En ce qui concerne le fichier PHP, il va falloir utiliser la fonction header() pour indiquer au navigateur que le contenu du fichier PHP est en réalité du JavaScript. Puis, il ne reste plus qu'à introduire la variable $\_GET['nick'] au sein du script JavaScript :

<?php header("Content-type: text/javascript"); ?>

var string = 'Bonjour <?php echo $\_GET['nick'] ?> !';

receiveMessage(string);

On constate maintenant que le script retourne bien le pseudo que l'utilisateur a entré.

### Le DSL et le format JSON

Le gros avantage du **Dynamic Script Loading** (DSL) est qu'il permet de récupérer du contenu sous forme d'objets JavaScript, comme un tableau ou tout simplement un objet littéral, et donc le fameux JSON. Si on récupère des données JSON via XMLHttpRequest, ces données sont livrées sous la forme de texte brut (récupérées via la propriété responseText). Il faut donc utiliser la méthode parse() de l'objet JSON pour pouvoir les interpréter. Avec le DSL, ce petit souci n'existe pas puisque c'est du JavaScript qui est transmis, et non du texte.

**Charger du JSON**

Comme dans l'exemple précédent, nous allons utiliser une page PHP pour générer le contenu du fichier JavaScript, et donc notre JSON. Les données JSON contiennent une liste d'éditeurs et pour chacun une liste de programmes qu'ils éditent :

<?php

header("Content-type: text/javascript");

echo 'var softwares = {

"Adobe": [

"Acrobat"

],

"Mozilla": [

"Firefox"

],

"Microsoft": [

"Office"

]

};';

?>

receiveMessage(softwares);

Au niveau de la page HTML, pas de gros changements. Nous allons juste coder une meilleure fonction receiveMessage() de manière à afficher, dans une alerte, les données issues du JSON. On utilise une boucle for in pour parcourir le tableau associatif, et une deuxième boucle for imbriquée pour chaque tableau :

<script>

function sendDSL() {

var scriptElement = document.createElement('script');

scriptElement.src = 'dsl\_script\_json.php';

document.body.appendChild(scriptElement);

}

function receiveMessage(json) {

var tree = '',

nbItems, i;

for (node in json) {

tree += node + "\n";

nbItems = json[node].length;

for (i = 0; i < nbItems; i++) {

tree += '\t' + json[node][i] + '\n';

}

}

alert(tree);

}

</script>

<p><button type="button" onclick="sendDSL()">Charger le JSON</button></p>

Avec ça, pas besoin de parser le JSON, c'est directement opérationnel ! Cette notion est souvent désignée par l'abréviation JSONP.

**Petite astuce pour le PHP**

Le PHP dispose de deux fonctions pour manipuler du JSON : json\_encode() et json\_decode(). La première, json\_encode(), permet de convertir une variable (un tableau associatif par exemple) en une chaîne de caractères au format JSON. La deuxième, json\_decode(), fait le contraire : elle recrée une variable à partir d'une chaîne de caractères au format JSON.

## TP : un système d'auto-complétion

Ce TP n'utilisera l'AJAX que par le biais de l'objet XMLHttpRequest. Cependant, il s'agit de la méthode la plus en vogue de nos jours, l'utilisation d'iframes et de DSL étant réservée à des cas bien plus particuliers.

Commençons tout d'abord par le code PHP du serveur :

<?php

$data = unserialize(file\_get\_contents('towns.txt')); // Récupération de la liste complète des villes

$dataLen = count($data);

sort($data); // On trie les villes dans l'ordre alphabétique

$results = array(); // Le tableau où seront stockés les résultats de la recherche

// La boucle ci-dessous parcourt tout le tableau $data, jusqu'à un maximum de 10 résultats

for ($i = 0 ; $i < $dataLen && count($results) < 10 ; $i++) {

if (stripos($data[$i], $\_GET['s']) === 0) { // Si la valeur commence par les mêmes caractères que la recherche

array\_push($results, $data[$i]); // On ajoute alors le résultat à la liste à retourner

}

}

echo implode('|', $results); // Et on affiche les résultats séparés par une barre verticale |

?>

Vient ensuite la structure HTML :

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>TP : Un système d'auto-complétion</title>

</head>

<body>

<input id="search" type="text" autocomplete="off" />

<div id="results"></div>

</body>

</html>

Et pour finir, voici le code JavaScript :

(function() {

var searchElement = document.getElementById('search'),

results = document.getElementById('results'),

selectedResult = -1, // Permet de savoir quel résultat est sélectionné : -1 signifie "aucune sélection"

previousRequest, // On stocke notre précédente requête dans cette variable

previousValue = searchElement.value; // On fait de même avec la précédente valeur

function getResults(keywords) { // Effectue une requête et récupère les résultats

var xhr = new XMLHttpRequest();

xhr.open('GET', './search.php?s='+ encodeURIComponent(keywords));

xhr.addEventListener('readystatechange', function() {

if (xhr.readyState == XMLHttpRequest.DONE && xhr.status == 200) {

displayResults(xhr.responseText);

}

});

xhr.send(null);

return xhr;

}

function displayResults(response) { // Affiche les résultats d'une requête

results.style.display = response.length ? 'block' : 'none'; // On cache le conteneur si

on n'a pas de résultats

if (response.length) { // On ne modifie les résultats que si on en a obtenu

response = response.split('|');

var responseLen = response.length;

results.innerHTML = ''; // On vide les résultats

for (var i = 0, div ; i < responseLen ; i++) {

div = results.appendChild(document.createElement('div'));

div.innerHTML = response[i];

div.addEventListener('click', function(e) {

chooseResult(e.target);

});

}

}

}

function chooseResult(result) { // Choisi un des résultats d'une requête et gère tout ce qui y est attaché

searchElement.value = previousValue = result.innerHTML; // On change le contenu du champ de recherche et on enregistre en tant que précédente valeur

results.style.display = 'none'; // On cache les résultats

result.className = ''; // On supprime l'effet de focus

selectedResult = -1; // On remet la sélection à "zéro"

searchElement.focus(); // Si le résultat a été choisi par le biais d'un clique alors le focus est perdu, donc on le réattribue

}

searchElement.addEventListener('keyup', function(e) {

var divs = results.getElementsByTagName('div');

if (e.keyCode == 38 && selectedResult > -1) { // Si la touche pressée est la flèche

"haut"

divs[selectedResult--].className = '';

if (selectedResult > -1) { // Cette condition évite une modification de childNodes[-1], qui n'existe pas, bien entendu

divs[selectedResult].className = 'result\_focus';

}

} else if (e.keyCode == 40 && selectedResult < divs.length - 1) { // Si la touche pressée est la flèche "bas"

results.style.display = 'block'; // On affiche les résultats

if (selectedResult > -1) { // Cette condition évite une modification de childNodes[-1], qui n'existe pas, bien entendu

divs[selectedResult].className = '';

}

divs[++selectedResult].className = 'result\_focus';

} else if (e.keyCode == 13 && selectedResult > -1) { // Si la touche pressée est "Entrée"

chooseResult(divs[selectedResult]);

} else if (searchElement.value != previousValue) { // Si le contenu du champ de recherche a changé

previousValue = searchElement.value;

if (previousRequest && previousRequest.readyState < XMLHttpRequest.DONE) {

previousRequest.abort(); // Si on a toujours une requête en cours, on l'arrête

}

previousRequest = getResults(previousValue); // On stocke la nouvelle requête

selectedResult = -1; // On remet la sélection à "zéro" à chaque caractère écrit

}

});

})();

## Qu'est-ce que le HTML5 ?

Alors que les langages HTML 4 et autres XHTML se focalisaient juste sur le contenu des pages Web, le HTML5 se focalise sur les applications Web et l'interactivité, sans toutefois délaisser l'accessibilité et la sémantique. Le HTML5 se positionne également comme concurrent des technologies Flash et Silverlight.

**Accessibilité et sémantique**

Le HTML5 apporte dès lors de nouveaux éléments comme <nav>, <header>, <article>, <figure> qui améliorent l'accessibilité, ainsi que des éléments comme <mark> ou <data> qui améliorent la sémantique (c'est-à-dire le sens qu'on donne aux textes).

**Applications Web et interactivité**

Le HTML5 apporte de nombreux éléments comme <video>, <datagrid>, <meter>, <progress>, <output> ainsi que de nouveaux types pour les éléments <input>, comme tel, url, date, number…

Le HTML5 spécifie aussi un certain nombre d'API JavaScript. Ces API JavaScript sont des techniques que nous allons pouvoir utiliser pour développer des applications Web et ajouter de l'interactivité. Parmi ces API JavaScript, on trouve par exemple l'API Drag & Drop. API signifie « Application Programming Interface ». Une API, dans le cadre d'une utilisation en JavaScript, est un ensemble d'objets, de méthodes et de propriétés réunis sous un même thème.

**Concurrencer Flash (et Silverlight)**

Le HTML5 se place en concurrent du Flash, en fournissant des outils analogues de façon native : <video>, <audio> et surtout <canvas>. Il est donc possible dès à présent de lire des vidéos en HTML5 sans nécessiter ni Flash, ni un autre plugin contraignant. L'élément <canvas> permettra de dessiner et donc de réaliser des animations, comme on le ferait avec Flash et Silverlight.

**Les API JavaScript**

Nous allons rapidement faire le tour des différentes API apportées par le HTML5.

**History : gérer l'historique**

Avec le JavaScript, il a toujours été possible d'avancer et de reculer dans l'historique de navigation, c'est-à-dire simuler l'effet des boutons Précédent et Suivant du navigateur. L'API History permet désormais de faire plus, notamment en stockant des données lors de la navigation. Cela est utile pour les applications basées sur l'AJAX, où il est rarement possible de revenir en arrière.

**Sélecteurs CSS : deux nouvelles méthodes**

Le HTML5 apporte les méthodes querySelector() et querySelectorAll(), qui permettent d'atteindre des éléments sur base de sélecteurs CSS, dont les nouveaux sélecteurs CSS3.

**Timers : rien ne change, mais c'est standardisé**

Le HTML5 standardise enfin les [fonctions temporelles](http://www.siteduzero.com/tutoriel-3-483967-la-gestion-du-temps.html#ss_part_2), comme setInterval(), clearInterval(), setTimeout() et clearTimeout().

**ContentEditable**

ContentEditable est une technique qui permet de rendre éditable un élément HTML. Cela permet à l'utilisateur d'entrer du texte dans un <div>, ou bien de créer une interface WYSIWYG, comme Word.

**Web Storage**

Cette API permet de conserver des informations dans la mémoire du navigateur, pendant le temps que de la navigation, ou pour une durée beaucoup plus longue. Les cookies fournissent plus ou moins 4 KB de stockage, alors que le Web Storage en propose 5 MB pour la plupart des navigateurs et 10 MB pour Internet Explorer. Cependant, le Web Storage n'est pas accessible par les serveurs Web, les cookies sont donc toujours de rigueur.

Pour enregistrer une valeur il suffit de faire :localStorage.setItem('nom-de-ma-cle', 'valeur de la clé');.

Il faut donc donner un nom à la clé pour pouvoir récupérer la valeur plus tard :

alert(localStorage.getItem('nom-de-ma-cle'));.

Si les données ne doivent être gardées en mémoire que pendant le temps de la navigation (elles seront perdues si l'utilisateur ferme son navigateur), il convient d'utiliser sessionStorage au lieu de localStorage.

**Web SQL Database**

C'est en quelque sorte une évolution du Web Storage. Le Web Storage ne permet que de stocker des valeurs sous forme de clé, alors que le Web SQL Database fournit une base de données complète ! C'est aussi plus complexe à utiliser, d'autant plus que Firefox ne l'implémente pas et utilise un autre type de base de données : IndexedDB.

**WebSocket**

Le WebSocket permet à une page Web de communiquer avec le serveur Web de façon bidirectionnelle : ça veut dire que le serveur peut envoyer des informations à la page, tout comme cette dernière peut envoyer des informations au serveur. C'est en quelque sorte une API approfondie du XMLHttpRequest. Cela nécessite un serveur adapté.

**Geolocation**

L'API de géolocalisation permet de détecter la position géographique du visiteur. Cela ne fonctionne que si l'utilisateur donne son accord, en réglant les paramètres de navigation de son navigateur.

**Workers et Messaging**

L'API Workers permet d'exécuter du code en tâche de fond. Ce code est alors exécuté en parallèle de celui de la page. Si le code de la page rencontre une erreur, ça n'affecte pas le code du Worker et inversement.

Le Worker est capable d'envoyer des messages au script principal via l'API Messaging. Le script principal peut aussi envoyer des messages au Worker. L'API Messaging peut aussi être utilisée pour envoyer et recevoir des messages entre une <iframe> et sa page mère, même si elles ne sont pas hébergées sur le même domaine.

**Offline Web Application**

Cette API sert à rendre disponible une page Web même si la connexion n'est pas active. Il suffit de spécifier une liste de fichiers que le navigateur doit garder en mémoire. Quand la page est hors ligne, il convient d'utiliser une API comme Web Storage pour garder en mémoire des données, comme des e-mails à envoyer une fois la connexion rétablie, dans le cas d'un webmail.

## L'audio et la vidéo

Les éléments <audio> et <video> permettent de jouer des sons et d'exécuter des vidéos, le tout nativement, c'est-à-dire sans plugins tels que Flash, QuickTime ou même Windows Media Player.

**L'audio**

Les éléments <audio> et <video> se ressemblent fortement.Ils sont représentés par le même objet, à savoir HTMLMediaElement. Comme ils dérivent du même objet, ils en possèdent les propriétés et méthodes.

L'insertion d'un élément <audio> est simple :

<audio id="audioPlayer" src="hype\_home.mp3"></audio>.

Ce bout de code suffit à insérer un lecteur audio qui lira le son hype\_home.mp3. Mais nous n'allons pas utiliser l'attribut src, mais plutôt deux éléments <source> :

<audio id="audioPlayer">

<source src="hype\_home.ogg">

<source src="hype\_home.mp3">

</audio>

De cette manière, si le navigateur est capable de lire le format .ogg, il le fera. Sans quoi, il lira le format .mp3.

Pour afficher un contrôleur de lecteur, il faut utiliser l'attribut booléen controls :

<audio controls="controls"></audio>

**Contrôles simples**

Ici, nous allons créer notre propre contrôleur de lecture. Voyons pour commencer comment recréer les boutons « Play », « Pause » et « Stop ». On commence par accéder à l'élément :

var player = document.querySelector('#audioPlayer');

Si on veut lancer la lecture, on utilise la méthode play() :

player.play();

Si on veut faire une pause, c'est la méthode pause() :

player.pause();

Par contre, il n'y a pas de méthode stop(). Si on appuie sur un bouton « Stop », la lecture s'arrête et se remet au début. Pour ce faire, il suffit de faire « Pause » et d'indiquer que la lecture doit se remettre au début, avec la propriété currentTime, exprimée en secondes :

player.pause();

player.currentTime = 0;

On va créer un petit lecteur, dont voici le code HTML de base :

<audio id="audioPlayer">

<source src="hype\_home.ogg">

<source src="hype\_home.mp3">

</audio>

<button class="control" onclick="play('audioPlayer', this)">Play</button>

<button class="control" onclick="resume('audioPlayer')">Stop</button>

Deux boutons ont été placés : le premier est un bouton « Play » et « Pause » en même temps (comme sur la plupart des lecteurs modernes), et le second permet de stopper et de rembobiner la lecture. Voici les fonctions play et resume :

function play(idPlayer, control) {

var player = document.querySelector('#' + idPlayer);

if (player.paused) {

player.play();

control.textContent = 'Pause';

} else {

player.pause();

control.textContent = 'Play';

}

}

function resume(idPlayer) {

var player = document.querySelector('#' + idPlayer);

player.currentTime = 0;

player.pause();

}

**Contrôle du volume**

L'intensité sonore se règle avec la propriété volume sur une échelle allant de 0 à 1. Si le volume est à 0, il est muet, et s'il est à 1, il est à fond. Pour le diminuer de moitié, on mettra 0,5. On va faire un système très simple : cinq barres verticales cliquables qui permettent de choisir un niveau sonore prédéfini :

<span class="volume">

<a class="stick1" onclick="volume('audioPlayer', 0)"></a>

<a class="stick2" onclick="volume('audioPlayer', 0.3)"></a>

<a class="stick3" onclick="volume('audioPlayer', 0.5)"></a>

<a class="stick4" onclick="volume('audioPlayer', 0.7)"></a>

<a class="stick5" onclick="volume('audioPlayer', 1)"></a>

</span>

Et la fonction associée :

function volume(idPlayer, vol) {

var player = document.querySelector('#' + idPlayer);

player.volume = vol;

}

**Barre de progression et timer**

Le HTML5 introduit un nouvel élément destiné à afficher une progression : l'élément <progress>. Il n'est toutefois utilisable qu'avec Firefox et Chrome. Mais nous n'allons pas l'utiliser ici, car cet élément n'est pas facilement personnalisable avec du CSS. Nous allons donc créer une barre de progression « à la main », avec des <div> et quelques calculs de pourcentages. Ajoutons ce code HTML après l'élément <audio> :

<div>

<div id="progressBarControl">

<div id="progressBar">Pas de lecture</div>

</div>

</div>

**Analyser la lecture**

Un élément HTMLMediaElement possède toute une série d'événements pour analyser et agir sur le lecteur. L'événement ontimeupdate va nous être utile pour détecter quand le média est en train d'être joué par le lecteur. Cet événement est déclenché continuellement pendant la lecture. Ajoutons donc cet événement sur notre élément <audio> :

<audio id="audioPlayer" ontimeupdate="update(this)">

Et commençons à coder la fonction update() :

function update(player) {

var duration = player.duration; // Durée totale

var time = player.currentTime; // Temps écoulé

var fraction = time / duration;

var percent = Math.ceil(fraction \* 100);

var progress = document.querySelector('#progressBar');

progress.style.width = percent + '%';

progress.textContent = percent + '%';

}

L'idée est de récupérer le temps écoulé et de calculer un pourcentage de manière à afficher la barre de progression (qui fait 100 % de large). Donc, si la chanson dure dix minutes et qu'on en est à une minute de lecture, on a lu 10 %. La propriété duration sert à récupérer la durée totale du média. Le calcul est simple : on divise le temps écoulé par la durée totale et on multiplie par 100. Comme ça ne tombera certainement pas juste, on arrondit avec Math.ceil(). Une fois le pourcentage récupéré, on définit la largeur de la barre de progression, et on affiche le pourcentage à l'intérieur.

**Améliorations**

L'interface réalisée précédemment est fonctionnelle, mais rudimentaire. Deux améliorations principales sont possibles :

* Afficher le temps écoulé.
* Rendre la barre de progression cliquable.

**Afficher le temps écoulé**

Il suffit d'utiliser la propriété currentTime. Le souci est que currentTime retourne le temps écoulé en secondes avec ses décimales. Il est donc possible de voir s'afficher un temps de lecture de 4,133968 secondes. Il convient donc de faire quelques opérations pour rendre ce nombre compréhensible :

function formatTime(time) {

var hours = Math.floor(time / 3600);

var mins = Math.floor((time % 3600) / 60);

var secs = Math.floor(time % 60);

if (secs < 10) {

secs = "0" + secs;

}

if (hours) {

if (mins < 10) {

mins = "0" + mins;

}

return hours + ":" + mins + ":" + secs; // hh:mm:ss

} else {

return mins + ":" + secs; // mm:ss

}

}

On peut donc ajouter ceci à notre fonction update() :

document.querySelector('#progressTime').textContent = formatTime(time);

Et modifier la barre de progression pour y ajouter un <span> dans lequel s'affichera le temps écoulé :

<div id="progressBarControl">

<div id="progressBar">Pas de lecture</div>

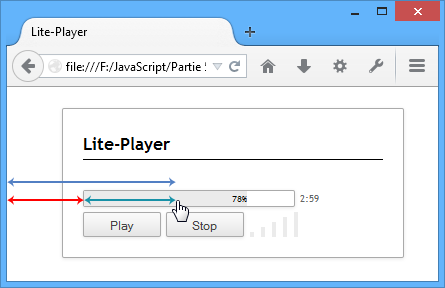
</div>

<span id="progressTime">00:00</span>

**Rendre la barre de progression cliquable**

Si on clique sur la barre de progression, le comportement attendu est la lecture du fichier audio à partir de cet endroit. Il va donc falloir calculer l'endroit où on a cliqué et positionner la lecture en conséquence, avec la propriété currentTime.

Pour savoir où l'on a cliqué au sein d'un élément, il faut connaître deux choses : les coordonnées de la souris et les coordonnées de l'élément. Ces coordonnées sont calculées à partir du coin supérieur gauche de la page :



Pour connaître la distance représentée par la flèche turquoise, il suffit de soustraire la distance représentée par la flèche rouge (la position de la barre sur l'axe des X) à la distance représentée par la flèche bleue (la position X du curseur de la souris). L'exemple ici ne permet que de reculer dans la lecture, puisque seule la barre de progression est cliquable. Il est bien évidemment possible de coder un système pour avancer dans la lecture, en rendant cliquable le conteneur de la barre de progression.

**Récupérer les coordonnées du curseur de la souris**

Nous allons créer la fonction getMousePosition() qui recevra comme paramètre un événement et qui retournera les positions X et Y du curseur :

function getMousePosition(event) {

return {

x: event.pageX,

y: event.pageY

};

}

Les propriétés pageX et pageY de l'objet event permettent respectivement de récupérer les positions sur l'axe des X et sur l'axe des Y.

**Récupérer les coordonnées d'un élément**

Comme l'élément n'est pas positionné de façon absolue, il n'est pas possible de connaître les coordonnées de son coin supérieur gauche via le CSS. Il va donc falloir calculer le décalage entre lui et son élément parent, puis le décalage entre cet élément parent et son parent… et ainsi de suite, jusqu'à arriver à l'élément racine, c'est-à-dire <html> :

function getPosition(element){

var top = 0, left = 0;

do {

top += element.offsetTop;

left += element.offsetLeft;

} while (element = element.offsetParent);

return { x: left, y: top };

}

**On clique !**

Maintenant que nous avons nos deux fonctions getMousePosition() et getPosition(), nous pouvons écrire la fonction clickProgress(), qui sera exécutée dès que l'internaute cliquera sur la barre de progression :

function clickProgress(idPlayer, control, event) {

var parent = getPosition(control); // La position absolue de la progressBar

var target = getMousePosition(event); // L'endroit de la progressBar où on a cliqué

var player = document.querySelector('#' + idPlayer);

var x = target.x - parent.x;

var wrapperWidth = document.querySelector('#progressBarControl').offsetWidth;

var percent = Math.ceil((x / wrapperWidth) \* 100);

var duration = player.duration;

player.currentTime = (duration \* percent) / 100;

}

On récupère la distance x, qui est la distance entre le bord gauche de la barre et l'endroit où on a cliqué. On divise x par la largeur totale du conteneur de la barre de progression (avec offsetWidth) et on multiplie par 100 pour obtenir un pourcentage. Ensuite, on calcule le currentTime en multipliant le temps total de la chanson par le pourcentage, le tout divisé par 100.

Et n'oublions pas de modifier le code HTML en conséquence :

<div id="progressBar" onclick="clickProgress('audioPlayer', this, event)">Pas de lecture</div>

**La vidéo**

Le principe de fonctionnement est exactement le même que pour les lectures audio. L'élément <video> possède toutefois quelques propriétés en plus :

|  |  |
| --- | --- |
| **Propriété** | **Description** |
| height | Hauteur de la zone de lecture. |
| width | Largeur de la zone de lecture. |
| poster | Récupère l'attribut poster. |
| videoHeight | La hauteur de la vidéo. |
| videoWidth | La largeur de la vidéo. |

En dehors de ça, la création et la personnalisation de la lecture d'une vidéo est rigoureusement identique à celle d'une piste audio. Il peut être utile d'utiliser un framework JavaScript destiné à la lecture d'éléments <audio> et <video> afin se faciliter la vie. De plus, ce genre de framework propose généralement une solution en Flash si le navigateur n'est pas à la hauteur du HTML5.

## L'élément Canvas

L'élément <canvas> est une zone dans laquelle il va être possible de dessiner au moyen du JavaScript. Cet élément fait son apparition dans la spécification HTML5.

**Premières manipulations**

La première chose à faire est d'insérer le canvas :

<canvas id="canvas" width="150" height="150">

<p>Désolé, votre navigateur ne supporte pas Canvas. Mettez-vous à jour</p>

</canvas>

Dès que c'est fait, on accède au canvas :

var canvas = document.querySelector('#canvas');

var context = canvas.getContext('2d');

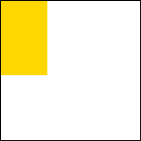
Une fois qu'on a le canvas, il faut accéder à ce qu'on appelle son contexte, avec getContext(). Il n'y a pour l'instant qu'un seul contexte disponible : la deux dimensions (2D). Il est prévu que les navigateurs gèrent un jour la 3D, mais cela reste expérimental à l'heure actuelle.

**Principe de fonctionnement**

Dessiner avec Canvas se fait par le biais de coordonnées. Le coin supérieur gauche du canvas est de coordonnées (0,0). Si on descend ou qu'on va vers la droite, on augmente les valeurs. Traçons un rectangle de 50 sur 80 pixels :

context.fillStyle = "gold";

context.fillRect(0, 0, 50, 80);



Dans un premier temps, on choisit une couleur avec fillStyle. Puis, avec fillRect(), on trace un rectangle. Les deux premiers paramètres sont les coordonnées du sommet supérieur gauche du rectangle que nous voulons tracer. Le troisième paramètre est la largeur du rectangle, et le quatrième est la hauteur. Autrement dit : fillrect(x, y, largeur, hauteur).

Si on veut centrer ce rectangle, il faut s’appliquer à quelques calculs pour spécifier les coordonnées :

context.fillRect(50, 35, 50, 80);

On recommence tout, et on centre le rectangle. Dès que c'est fait, on ajoute un carré de 40 pixels d'une couleur semi-transparente :

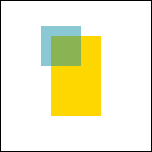
context.fillStyle = "gold";

context.fillRect(50, 35, 50, 80);

context.fillStyle = "rgba(23, 145, 167, 0.5)";

context.fillRect(40, 25, 40, 40);

La propriété fillStyle peut recevoir diverses valeurs : le nom de la couleur, un code hexadécimal (sans oublier le # devant), une valeur RGB, HSL ou HSLA ou, comme ici, une valeur RGBA. Dans le cas d'une valeur RGBA, le quatrième paramètre est l'opacité, définie sur une échelle de 0 à 1, le 0 étant transparent et le 1 opaque :



**Le fond et les contours**

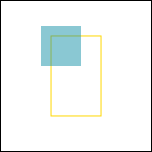
Nous avons créé des formes pleines, mais il est également possible de créer des formes creuses, avec juste un contour. Canvas considère deux types de formes : fill et stroke. Une forme fill est une forme remplie, comme nous avons fait précédemment, et une forme stroke est une forme vide pourvue d'un contour. Si pour créer un rectangle fill on utilise fillRect(), pour créer un rectangle stroke on va utiliser strokeRect() :

context.strokeStyle = "gold";

context.strokeRect(50, 35, 50, 80);

context.fillStyle = "rgba(23, 145, 167, 0.5)";

context.fillRect(40, 25, 40, 40);



Comme il s'agit d'un contour, il est possible de choisir l'épaisseur à utiliser. Cela se fait avec la propriété lineWidth : context.lineWidth = "5";.

**Effacer**

Une dernière méthode existe en ce qui concerne les rectangles : clearRect(x, y, largeur, hauteur). Cette méthode agit comme une gomme, c'est-à-dire qu'elle va effacer du canvas les pixels délimités par le rectangle. Tout comme fillRect(), on lui fournit les coordonnées des quatre sommets. clearRect() est utile pour faire des découpes au sein des formes, ou tout simplement pour effacer le contenu du canvas :

context.strokeStyle = "gold";

context.strokeRect(50, 35, 50, 80);

context.fillStyle = "rgba(23, 145, 167, 0.5)";

context.fillRect(40, 25, 40, 40);

context.clearRect(45, 40, 30, 10);

***Formes géométriques***

Canvas fournit peu de formes géométriques. Il y a uniquement le rectangle, les arcs. Mais pour compléter ce manque, Canvas dispose de chemins ainsi que de courbes de Bézier cubiques et quadratiques.

**Les chemins simples**

Les chemins vont nous permettre de créer des lignes et des polygones. Pour ce faire, il existe plusieurs méthodes : beginPath() et closePath(), moveTo(), lineTo(), stroke() et son équivalent fill().

Comme pour la création de rectangles, la création de chemins se fait par étapes successives. On commence par initier un nouveau chemin avec beginPath(). Ensuite, avec moveTo(), on déplace le « crayon » à l'endroit où on souhaite commencer le tracé : c'est le point de départ du chemin. Puis, on utilise lineTo() pour indiquer un deuxième point, un troisième, etc. Une fois tous les points du chemin définis, on applique au choix stroke() ou fill() :

context.strokeStyle = "rgb(23, 145, 167)";

context.beginPath();

context.moveTo(20, 20); // 1er point

context.lineTo(130, 20); // 2e point

context.lineTo(130, 50); // 3e

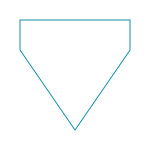
context.lineTo(75, 130); // 4e

context.lineTo(20, 50); // 5e

context.closePath(); // On relie le 5e au 1er

context.stroke();

closePath() n'est pas nécessaire. Il termine le chemin pour nous, en reliant le dernier point au premier. Si on veut une forme fermée, via stroke(), c'est assez pratique. Par contre, si on veut remplir la forme avec fill(), la forme sera fermée automatiquement, donc closePath() est inutile.



**Les arcs**

En plus des lignes droites, il est possible de tracer des arcs de cercle, avec la méthode arc(x, y, rayon, angleDepart, angleFin, sensInverse). Les angles sont exprimés en radians. Avec les arcs, x et y sont les coordonnées du centre de l'arc. Les paramètres angleDepart et angleFin définissent les angles de début et de fin de l'arc. Pour rappel, pour obtenir des radians il suffit de multiplier les degrés par π divisé par 180 : (Math.PI / 180) \* degrees.

context.beginPath(); // Le cercle extérieur

context.arc(75, 75, 50, 0, Math.PI \* 2); // Ici le calcul est simplifié

context.stroke();

context.beginPath(); // La bouche, un arc de cercle

context.arc(75, 75, 40, 0, Math.PI); // Ici aussi

context.fill();

context.beginPath(); // L'œil gauche

context.arc(55, 70, 20, (Math.PI / 180) \* 220, (Math.PI / 180) \* 320);

context.stroke();

context.beginPath(); // L'œil droit

context.arc(95, 70, 20, (Math.PI / 180) \* 220, (Math.PI / 180) \* 320);

context.stroke();



Pour chaque arc, il est plus propre et plus facile de commencer un nouveau chemin avec beginPath().

**Utilisation de moveTo()**

moveTo() permet de déplacer le « crayon » à l'endroit où l'on souhaite commencer un chemin. Mais cette méthode peut aussi être utilisée pour effectuer des « levées de crayon » au sein d'un même chemin :

context.beginPath(); // La bouche, un arc de cercle

context.arc(75, 75, 40, 0, Math.PI);

context.fill();

context.beginPath(); // Le cercle extérieur

context.arc(75, 75, 50, 0, Math.PI \* 2);

context.moveTo(41, 58); // L'œil gauche

context.arc(55, 70, 20, (Math.PI / 180) \* 220, (Math.PI / 180) \* 320);

context.moveTo(81, 58); // L'œil droit

context.arc(95, 70, 20, (Math.PI / 180) \* 220, (Math.PI / 180) \* 320);

context.stroke();

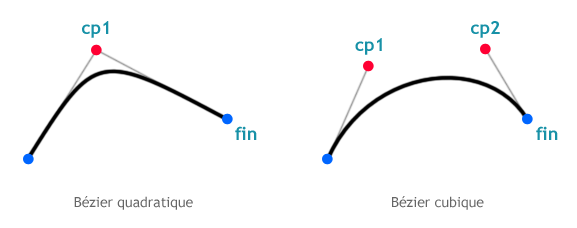


Si on retire les deux moveTo(), on obtient quelque chose comme ça :



**Les courbes de Bézier**

Il est possible de réaliser des courbes, par le biais de courbes de Bézier. Deux types de courbes sont disponibles : cubique et quadratique. Les courbes sont définies par les coordonnées des tangentes qui servent à la construction des courbes. Voici les deux types de courbes, avec les tangentes colorées en gris :



Une courbe quadratique sera dessinée par quadraticCurveTo(), alors qu'une courbe cubique le sera par bezierCurveTo() :

quadraticCurveTo(cp1X, cp1Y, x, y)

bezierCurveTo(cp1X, cp1Y, cp2X, cp2Y, x, y)

Les courbes sont définies par leurs points d'arrivée (x et y) et par les points de contrôle. Dans le cas d'une courbe de Bézier cubique, deux points sont nécessaires. La difficulté des courbes de Bézier est de connaître les valeurs utiles pour les points de contrôle. C'est d'autant plus complexe qu'on ne voit pas en temps réel ce qu'on fait. Ce genre de courbes est donc puissant, mais complexe à mettre en œuvre. Il existe toutefois des plugins qui permettent de convertir des dessins vectoriels en instructions Canvas. C'est par exemple le cas de Ai2Canvas, un plugin pour Adobe Illustrator.

Voici une variante du logo JavaScript à partir d'un rectangle arrondi :

context.beginPath();

context.moveTo(131, 119);

context.bezierCurveTo(131, 126, 126, 131, 119, 131);

context.lineTo(30, 131);

context.bezierCurveTo(23, 131, 18, 126, 18, 119);

context.lineTo(18, 30);

context.bezierCurveTo(18, 23, 23, 18, 30, 18);

context.lineTo(119, 18);

context.bezierCurveTo(126, 18, 131, 23, 131, 30);

context.lineTo(131, 119);

context.closePath();

context.fillStyle = "rgb(23, 145, 167)";

context.fill();

context.font = "68px Calibri,Geneva,Arial";

context.fillStyle = "white";

context.fillText("js", 84, 115);



**Images et textes**

Il est possible d'insérer des images au sein d'un canvas. Pour ce faire, on utilisera la méthode drawImage(image, x, y), mais attention : pour qu'une image puisse être utilisée, elle doit au préalable être accessible via un objet Image ou un élément <img />. Il est également possible d'insérer un canvas dans un canvas. En effet, le canvas que l'on va insérer est considéré comme une image.

Insérons l'âne Zozor du Site du Zéro au sein du canvas :

var zozor = new Image();

zozor.src = 'zozor.png'; // Image de 80x80 pixels

context.drawImage(zozor, 35, 35);

On aurait pu récupérer une image déjà présente dans la page : <img id="myZozor" src="zozor.png" alt="Zozor assis" />.

var zozor = document.querySelector('#myZozor');

context.drawImage(zozor, 35, 35);

Attention aux grandes images : si l'image est trop longue à charger, elle sera affichée de façon saccadée au sein du canvas. Une solution est d'utiliser onload pour déclencher le dessin de l'image une fois qu'elle est chargée :

var zozor = new Image();

zozor.src = 'zozor.png';

zozor.addEventListener('load', function() {

context.drawImage(zozor, 35, 35);

});



**Mise à l'échelle**

drawImage(image, x, y, largeur, hauteur) possède deux paramètres supplémentaires facultatifs : largeur et hauteur, qui permettent de définir la largeur et la hauteur que l'image occupera une fois incrustée dans le canvas. Si la diminution de la taille des images ne pose pas trop de problèmes, il faut éviter de les agrandir, au risque de voir vos images devenir floues :

context.drawImage(zozor, 35, 35, 40, 40);

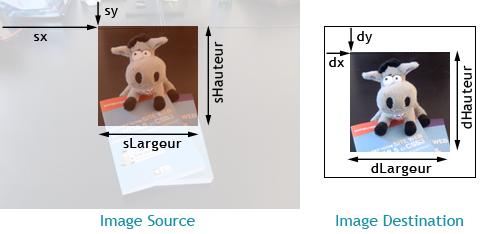
Ici, l'image est réduite de moitié, puisque de base elle fait 80 pixels sur 80 pixels.

**Recadrage**

Quatre paramètres supplémentaires et optionnels s'ajoutent à drawImage(). Ils permettent de recadrer l'image, c'est-à-dire de prélever une zone rectangulaire au sein de l'image afin de la placer dans le canvas :

drawImage(image, sx, sy, sLargeur, sHauteur, dx, dy, dLargeur, dHauteur)

Les paramètres commençant par s indiquent la source, c'est-à-dire l'image, ceux commençant par d indiquent la destination, autrement dit le canvas :



var zozor = document.querySelector('#plush');

context.drawImage(zozor, 99, 27, 100, 100, 25, 25, 100, 100);

**Les patterns**

Pour faire se répéter une image pour, par exemple, créer un fond, il est possible de faire une double boucle for et d'insérer plusieurs fois la même image. Mais il y a plus simple : les **patterns**. On parle aussi de motifs en français. Un pattern est une image qui se répète comme un papier peint. Pour en créer un, on utilise la méthode createPattern(image, type). Le premier argument est l'image à utiliser, et le deuxième est le type de pattern. Différents types existent, mais seul repeat semble reconnu par la plupart des navigateurs :

var zozor = new Image();

zozor.src = 'zozor.png';

zozor.addEventListener('load', function() {

var pattern = context.createPattern(zozor, 'repeat');

context.fillStyle = pattern;

context.fillRect(0, 0, 150, 150);

});



La façon de procéder est un peu étrange, puisqu'il faut passer le pattern à fillStyle, et ensuite créer un rectangle plein qui recouvre l'entièreté du canvas. En clair, il s'agit de créer un rectangle avec une image qui se répète comme fond. Nous devons absolument passer par l’événement load, sinon le pattern ne s'affichera pas correctement si l'image n'est pas chargée.

**Le texte**

Pour écrire du texte au sein d'un canvas, il y a les méthodes fillText() et strokeText(), secondées par la propriété font, qui permet de définir le style du texte :

context.fillStyle = "rgba(23, 145, 167, 1)";

context.fillRect(50, 50, 50, 50);

context.font = "bold 22pt Calibri,Geneva,Arial";

context.fillStyle = "#fff";

context.fillText("js", 78, 92);



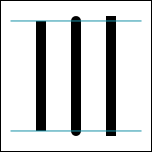
Les méthodes fillStyle et strokeStyle sont toujours utilisables, puisque les textes sont considérés comme des formes au même titre que les rectangles ou les arcs. La propriété font reçoit des informations sur la police à utiliser, à l'exception de la couleur, qui est gérée par strokeStyle et fillStyle. Dans l'exemple qui va suivre, nous allons utiliser un texte en Calibri, de 22 points et mis en gras. fillText() reçoit trois paramètres : le texte et les positions x et y de la ligne d'écriture du texte :



Un quatrième paramètre peut être ajouté : la largeur maximale que le texte doit utiliser.

**Les styles de lignes**

La propriété lineCap permet de définir la façon dont les extrémités des chemins sont affichées. Trois valeurs sont admises : butt, celle par défaut, round et square. Voici trois lignes, chacune avec un lineCap différent :



lineCap s'utilise de la même façon que linewidth :

context.beginPath();

context.lineCap = 'round';

context.moveTo(75, 20);

context.lineTo(75, 130);

context.stroke();

**Les intersections**

Pour gérer la façon dont les angles des chemins sont affichés, il faut utiliser lineJoin. Cette propriété reçoit elle aussi trois valeurs différentes : round, bevel et miter, ce dernier étant la valeur par défaut :



**Les dégradés**

Canvas propose deux types de dégradés : linéaire et radial. Pour créer un dégradé, on commence par créer un objet canvasGradient que l'on va assigner à fillStyle. Pour créer un tel objet, on utilise au choix createLinearGradient() ou createRadialGradient().

**Dégradés linéaires**

On a besoin de quatre paramètres pour créer un dégradé linéaire :

createLinearGradient(debutX, debutY, finX, finY)

debutX et debutY sont les coordonnées du point de départ du dégradé, et finX et finY sont les coordonnées de fin :

var linear = new context.createLinearGradient(0, 0, 150, 150);

context.fillStyle = linear;

Il manque cependant les informations sur les couleurs. Cela s’ajoute avec addColorStop(position, couleur). Le premier paramètre, position, est une valeur comprise entre 0 et 1. C'est la position relative de la couleur par rapport au dégradé. Si on met 0.5, la couleur commencera au milieu :

var linear = context.createLinearGradient(0, 0, 0, 150);

linear.addColorStop(0, 'white');

linear.addColorStop(1, '#1791a7');

context.fillStyle = linear;

context.fillRect(20, 20, 110, 110);

Pour modifier l'inclinaison du dégradé, il faut modifier les paramètres de createLinearGradient(). Par exemple, si on met createLinearGradient(0, 0, 150, 150), la fin du dégradé sera dans le coin inférieur droit, et donc incliné à 45 degrés. Il est possible de mettre plus de deux avec addColorStop(). Voici un exemple avec quatre :

var linear = context.createLinearGradient(0, 0, 0, 150);

linear.addColorStop(0, 'white');

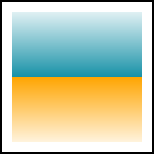
linear.addColorStop(0.5, '#1791a7');

linear.addColorStop(0.5, 'orange');

linear.addColorStop(1, 'white');

context.fillStyle = linear;

context.fillRect(10, 10, 130, 130);



**Dégradés radiaux**

En ce qui concerne les dégradés radiaux, il faut six paramètres :

createRadialGradient(centreX, centreY, centreRayon, finX, finY, finRayon)

Un dégradé radial est défini par deux choses : un premier cercle (le centre) qui fait office de point de départ et un second qui fait office de fin. Ce qui est pratique, c'est que les deux cercles n'ont pas besoin d'avoir la même origine, ce qui permet d'orienter le dégradé :

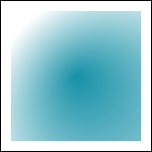
var radial = context.createRadialGradient(75, 75, 0, 130, 130, 150);

radial.addColorStop(0, '#1791a7');

radial.addColorStop(1, 'white');

context.fillStyle = radial;

context.fillRect(10, 10, 130, 130);



Ici, le cercle du centre estau centre du canvas, et celui de fin en bas à droite. Grâce aux dégradés radiaux, il est possible de créer des bulles assez facilement. La seule condition est que la couleur de fin du dégradé soit transparente, ce qui nécessite l'utilisation d'une couleur RGBA ou HSLA :

var radial1 = context.createRadialGradient(0, 0, 10, 100, 20, 150); // fond

radial1.addColorStop(0, '#ddf5f9');

radial1.addColorStop(1, '#ffffff');

var radial2 = context.createRadialGradient(75, 75, 10, 82, 70, 30); // bulle orange

radial2.addColorStop(0, '#ffc55c');

radial2.addColorStop(0.9, '#ffa500');

radial2.addColorStop(1, 'rgba(245,160,6,0)');

var radial3 = context.createRadialGradient(105, 105, 20, 112, 120, 50); // bulle turquoise

radial3.addColorStop(0, '#86cad2');

radial3.addColorStop(0.9, '#61aeb6');

radial3.addColorStop(1, 'rgba(159,209,216,0)');

context.fillStyle = radial1;

context.fillRect(10, 10, 130, 130);

context.fillStyle = radial2;

context.fillRect(10, 10, 130, 130);

context.fillStyle = radial3;

context.fillRect(10, 10, 130, 130);

Ce qui donne un dégradé de fond avec deux bulles de couleur :



**Opérations**

La méthode save() a pour fonction de sauvegarder l'état graphique du canvas, c'est-à-dire les informations concernant les styles appliqués au canvas. Ces informations sont fillStyle, strokeStyle, lineWidth, lineCap, lineJoin ainsi que translate() et rotate().À chaque appel de la méthode save(), l'état graphique courant est sauvegardé dans une pile. Pour restaurer l'état précédent, il faut utiliser restore().

**Les translations**

La translation permet de déplacer le repaire d'axes du canvas. L'idée est de placer le point (0,0) à l'endroit où l'on souhaite dessiner une forme. De cette manière, on dessine la forme sans se soucier des calculs de son emplacement, ce qui peut se révéler utile quand on insère des formes complexes. Une fois que les formes sont dessinées, on replace les axes à leur point d'origine. save() et restore() prennent en compte les translations.

Les translations se font avec la méthode translate(x, y). x est l'importance du déplacement sur l'axe des abscisses et y sur l'axe des ordonnées : les valeurs peuvent donc être négatives.

context.save();

context.translate(40, 40);

context.fillStyle = "teal";

context.fillRect(0, 0, 50, 50);

context.restore();

context.fillStyle = "orange";

context.fillRect(0, 0, 50, 50);



On commence par sauvegarder l'état du canvas. Ensuite, on déplace l'origine des axes au point (40,40) et on y dessine un carré bleu-gris. Dès que c'est fait, on restaure l'état, ce qui a pour conséquence de replacer l'origine des axes au point (0,0) du canvas. Là, on dessine le carré orange. Grâce à la translation, on a pu laisser (0,0) comme coordonnées de fillRect().

**Les rotations**

Les rotations permettent d'appliquer une rotation aux axes du canvas. Le canvas tourne autour de son point d'origine (0,0). La méthode rotate() reçoit un seul paramètre : l'angle de rotation spécifié en radians. Il est possible de combiner une rotation et une translation :

context.translate(75,75);

context.fillStyle = "teal";

context.rotate((Math.PI / 180) \* 45);

context.fillRect(0, 0, 50, 50);

context.fillStyle = "orange";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);

context.fillStyle = "teal";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);

context.fillStyle = "orange";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);



On place l'origine des axes au centre du canvas avec translate(). On opère une première rotation de 45 degrés et on dessine un carré bleu-gris. Ensuite, on fait une deuxième rotation de 90 degrés et on dessine un carré orange. On continue de tourner les axes de 90 degrés et on dessine un nouveau carré bleu-gris. On fait une dernière rotation et on dessine un carré orange.

**Animations**

La gestion des animations avec Canvas est quasi inexistante. En effet, Canvas ne propose rien pour animer les formes, les déplacer, les modifier. Pour arriver à créer une animation avec Canvas, il faut :

* Dessiner une image.
* Effacer tout.
* Redessiner une image, légèrement modifiée.
* Effacer tout.
* Redessiner une image, légèrement modifiée.
* Et ainsi de suite.

En clair, il suffit d'appeler une fonction qui, toutes les x secondes, va redessiner le canvas. Il est également possible d'exécuter des fonctions à la demande de l'utilisateur.

**Une question de « framerate »**

« Framerate » est un mot anglais pour évoquer le nombre d'images affichées par seconde. Les standards actuels définissent que chaque animation est censée, en théorie, afficher un framerate de 60 images par seconde pour paraître fluide pour l’œil humain. Parfois, ces 60 images peuvent ne pas être toutes affichées en une seconde à cause d'un manque de performances, on appelle cela une baisse de framerate et cela est généralement perçu par l'œil humain comme étant un ralenti saccadé. Ce problème est peu appréciable et est malheureusement trop fréquent avec les fonctions setTimeout() et setInterval(), qui n'ont pas été conçues à l'origine pour ce genre d'utilisations.

Une solution à ce problème a été développée : requestAnimationFrame(). À chacune de ses exécutions, cette fonction va déterminer à quel moment elle doit se redéclencher de manière à garder un framerate de 60 images par seconde.

**Un exemple concret**

Reprenons le canvas que nous venons de réaliser :



En nous basant sur son code, nous allons faire tourner le dessin dans le sens des aiguilles d'une montre. Pour commencer, il faut créer une fonction qui sera appelée par window.requestAnimationFrame(). Il s'agira de la fonction draw(angle). Cette fonction efface le canvas et le redessine avec un angle de rotation incrémenté de quelques degrés :

window.addEventListener('load', function() {

var canvas = document.querySelector('#canvas');

var context = canvas.getContext('2d');

function draw(angle) {

context.save();

context.clearRect(0, 0, 150, 150);

context.translate(75,75);

context.fillStyle = "teal";

context.rotate((Math.PI / 180) \* (45 + angle));

context.fillRect(0, 0, 50, 50);

context.fillStyle = "orange";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);

context.fillStyle = "teal";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);

context.fillStyle = "orange";

context.rotate(Math.PI / 2);

context.fillRect(0, 0, 50, 50);

context.restore();

angle = angle + 2;

if (angle >= 360) angle = 0;

window.requestAnimFrame(function() { draw(angle) });

}

draw(0);

});

La variable angle représente le décalage. Lors du premier appel de draw(), le décalage vaut 0. Après le premier appel, on incrémente angle de 2 degrés, et donc, lors du prochain appel, tout le canvas sera dessiné avec un décalage de 2 degrés. On réincrémente de 2, et on redessine. Ainsi de suite, pour donner l'illusion que toute la forme bouge, alors qu'on ne fait que spécifier un angle de rotation de départ qui va en augmentant.

Les possibilités d'animation de Canvas sont toutes basées sur le même principe : window.requestAnimationFrame(). Ici, il s'agissait de créer un effet de rotation, mais il est possible de créer une courbe qui s'étire (une courbe de Bézier pour laquelle on incrémente les valeurs), d'animer une balle qui rebondit etc…

Enfin, il existe des frameworks qui permettent de simplifier le dessin, et même d'ajouter des événements aux "formes".

## L'API File

L'API que nous allons découvrir n'est pas utilisable seule. Autrement dit, elle nécessite d'être appelée par diverses technologies permettant son accès et lui fournissant les fichiers qu'elle peut manipuler. Cette API a été conçue de cette manière afin d'éviter que ce ne soit nous, développeurs, qui choisissions quel fichier lire sur l'ordinateur du client. Si cette sécurité n’existait pas, les conséquences pourraient être désastreuses. L'API File ne permet, actuellement, d'écrire un fichier stocké sur l'ordinateur d'un client. Nous ne pourrons que le lire ou bien l'uploader pour le modifier sur un serveur, l'écriture d'un fichier sur l'ordinateur du client est encore en cours d'étude à l'heure où nous écrivons ces lignes.

Afin de pouvoir utiliser notre API, il va nous falloir définir comment les fichiers vont pouvoir être choisis par l'utilisateur. La solution la plus simple pour commencer est l'utilisation d'une balise <input type="file" />, qui va nous permettre d'accéder aux propriétés des fichiers sélectionnés par l'utilisateur. Ces propriétés constituent une partie de l'API File.

Pour accéder au fichier il va nous falloir passer par la propriété files de notre balise <input>. Celle-ci va nous permettre d'accéder à une collection d'objets utilisables de la même manière qu'un tableau, chaque objet représentant un fichier. Une collection d'objets parce que le HTML5 a ajouté la possibilité de choisir plusieurs fichiers pour un seul et même input. Il suffit d'y ajouter l'attribut multiple pour que cela soit autorisé au client :

<input id="file" type="file" multiple />

La propriété files est la même pour tous, que la sélection de fichiers soit multiple ou non. Si nous voulons lire le fichier d'un <input> ne gérant qu'un seul fichier, alors nous utiliserons files[0] et non pas file.

Essayons maintenant d'obtenir le nom du fichier sélectionné grâce à la propriété name contenue dans chaque objet de type File :

document.querySelector('#file').addEventListener('change', function() {

alert(this.files[0].name);

});

**L’objet Blob**

Un objet de type Blob est une structure représentant des données binaires disponibles uniquement en lecture seule. Ces objets représentent les données binaires du fichier ciblé. Les objets Blob possèdent deux propriétés nommées size et type qui permettent respectivement de récupérer la taille en octets des données manipulées par le Blob ainsi que leur type MIME.

**L'objet File**

Les objets File possèdent un nom bien représentatif puisqu'ils permettent de manipuler les fichiers. Leur particularité est qu'ils héritent des propriétés et méthodes des objets Blob. En plus des propriétés et méthodes des objets Blob, les objets File possèdent deux propriétés supplémentaires qui sont name pour obtenir le nom du fichier et lastModifiedDate pour obtenir la date de la dernière modification du fichier (sous forme d'objet Date). On peut également aborder leur lecture grâce aux objets de type FileReader.

**Lire les fichiers**

Nous allons aborder la lecture des fichiers grâce à l'objet FileReader. Son instanciation s'effectue sans aucun argument :

var reader = new FileReader();

Cet objet permet la lecture asynchrone de fichiers, grâce à trois méthodes différentes :

|  |  |
| --- | --- |
| **Nom** | **Description** |
| readAsArrayBuffer() | Stocke les données dans un objet de type ArrayBuffer. Ces objets ont été conçus pour permettre l'écriture et la lecture de données binaires directement dans leur forme native. Ils sont surtout utilisés dans des domaines exigeants tels que le WebGL. |
| readAsDataURL() | Les données sont converties dans un format nommé DataURL. Ce format consiste à convertir toutes les données binaires d'un fichier en base64 pour ensuite stocker le résultat dans une chaîne de caractères. Cette dernière est complétée par la spécification du type MIME du fichier concerné. Les DataURL permettent donc de stocker un fichier sous forme d'une URL lisible par les navigateurs récents. |
| readAsText() | Les données ne subissent aucune modification, elles sont tout simplement lues puis stockées sous forme d'une chaîne de caractères. |

Ces trois méthodes prennent chacune en paramètre un argument de type Blob ou File. La méthode readAsText() possède un argument supplémentaire (et facultatif) permettant de spécifier l'encodage du fichier :

reader.readAsText(file, 'UTF-8');

reader.readAsText(file, 'ISO-8859-1');

Nous avons bien précisé que la lecture d'un fichier est asynchrone. Il faut donc partir du principe que nous allons avoir plusieurs événements à notre disposition. Ces événements diffèrent peu de ceux que l'on rencontre avec la seconde version de l'objet XMLHttpRequest :

|  |  |
| --- | --- |
| **Nom** | **Description** |
| loadstart | La lecture vient de commencer. |
| progress | Tout comme avec les objets XHR, l'événement progress se déclenche à intervalles réguliers durant la progression de la lecture. Il fournit, lui aussi, un objet en paramètre possédant deux propriétés, loaded et total, indiquant respectivement le nombre d'octets lus et le nombre d'octets à lire en tout. |
| load | La lecture vient de se terminer avec succès. |
| loadend | La lecture vient de se terminer (avec ou sans succès). |
| abort | Se déclenche quand la lecture est interrompue (avec la méthode abort() par exemple). |
| error | Se déclenche quand une erreur a été rencontrée. La propriété error contiendra alors un objet de type FileError pouvant fournir plus d'informations. |

Une fois les données lues, il ne reste plus qu'à les récupérer dans la propriété result. Ainsi, afin de lire un fichier texte, il suffit de suivre l’exemple suivant :

<input id="file" type="file" />

<script>

var fileInput = document.querySelector('#file');

fileInput.addEventListener('change', function() {

var reader = new FileReader();

reader.addEventListener('load', function() {

alert('Contenu du fichier "' + fileInput.files[0].name + '" :\n\n' + reader.result);

});

reader.readAsText(fileInput.files[0]);

});

</script>

Pour finir sur la lecture des fichiers, l'objet FileReader possède aussi une propriété readyState permettant de connaître l'état de la lecture. Il existe trois états différents représentés par des constantes spécifiques aux objets FileReader :

|  |  |  |
| --- | --- | --- |
| **Constante** | **Valeur** | **Description** |
| EMPTY | 0 | Aucune donnée n'a encore été chargée. |
| LOADING | 1 | Les données sont en cours de chargement. |
| DONE | 2 | Toutes les données ont été chargées. |

Tout comme avec un objet XHR, nous pouvons vérifier l'état de la lecture, soit avec la constante :

if (reader.readyState === reader.LOADING) {

// La lecture est en cours...

}

soit directement avec la valeur de la constante :

if (reader.readyState === 1) {

// La lecture est en cours...

}

**Mise en pratique**

Le principal intérêt de cette API réside en fait dans son utilisation avec d'autres ressources. Un petit peu plus loin dans ce chapitre, nous allons étudier comment l'utiliser conjointement avec l'objet XMLHttpRequest afin d'effectuer des uploads. Nous verrons aussi, dans un chapitre ultérieur, comment s'en servir efficacement avec le Drag & Drop.

Nous allons ici faire une mise en pratique de cette API. Le scénario est le suivant : nous souhaitons créer un site d'hébergement d'images interactif. Le principe est simple, l'utilisateur sélectionne les images qu'il souhaite uploader, elles sont alors affichées en prévisualisation sur la page et l'utilisateur n'a plus qu'à cliquer sur le bouton d'upload une fois qu'il aura vérifié qu'il a bien sélectionné les bonnes images.

Notre objectif, ici, est de créer la partie concernant la sélection et la prévisualisation des images, l'upload ne nous intéresse pas. Afin d'obtenir le résultat escompté, nous allons devoir utiliser l'API File, qui va nous permettre de lire le contenu des fichiers avant même d'effectuer un quelconque upload.

Commençons par construire la page HTML qui va accueillir notre script :

<input id="file" type="file" multiple />

<div id="prev"></div>

Il n'y a pas besoin de plus, nous avons notre balise <input> pour sélectionner les fichiers (avec l'attribut multiple afin de permettre la sélection de plusieurs fichiers) ainsi qu'une balise <div> pour y afficher les images à uploader.

Il nous faut maintenant passer au JavaScript. Commençons par mettre en place la structure principale de notre script :

(function() {

var allowedTypes = ['png', 'jpg', 'jpeg', 'gif'],

fileInput = document.querySelector('#file'),

prev = document.querySelector('#prev');

fileInput.addEventListener('change', function() {

// Analyse des fichiers et création des prévisualisations

});

})();

Ce code déclare les variables et les événements nécessaires. Il existe une variable allowedTypes qui contient un tableau listant les extensions d'images dont nous autorisons l'upload. L'analyse des fichiers peut maintenant commencer. Sachant que nous avons autorisé la sélection multiple de fichiers, nous allons devoir utiliser une boucle afin de parcourir les fichiers sélectionnés. Il nous faudra aussi vérifier quels sont les fichiers à autoriser :

fileInput.addEventListener('change', function() {

var files = this.files,

filesLen = files.length,

imgType;

for (var i = 0; i < filesLen; i++) {

imgType = files[i].name.split('.');

imgType = imgType[imgType.length - 1].toLowerCase(); // On utilise toLowerCase() pour éviter les extensions en majuscules

if (allowedTypes.indexOf(imgType) != -1) {

createThumbnail(files[i]); // Le fichier est bien une image supportée, il ne reste plus qu'à

l'afficher

}

}

});

Il nous faut maintenant afficher l'image. L'affichage d'une image, en HTML, se fait grâce à la balise <img>, or celle-ci n'accepte qu'une URL en guise de valeur pour son attribut src. Nous pourrions lui fournir l'adresse du fichier à afficher, mais nous ne connaissons que son nom, pas son chemin. La réponse se trouve dans les DataURL. En effet, nous avions précisé que les DataURL permettaient de stocker des données dans une URL, c’est exactement ce qu'il nous faut. Nous pouvons maintenant passer à la création de notre fonction createThumbnail() :

function createThumbnail(file) {

var reader = new FileReader();

reader.addEventListener('load', function() {

// Affichage de l'image

var imgElement = document.createElement('img');

imgElement.style.maxWidth = '150px';

imgElement.style.maxHeight = '150px';

imgElement.src = this.result;

prev.appendChild(imgElement);

});

reader.readAsDataURL(file);

}

Notre script complet aura donc la forme suivnate :

(function() {

function createThumbnail(file) {

var reader = new FileReader();

reader.addEventListener('load', function() {

var imgElement = document.createElement('img');

imgElement.style.maxWidth = '150px';

imgElement.style.maxHeight = '150px';

imgElement.src = this.result;

prev.appendChild(imgElement);

});

reader.readAsDataURL(file);

}

var allowedTypes = ['png', 'jpg', 'jpeg', 'gif'],

fileInput = document.querySelector('#file'),

prev = document.querySelector('#prev');

fileInput.addEventListener('change', function() {

var files = this.files,

filesLen = files.length,

imgType;

for (var i = 0; i < filesLen; i++) {

imgType = files[i].name.split('.');

imgType = imgType[imgType.length - 1];

if (allowedTypes.indexOf(imgType) != -1) {

createThumbnail(files[i]);

}

}

});

})();

**Upload de fichiers avec l'objet XMLHttpRequest**

Il était auparavant impossible d'uploader des données binaires avec l'objet XMLHttpRequest, car celui-ci ne supportait pas l'utilisation de l'objet FormData. Depuis l'arrivée de ce nouvel objet ainsi que de la deuxième version du XMLHttpRequest, cette « prouesse » est maintenant réalisable facilement. Il est très simple de créer des données binaires (grâce à un Blob) pour les envoyer sur un serveur.

Afin d'effectuer un upload de fichiers, il faut tout d'abord récupérer un objet de type File, il nous faut donc un <input> :

<input id="file" type="file" />

Il faut ajouter à cela un code JavaScript qui récupère le fichier spécifié et s'occupe de créer une requête XMLHttpRequest :

var fileInput = document.querySelector('#file');

fileInput.addEventListener('change', function() {

var xhr = new XMLHttpRequest();

xhr.open('POST', 'http://exemple.com'); // Rappelons qu'il est obligatoire d'utiliser la méthode POST quand on souhaite utiliser un FormData

xhr.addEventListener('load', function() {

alert('Upload terminé !');

});

// Upload du fichier…

});

Il suffit maintenant de passer notre objet File à un objet FormData et d'uploader ce dernier :

var form = new FormData();

form.append('file', fileInput.files[0]);

xhr.send(form);

L'upload de fichiers par le biais d'un objet XHR permet juste de simplifier les choses puisque l'on n'a plus à s'embêter à passer par le biais d'une <iframe>. II nous reste encore à afficher la progression de l'upload.

L'objet XHR possède une propriété upload donnant accès à plusieurs événements dont l'événement progress. Ce dernier fonctionne exactement de la même manière que le précédent événement progress que nous avions étudié dans le chapitre consacré à l'objet XHR :

xhr.upload.addEventListener('progress', function(e) {

e.loaded; // Nombre d'octets uploadés

e.total; // Total d'octets à uploader

});

Ainsi, il est facile de faire une barre de progression avec cet événement et la balise HTML5 <progress> :

<input id="file" type="file" />

<progress id="progress"></progress>

var fileInput = document.querySelector('#file'),

progress = document.querySelector('#progress');

fileInput.addEventListener('change', function() {

var xhr = new XMLHttpRequest();

xhr.open('POST', 'upload.html');

xhr.upload.addEventListener('progress', function(e) {

progress.value = e.loaded;

progress.max = e.total;

});

xhr.addEventListener('load', function() {

alert('Upload terminé !');

});

var form = new FormData();

form.append('file', fileInput.files[0]);

xhr.send(form);

});

## Le Drag & Drop

Grâce au HTML5, il est possible de permettre un déplacement de texte, de fichier ou d'autres éléments depuis n'importe quelle application jusqu'à notre navigateur.

**Aperçu de l'API**

Afin de rendre un élément déplaçable, il suffit d'utiliser son attribut draggable et de le mettre à true (que ce soit en HTML ou en JavaScript). (Il est possible de rencontrer des problèmes avec Firefox. En effet, ce navigateur nécessite une information supplémentaire que nous verrons plus loin.)

Parmi les huit événements que l'API Drag & Drop fournit, l'élément déplaçable peut en utiliser deux : dragstart et dragend. L'événement dragstart se déclenche lorsque l'élément ciblé commence à être déplacé. Cet événement est particulièrement utile pour initialiser certains détails utilisés tout au long du processus de déplacement. Pour cela, il nous faudra utiliser l'objet dataTransfer que nous étudierons plus loin. L'événement dragend permet de signaler à l'objet déplacé que son déplacement est terminé, que le résultat soit un succès ou non.

**Initialiser un déplacement avec l'objet dataTransfer**

L'objet dataTransfer est généralement utilisé au travers de deux événements : dragstart et drop. Cet objet permet de définir et de récupérer les informations relatives au déplacement en cours d'exécution. Ici, nous n'allons aborder l'objet dataTransfer que dans le cadre de l'initialisation d'un déplacement, son utilisation pour la fin d'un processus de drag & drop sera étudiée plus tard.

L'objet dataTransfer permet de réaliser trois actions (toutes facultatives) :

* Sauvegarder une chaîne de caractères qui sera transmise à l'élément HTML qui accueillera l'élément déplacé. La méthode à utiliser est setData().
* Définir une image utilisée lors du déplacement. La méthode concernée est setDragImage().
* Spécifier le type de déplacement autorisé avec la propriété effectAllowed. Cette propriété ayant un usage assez restreint, nous ne l'aborderons pas ici. Cela est aussi valable pour sa congénère dropEffect.

La méthode setData() prend deux arguments en paramètres. Le premier est le type MIME des données (sous forme de chaîne de caractères) que nous allons spécifier dans le deuxième argument. Précisons que le deuxième argument est obligatoirement une chaîne de caractères, ce qui signifie que le type MIME qui sera spécifié n'a que peu d'intérêt, nous utiliserons généralement le type text/plain pour des raisons de simplicité :

draggableElement.addEventListener('dragstart', function(e) {

e.dataTransfer.setData('text/plain', "Ce texte sera transmis à l'élément HTML de réception");

});

En temps normal, nous pourrions penser que méthode est inutile puisqu'il suffirait de stocker les données dans une variable plutôt que par le biais de setData(). En travaillant sur la même page oui. Cependant le Drag & Drop en HTML5 possède la faculté de s'étendre bien au-delà de notre page Web actuelle et donc de faire un glisser-déposer d'une page à une autre, que ce soit d'un onglet à un autre ou bien même d'un navigateur à un autre. Le transfert de données entre les pages Web n'étant pas possible (tout du moins pas sans « tricher »), il est utile d'utiliser la méthode setData().

Attention, l'utilisation de la méthode setData() est obligatoire avec Firefox. Cela est stupide, car nous n'avons pas forcément quelque chose à y stocker, mais nous n'avons le choix. Il faut donc utiliser le type MIME de notre choix et lui passer une chaîne de caractères vide, comme ceci : setData('text/plain', '');.

La méthode setDragImage() est extrêmement utile pour qui souhaite personnaliser l'affichage de sa page Web. Elle permet de définir une image qui se placera sous le curseur pendant le déplacement de l'élément concerné. La méthode prend trois arguments en paramètres. Le premier est un élément <img> contenant l'image souhaitée, le deuxième est la position horizontale de l'image et le troisième est la position verticale :

var dragImg = new Image(); // Il est conseillé de précharger l'image, sinon elle risque de ne pas s'afficher pendant le déplacement

dragImg.src = 'drag\_img.png';

document.querySelector('\*[draggable="true"]').addEventListener('dragstart', function(e) {

e.dataTransfer.setDragImage(dragImg, 40, 40); // Une position de 40x40 pixels centrera l'image (de 80x80 pixels) sous le curseur

});

**Définir une zone de « drop »**

Un élément en cours de déplacement ne peut pas être déposé n'importe où, il faut pour cela définir une zone de « drop » qui ne sera, au final, qu'un simple élément HTML. Les zones de drop prennent généralement en charge quatre événements :

* dragenter, qui se déclenche lorsqu'un élément en cours de déplacement entre dans la zone de drop.
* dragover, qui se déclenche lorsqu'un élément en cours de déplacement se déplace dans la zone de drop.
* dragleave, qui se déclenche lorsqu'un élément en cours de déplacement quitte la zone de drop.
* drop, qui se déclenche lorsqu'un élément en cours de déplacement est déposé dans la zone de drop.

Par défaut, le navigateur interdit de déposer un quelconque élément où que ce soit dans la page Web. Notre but est donc d'annuler cette action par défaut, et qui dit « annulation d'une action par défaut », dit preventDefault(). Cette méthode va devoir être utilisée au travers de l'événement dragover.

Prenons un exemple simple :

<div id="draggable" draggable="true">Je peux être déplacé !</div>

<div id="dropper">Je n'accepte pas les éléments déplacés !</div>

Cet exemple ne fonctionnera pas. Le navigateur affichera un curseur montrant une interdiction lorsque nous survolez le deuxième <div>. Afin d'autoriser cette action, il va nous falloir ajouter un code JavaScript très simple :

document.querySelector('#dropper').addEventListener('dragover', function(e) {

e.preventDefault(); // Annule l'interdiction de drop

});

Avec ce code, le curseur n'affiche plus d'interdiction en survolant la zone de drop, cependant il ne se passe rien si nous relâchons notre élément sur la zone de drop. Cela est parfaitement normal, car c'est à nous de définir la manière dont la zone de drop doit gérer les éléments qu'elle reçoit. Avant toute chose, pour agir suite à un drop d'élément, il nous faut détecter ce fameux drop, nous allons donc devoir utiliser l'événement drop :

document.querySelector('#dropper').addEventListener('drop', function(e) {

e.preventDefault(); // Cette méthode est toujours nécessaire pour éviter une éventuelle redirection inattendue

alert('Vous avez bien déposé votre élément !');

});

Essayons maintenant les événements dragenter, dragleave et un petit oublié qui se nomme dragend :

var dropper = document.querySelector('#dropper');

dropper.addEventListener('dragenter', function() {

dropper.style.borderStyle = 'dashed';

});

dropper.addEventListener('dragleave', function() {

dropper.style.borderStyle = 'solid';

});

// Cet événement détecte n'importe quel drag & drop qui se termine, autant le mettre sur « document » :

document.addEventListener('dragend', function() {

alert("Un Drag & Drop vient de se terminer mais l'événement dragend ne sait pas si c'est un succès ou non.");

});

Avant d'essayer ce code, il nous faut réfléchir à une chose : nous appliquons un style lorsque l'élément déplacé entre dans la zone de drop puis nous le retirons lorsqu'il en sort. Cependant, que se passe-t-il si nous relâchons notre élément dans la zone de drop ? Eh bien le style reste en place, car l'élément n'a pas déclenché l'événement dragleave. Il nous faut donc retirer le style en modifiant notre événement drop :

dropper.addEventListener('drop', function(e) {

e.preventDefault(); // Cette méthode est toujours nécessaire pour éviter une éventuelle redirection inattendue

alert('Vous avez bien déposé votre élément !');

// Il est nécessaire d'ajouter cela car sinon le style appliqué par l'événement « dragenter » restera en place même après un drop :

dropper.style.borderStyle = 'solid';

});

**Terminer un déplacement avec l'objet dataTransfer**

L'objet dataTransfer a deux rôles importants lors de la fin d'un drag & drop. Le premier consiste à récupérer, grâce à la méthode getData(), le texte sauvegardé par setData() lors de l'initialisation du drag & drop :

dropZone.addEventListener('drop', function(e) {

alert(e.dataTransfer.getData('text/plain')); // Affiche le contenu du type MIME « text/plain »

});

Quant au deuxième rôle, celui-ci consiste à récupérer les éventuels fichiers qui ont été déposés par l'utilisateur. Cela fonctionne plus ou moins de la même manière qu'avec une balise <input type="file" />, il nous faut toujours accéder à une propriété files, sauf que celle-ci est accessible dans l'objet dataTransfer dans le cadre d'un drag & drop :

dropZone.addEventListener('drop', function(e) {

e.preventDefault();

var files = e.dataTransfer.files,

filesLen = files.length,

filenames = "";

for (var i = 0 ; i < filesLen ; i++) {

filenames += '\n' + files[i].name;

}

alert(files.length + ' fichier(s) :\n' + filenames);

});

**Mise en pratique**

Notre but ici est de créer une page Web avec deux zones de drop et quelques éléments que l'on peut déplacer d'une zone à l'autre. Voici le code HTML utilisé et le CSS associé :

<div class="dropper">

<div class="draggable">#1</div>

<div class="draggable">#2</div>

</div>

<div class="dropper">

<div class="draggable">#3</div>

<div class="draggable">#4</div>

</div>

.dropper {

margin: 50px 10px 10px 50px;

width: 400px;

height: 250px;

background-color: #555;

border: 1px solid #111;

border-radius: 10px;

transition: all 200ms linear;

}

.drop\_hover {

box-shadow: 0 0 30px rgba(0, 0, 0, 0.8) inset;

}

.draggable {

display: inline-block;

margin: 20px 10px 10px 20px;

padding-top: 20px;

width: 80px;

height: 60px;

color: #3D110F;

background-color: #822520;

border: 4px solid #3D110F;

text-align: center;

font-size: 2em;

cursor: move;

transition: all 200ms linear;

user-select: none;

}

Le code HTML est extrêmement simple et la seule chose à comprendre au niveau du CSS est que la classe .drop\_hover sera appliquée à une zone de drop lorsque celle-ci sera survolée par un élément HTML déplaçable.

Il nous faut, avant toute chose, une structure pour notre code. Nous avons décidé de partir sur un code basé sur cette forme :

(function() {

var dndHandler = {

// Cet objet est conçu pour être un namespace et va contenir les méthodes que nous allons créer pour notre système de drag & drop

};

// Ici se trouvera le code qui utilisera les méthodes de notre namespace « dndHandler »

})();

Pour commencer à exploiter notre structure, il nous faut une méthode capable de donner la possibilité aux éléments concernés d'être déplacés. Les éléments concernés sont ceux qui possèdent une classe .draggable. Afin de les paramétrer, nous allons créer une méthode applyDragEvents() dans notre objet dndHandler :

var dndHandler = {

applyDragEvents: function(element) {

element.draggable = true;

}

};

Ici, notre méthode s'occupe de rendre déplaçables tous les objets qui lui seront passés en paramètres. Cependant, cela ne suffit pas pour deux raisons :

* Nos zones de drop devront savoir quel est l'élément qui sera déposé, nous allons utiliser une propriété draggedElement pour sauvegarder ça.
* Firefox nécessite l'envoi de données avec setData() pour autoriser le déplacement d'éléments.

Ces deux ajouts sont faciles à mettre en place :

var dndHandler = {

draggedElement: null, // Propriété pointant vers l'élément en cours de déplacement

applyDragEvents: function(element) {

element.draggable = true;

var dndHandler = this; // Cette variable est nécessaire pour que l'événement « dragstart » accède facilement au namespace « dndHandler »

element.addEventListener('dragstart', function(e) {

dndHandler.draggedElement = e.target; // On sauvegarde l'élément en cours de déplacement

e.dataTransfer.setData('text/plain', ''); // Nécessaire pour Firefox

});

}

};

Ainsi, nos zones de drop n'auront qu'à lire la propriété draggedElement pour savoir quel est l'élément qui a été déposé.

Passons maintenant à la création de la méthode applyDropEvents() qui, comme son nom l'indique, va se charger de gérer les événements des deux zones de drop. Nous allons commencer par gérer les deux événements les plus simples : dragover et dragleave.

var dndHandler = {

// […]

applyDropEvents: function(dropper) {

dropper.addEventListener('dragover', function(e) {

e.preventDefault(); // On autorise le drop d'éléments

this.className = 'dropper drop\_hover'; // Et on applique le style adéquat à notre zone de drop quand un élément la survole

});

dropper.addEventListener('dragleave', function() {

this.className = 'dropper'; // On revient au style de base lorsque l'élément quitte la zone de drop

});

}

};

Notre but maintenant est de gérer le drop d'éléments. Notre système doit fonctionner de la manière suivante :

* Un élément est « droppé ».
* Notre événement drop va alors récupérer l'élément concerné grâce à la propriété draggedElement.
* L'élément déplacé est cloné.
* Le clone est alors ajouté à la zone de drop concernée.
* L'élément d'origine est supprimé.
* Et pour terminer, le clone se voit réattribuer les événements qu'il aura perdus du fait que la méthode cloneNode() ne conserve pas les événements.

dropper.addEventListener('drop', function(e) {

var target = e.target,

draggedElement = dndHandler.draggedElement, // Récupération de l'élément concerné

clonedElement = draggedElement.cloneNode(true); // On créé immédiatement le clone de cet élément

target.className = 'dropper'; // Application du style par défaut

clonedElement = target.appendChild(clonedElement); // Ajout de l'élément cloné à la zone de drop actuelle

dndHandler.applyDragEvents(clonedElement); // Nouvelle application des événements qui ont été perdus lors du cloneNode()

draggedElement.parentNode.removeChild(draggedElement); // Suppression de l'élément d'origine

});

Nos deux méthodes sont maintenant terminées, il ne nous reste plus qu'à les appliquer aux éléments concernés :

(function() {

var dndHandler = {

// […]

};

var elements = document.querySelectorAll('.draggable'),

elementsLen = elements.length;

for (var i = 0 ; i < elementsLen ; i++) {

dndHandler.applyDragEvents(elements[i]); // Application des paramètres nécessaires aux éléments déplaçables

}

var droppers = document.querySelectorAll('.dropper'),

droppersLen = droppers.length;

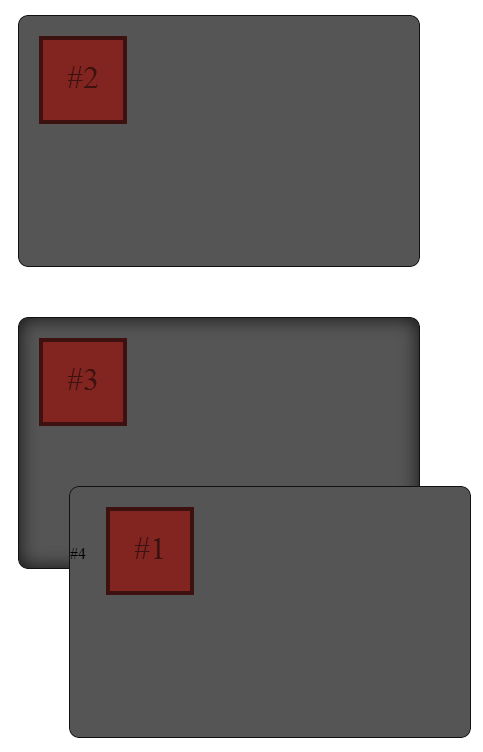
for (var i = 0 ; i < droppersLen ; i++) {

dndHandler.applyDropEvents(droppers[i]); // Application des événements nécessaires aux zones de drop

}

})();

Notre code est terminé, cependant il a un bug majeur qui se produit si nous essaons de déplacer un élément directement sur un autre élément plutôt que sur une zone de drop. Si on déplace par exemple l'élément #1 sur l'élément #4, nous aurons alors quelque chose qui ressemble à l'image suivante :



Ici, l’élément #4 s’est transformé. Cela s'explique par le fait que l'événement drop est hérité par les éléments enfants, ce qui signifie que les éléments possédant la classe .draggable se comportent alors comme des zones de drop.

Une solution serait d'appliquer un événement drop aux éléments déplaçables refusant tout élément HTML déposé, mais cela obligerait alors l'utilisateur à déposer son élément en faisant bien attention à ne pas se retrouver au-dessus d'un élément déplaçable :

applyDragEvents: function(element) {

// […]

element.addEventListener('drop', function(e) {

e.stopPropagation(); // On stoppe la propagation de l'événement pour empêcher la zone de drop d'agir

});

},

Cette solution ne sera pas retenue car trop pénible pour l’utilisateur.

Jusqu’ici, nous n'avions pas encore étudié la méthode stopPropagation(), car celle-ci nécessite un cas concret d'utilisation, ce qui est le cas ici. Cette méthode sert à stopper la propagation des événements. Dans une phase de bouillonnement, si un élément enfant possède un événement du même type qu'un de ses éléments parents, alors son événement se déclenchera en premier, puis viendra celui de l'élément parent. La méthode stopPropagation() sert à brider ce fonctionnement.

Dans le cadre d'une phase de bouillonnement, en utilisant cette méthode dans l'événement de l'élément enfant, nous empêcherons alors l'élément parent d'exécuter son événement. Dans le cadre d'une phase de capture, en utilisant cette méthode sur l'élément parent, nous empêcherons alors l'élément enfant de déclencher son événement.

La solution la plus pratique pour l'utilisateur serait donc de faire en sorte de « remonter » les éléments parents (avec parentNode) jusqu'à tomber sur une zone de drop :

dropper.addEventListener('drop', function(e) {

var target = e.target,

draggedElement = dndHandler.draggedElement, // Récupération de l'élément concerné

clonedElement = draggedElement.cloneNode(true); // On crée immédiatement le clone de cet élément

while (target.className.indexOf('dropper') == -1) { // Cette boucle permet de remonter jusqu'à la zone de drop parente

target = target.parentNode;

}

target.className = 'dropper'; // Application du style par défaut

clonedElement = target.appendChild(clonedElement); // Ajout de l'élément cloné à la zone de drop actuelle

dndHandler.applyDragEvents(clonedElement); // Nouvelle application des événements qui ont été perdus lors du cloneNode()

draggedElement.parentNode.removeChild(draggedElement); // Suppression de l'élément d'origine

});

Si target (qui représente l'élément ayant reçu un élément déplaçable) ne possède pas la classe .dropper, alors la boucle va passer à l'élément parent et va continuer comme cela jusqu'à tomber sur une zone de drop.

Le code Javascript complet de notre exemple sera donc le suivant :

(function() {

var dndHandler = {

draggedElement: null, // Propriété pointant vers l'élément en cours de déplacement

applyDragEvents: function(element) {

element.draggable = true;

var dndHandler = this; // Cette variable est nécessaire pour que l'événement « dragstart » ci-

dessous accède facilement au namespace « dndHandler »

element.addEventListener('dragstart', function(e) {

dndHandler.draggedElement = e.target; // On sauvegarde l'élément en cours de

déplacement

e.dataTransfer.setData('text/plain', ''); // Nécessaire pour Firefox

});

},

applyDropEvents: function(dropper) {

dropper.addEventListener('dragover', function(e) {

e.preventDefault(); // On autorise le drop d'éléments

this.className = 'dropper drop\_hover'; // Et on applique le style adéquat à notre zone de

drop quand un élément la survole

});

dropper.addEventListener('dragleave', function() {

this.className = 'dropper'; // On revient au style de base lorsque l'élément quitte la zone

de drop

});

var dndHandler = this; // Cette variable est nécessaire pour que l'événement « drop » ci-

dessous accède facilement au namespace « dndHandler »

dropper.addEventListener('drop', function(e) {

var target = e.target,

draggedElement = dndHandler.draggedElement, // Récupération de l'élément concerné

clonedElement = draggedElement.cloneNode(true); // On créé immédiatement le clone

de cet élément

while (target.className.indexOf('dropper') == -1) { // Cette boucle permet de remonter

jusqu'à la zone de drop parente

target = target.parentNode;

}

target.className = 'dropper'; // Application du style par défaut

clonedElement = target.appendChild(clonedElement); // Ajout de l'élément cloné à la zone

de drop actuelle

dndHandler.applyDragEvents(clonedElement); // Nouvelle application des événements qui

ont été perdus lors du cloneNode()

draggedElement.parentNode.removeChild(draggedElement); // Suppression de l'élément

d'origine

});

}

};

var elements = document.querySelectorAll('.draggable'),

elementsLen = elements.length;

for (var i = 0; i < elementsLen; i++) {

dndHandler.applyDragEvents(elements[i]); // Application des paramètres nécessaires aux

éléments déplaçables

}

var droppers = document.querySelectorAll('.dropper'),

droppersLen = droppers.length;

for (var i = 0; i < droppersLen; i++) {

dndHandler.applyDropEvents(droppers[i]); // Application des événements nécessaires aux zones

de drop

}

})();

## Aller plus loin

**Les frameworks**

Un framework a pour but de fournir une « surcouche » au JavaScript afin de simplifier l'utilisation des domaines les plus utilisés de ce langage tout en facilitant la compatibilité de nos codes entre les navigateurs Web. Par exemple, quelques frameworks disposent d'une fonction $() s'utilisant de la même manière que la méthode querySelector(), et ce, sur tous les navigateurs Web, facilitant ainsi la sélection d'éléments HTML.

L'atout numéro un d'un framework est sa capacité à s'adapter à toutes les utilisations du JavaScript et à fournir un système performant de plugins afin qu'il puisse être étendu à des utilisations non envisagées par son système de base. Grâce à ces deux points, un framework permet de simplifier et d'accélérer considérablement le développement d'applications Web.

Il existe de nombreux frameworks en JavaScript en raison de la pauvreté de ce langage en terme de fonctions natives. Nous allons voir les plus connus d'entre eux :

* **jQuery :** il s'agit du framework JavaScript le plus connu. Réputé pour sa simplicité d'utilisation et sa communauté gigantesque, il est clairement incontournable. Cependant, il n'est pas toujours apprécié en raison de sa volonté de s'écarter de la syntaxe de base du JavaScript grâce au chaînage de fonctions, que nous pouvez constater dans l'exemple qui suit :

$("p.neat").addClass("ohmy").show("slow");.

* **MooTools :** un framework puissant et presque tout aussi connu que jQuery, bien qu'énormément moins utilisé. Il est réputé pour sa modularité et son approche différente plus proche de la syntaxe de base du JavaScript. En revanche, bien que ce framework soit « segmentable » (nous ne téléchargons que ce dont nous avons besoin), il reste nettement plus lourd que jQuery.
* **Dojo :** connu pour sa capacité à permettre la conception d'interfaces Web extrêmement complètes, il possède des atouts indéniables face aux plus grands frameworks et tout particulièrement jQuery UI, une extension de jQuery conçue pour faire des interfaces Web. Ce framework est l'un des plus modulaires que l'on puisse trouver sur Internet.
* **YUI :** il est souvent oublié par les développeurs Web, mais l'entreprise Yahoo! n'a pourtant pas dit son dernier mot avec des projets ambitieux parmi lesquels compte YUI. Ce framework est pour le moins complet, modulable et relativement performant, bien qu'il bénéficie d'une communauté assez restreinte.

**Les bibliothèques**

Contrairement aux frameworks, les bibliothèques (libraries en anglais) ont un but bien plus spécialisé. L'intérêt face aux frameworks se situe à la fois sur le poids du fichier JavaScript à utiliser (une bibliothèque sera généralement plus légère qu'un framework) et sur la spécialisation des bibliothèques. Ces dernières ont souvent tendance à aller plus loin que les frameworks, vu qu'elles n'agissent que sur un domaine bien précis, ce qui simplifie d'autant plus notre développement dans le domaine concerné par la bibliothèque.

De plus, il existe un principe important dans le développement (Web ou logiciel) : l'optimisation. Utiliser un framework uniquement pour faire une malheureuse animation n'est vraiment pas conseillé. Il est préférable d’utiliser une bibliothèque :

* **Sizzle, Qwery :** deux bibliothèques conçues pour fonctionner de la même manière que la méthode querySelector() (ce type de bibliothèques est d'ailleurs à l'origine de l'implémentation officielle de la méthode en question). La première est le moteur de sélection du framework jQuery mais est relativement lourde, la seconde a l'avantage d'être particulièrement légère et un peu plus rapide en terme d'exécution.
* **Popcorn.js :** une bibliothèque permettant une manipulation aisée des balises <audio> et <video>. Il devient très simple d'interagir avec ces balises afin de mettre en place des sous-titres, des commentaires placés à un moment précis de la piste audio ou vidéo, etc.
* [**Raphaël**](http://raphaeljs.com/)**, Three.js :** bibliothèques spécialisées dans le graphisme. La première fonctionne exclusivement avec les images SVG, la deuxième s'est spécialisée dans la 3D et gère à la fois SVG, <canvas> et surtout la bibliothèque WebGL, qui sait tirer parti du moteur OpenGL.
* **Modernizr :** les langages Web sont plus ou moins bien supportés selon les navigateurs, surtout quand il s'agit de fonctionnalités récentes. Cette bibliothèque a pour but d’ aider à détecter la présence de telle ou telle fonctionnalité. Il ne restera alors plus qu'à fournir un script résolvant ce problème (un polyfill) ou exploitant une solution alternative.

**Diverses applications du JavaScript**

Comme son nom l'indique, le JavaScript est un langage de script. Cela signifie qu'il s'agit d'un langage interprété (et non compilé) qui peut être exécuté dans divers environnements. Notre navigateur Web est un environnement, mais notre système d'exploitation aussi ! Ainsi, par exemple, Windows supporte l'exécution de fichiers JavaScript au sein même de son système.

Ainsi, le JavaScript est bien loin d'être limité au Web, bien qu'il s'agisse de sa principale utilisation. Ce langage se retrouve notamment dans le code des extensions des navigateurs Web tels que Firefox ou Chrome. Cela permet de coder facilement et rapidement des extensions basées sur le HTML et le JavaScript avec des droits bien plus étendus que ceux d'une page Web classique. Nous pouvons, par exemple, gérer les onglets et les fenêtres grâce à une API fournie par le navigateur et utilisable en JavaScript.

**Des logiciels développés en JavaScript**

Dans un tout autre domaine que le Web, la plateforme Adobe Air propose le développement d'applications sur de multiples environnements parmi lesquels nous retrouvons (à l'heure actuelle) Windows, Mac OS, Android, iOS et BlackBerry Tablet OS. L'avantage de cette plateforme est qu'elle supporte plusieurs ensembles de langages dont l'ensemble HTML/CSS + JavaScript. Ainsi, il est maintenant possible de développer des applications PC et mobiles simplement grâce à du HTML et du JavaScript. Adobe Air est conçu de telle sorte que le JavaScript utilisé au travers de cette plateforme bénéficiera d'API supplémentaires telles que celle permettant de manipuler les fichiers.

**Des applications pour SmartPhones en JavaScript**

A l'instar D'Adobe Air existe Adobe PhoneGap, qui propose de réaliser une application pour SmartPhone (Android, iOS...) en utilisant des technoloies Web (HTML, CSS, JavaScript + API) et de la compiler avec PhoneGap pour en faire une application exécutable.

**Du JavaScript sur le serveur**

Retour au domaine du Web, mais du côté des serveurs. **Node.js** est une plateforme permettant l'exécution du langage JavaScript côté serveur afin de remplacer ou venir en complément de langages serveur plus traditionnels tels que le PHP. L'intérêt de ce projet réside essentiellement dans son système non bloquant. Prenons le PHP par exemple. Pour lire un fichier en entier, nous serons obligés d'attendre la lecture complète avant de pouvoir passer à la suite du code, tandis qu'avec **Node.js** nous utilisons un langage (le JavaScript) conçu pour gérer tout ce qui est événementiel. Ainsi, nous ne sommes pas obligés d'attendre la fin de la lecture du fichier pour passer à autre chose.