

bf.cepegra

JavaScript & jQuery



Nicolas Bauwens

Conseiller pédagogique - Formateur
n.bauwens@bruxellesformation.be

Contenu

Introduction	5
Le B-A-BA d'Internet	5
Qu'est-ce que JavaScript & jQuery ?	6
Définition	6
Historique.....	6
Outils	6
Quelques mots sur jQuery	7
Inclure du JavaScript dans une page HTML	8
Les éléments du langage.....	9
Variables, types et valeurs	9
Déclaration et initialisation d'une variable	9
Types primitifs.....	10
Types objets	12
Transformations de types	14
Tests et conditions	16
Les opérateurs de comparaison.....	16
Les opérateurs logiques	17
Types de tests	18
Tester <i>null</i> et <i>undefined</i>	19
Les boucles.....	22
L'incrémentation.....	24
Le type array	24
Types de boucles.....	25
Les fonctions	27
Déclaration.....	27
Portée des variables.....	28
Appel	28
Les fonctions anonymes.....	29
Utiliser jQuery	30
Qu'est-ce que jQuery ?	30
Inclure jQuery dans son projet	30
Manipulation du DOM	32
Comprendre le Document Objet Model	32

Rappel sur les objets	33
Les objets window et document.....	34
Lecture et modification du DOM	35
En JavaScript pur.....	36
En jQuery.....	38
Agir sur plusieurs éléments d'une sélection	45
Les filtres	47
Les événements	48
Le concept d'événement.....	48
Associer un événement à un élément	48
Attendre le chargement de l'HTML	51
L'objet <i>Event</i>	52
Stopper le comportement normal d'un événement	53
Les effets visuels avec jQuery	53
Apparaître et disparaître.....	54
Les animations	54
Parcourir le DOM avec jQuery	55
Les formulaires avec jQuery.....	56
Interagir avec un formulaire	56
Valider un formulaire	60
Requêtes AJAX avec jQuery	63
Introduction	63
Fonctionnement.....	63
Méthodes jQuery	64
<i>.load()</i>	64
<i>\$.get()</i> et <i>\$.post()</i>	64
<i>\$.getJSON()</i>	64
<i>\$.ajax()</i>	64
Récupération des résultats et des erreurs.....	65
Les plug-ins.....	66
jQuery UI	66
Autres plug-ins	66
Liens et ressources à propos de jQuery	67
Plus loin avec JavaScript.....	68

Le debug et la gestion des erreurs.....	68
Le debug.....	68
La gestion des erreurs.....	68
Le testing.....	69
Orienté objet.....	69
Objet littéral.....	69
Constructeur	71
Structurer son code grâce aux objets	72
Quelques trucs en plus.....	72
Isoler son code grâce au <i>module pattern</i>	72
Optimiser grâce à la minification	73
La gestion des cookies.....	73
Le futur... et le présent de JavaScript.....	75
API HTML5.....	75
Single page applications et frameworks MVC	75
End-to-end JavaScript	77
Sources.....	78

Introduction

Au terme de ce cours, vous serez capable de :

- maîtriser les bases de la programmation
- comprendre les grands concepts du JavaScript
- lire un script jQuery existant et le modifier
- utiliser des plug-ins jQuery et paramétrer leur fonctionnement
- rédiger votre propre script avec jQuery et JavaScript afin de dynamiser votre site web avec des effets et des animations

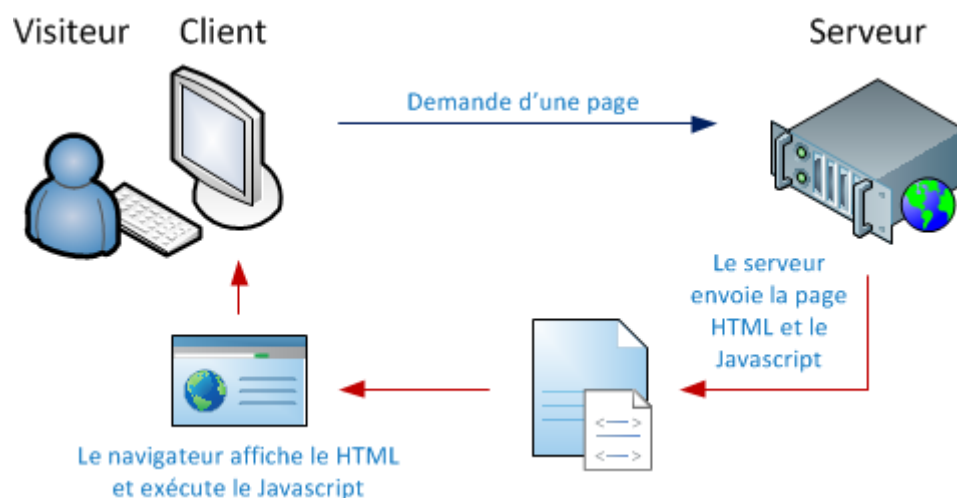
Le JavaScript c'est le 3^{ème} pilier du web design. Le premier c'est l'HTML, il décrit la **structure** d'une page web. Le second, c'est la CSS, qui définit la **présentation** de la page. Le JavaScript lui définit le **comportement** de la page : il met vie à une page web qui serait autrement complètement statique.

Le B-A-BA d'Internet

Petit rappel pour ceux qui savent, et petite explication pour ceux qui se sont toujours posé la question « mais comment ça marche Internet » ?

Pour faire simple, quand vous tapez <http://www.google.com> dans votre navigateur, votre requête est envoyée vers un serveur web, dans ce cas-ci un serveur de chez Google, qui va simplement vous renvoyer un fichier HTML accompagné de fichiers CSS et éventuellement de JavaScript. Ce transfert de données utilise le protocole HTTP (HyperText Transmission Protocol). Un protocole est une spécification des règles de communication entre matériels électroniques.

Une fois la page web reçue par le navigateur, celui-ci va décoder tout ça (*parsing* en anglais) et vous afficher une belle page web !



Qu'est-ce que JavaScript & jQuery ?

Définition

JavaScript est un langage de programmation interprété.

Un **langage de programmation**, c'est un langage qui permet d'écrire du code source. Le code source quant à lui:

- communique des actions à un ordinateur sous forme d'instructions (variables, tests, boucles, etc...)
- peut être compilé: le code est d'abord converti en langage binaire (0 et 1) par un compilateur afin d'être compris par l'ordinateur
- ou **interprété**: l'ordinateur comprend directement le code via un interpréteur. On parle souvent du mot **script** pour parler de code interprété.

Il peut être utilisé côté **client**: dans ce cas il est interprété par les interpréteurs des navigateurs (*SpiderMonkey* pour Firefox, *JScript* pour IE, *V8* pour Chrome) donc sur la machine de l'internaute. Dans le cadre de ce cours, nous travaillerons avec du JavaScript côté client.

Mais il peut également être utilisé côté **serveur**, comme PHP, C#, Java,... qui s'exécutent sur le serveur et renvoient de l'HTML (ou des données) aux machines clientes ; grâce à Node.js, une plateforme construite sur V8.

Quand JavaScript tourne sur une machine client, cela peut poser divers problèmes: lenteur de l'exécution du script due aux performances de la machine client, problème de compatibilité avec des navigateurs obsolètes voir la désactivation complète de JavaScript sur la machine cliente. Nous reviendrons sur ces problématiques dans le cours.

Historique

Il ne faut pas retenir tout ça mais ça va éclaircir certains termes que vous risquez de rencontrer quand vous travaillerez avec du JavaScript. JavaScript a été inventé par Netscape en 1995 et fut ensuite standardisé par l'ECMA (un organisme qui met en place des standards dans le monde de l'informatique). La version standardisée s'appelle donc **ECMAScript**, la version stable la plus récente étant ECMAScript 5. Plusieurs implémentations d'ECMAScript existent à l'heure actuelle, dont JScript (implémentation dans IE) ou ActionScript (dans Flash).

D'où quelques différences de comportement entre Mozilla et Internet Explorer, JScript permettant de faire des choses qui ne sont pas possible par défaut avec JavaScript.

Outils

Pour écrire du JavaScript, il ne faut pas d'outils particuliers. Votre éditeur HTML fait l'affaire, aussi bien qu'un outil comme Notepad++. Afin de nous faciliter la vie, nous utiliserons l'outil WebStorm qui fait de l'*autocomplétion* et offre des petites fonctionnalités sympathiques comme le debug de code.

Vous pouvez aussi utiliser le site <http://jsfiddle.net/> qui permet de tester rapidement du code JavaScript sans devoir ouvrir un éditeur, créer un fichier, le lancer dans Firefox etc... C'est très pratique quand vous voulez tester du code rapidement ! Les exemples de ce cours seront dans jsfiddle.

Enfin, outil ô combien indispensable pour déboguer du JavaScript : [Firebug](#). FireBug est une extension de Firefox et permet d'inspecter le code HTML et les CSS mais aussi de déboguer du JavaScript, voir les requêtes AJAX, etc... Vous pouvez utiliser la méthode `console.log()` dans votre code pour déboguer, mais attention, retirez cette instruction avant de mettre le site en ligne !

Quelques mots sur jQuery

jQuery est une librairie JavaScript, en gros, du code source qui permet de faire rapidement et facilement des choses qui seraient très compliquées à faire soi-même en JavaScript pur !

jQuery est principalement utilisé pour [manipuler le DOM](#) (la page HTML) mais permet aussi de faire des appels asynchrones via [AJAX](#).

De plus, jQuery s'occupe tout seul de la compatibilité avec les différents navigateurs, même IE 6 – pour information Internet Explorer 6 et même 7 sont les pires cauchemars d'un intégrateur car ils regorgent de spécificités propres à eux.

Sa syntaxe est claire et facilement compréhensible, le seul point négatif étant qu'il impacte les performances techniques car la librairie est assez lourde à charger.

C'est très chouette, mais pour bien jouer avec jQuery il faut quand même comprendre les bases du JavaScript, et plus précisément sa syntaxe. Comme si vous appreniez une nouvelle langue, nous allons voir les rudiments de grammaire et de vocabulaire du langage dans les prochains chapitres.

Inclure du JavaScript dans une page HTML

Du code JavaScript est lié à la page HTML via le tag `<script>`. De manière similaire à des styles, le code JavaScript peut soit:

- être inclus directement dans la page HTML, sous cette forme

```
</> <script type="text/javascript">  
    [CODE JAVASCRIPT]  
</script>
```

- se trouver dans un fichier externe (extension .js), sous cette forme (la localisation du fichier peut être absolue ou relative, ici il se trouve dans le fichier *function.js*)

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

Prenez l'habitude de mettre vos scripts dans des fichiers séparés, par soucis de performances (le fichier sera mis en cache par le navigateur) et pour rendre le script *inobstrusive* (nous reviendrons sur cette notion à la fin du cours)

Il est souvent mentionné que le tag `<script>` doit se trouver dans le tag `<head>`, cependant une page Web est lue par le navigateur de façon linéaire, c'est-à-dire qu'il lit d'abord le `<head>`, puis les éléments de `<body>` les uns à la suite des autres. Si vous appelez un fichier JavaScript dès le début du chargement de la page, le navigateur va donc charger ce fichier, et si ce dernier est volumineux, le chargement de la page s'en trouvera ralenti. C'est normal puisque le navigateur va charger le fichier avant de commencer à afficher le contenu de la page.

Pour pallier ce problème, il est conseillé de placer les tags `<script>` juste avant la fermeture du tag `<body>`.

Les éléments du langage

Tout langage de programmation se compose de ces différents éléments: les variables, les tests, les boucles. Nous allons plonger dans la base de la programmation !

Variables, types et valeurs

Déclaration et initialisation d'une variable

Vous vous souvenez de votre cours de math ? On y parlait déjà de variables, par exemple dans le calcul suivant, on trouve 3 variables: x, a et b.

$$x = (a + b)/4$$

On **assigne** des valeurs à a et b afin de trouver la valeur de x.

C'est le même principe en informatique, une variable permet de stocker une valeur, une information, qu'elle soit numérique, sous forme de texte, ou autre.

Afin d'utiliser une variable il faut d'abord la **déclarer**. Il s'agit tout simplement de lui réserver un espace de stockage en mémoire, rien de plus. Une fois la variable déclarée, vous pouvez commencer à y stocker des données sans problème.

Pour déclarer une variable, il suffit d'écrire par exemple:

JS

```
var myVariable;
```

Notez la présence d'un point-virgule. Après chaque instruction, il faut mettre un point-virgule. C'est comme ça 😊



le JavaScript est sensible à la casse ou *case-sensitive*, ce qui veut dire que les majuscules et minuscules comptent.

```
var myVariable;  
var MyVariable;
```

sont 2 variables différentes !

On peut nommer une variable comme on le souhaite, cependant il existe certains noms réservés qu'on ne peut pas utiliser, sinon le code ne marchera pas. [La liste complète des mots réservés](#)

Une fois une variable déclarée on peut **l'initialiser**, c'est-à-dire lui assigner une valeur, par exemple

JS

```
var myVariable;  
myVariable = 2;
```

Généralement, les variables sont déclarées au tout début du bloc de code.

Contrairement à de nombreux langages, le JavaScript est un langage typé *dynamiquement*. Cela veut dire, généralement, que toute déclaration de variable se fait avec le mot-clé `var` sans distinction du contenu, tandis que dans d'autres langages, comme le Java, il est nécessaire de préciser quel type de contenu la variable va devoir contenir (un nombre, du texte, etc...)

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. Il faut donc être très attentif !

Les types en JavaScript peuvent être divisés en 2 catégories : les types *primitifs* (nombres, textes, booléens, null et undefined) et les types *objets* (toute autre valeur qui n'est pas de type primitif)

Types primitifs

Numérique (Number)

Ce type représente tout nombre, qu'il soit entier, négatif, décimal,...

Pour assigner un type numérique à une variable, il vous suffit juste d'écrire le nombre seul :



```
var number = 5;
```

Attention, dans le cas d'un nombre décimal, il faut utiliser le `.` et non la virgule, par exemple:



```
var number = 5.5;
```

Quelle est l'utilité ? De pouvoir faire des calculs par exemple ! Et à cette fin, vous pouvez utiliser les opérateurs arithmétiques que vous connaissez tous comme `+`, `-`, `/`, `*`.

Mais également, moins connu, l'opérateur `%`, qui permet de calculer le modulo (*le reste de la division entière*). A quoi sert le modulo ? A vérifier si un nombre est pair par exemple, ou multiple d'un chiffre.

Chaîne de caractère (String)

Ce type représente n'importe quel texte. On peut l'assigner de deux façons différentes :



```
var text1 = "Mon premier texte"; // Avec des guillemets  
var text2 = 'Mon deuxième texte'; // Avec des apostrophes
```

Il est important de préciser que si vous écrivez `var myVariable = '2'`; alors le type de cette variable est une chaîne de caractères et non pas un type numérique.

Une autre précision importante, si vous utilisez les apostrophes pour « encadrer » votre texte et que vous souhaitez utiliser des apostrophes dans ce même texte, il vous faudra alors « échapper » vos apostrophes de cette façon :

JS

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

Pourquoi ? Car si vous n'échappez pas votre apostrophe, le JavaScript croira que votre texte s'arrête à l'apostrophe contenue dans le mot « c'est ». Du coup, une erreur sera générée et la suite de votre script ne s'exécutera pas.

À noter que ce problème est identique pour les guillemets. Une autre possibilité est d'encadrer le texte avec des guillemets quand vous utilisez des apostrophes dedans:

JS

```
var text = "Ça c'est quelque chose !";
```

Une opération intéressante avec le texte s'appelle la **concaténation**, elle permet de « coller » 2 bouts de texte ensemble pour créer un nouveau texte. Il suffit d'utiliser l'opérateur + entre 2 textes :

JS

```
var text = "Bonjour" + " " + "Madame";  
//text vaudra "Bonjour Madame"
```

Booléens (Boolean)

Ce type permet simplement de stocker "vrai" ou "faux" :

JS

```
var isTrue = true;  
var isFalse = false;
```

Quelle est l'utilité ? De pouvoir vérifier des conditions par exemple.

Sachez qu'il est également possible de "convertir" une variable d'un type à l'autre, par exemple de transformer une string qui contient "42" en une variable numérique qui contient le nombre 42 et inversement.

null et undefined

Il arrive qu'on doive dire qu'une variable est déclarée mais **ne contient rien**. Dans ce cas, on utilisera *null*.

Si une variable **n'a pas été initialisée**, elle sera *undefined*. Evidemment, si vous exécutez une action avec une variable *undefined*, le code va planter.



Exemples de *undefined* et de *null*.

Ce test appelle *console.log()*, pensez à activer Firebug ;-)

<http://jsfiddle.net/xa4zs/>

On peut bien sûr [tester si une variable est null ou undefined](#)

Types objets

JavaScript possède d'autres types natifs que ceux vus précédemment. Dans le cas des types primitifs, des nombres, textes ou booléens littéraux sont généralement utilisés – comme dans les exemples vu précédemment.

Dans le cas des types objets, nous devons passer par l'initialisation d'un objet. Un objet est un type de données qui permet de stocker des propriétés (des variables propres à l'objet) qu'on pourrait imaginer comme étant des **caractéristiques** – par exemple, une télévision a plusieurs caractéristiques : la taille de l'écran, la résolution, la couleur,...

Nous le verrons plus tard, un objet peut aussi posséder des **comportements**, appelées méthodes; dans le cas d'une télévision : allumer, éteindre, changer de chaîne, augmenter le volume,...

Un objet s'initialise ou s'instancie à l'aide de ce qu'on appelle un **constructeur** et via le mot-clé *new* suivi du « type » de l'objet.

Le type Date

Le constructeur *Date()* permet de créer des objets qui représentent des dates et des heures. Dans l'exemple ci-dessous on génère une date d'expiration :

JS

```
//quand on instancie une nouvelle Date, elle sera à la date courante
var date = new Date();
//on lui rajoute le nombre de jours en millisecondes
date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
```

Important à savoir, JavaScript stocke les dates en millisecondes. Mais heureusement pour nous, le constructeur *Date* peut prendre en charge plusieurs paramètres pour nous permettre de construire une date, mais retenez surtout ces deux-cis :

JS

```
new Date();
new Date(année, mois, jour [, heure, minutes, secondes, millisecondes ]);
```

- 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 manuellement chaque composant qui constitue une date, nous retrouvons donc en paramètres obligatoires : l'année, le mois et le jour. Les quatre derniers paramètres sont facultatifs (c'est pour cela que vous voyez des crochets, ils signifient que les paramètres sont facultatifs). Ils seront initialisés à 0 s'ils ne sont pas spécifiés, nous y retrouvons : les heures, les minutes, les secondes et les millisecondes.

Une fois un objet *Date* instancié, vous pouvez utiliser ces méthodes :

- *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).



Exemples de calcul de dates, avec jQuery et *datepicker*

<http://jsfiddle.net/fkKMv/>



Moment.js, une librairie JavaScript incontournable pour travailler avec les dates :

<http://momentjs.com/>

Le type *Array*

Pour créer des tableaux (liste de valeurs) : [Voir ce chapitre](#)

Le type *RegExp*

Pour travailler avec des expressions régulières afin d'effectuer des recherches ou remplacements performants dans du texte. Les expressions régulières ne sont pas abordées dans ce cours.



Plus sur les expressions régulières avec *RegExp* :

http://www.w3schools.com/js/js_obj_regexp.asp

Le type *Math*

Contrairement aux autres types objets, *Math* n'est pas un constructeur mais un objet global à partir duquel on peut effectuer des opérations mathématiques comme : arrondir à la valeur supérieure ou inférieure, calculer des exposants, générer des nombres aléatoires, obtenir le nombre Pi, etc...



Plus sur *Math* :

http://www.w3schools.com/js/js_obj_math.asp

Transformations de types

Imaginez le cas où un utilisateur renseigne un montant dans un champ texte et vous voulez calculer le montant TVA comprise, vous serez bien obligé de transformer le texte rentré dans le champ par l'utilisateur en valeur numérique.

Il est bien sûr possible de transformer une valeur d'un certain type en un autre type, par exemple de convertir un texte qui représente un nombre en nombre. Ou bien l'inverse.

Dans ce cas, vous pouvez effectuer une conversion explicite en utilisant les fonctions *Number()*, *String()* ou *Boolean()*

JS

```
Number("3"); // => 3  
String(false) ; // => "false"  
Boolean(0); // => false
```

Pour convertir des nombres en texte, il est également possible d'utiliser plusieurs méthodes de *Number()* : *.toFixed()* pour préciser un nombre de chiffres après la virgule, *.toPrecision()* pour préciser le nombre de chiffres au total. Mais aussi *.toString()* et *.toExponential()*.

JS

```
var n = 3.7514615;  
console.log(n.toFixed(2)); // 3.75  
console.log(n.toPrecision(5)); // 3.7515
```



Plus sur les méthodes de conversion de *Number()* :

http://www.w3schools.com/jsref/jsref_obj_number.asp

Pour convertir du texte en valeur numérique, il est également possible d'utiliser les méthodes globales *parseInt()* pour transformer vers un entier ou *parseFloat()* pour entier ou décimal. Ces méthodes sont plus flexibles que *Number()* mais sont donc aussi moins fiables si vous cherchez une conversion « stricte »

JS


```
parseInt("10 burgers"); // => 10  
parseFloat("34.12"); // => 34.12
```



Article super intéressant sur la conversion de texte en nombre en JavaScript :

<http://www.js-attitude.fr/2012/12/26/convertir-un-nombre-en-texte-en-javascript/>

En dehors de ces conversions explicites, il existe des conversions implicites. En effet, JavaScript est très flexible sur ce point : si JavaScript veut du texte, il va convertir ce que vous lui donnez en texte ; s'il veut un nombre, il va convertir ce que vous voulez en nombre. Cela peut aller très très loin, voici juste quelques exemples :

	<pre>10 + " burgers"; // => "10 burgers" "7" * "4"; // => 28 +"10" ; // => 10</pre>
---	--

	Un « style » de programmation complètement cryptique qui se base sur les conversions implicites en JS : http://www.jsfuck.com/
---	---

Tests et conditions

Les tests permettent d'exécuter une action (ou de ne pas l'exécuter !) en fonction de certains critères, ou *conditions*. Les tests s'appliquent toujours au type booléen, c'est-à-dire à "vrai" ou "faux". Un test est toujours du type "**Si** la condition est vraie, **alors** j'exécute ceci, **sinon** j'exécute cela"

Par exemple, "si l'utilisateur est belge, alors afficher un message à l'écran" ou "si l'utilisateur choisit le thème 'vert' alors charger la CSS 'vert.css'" ou encore "si la résolution de l'écran est minimum de 640*480, alors charger les scripts lightbox"

Exemples :

Charger des CSS ou Scripts conditionnellement:

<http://japborst.net/blog/conditionally-load-css-or-js.html>

Tester si l'utilisateur vient pour la première fois sur le site

<http://www.ravelrumba.com/blog/firstimpression-js-library-detecting-new-visitors/>



Détecter le pays de l'utilisateur et afficher un message approprié

<http://jsfiddle.net/9hNp4/>



une librairie JavaScript, YepNope, permet de faire du chargement conditionnel de CSS ou de Script: <http://yepnopejs.com/>

De quoi sont constituées les conditions ? De valeurs à tester et de deux types d'opérateurs : de logique et de comparaison. Pour être plus clair: une condition sera vraie ou fausse en fonction du résultat d'une comparaison et/ou d'un test logique.

Les opérateurs de comparaison

Ces opérateurs vont permettre de comparer diverses valeurs entre elles. En tout, ils sont au nombre de huit, les voici :

Opérateur	Signification
==	égal à
!=	différent de
===	contenu et type égal à (<i>à privilégier</i>)
!==	contenu ou type différent de (<i>à privilégier</i>)
>	supérieur à
>=	supérieur ou égal à
<	inférieur à
<=	inférieur ou égal à

Le résultat d'une comparaison renvoie **vrai** ou **faux**, en bref, une valeur de type **booléen** ;-)

Exemple:

```
JS var hour = getHour();
    if(hour === 12){
        document.write('le dîner est prêt !');
    }
```

Les opérateurs logiques

Pourquoi ces opérateurs sont-ils nommés comme étant « logiques » ? Car ils fonctionnent sur le même principe qu'une table de vérité en électronique. Avant de décrire leur fonctionnement, il nous faut d'abord les lister, ils sont au nombre de trois :

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

Lorsque qu'on teste plusieurs valeurs, il faut avoir en tête les résultats possibles, voici la table de vérité des opérateurs "ET" et "OU". Les résultats d'un test logique est aussi un **booléen**

valeur1	Opérateur	valeur2	Résultat
Faux	ET	Faux	Faux
Faux	ET	Vrai	Faux
Vrai	ET	Faux	Faux
Vrai	ET	Vrai	Vrai
Faux	OU	Faux	Faux
Faux	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Vrai	OU	Vrai	Vrai

L'opérateur **!** quant à lui renvoie l'inverse d'une valeur booléenne. Si une variable est « vraie », son inverse sera « faux ».

Du coup, il est possible de combiner au sein d'un même test, des opérateurs de comparaison et des opérateurs logiques, comme dans l'exemple ci-dessous

```
JS var hour = getHour();
    var name = getUsername();
    if(hour === 12 && name === 'Michel'){
        document.write('le dîner est prêt pour Michel !');
    }
```

if-else

Nous l'avons déjà vue dans les exemples précédents, retenez-le bien car c'est le type de test le plus rencontré. Il permet d'exécuter une action si une condition est vraie, et une autre action si elle est fausse.

Sa syntaxe se présente sous cette forme

```
if (test) {  
    action;  
}  
else {  
    autreAction;  
}
```

Il est possible d'enchaîner les tests if-else, par exemple pour dire, "**si** le visiteur est Belge, **alors** afficher un message vert, **sinon si** le visiteur est Français, **alors** afficher un message bleu, **sinon** afficher un message jaune"

La syntaxe sera telle que:

```
if(test) {  
    action;  
}  
else if (test2) {  
    autreAction;  
}  
else {  
    actionParDéfaut;  
}
```

switch

C'est une variante du if-else. Si on doit tester plusieurs pays avec des if-else, ça risque de vite devenir ennuyeux. Dans le cas où il faut faire beaucoup de tests sur une même donnée (ici, le code pays du visiteur), alors il est préférable d'utiliser un *switch* car il permet de mieux s'y retrouver. Cette instruction effectue une comparaison par rapport à la valeur **et au type**.

Sa syntaxe :

```
switch(variable) {  
    case valeurPossible1 :  
        action1;  
        break ;  
    case valeurPossible2 :  
        action2;
```

```

        break ;
    case valeurPossible3 :
        action3;
        break ;
    default :
        actionParDéfaut;
        break ;
}

```

L'instruction *break* permet de « casser » le switch et d'en sortir dès que l'action adéquate a été exécutée. Si on ne mentionne pas le *break*, les autres cas vont également s'exécuter !

default permet lui de définir une action par défaut si aucune des valeurs des *case* ne correspond.



Détecter le pays de l'utilisateur avec un switch et afficher un message approprié

<http://jsfiddle.net/zrUtk>

ternaire

Une autre variante du if-else. Si vous voulez épargner vos doigts lors de l'écriture de if-else, vous pouvez passer par une condition ternaire. C'est la même chose en plus court, mais en parfois moins lisible... Préférez le if-else, mais sachez reconnaître celui-ci.

Syntaxe:

(test) ? action si test est vrai : action si test est faux

ou

résultat = (test) ? valeur si test est vrai : valeur si test est faux



Détecter le pays de l'utilisateur avec l'opérateur ternaire et afficher un message approprié

<http://jsfiddle.net/Xns2Z/>

Tester *null* et *undefined*

Pour vérifier l'existence d'une variable (qu'elle ne soit pas *undefined*) ou son contenu (qu'elle ne soit pas *null* ou vide), on peut le faire de la manière classique ci-dessous :

```

JS  var notSet = null;
    if(notSet === null)
    {
        alert("la variable n'a pas de valeur")
    }

```

JS

```
var notInit;  
if(notInit === undefined)  
{  
    alert("variable pas initialisée")  
}
```

Du coup, pour tester *null* et *undefined* mais aussi par exemple une string vide, il faudrait combiner plusieurs conditions. Il existe une notation beaucoup plus concise !

Vous savez que les variables peuvent être de plusieurs types : les nombres, les chaînes de caractères, etc. En fait 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.

Quels contenus seront « faux » ? Un nombre qui vaut zéro, une string vide, *null* ou *undefined*. Tous les autres cas seront « vrais ». Du coup, on peut écrire quelque chose comme ça, qui est nettement plus rapide :

JS

```
var notSet = null;  
if(notSet)  
{  
    alert("la variable n'a pas de valeur")  
}
```



Quand est-on dans le vrai en JavaScript ? <http://www.js-attitude.fr/2012/09/10/truthy-ou-falsy-en-javascript/>

Une autre petite subtilité du JavaScript nous est offerte par l'opérateur **OU**. Celui-ci, en plus de sa fonction principale, permet de renvoyer la première variable possédant une valeur évaluée à *true*.

JS

```
var conditionTest1 = '', conditionTest2 = 'Une chaîne de caractères';  
  
alert(conditionTest1 || conditionTest2);  
//affichera 'Une chaîne de caractères'
```

Concrètement, à quoi cela peut-il servir ?

Par exemple, certains merveilleux navigateurs comme Internet Explorer ont des fonctionnalités qui sont les mêmes que dans le standard ECMA mais appelées différemment ! Chouette hein ? Hé bien cette notation permettra au navigateur de choisir la fonctionnalité qui existe pour lui.

Si vous êtes déjà curieux, voici un petit exemple : on souhaite récupérer le contenu texte d'un DIV grâce au [DOM](#) (nous verrons ceci en détail dans le chapitre ad hoc). Avec IE on doit utiliser « *innerText* » et pour les autres navigateurs avec le standard ECMA, « *textContent* ». Une manière concise de l'écrire serait :



```
var txt = div.textContent || div.innerText || '';
```



Tester l'existence et le contenu de variables

<http://jsfiddle.net/ZZbh5>

Les boucles

Quand on parle de répéter un album en boucle, ça vous évoque quoi ? Le fait de répéter plusieurs fois, automatiquement, les chansons de l'album. C'est le même principe en programmation. Les boucles permettent de répéter une action plusieurs fois tant qu'une condition est respectée, ou jusqu'à ce qu'une condition soit respectée.

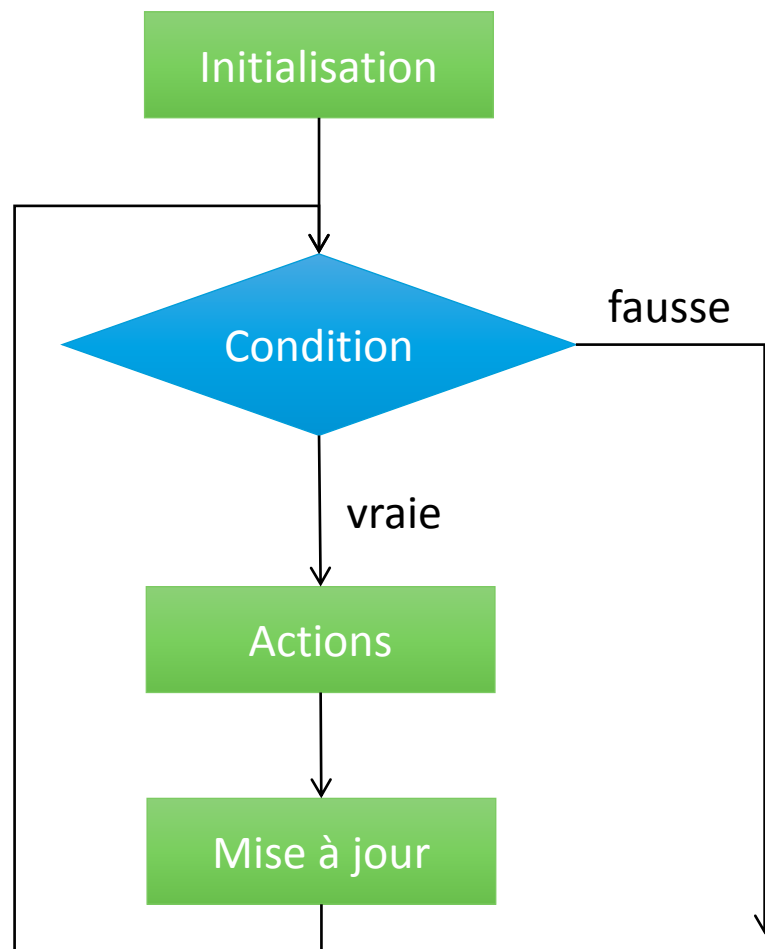
Et oui, nous allons, comme pour les tests, utiliser les conditions dans les boucles. Il faut, bien entendu, qu'un moment la condition devienne fausse, sinon on génère une boucle infinie... vous vous rappelez des sites qui ouvraient milles fenêtres popup ? ☺ C'est un beau cas d'utilisation d'une boucle infinie, une boucle qui ne s'arrête jamais ! Haaaaaaa !

Une boucle peut être exprimée de cette façon « **Tant que** la condition est vraie, exécuter ces actions » ou bien de cette façon « **Pour** toutes les données que voici, exécuter ces actions ».

Une boucle fonctionne de cette manière :

1. On aura généralement besoin d'une variable de boucle, qu'on va initialiser avant la boucle avec une valeur qui **ne va pas rencontrer la condition** de la boucle (sinon la boucle ne s'enclencherait pas !).
2. A chaque passage de boucle ou **itération**, on vérifie si la condition est toujours vraie, si c'est le cas on va rentrer dans la boucle, c'est-à-dire exécuter les actions dans la boucle. Si la condition est fausse, on sort de la boucle et on continue à exécuter le code suivant.
3. On doit, au sein de la boucle, à la fin de toutes les actions, mettre à jour la variable de boucle pour que, à un moment, sa valeur ne corresponde plus à la condition et que la boucle s'arrête.
4. On retourne au point 2

Ce flux est illustré dans le schéma suivant :



L'incrémentation

Avant de plonger dans les types de boucles, arrêtons-nous un instant car c'est l'endroit idéal pour parler de l'incrémentation. Qu'est-ce que l'incrémentation ? C'est le fait d'ajouter une unité à un nombre au moyen d'une syntaxe courte. À l'inverse, la décrémentation permet de soustraire une unité.

Par exemple, pour incrémenter une variable numérique on peut le faire de cette manière

JS

```
var incrementVariable = 0;  
incrementVariable = incrementVariable + 1;
```

Mais on peut l'écrire de manière plus courte avec l'opérateur ++ (ou -- pour une décrémentation)

JS

```
var incrementVariable = 0;  
incrementVariable++;
```

Vous allez comprendre l'intérêt de l'incrémentation dans les exemples des boucles qui utilisent des *Array*.

Le type array

Bon, les boucles utilisent très régulièrement un autre type de données qu'on appelle *array*, ou tableaux en Français. Pour ne pas confondre avec les tables HTML, on va garder le terme *array*!

Là où les variables de type numérique ou string ne contiennent qu'une seule valeur, un *array* en anglais, est une variable qui contient plusieurs valeurs, appelées **items**. Chaque item est accessible au moyen d'un **indice** (*index* en anglais) et dont la numérotation commence **à partir de 0**. Voici un schéma représentant un tableau, qui stocke cinq items :

Index	0	1	2	3	4
Item	« HTML 5 »	« CSS 3 »	« Photoshop »	« JavaScript »	« jQuery »

Un *array* contenant 5 éléments aura donc un *index* maximal de 4, c'est-à-dire « nombre d'éléments – 1 ».

En JavaScript, on écrirait l'*array* ci-dessus de cette manière.

JS

```
var skills = ['HTML 5', 'CSS 3', 'Photoshop', 'JavaScript',  
             'jQuery'];
```

Et pour récupérer et modifier des valeurs, il faut préciser le numéro de l'*index* en crochets :



```
skills[1] = 'PHP'; //Modifier  
console.log(skills[1]); //Accéder
```

Types de boucles

while

C'est celle qui colle le plus au flux décrit ci-dessus. Voici sa syntaxe :

```
while (condition) {  
    action1;  
    action2;  
}
```

Exemple :



Tant qu'on n'atteint pas un *div* défini d'une collection de *div*, on rajoute du code HTML aux *div* parcourus

<http://jsfiddle.net/g8XJx/>

Une existe une variante de la boucle *while*, la boucle *do while*. Elle 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.

Cependant vous allez rarement l'utiliser, mais il faut savoir la reconnaître. Sa syntaxe :

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

for

La boucle que vous allez utiliser le plus souvent est la boucle *for*. Elle fonctionne de la même manière qu'un *while* mais l'initialisation, la condition et l'incréméntation se déclarent sur une seule ligne, comme ceci :

```
for (initialisation; condition; incréméntation) {  
    action;  
}
```

1. Dans le premier bloc, *l'initialisation*, on initialise une variable, généralement à 0.
2. On définit dans *condition*, la condition d'arrêt de la boucle, afin de ne pas créer la fameuse boucle infinie. Généralement on dit que la boucle continue tant que la variable initialisée est plus petite qu'une valeur définie.

3. Enfin, dans le bloc *d'incrémentation*, on indique que la variable sera **incrémentée** à chaque passage de la boucle, afin de la faire arriver à la valeur d'arrêt

Voici un exemple qui ne sert à rien mais qui aide à comprendre comment ça s'écrit

```
JS for (var iter = 0; iter < 5; iter++) {  
    alert('Itération n°' + iter);  
}
```

On peut aussi partir « à l'envers » en décrémentant jusqu'à 0.

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
- *Condition* : avant chaque passage de boucle
- *Incrémentation* : après chaque passage de boucle. Cela veut dire que, si vous faites un *break* dans une boucle *for*, le passage dans la boucle lors du *break* ne sera pas comptabilisé.

Un *break* permet de « casser » l'exécution d'une boucle et donc d'en sortir directement. L'instruction *continue*, quant à elle, 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. Mais vous l'utiliserez rarement !

Le fonctionnement même du JavaScript fait que la boucle *for* est nécessaire dans la majorité des cas comme la manipulation des *array*. Nous verrons aussi un peu plus tard une variante de la boucle *for*, appelée *for in*.

Exemples :

 Adapter la hauteur de 3 *div* en fonction du plus haut des *div*
<http://jsfiddle.net/WZ6ac/>

 Créer des *span* sous forme de « tags » à partir d'un *array*
<http://jsfiddle.net/EvBmX/>

Les fonctions

Que feriez-vous si vous devez exécuter plusieurs fois le même code ou un code similaire mais où les variables utilisées sont différentes ? Le copier-coller n'est pas une super bonne idée, c'est ennuyeux, ça provoque de la redondance, ça alourdit le code et le rend moins lisible et ça risque d'introduire des erreurs... D'où l'intérêt d'utiliser des fonctions !

Ici, oubliez le cours de math, une fonction c'est un bout de code qu'on a mis « à part » et qu'on a nommé (le nom de la fonction) afin de pouvoir le réutiliser facilement !

Déclaration

Vous en avez vu dans les exemples précédents, une fonction se déclare comme ceci (les crochets indiquent des éléments facultatifs)

```
function [nomDeLaFonction]([param1, param2,...]) {  
    CODE  
}
```

La déclaration d'une fonction contient les éléments suivants :

- Le mot-clé *function*
- Un **nom** (facultatif, nous le verrons plus loin)
- Une liste de **paramètres**. Ils sont mis entre crochets ci-dessus car ils sont facultatifs. Ils permettent de donner des paramètres, des données en entrée à la fonction. Ils ne sont utilisables que dans le corps de la fonction.
- Un **corps** ; du code quoi ; qui peut éventuellement renvoyer une **valeur de retour**. La fonction peut se contenter de n'exécuter que du code, mais elle peut également retourner, renvoyer un résultat ! Voilà qui est pratique 😊

Par exemple, nous avons vu dans un exemple précédent :

```
JS function hasClass(element, cls) {  
    return ( ' ' + element.className + ' ').indexOf( ' ' + cls + '  
    ' ) > -1;  
}
```

La fonction s'appelle *hasClass*, elle a 2 arguments, *element* et *cls* et elle retourne un résultat, booléen dans ce cas-ci.

Bien que ça ne soit pas requis par le langage, on déclare généralement les fonctions en haut du bloc, du *scope* où elle est utilisée.

Quand une fonction est nommée, ce nom est valide à travers le *scope* dans lequel cette fonction a été déclarée. On peut dire que le *scope*, c'est comme une « bulle ». Vous pouvez avoir plusieurs fonctions avec le même nom dans votre code, à condition que toutes ces fonctions soient restreintes au niveau de leur portée ; c'est-à-dire que chaque fonction soit limitée à sa propre « bulle ».

Portée des variables

On peut déclarer des variables dans le corps d'une fonction. De telles variables s'appellent des **variables locales**, car elles sont propres à la fonction, au contraire des variables globales, définies en dehors de toute fonction et disponible à travers tout le code – et donc, également, au sein des fonctions.

Une fois sorti du *scope* de la fonction, il n'est plus possible d'accéder à ses variables locales, si vous faites ça, votre code va bien planter ☹

Si vous déclarez et initialisez une variable locale du même nom qu'une variable globale, la valeur prise en compte dans la fonction sera celle de la variable locale. Une fois sorti de la fonction, la valeur globale reprendra le dessus. Veillez à utiliser des noms différents afin d'éviter toute confusion.

Si les variables sont propres à la fonction, mieux vaut les créer comme variables locales, c'est plus propre car il faut user des variables globales avec modération !



Créer des *span* sous forme de « tags » à partir d'un *array*, mais cette fois avec une fonction et utilisation d'une variable locale

<http://jsfiddle.net/cLLS2/>

Les variables sont dans le *scope* de la fonction dès leur déclaration jusqu'à la fin de la fonction, indépendamment de leur imbrication dans des blocs tels que des conditions ou des boucles.



```
if(true){  
    var dummy = "Hello";  
}  
alert(dummy);
```

En JavaScript, le code ci-dessus est tout à fait valide ! Alors que dans des langages comme le C, le Java,... ce code ne marcherait pas et génèrerait une erreur.

Appel

Une fonction s'appelle par son nom, et si des arguments sont passés entre les parenthèses, ils seront assignés aux paramètres spécifiés dans la déclaration de la fonction dans le même ordre.

Toujours par rapport à l'exemple précédent, la fonction est appelée de cette manière



```
var allSpans = document.querySelectorAll(".span3");  
var result = hasClass(allSpans[0], "alert-pink")
```

Si un nombre différent d'arguments qu'il n'y a de paramètres est passé lors de l'appel, JavaScript gère la situation comme ceci:

- S'il y a plus d'arguments que de paramètres définis dans la déclaration, les arguments « excédentaires » ne sont tout simplement pas assignés

- S'il y a plus de paramètres que d'arguments, les paramètres qui n'ont pas d'arguments correspondant sont mis à *undefined*



N'oubliez pas les () lors de l'appel, c'est ce qui différencie l'appel d'une fonction de sa référence.

Les fonctions anonymes

Les fonctions anonymes sont des fonctions qui n'ont simplement pas de nom ! Mais à quoi ça sert alors ? On ne saura jamais les appeler ?

En fait les fonctions anonymes sont très pratiques pour effectuer du code lors d'une action, d'un événement, sans devoir déclarer une fonction qui ne servirait qu'à une seule chose bien précise. Nous verrons cela plus en détails dans le chapitre sur le DOM et les événements.

Sachez déjà qu'il est possible d'assigner une fonction anonyme à une variable. Je vois que vous fronchez les sourcils... En fait il s'agit plutôt d'assigner une **référence** vers la fonction à une variable. Ça s'écrit comme ceci :

JS

```
//on assigne une référence vers une fonction anonyme
var changeBackground = function () {
    document.body.style.background =
    "url(\"http://subtlepatterns.com/patterns/escheresque_ste.png
    \") repeat scroll 0% 0% transparent";
};

//on appelle la fonction
changeBackground();
```



Testez ce code sur jsFiddle

<http://jsfiddle.net/fAUwy/>

Notez la présence d'un point-virgule à la fin de la déclaration de la fonction. Comme nous sommes dans le cas d'une assignation, il est nécessaire, alors qu'il ne l'est pas quand on déclare une fonction « normale ».

Vous verrez très souvent ce genre de syntaxe en JavaScript donc mieux vaut déjà s'y habituer !

Utiliser jQuery

Qu'est-ce que jQuery ?

Ce qui rebute souvent les développeurs concernant le JavaScript, c'est lors de la manipulation de la page HTML. En effet, faire de petites choses demande d'écrire pas mal de lignes de code et puis il faut toujours tenir compte de la compatibilité entre les différents navigateurs. Bref, pas toujours simple.

jQuery va nous être très précieux. Son slogan est « *write less, do more* » et de fait, cette librairie a l'avantage d'avoir une syntaxe courte et facile à comprendre. Elle permet de faire en 1 ligne ce qui demanderait des centaines de lignes de code en JavaScript pur. De plus, elle s'occupe pour nous de la compatibilité entre les navigateurs, c'est pas beau la vie ?

Un autre avantage non négligeable ce sont les plug-ins ! Il existe une flopée de ces « mini-librairies » qui se basent sur jQuery, programmées par des gens comme vous et moi et qui ont un but bien précis : un petit système de *rating*, la gestion du scroll, faire des *parallax*, faciliter l'utilisation de Google Maps, etc...

jQuery utilise tout simplement du JavaScript pur et dur, celui que nous avons vu précédemment, il n'a donc rien de « magique » mais une fois que vous y aurez goûté, vous aurez du mal à vous en passer ! Et tout ça gratuitement ☺

jQuery 2.0 délaisse le support de de IE6 à IE8 et est donc plus léger.

Ce n'est pas la seule librairie JavaScript qui existe sur le marché, cependant elles n'ont pas toutes la même philosophie et but que jQuery, certaines sont destinées à des programmeurs JavaScript un peu plus chevronnés ! Voici une petite liste de ses concurrents : MooTools, DojoToolkit, Prototype, Yahoo UI. Zepto, autre librairie, quant à elle utilise exactement la même syntaxe que jQuery mais est beaucoup plus légère car elle ne se préoccupe que des navigateurs récents et délaisse certaines fonctionnalités offertes par jQuery.

Inclure jQuery dans son projet

En local

Vous pouvez tout simplement télécharger la dernière version de jQuery sur <http://jquery.com/download/>, préférez la version « compressée » ou *minifiée*. Ensuite, ajoutez le fichier à votre projet et ajoutez le tag *script* aux pages qui nécessitent jQuery comme décrit dans le chapitre [Inclure du JavaScript dans une page HTML](#)

Via CDN

Une autre possibilité est d'utiliser un *Content Delivery Network*, ou CDN. En gros, c'est un site web qui héberge une version de jQuery et vous pouvez directement pointer vers ce site pour inclure jQuery dans vos pages.

Il en existe plusieurs : chez Microsoft, chez Google, et même chez jQuery. Il existe de multiples avantages :

- Vous ne devez pas gérer le fichier vous-même
- Les CDNs sont des serveurs répartis partout dans le monde, donc vos visiteurs téléchargeront un fichier d'un serveur près de chez eux, c'est donc plus rapide
- Comme d'autres sites utilisent les CDNs, vos visiteurs auront peut-être déjà jQuery dans le cache de leur navigateur vu qu'ils seront passés sur un de ces sites précédemment, votre navigateur ne devra dès lors plus télécharger le fichier.

Avec le CDN jQuery (pour la version 1.9.1 minifiée)

```
</> <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
```

Si vous souhaitez toujours la dernière version, vous pouvez utiliser ceci

```
</> <script src="http://code.jquery.com/jquery-latest.min.js"></script>
```

Avec le CDN Microsoft (pour la version 1.9.0 minifiée)

```
</> <script src="http://ajax.aspnetcdn.com/ajax/jquery/jquery-1.9.0.min.js"></script>
```

Avec le CDN Google (pour la version 1.9.1 minifiée)

```
</> <script  
src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></  
script>
```

Plus d'infos sur <http://jquery.com/download/>

Manipulation du DOM

Comprendre le Document Objet Model

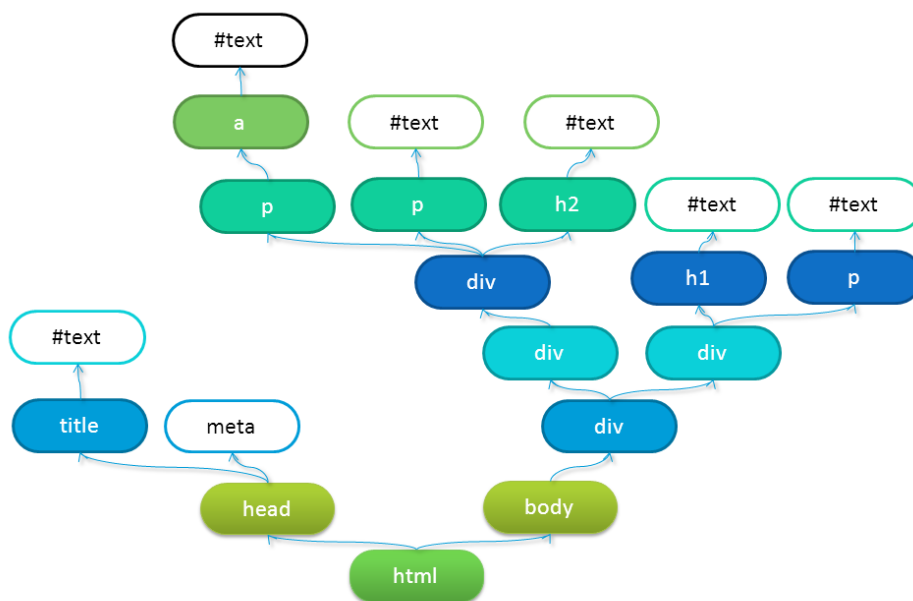
Comme vous êtes déjà familiers avec l'HTML, vous savez que le code HTML peut être représenté sous forme d'un arbre. Le code HTML suivant :

```
</> <html lang="en">
  <head>
    <title>Le titre de ma page</title>
    <meta name="description" content="youhouuu">
  </head>

  <body>
    <div class="container">
      <div class="hero-unit">
        <h1>Hello, world!</h1>
        <p>Texte d'introduction</p>
      </div>

      <div class="row">
        <div class="span4">
          <h2>Heading</h2>
          <p>Donec id elit non mi porta gravida at eget metus.</p>
          <p><a class="btn" href="#">Plus d'infos</a></p>
        </div>
      </div>
    </div>
  </body>
</html>
```

Peut être représenté sous cette forme



Le DOM ou **Document Object Model**, c'est une représentation du code HTML sous forme d'arbre d'**objets** (appelés *Nodes*). C'est sous cette forme que le HTML est « mémorisé » par le navigateur. Il est possible, en JavaScript, de parcourir cet arbre et donc d'accéder à tous les éléments d'une page HTML en vue de la modifier en rajoutant, modifiant ou supprimant des éléments!

Une remarque sur le schéma : le texte qui se trouve dans un tag (par exemple le texte d'un lien) est vu par le DOM comme un *Node* spécifique (de type *#text*).



Note de vocabulaire : dans un cours sur le HTML, on parlera de balises HTML (une paire de balises en réalité : une balise ouvrante et une balise fermante). Ici, en JavaScript, on parlera d'*élément HTML*, pour la simple raison que chaque paire de balises (ouvrante et fermante) est vue comme un élément.

Rappel sur les objets

Mais que signifie terme « objet » dans « Document Object Model »? Vous avez peut-être déjà entendu de « langage orienté objet », comme le Java ou le C#. Et bien JavaScript lui aussi est « orienté objet ».

C'est un peu abstrait alors on va éviter de partir dans des considérations trop informatiques mais pour faire simple disons qu'un objet c'est une manière de structurer des informations.

En fait, les types primitifs *Number*, *String* et *Boolean* rencontrés précédemment sont des types d'objets, mais nous ne devons pas nous en soucier car les nombres, textes ou booléens littéraux sont automatiquement convertis en objets.

On reconnaît les objets car ils peuvent contenir 2 types d'information: des **propriétés** et des **méthodes**. Une propriété est une variable qui appartient à l'objet et une méthode est une fonction propre à l'objet.

Un objet s'initialise ou *s'instancie* à l'aide de ce qu'on appelle un **constructeur**.

JS

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

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

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

Donc, chaque élément du DOM est un objet et possède donc des propriétés et des méthodes. Par exemple, si on récupère un élément qui correspond à un *div*, on va pouvoir faire :



Changer le *style* d'un *div* et mettre le contenu d'un *div* enfant en majuscules

<http://jsfiddle.net/kbAQU/>

Le *div* a une propriété *style* qui elle-même a une propriété *backgroundColor* et *padding* qui permettent respectivement d'influer sur les styles CSS *background-color* et *padding*.

Dans la majorité des exemples vus précédemment, on utilisait des objets ainsi que leurs propriétés et méthodes.



Note de vocabulaire : dans le vocabulaire du DOM, on parle souvent de *Node* (nœud) et d'*Element* (élément)

1. Un *Node* peut être n'importe quel type d'objet dans le DOM
2. Un *Element* est un type de *Node* spécifique, qui peut être exprimé sous forme de balise HTML, qui peut avoir des attributs comme *id* ou *class*, peut avoir des éléments enfants, etc...

On dit qu'un *Element* **hérite** d'un *Node*. Un *Element* possèdera donc toutes les propriétés et méthode d'un *Node*.

Attention : le *Node* **#text** qui représente du texte simple, est un *Node* et pas un *Element* !

Les objets *window* et *document*

L'objet window

L'objet *window* est le point de départ du DOM, il représente la fenêtre du navigateur. En JavaScript, l'objet *window* est dit **implicite**, c'est-à-dire qu'il n'y a généralement pas besoin de le spécifier dans son code.

De même, lorsque vous déclarez une variable dans le contexte global de votre script, cette variable deviendra en vérité une propriété de l'objet *window*. Il en va de même avec les fonctions, les fonctions globales sont en fait des méthodes de l'objet *window* !



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 vous utilisez cette variable ! Ça peut être très pratique mais utilisez cette particularité en connaissance de cause et préférez le mot-clé *var* pour déclarer vos variables.

Quelques propriétés pratiques de l'objet *window* :

- *history* : permet d'accéder à l'historique de navigation pour la fenêtre en cours
- *document* : permet d'accéder au document HTML (voir chapitre suivant)
- *location* : permet d'obtenir des informations sur l'URL courant
- *navigator* : permet d'obtenir des informations du navigateur



Affichage de quelques propriétés de l'objet *window* et de ses sous-objets

<http://jsfiddle.net/4v3Ed/>

Quelques méthodes pratiques de l'objet *window* :

- *alert()* : pour afficher un message (pratique pour débogger ;))
- *open()* : pour ouvrir une nouvelle fenêtre, comme une bonne vieille pop-up – maintenant bloquée par les navigateur
- *scrollTo()* : pour positionner le scroll jusque un point précis



Toutes les propriétés et méthodes de l'objet *window*

<https://developer.mozilla.org/en-US/docs/DOM/window>

L'objet document

L'objet *document* est une propriété de l'objet *window*, il correspond au tag *<html>*. C'est cet objet qui représente la page web et c'est via lui que l'on va pouvoir accéder aux éléments HTML du DOM pour ensuite les modifier.

Lecture et modification du DOM

Il est donc possible, en JavaScript et jQuery, de modifier le DOM. C'est en modifiant le DOM qu'on peut par exemple afficher un tooltip quand on clique sur une image, une fenêtre modale, un calendrier ou des messages d'erreurs lorsqu'un formulaire n'est pas valide. Chaque modification consiste en :

1. Sélectionner l'élément HTML qu'on souhaite modifier (par exemple, un *div* à afficher en tant que tooltip ou fenêtre modale)
2. Faire quelque chose avec cet élément, comme :
 - a. Changer ses propriétés, par exemple sa position, sa couleur de fond,...
 - b. Ajouter du contenu à l'élément
 - c. Supprimer l'élément de la page HTML
 - d. Récupérer de l'information de l'élément, par exemple le texte d'un *input*
 - e. Ajouter/supprimer des classes CSS, par exemple pour changer sa visibilité, sa typographie,...

Une fois l'élément sélectionné, on peut évidemment lui appliquer plusieurs actions.

Nous allons voir comment cela est possible en JavaScript pur et en jQuery.

En JavaScript pur

Dans ce chapitre nous allons brièvement exposer les propriétés et méthodes des objets *Node* et *Element* qui permettent d'accéder plus en profondeur au DOM et de le modifier.

Nous n'allons pas nous attarder dessus, les mêmes possibilités existent en jQuery et nous les verrons dans le chapitre ad hoc. Mais cela vous donnera déjà un aperçu des possibilités offertes « de base » par JavaScript.

Sélectionner un élément (méthode classique)

L'objet *document* possède 3 méthodes qui permettent d'accéder aux éléments du DOM :

- *getElementById()* : permet d'accéder à un élément en spécifiant son *id* (attribut *id* en HTML)
- *getElementsByTagName()* : retourne un *array* d'éléments qui correspondent au tag spécifié
- *getElementsByName()* : retourne un *array* d'éléments d'un formulaire dont le l'attribut *name* correspond au *name* spécifié

Bon, ces méthodes sont quand un peu limitées pour des actions complexes, si par exemple on veut accéder à tous les *td* d'une table, on va devoir écrire un code assez moche et lourd, avec des boucles... mais disons que c'est pratique et performant pour des actions très simples et ciblées.

Sélectionner un élément (méthode récente)

Du coup, deux méthodes sont récemment venues compléter les 3 « classiques » pour accéder plus rapidement aux éléments du DOM. Malheureusement elles ne sont pas supportées par tous les browsers. Ce qui les rend un peu inutiles...

Elles utilisent, comme jQuery d'ailleurs, **les sélecteurs CSS** sous forme de *string* en paramètre.

- *querySelector()* : renvoie le premier élément trouvé qui correspond au sélecteur CSS
- *querySelectorAll()* : revoie un *array* avec **tous** les éléments qui correspondent au sélecteur CSS



Tant qu'on n'atteint pas un *div* défini d'une collection de *div*, on rajoute du code HTML aux *div* parcourus – notez l'utilisation de *querySelectorAll()*

<http://jsfiddle.net/g8XJx/>



Adapter la hauteur de 3 *div* en fonction du plus haut des *div* – notez l'utilisation de *getElementById()* et *children*

<http://jsfiddle.net/WZ6ac/>

Modifier un élément

Propriétés et méthodes de l'objet *Element* permettant de le modifier:

- *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.
- Attributs accessibles comme *id*, *title*, *href*, *style*, *value*, etc...
- *className* : récupérer et éditer l'attribut HTML *class* (attention, c'est une *string* avec toutes les classes sous format texte)
- *innerHTML* : récupérer et éditer l'HTML enfant d'un élément sous format texte
- *innerText* (< IE 9) et *textContent* : le même que *innerHTML* excepté le fait que seul le texte est récupéré, et non les balises.

Propriétés et méthodes de l'objet *Node* permettant d'ajouter ou retirer des éléments enfant:

- *appendChild()* : ajouter un *Node* aux enfants
- *insertBefore()* : ajouter un *Node* aux enfants avant un autre *Node*
- *insertAfter()* : ajouter un *Node* aux enfants après un autre *Node*
- *removeChild()* : retirer un *Node* enfant
- *replaceChild()* : remplacer un *Node* enfant
- *hasChildNode()* : retourne un booléen qui permet de savoir si le *Node* a des enfants



Créer des *span* sous forme de « tags » à partir d'un *array* – notez l'utilisation de *createElement()*, *createTextNode()* et *appendChild()*

<http://jsfiddle.net/cLLS2/>

Créer un nouvel élément

Il est également possible de créer des nouveaux *Element* en JavaScript via la méthode *document.createElement()*. Cette méthode demande un paramètre une *string* avec le nom du tag correspondant à l'élément souhaité, par exemple :

JS

```
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. La méthode `document.createTextNode()` permet quant à elle de créer un *Node #text*. Ces méthodes sont beaucoup plus performantes que la création d'éléments en jQuery, que nous verrons également.

Cela dit, on peut déjà travailler sur l'élément créée, en lui ajoutant des attributs ou même des événements (que nous verrons dans le chapitre suivant). On peut l'ajouter au DOM via la méthode `appendChild()` de l'objet *Node*.

En jQuery

jQuery est beaucoup plus puissant pour ce qui est de l'accès et la modification du DOM. Il utilise intensivement les sélecteurs CSS et s'occupe de la compatibilité avec les navigateurs, même avec le bon vieux IE6.

Sélectionner un élément à l'aide des sélecteur CSS

Si vous êtes familiers avec les CSS, vous avez les armes pour travailler avec jQuery ! Ils permettent de sélectionner un élément HTML à partir de son *style*.

La syntaxe de base est celle-ci :

`$('sélecteur')`

Le caractère `$` est ce qu'on appelle *l'objet jQuery*. C'est le point de départ de n'importe quelle action que vous allez effectuer avec jQuery. Vous verrez de temps en temps cette syntaxe :

`jQuery('sélecteur')`

Ce qui revient à la même chose que le caractère `$`, mais qu'on utilise parfois par soucis de compatibilité – si une autre librairie utilise déjà le symbole `$`.

Les sélecteurs CSS de base comme l'ID, la classe ou le nom de l'élément sont le cœur des CSS

ID

Comme [`document.getElementById\(\)`](#), on peut sélectionner un élément à partir de son ID avec jQuery

JS

```
//façon oldschool  
var message = document.getElementById( 'message' );
```

JS

```
//façon jQuery  
var message = $( '#message' );
```

Notez l'utilisation du caractère # avant l'ID en question, comme en CSS

Nom de l'élément

Comme [document.getElementsByTagName\(\)](#), jQuery permet de sélectionner des éléments HTML à partir de leur nom de tag.

JS

```
//façon oldschool  
var linksList = document.getElementsByTagName('a');
```

JS

```
//façon jQuery  
var linksList = $('a');
```

C'est quand même plus court non ? ☺

Classe

Un autre sélecteur super pratique qui permet de sélectionner un ou plusieurs éléments à partir de leur **classe CSS**.

Cette façon de faire est possible avec les nouveautés du DOM [document.querySelector\(\)](#) et [document.querySelectorAll\(\)](#) mais comme déjà dit, ces méthodes ne sont pas supportées par tous les navigateurs. Autant donc passer directement par jQuery !

Un exemple d'utilisation :

JS

```
//façon jQuery  
var subMenus = $('.submenu');
```

Notez l'utilisation du . qui définit une classe comme en CSS.



Sélecteurs CSS de base – activez la console Firebug pour voir les résultats

<http://jsfiddle.net/aASEX/>

Vérifier si un élément existe

Pour simplement vérifier qu'un élément existe en jQuery, il faut sélectionner l'élément et vérifier si sa propriété *length* est plus grande que 0

JS

```
if($("#blabla").length > 0){  
    console.log(« OK ») ;  
}
```

Les sélecteurs avancés

Bien souvent, vous n'utiliserez que les **sélecteurs de base**. Mais pour des opérations plus complexes vous pourriez être amenés à combiner des sélecteurs et utiliser des fonctions avancées telles que :

- **Les sélecteurs descendants** : pour cibler un tag qui se trouve dans un autre tag. Par exemple pour pouvoir sélectionner tous les liens situés dans un *ul* qui aurait l'ID « navbar » vous écririez :

JS

```
var navbarLinks = $('#navbar a');
```

- **Les sélecteurs enfants** : pour cibler un tag qui est enfant d'un autre. Par exemple, sélectionner les *p* qui se trouve **directement** sous le *body* :

JS

```
var childParagraphs = $('body > p');
```

- **Les sélecteurs adjacents** : pour sélectionner un tag qui se trouve juste après un autre. Si vous voulez par exemple sélectionner un *div* qui se trouve dans l'HTML juste après un *h2* avec l'ID « header », vous séparez les 2 sélecteurs par un '+'

JS

```
var adjacentDiv = $('#header + div');
```

- **Les sélecteurs d'attributs** : pour sélectionner des éléments qui possèdent un certain attribut ou dont un certain attribut possède une valeur spécifique. Par exemple si vous voulez sélectionner tous les tags *img* qui possèdent un attribut *alt*, ou tous les tags *a* qui pointent vers un site extérieur (dont le *href* commence par 'http://')

Voici les différentes possibilités :

- `[attribut]` : sélectionne les éléments qui **possède** l'attribut spécifié
- `[attribut= « valeur »]` : sélectionne les éléments dont l'attribut spécifié **possède** la **valeur** indiquée
- `[attribut^= « valeur »]` : sélectionne les éléments dont l'attribut spécifié **commence** avec la valeur indiquée

- `[attribut$= « valeur »]` : sélectionne les éléments dont l'attribut spécifié **termine** avec la valeur indiquée
- `[attribut*= « valeur »]` : sélectionne les éléments dont l'attribut spécifié **contient** avec la valeur indiquée



Plus sur les sélecteurs : <http://api.jquery.com/category/selectors/>

Modifier un élément

Une fois votre élément sélectionné, vous pouvez effectuer une série d'action à partir de cet élément.

Lire et modifier son texte

La méthode `text()` permet de modifier le texte d'un élément HTML, par exemple le texte d'un lien, d'un h2, etc... Si vous envoyez du code HTML à cette méthode, elle affichera le code HTML dans la page, comme si vous vouliez montrer du code aux visiteurs.

JS

```
$('#errors h2').text('Pas d'erreurs trouvées');
```

Si vous ne donnez aucun texte en paramètre à la méthode, dans ce cas vous pouvez récupérer la valeur du texte de l'élément, pour le mettre dans une variable par exemple.

La méthode `html()` quant à elle permet de modifier le contenu HTML d'un élément. Si l'élément sélectionné contient déjà du *inner HTML*, celui-ci sera remplacé par celui que vous spécifiez en paramètre.

JS

```
$('#errors h2').html('<span>Pas d'erreurs trouvées</span>');
```

Si vous ne donnez aucun texte en paramètre à la méthode, dans ce cas vous pouvez récupérer la valeur du *inner HTML* l'élément, pour le mettre dans une variable par exemple.



Modification de contenu avec `text()` et `html()`

<http://jsfiddle.net/kmRBL/>

Lire et modifier ses attributs

La méthode `attr()` permet de lire la valeur d'un attribut d'un élément. Par exemple, pour récupérer l'adresse d'une image dans la page, vous pourriez faire ceci

JS

```
var imagePath = $('#banner img').attr('src');
```

Si vous passez un 2^{ème} attribut à la méthode *attr()*, vous pouvez alors définir la valeur de l'attribut en question. Si vous souhaitez par exemple modifier le chemin vers l'image de l'exemple précédent :

JS

```
$('#banner img').attr('src', 'images/newImage.png');
```

Si vous souhaitez effacer un attribut, là il faut utiliser la méthode *removeAttr()*. Par exemple pour retirer l'attribut *alt* de l'image :

JS

```
$('#banner img').removeAttr('alt');
```



Modification d'attributs avec *attr()* et *removeAttr()*

<http://jsfiddle.net/MLtMt/>

Lire et modifier son style

Via les classes CSS

jQuery offre plusieurs méthodes pour manipuler l'attribut *class* d'un élément.

addClass() ajoute la classe CSS spécifiée en paramètre à l'élément sélectionné. Cette méthode n'écrase pas les classes définies précédemment mais se contente de rajouter la classe spécifiée.

JS

```
$('.btn').addClass('btn-success');
```

removeClass() fait exactement l'inverse en enlevant la classe CSS spécifiée de l'élément sélectionné.

JS

```
$('.btn').removeClass('btn-success');
```

toggleClass() quant à elle ajoute la classe si elle n'existe pas et l'enlève si elle existe. C'est pratique pour des éléments qui ont un état « on/off » par exemple.



Modification de classes CSS avec *addClass()* et *removeClass()*

<http://jsfiddle.net/zgSF8/>

Via les propriétés CSS

Imaginons que vous voulez directement changer le style d'un élément sans passer par des classes CSS. C'est possible via la méthode *css()*. Vous pouvez utiliser de 3 manières différentes :

- pour trouver la valeur existante d'une propriété CSS
- pour modifier une propriété CSS d'un élément
- pour modifier plusieurs propriétés à la fois

Vous allez surtout utiliser cette méthode pour modifier une propriété. Par exemple :

JS

```
$( '.btn' ).css( 'font-size', '200%' );
```



Modification de styles CSS avec *css()*

<http://jsfiddle.net/fHPYs/>

Supprimer un élément

Vous pouvez bien entendu effacer un ou plusieurs éléments de la page web avec *remove()*.

Mettons que vous avez un *div* avec l'ID « popup », qui serait une fenêtre modale, et que vous voulez l'effacer de la page :

JS

```
$( '#popup' ).remove();
```

Cela marche aussi si vous spécifiez un sélecteur qui concerne plusieurs éléments – ils seront tous effacés.

Vous pouvez également remplacer un élément par autre chose avec *replaceWith()*. Disons que vous avez une *img* avec l'ID « product101 » et vous voulez mettre du texte à la place car le produit lié a été ajouté à un panier d'achat.

JS

```
$( '#product101' ).replaceWith( '<p>Ajouté au panier</p>' );
```

Ajouter du contenu à la page

Créer un nouvel élément

Il est bien sûr possible de créer du nouveau contenu « dynamiquement » en JavaScript et jQuery de le rajouter à la page. Pour ce faire, il faut spécifier à l'objet jQuery le tag (avec balises) qui correspondant à l'élément que vous souhaitez créer.

Par exemple, pour créer un nouvel élément qui correspond à un *span* :

JS

```
var newSpan = $("<span></span>");
```



Vous pouvez également créer un nouvel élément en utilisant la manière JavaScript pure avec [*document.createElement\(\)*](#) – cette façon de faire est plus rapide que la manière jQuery.

Rajouter un nouvel élément à la page

Une fois le nouvel élément créée, vous pouvez utiliser les méthodes *append()*, *prepend()*, *after()* ou *before()* pour l'ajouter à la page web :

- *append()* : ajoute l'élément (ou le code HTML spécifié en string) en paramètre comme dernier enfant de l'élément sélectionné
- *appendTo()* : ajouter l'élément sélectionné à un parent, comme premier enfant
- *prepend()* : ajoute l'élément (ou le code HTML spécifié en string) en paramètre comme premier enfant de l'élément sélectionné
- *prependTo()* : ajouter l'élément sélectionné à un parent, comme dernier enfant
- *after()* : ajoute l'élément (ou le code HTML spécifié en string) directement après l'élément sélectionné
- *before()* : ajoute l'élément (ou le code HTML spécifié en string) directement avant l'élément sélectionné

JS

```
var newSpan = $("<span></span>");  
$('#tags').append(newSpan);
```



Tant qu'on n'atteint pas un *div* défini d'une collection de *div*, on rajoute du code HTML aux *div* parcourus – on utilise jQuery et la méthode *append()*

<http://jsfiddle.net/4XbR5/>

Enchaînement de méthodes

Une fonctionnalité super intéressante de jQuery est ce qu'on appelle l'enchaînement de méthodes. En fait, toutes les méthodes que nous venons de voir peuvent être connectées, mises à la chaîne l'une de l'autre. Cela permet de faire plusieurs choses en une seule ligne de code !

Par exemple, si nous voulons en une seule fois :


- créer un élément *span*
- lui ajouter une classe « *orange* »
- lui ajouter du texte « *HTML 5* »
- le rajouter à un autre élément avec l'ID « *tags* »

Nous pouvons écrire ceci :

```
JS $("#tags").append(
    $("<span></span>").addClass("orange").text('HTML 5')
);
```

Génial non ? Si vous trouvez cela dur à lire, vous pouvez écrire chaque méthode à la ligne, comme ceci :

```
JS $("#tags").append(
    $("<span></span>")
    .addClass("orange")
    .text('HTML 5')
);
```

 Créer des *span* sous forme de « *tags* » à partir d'un *array* – on utilise jQuery avec la création d'élément, *append()* et *text()*, avec enchaînement des méthodes

<http://jsfiddle.net/GDALW/>

Agir sur plusieurs éléments d'une sélection

Si on voulait effectuer une même action sur plusieurs éléments, comment s'y prendrait-on ?

Par exemple, imaginons qu'on ait une page avec plusieurs chapitres et que nous voulons générer une table des matières automatiquement, sans se prendre la tête. Si un jour on rajoute un chapitre, la table des matières se mettrait à jour toute seule.

Vous vous rappelez des boucles et plus précisément de la boucle *for* ? Elle nous permettait de boucler sur un ensemble d'éléments, hé bien une possibilité similaire existe en jQuery : la méthode *each()*.

La méthode *each()* peut être chaînée à une sélection afin de « boucler » sur chaque élément de la sélection et leur appliquer la même action, au sein d'une fonction anonyme. Sa syntaxe est celle-ci :

```

$( 'sélecteur' ).each(function() {
    action1;
    action2;
    action3;
    ...
});

```

Ça fait beaucoup de (et de { tout ça hein ? Les parenthèses et les accolades sont mises en évidence pour plus de clarté. Commencez toujours par écrire le bloc d'instruction sans code à l'intérieur puis remplissez avec votre code, vous ferez moins d'erreurs.

Une fois à l'intérieur du each(), vous allez bien souvent devoir accéder à l'élément courant afin de lire ou modifier ses propriétés. Dans le cas de notre table des matières, nous devrons accéder au nom du chapitre et à son ID pour pouvoir créer un lien (ancree).

Cela est possible via le mot-clé *this*, ce mot-clé correspond à l'élément courant du DOM (*Element*). A chaque passage de boucle, la valeur de *this* changera : il correspondra au premier élément au premier passage, au deuxième élément au deuxième, ainsi de suite... Cependant *this* ne vous permettra d'accéder qu'aux méthodes et propriétés standard de l'objet *Element*, si vous voulez faire du jQuery avec, il va falloir le « transformer » en équivalent jQuery en écrivant *\$(this)*.

JS

```

$("h2").each(function() {
    //on récupère le texte
    var chapterName = $(this).text();
    console.log(chapterName);
});

```

∞

Adapter la hauteur de 3 *div* en fonction du plus haut des *div* –on utilise jQuery et les méthodes *each()* et *css()*

<http://jsfiddle.net/jY4bL/>

∞

Créer une table des matières automatiquement en jQuery

<http://jsfiddle.net/GNGRm/>

i

Plus sur le DOM: <http://api.jquery.com/category/manipulation/>

Les filtres

jQuery donne la possibilité de filtrer les sélection en fonction de certains critères. Par exemple, si vous ajoutez `:even` à la suite d'un sélecteur, alors vous ne récupérez que les éléments pairs d'un groupe d'éléments.

De plus, vous pouvez des éléments qui contiennent un tag particulier, un texte spécifique, les éléments cachés et même ceux qui ne correspond pas à un certain sélecteur. Pour utiliser un filtre, rajouter un deux-points suivi du nom du filtre :

JS

```
//ne renverra que les lignes paires  
$('tr:even');
```

Voici une liste des sélecteurs de filtrage possibles :

- `:even` et `:odd` : ne renvoie que les éléments pairs ou impairs d'un groupe d'éléments – attention, ils se basent sur l'index, comme dans un tableau, c'est-à-dire 0,1,2,3,... `:even` correspondra donc en fait, aux éléments impairs, c'est un peu bizarre mais c'est comme ça ☺
- `:first` et `:last` : sélectionne le premier ou dernier élément d'un groupe d'éléments
- `:hidden` : trouve les éléments cachés (avec `display:none` ou sur lesquels `.hide()` a été appelé)
- `:visible` : le contraire de `:hidden`
- `contains()` : trouve les éléments qui contiennent un texte spécifique spécifié en paramètre (s'utilise comme une méthode)

Mais il existe aussi des méthodes pour filtrer les éléments d'une sélection :

- `filter()` : permet de sélectionner des éléments qui rencontrent le sélecteur spécifié en paramètre
- `not()` : permet de sélectionner des éléments qui ne rencontrent pas le sélecteur spécifié en paramètre
- `has()` : trouve les éléments qui contiennent un autre sélecteur spécifié en paramètre
- `is()` : vérifie les éléments de la sélection correspondent au sélecteur spécifié en paramètre



Exemples de filtres

<http://jsfiddle.net/5NEuW/>



Plus sur les filtres : <http://api.jquery.com/category/traversing/filtering/>

Les événements

Le concept d'événement

On dit souvent que JavaScript permet de faire des sites web *interactifs*, en effet, on voit souvent dans les sites web des actions qui se déclenchent lorsqu'on interagit avec des éléments sur la page : boîte de dialogue qui s'ouvre lors d'un clic de souris sur un lien, des éléments qui changent de taille ou de couleur quand la souris passe dessus, des interactions possibles avec le clavier, etc... tout cela est possible grâce aux **événements**.

Un événement représente le moment précis où quelque chose arrive, pour vous permettre de mieux comprendre, voici une liste des événements principaux des éléments HTML :

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

Mais il existe des événements spécifiques pour l'objet [*window*](#) également, comme :

Nom	Action pour le déclencher
<i>resize</i>	Redimensionner la fenêtre du navigateur
<i>scroll</i>	Scroller dans la fenêtre
<i>load</i>	La page web a fini d'être chargée (tous les objets du DOM sont chargés ainsi que les images et sous-frames).

Pour info, l'événement *resize* sera très utile dans le cas où vous voulez faire un site web *responsive*, c'est-à-dire qui s'adapte en fonction de l'écran du visiteur (écran normal, tablette, smartphone, etc...)

Associer un événement à un élément

Associer un événement à un élément en JavaScript ou jQuery ressemble très fort à ce que nous avons vu pour modifier les propriétés d'un élément, cela consiste en 3 étapes :

1. Sélectionner le (ou les) élément(s) HTML avec le(s)quel(s) le visiteur va interagir

2. Assigner l'événement à l'élément : on définira le nom de l'événement par rapport au tableau du chapitre précédent.
3. Passer une fonction à l'événement : cette fonction contient le code, l'action qui va se passer lors du déclenchement de l'événement. On appelle cela le *handler*.

En JavaScript

Concrètement il existe plusieurs manières d'associer un événement à un élément HTML en JavaScript pur, retenez surtout la première :

1. Via le DOM, en associant l'événement en JavaScript de la manière « DOM-0 ». On assigne une fonction anonyme à une propriété qui se nomme « **on** » + **nom de l'événement**, comme ceci :

JS

```
//on récupère l'élément HTML
var myButton = document.getElementById('myButton');

//on assigne une fonction anonyme à sa propriété 'onclick'
myButton.onclick = function() {
    console.log("Clic sur le bouton");
};
```

2. Via le DOM, à la manière « DOM-2 » via la méthode *addEventListener()*. C'est plus « propre » mais Internet Explorer version 6 à 8 utilise une autre manière (*attachEvent()*)

JS

```
//on récupère l'élément HTML
var myButton = document.getElementById('myButton');

myButton.addEventListener('click', function() {
    console.log("Clic sur le bouton");
}, false);
```

3. Directement depuis l'HTML. Cette méthode est à proscrire car elle mélange le code et l'HTML et ne respecte donc pas la *separation of concerns* et ce n'est pas ce qu'on appelle du *unobstrusive JavaScript*. C'est comme quand vous mettez des styles directement depuis l'attribut *style* sans passer par une CSS, pas terrible quoi. Cela s'écrit de cette manière :

</>

```
<button id="myButton" onclick="console.log('Clic sur le bouton');">Un bouton</button>
```

Avec jQuery

Avec jQuery, nous allons comme toujours utiliser les sélecteurs CSS pour sélectionner le(s) élément(s) au(x)quel(s) associer l'événement.

Ensuite nous allons chaîner une méthode dont le nom correspond exactement [au tableau des événements](#). Par exemple *click()* ou *mouseover()*.

Enfin nous allons passer une fonction (*handler*), généralement une fonction anonyme.

Syntaxe :

```
$(‘sélecteur’).événement(function() {  
    action1;  
    ...  
});
```

JS

```
//action sur le clic du bouton avec l’ID « start »  
$(‘#start’).click(function() {  
    console.log(‘Vous avez cliqué !’);  
});
```



Créer des *span* sous forme de « tags » à partir d’une *array* – on utilise jQuery avec des événements souris

<http://jsfiddle.net/S8nt7/>

Il est possible d’appeler un événement manuellement (si vous voulez par exemple, simuler un clic de souris sur un élément à partir d’une autre action), dans ce cas, il suffit d’appeler la méthode correspondant à l’événement sur l’élément sélectionné mais sans fournir de paramètre à la méthode.

Il existe un événement supplémentaire en jQuery qui n’existe pas nativement en JavaScript, c’est l’événement *hover()*. Il est à utiliser quand vous voulez effectuer une action lorsque la souris passe sur un élément et puis une autre action lorsque la souris « sort » de l’élément.

Il faut donc lui donner 2 « actions », en code cela va se traduire par 2 fonctions anonymes à passer à l’événement. Voici sa syntaxe :

```
$(‘sélecteur’).hover(  
    function() {  
        //Action quand la souris passe sur l’élément  
    },  
    function() {  
        //Action quand la souris part de l’élément  
    }  
);
```

Par exemple, supposons que le visiteur passe sa souris au-dessus d’un lien avec l’ID « *menu* », vous voulez faire apparaître un *div* (par défaut non-visible) avec l’ID « *submenu* ». Quand la souris quitte le lien, vous voulez faire de nouveau disparaître le « *submenu* »

JS

```
$(‘#menu’).hover(function() {  
    $(‘#submenu’).show();  
}, function() {  
    $(‘#submenu’).hide();  
});
```

```
});
```



Afficher la table des matières auto-générée lors d'un passage de la souris

<http://jsfiddle.net/r8rPS/>

.on(), quand les éléments n'existent pas encore

Quand un événement est associé à un ou plusieurs éléments, ceux-ci doivent être présents dans la page HTML lors du chargement de la page. Cela veut dire que si vous associez un click de la manière vue précédemment et qu'un nouvel élément qui correspond au sélecteur associé est créée dynamiquement (par jQuery), l'événement ne sera pas lancé pour ce nouvel élément.

Dans ce cas, il faut utiliser `.on()` pour associer l'événement. Cette méthode demande 3 paramètres :

- Le nom de l'événement
- Un sélecteur auquel sera associé l'événement
- Un *handler*

Cette méthode doit donc être appliquée sur un élément qui existe au chargement de la page et qui sera un parent du nouvel élément créée auquel l'événement sera associé. Dans l'exemple ci-dessous, `.on()` est appliqué au *div* qui possède la classe « container » et le code du *handler* sera exécuté lors d'un *click* sur un lien enfant de ce div, que ce lien existe au chargement de la page ou non.



```
$("#div.container").on("click", "a", function(){  
    console.log("I clicked on the link");  
});
```

A noter que la manière de faire ci-dessus est aussi plus performante dans le cas où vous devez associer un événement à beaucoup d'éléments.



Plus d'infos sur `.on()`: <https://api.jquery.com/on/>

Il est également possible de **désassocier un événement** attaché avec `.on()` avec la [méthode `.off\(\)`](#)

Attendre le chargement de l'HTML

Quand une page web se charge, le navigateur va immédiatement essayer de lancer tous les scripts qu'il rencontre. Ça peut être problématique avec un script qui manipule le contenu de la page et qui se lance avant que tout le contenu de la page soit chargé... Cela générera plein d'erreurs !

Il faut donc dire au script « attend que la page soit chargée pour te lancer ».

C'est possible en JavaScript via la méthode *window.load()* mais cette méthode attend que tous les contenus associés (images, CSS, scripts, animations Flash, vidéos...) soient chargés pour se lancer et ça peut prendre pas mal de temps ! Heureusement, jQuery fournit la méthode **ready()** qui attend que la page soit chargée mais n'attend pas tous les contenus comme les images, vidéos, ou autre.

La syntaxe de cette instruction est celle-ci :

```
$(document).ready(function() {  
    //votre code ici  
});
```

Une variante plus courte :

```
$(function() {  
    //votre code ici  
});
```

Vous devez rajouter ce code à chaque page où vous souhaitez exécuter du code lorsque la page est chargée. Si vous vous retrouvez avec plusieurs *ready()*, ce n'est pas un problème.

L'objet *Event*

Vous vous demandez peut-être : « mais comment je fais pour récupérer la position de la souris alors ? » ou « comment je peux savoir sur quelle touche l'utilisateur vient d'appuyer ? ». On peut bien sûr récupérer ces informations, ça se fait via l'objet *Event*.

Cet objet est bien 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 en jQuery

```
JS //variable e en paramètre : l'objet Event !  
$( '#myButton' ).click(function(e) {  
    console.log(e.pageX);  
    console.log(e.pageY);  
});
```

L'objet *Event* contiendra des propriétés différentes en fonction du type d'événement, si l'événement est une action avec la souris il contiendra *pageX* et *pageY* (les coordonnées de la souris), tandis que si c'est un *keypressed* il contiendra *which* (le code ASCII de la touche).



Passer d'un chapitre à l'autre avec les touches du clavier (« p » et « n »), à la Google Reader. Notez l'utilisation d'un **constructeur** et de la méthode **push()** d'un *array*

<http://jsfiddle.net/sfWns/>



Passer d'un chapitre à l'autre avec les touches du clavier (« p » et « n ») à la Google Reader. Notez l'utilisation d'un **constructeur** et de la méthode **push()** d'un **array**.

Remarquez la gestion du scroll pour faire apparaitre un lien « retour au début »

<http://jsfiddle.net/Wfgcm/>



Une librairie jQuery pour gérer les positions du scroll

<https://github.com/sxalexander/jquery-scrollspy>

Stopper le comportement normal d'un événement

Certains éléments HTML ont une réponse préprogrammée aux événements, par exemple un lien charge une page quand on clique dessus, un formulaire envoie ses données à un serveur web, etc... Il arrive que l'on veuille « couper » le comportement normal de cet événement : que le lien ne redirige pas vers la page ou que le formulaire ne poste pas ses données vers le serveur car certains de ses champs ne sont pas remplis correctement.

Dans ce cas, il suffit, une fois dans le corps de la fonction associée à l'événement, d'appeler la méthode *preventDefault()* de l'objet Event ou de simplement retourner la valeur *false* qui aura pour effet d'appeler *preventDefault()* et *stopPropagation()* (méthode empêchant à l'événement de « remonter », de se propager aux éléments parents).

JS

```
$('#myLink').click(function(e) {  
    e.preventDefault();  
});
```

Equivalut à

JS

```
$('#myLink').click(function(e) {  
    return false;  
});
```



Plus sur les événements : <http://api.jquery.com/category/events/>

An introduction to DOM events : <http://coding.smashingmagazine.com/2013/11/12/an-introduction-to-dom-events/>

Les effets visuels avec jQuery

Dans les chapitres précédents, vous avez appris la base de jQuery : sélectionner des éléments, les modifier, et leur attacher des événements.

Maintenant c'est l'heure de passer au côté le plus fun de jQuery, les effets visuels ! Comme pour la modification et les événements, il faut :

1. Sélectionner le (ou les) élément(s) HTML avec le(s)quel(s) sur le(s)quel(s) on veut appliquer un effet
2. Appliquer l'effet via la méthode ad hoc en donnant des paramètres à la méthode d'effet pour le « contrôler »

Apparaître et disparaître

Les effets les plus simples sont aussi les plus souvent utilisés. Voici les méthodes qui permettent de faire apparaître et disparaître des éléments.

- *show()* : affiche un élément caché
- *hide()* : cache un élément affiché
- *toggle()* : si l'élément est caché, il s'affiche ; s'il est affiché, il se cache
- *fadeIn()* : affiche un élément caché de manière progressive
- *fadeOut()* : cache un élément affiché de manière progressive
- *fadeToggle()* : si l'élément est caché, il s'affiche ; s'il est affiché, il se cache mais de manière progressive
- *slideDown()* : affiche un élément caché avec un effet de glissement
- *slideUp()* : cache un élément affiché avec un effet de glissement
- *slideToggle()* : si l'élément est caché, il s'affiche ; s'il est affiché, il se cache avec un effet de glissement

Toutes ces méthodes peuvent recevoir un paramètre pour contrôler la vitesse d'apparition ou de disparition. Il se précise soit en string : *'slow'*, *'normal'* (par défaut) ou *'fast'* ; soit en nombre de millisecondes. Par exemple :



```
//fait disparaître lentement tous les div  
$('div').fadeOut('slow');
```

Un autre effet permet de faire passer un élément à une certaine opacité : *fadeTo()*. Il prend en paramètre une valeur de vitesse, comme les autres, mais aussi un pourcentage d'opacité.



Effets de base (disparition et apparition)

<http://jsfiddle.net/r3XH9/>

Les animations

Il est possible de faire des animations plus personnalisées grâce à la méthode *animate()* qui permet de jouer avec toutes les propriétés CSS numériques (taille de la police, position d'un élément, opacité, largeur d'une bordure,...)

Cette méthode prend plusieurs paramètres :

- un objet littéral avec la liste des propriétés CSS qu'on désire changer et les valeurs que doivent atteindre l'animation. Par exemple si vous voulez animer un élément en le déplaçant de 650px vers la gauche, en changeant son opacité à 50% et agrandir sa police à 24px :

JS

```
{
  left : '650px',
  opacity : 0.5,
  fontSize : '24px'
}
```

- la durée de l'animation, en millisecondes - facultatif, 400 par défaut
- l'effet de transition (*easing*) – facultatif. De base uniquement les effets 'swing' ou 'linear' sont possibles mais vous pouvez en avoir plus via cette librairie <http://gsgd.co.uk/sandbox/jquery/easing/>
- Une méthode de callback – facultatif. Il est possible de dire à la méthode *animate()* de déclencher une méthode une fois que l'animation est terminée.



Animation tout simple grâce à *animate()*

<http://jsfiddle.net/WsUcH/>



Afficher la table des matières auto-générée lors d'un passage de la souris avec un *slide* horizontal

<http://jsfiddle.net/r8rPS/>



Disparition et apparition avec *animate()* et callback

<http://jsfiddle.net/dTtpT/>



Plus sur les effets visuels : <http://api.jquery.com/category/effects/>

Parcourir le DOM avec jQuery

Bien souvent, après avoir [sélectionné un élément](#) vous allez vouloir trouver d'autres éléments en relation avec celui-ci. Par exemple, dans vous vous retrouvez dans la fonction liée à l'événement *click* sur une image, vous voulez retrouver un *span* enfant pour le faire apparaître ou disparaître ou un *div* parent pour lui appliquer un style. C'est là que les méthodes qui permettent de traverser le DOM entrent en piste. Elles permettent, à partir d'un élément sélectionné, de trouver d'autres éléments, toujours via un sélecteur CSS.

Voici la liste des méthodes jQuery qui permettent de parcourir le DOM :

- *find()* : permet de trouver des éléments particuliers à partir de la sélection en cours
- *children()* : comme *find()* mais se limite aux éléments qui sont des enfants direct de la sélection en cours

- *parent()* : permet de « remonter » et de récupérer le parent direct de la sélection en cours
- *closest()* : permet de trouver l'ancêtre le plus proche qui correspond à un sélecteur
- *siblings()* : permet de récupérer tous les éléments qui sont au même niveau que la sélection en cours
- *next()* : permet de trouver l'élément de même niveau que la sélection en cours et qui le suit directement dans l'HTML
- *prev()* : même chose que *next()* mais pour l'élément précédent



Plus sur le parcours du DOM : <http://api.jquery.com/category/traversing/>

Les formulaires avec jQuery

Les formulaires sont une part importante d'un site web. Et jQuery permet de récupérer (et de modifier) facilement les valeurs d'un formulaire, mais aussi de construire des interactions plus complète et de permettre de valider les données avant l'envoi.

Interagir avec un formulaire

Sélectionner des éléments

Vous pouvez bien sûr attribuer un ID à un élément de formulaire et y accéder comme nous l'avons vu précédemment, mais jQuery offre des sélecteurs spécifiques aux champs de formulaire :

Sélecteur	But
:input	Sélectionner tous les éléments de formulaire
:text	Sélectionner tous les champs texte
:password	Sélectionner tous les champs mot de passe
:radio	Sélectionner tous les boutons radio
:checkbox	Sélectionner toutes les cases à cocher
:submit	Sélectionne tous les boutons submit
:image	Sélectionne tous les boutons image
:reset	Sélectionne tous les boutons reset
:button	Sélectionne tous les champs avec le type « button »
:file	Sélectionne tous les champs d'upload
:hidden	Sélectionner tous les champs cachés

Ces sélecteurs peuvent bien sûr être combinés à d'autres sélecteurs. Ils s'utilisent, par exemple comme ceci (sélectionner tous les champs cachés du formulaire *myForm*) :

JS

```
var hiddenFields = $('#myForm :hidden');
```


En plus, jQuery offre 2 sélecteurs pour permettre de récupérer les valeurs sélectionnées dans des boutons radio/cases à cocher et dans des listes déroulantes :

Sélecteur	But
:checked	Sélectionne les éléments cochés (dans le cas de boutons radio ou cases à cocher)
:selected	Sélectionne les éléments sélectionnés dans une liste déroulante

Exemple d'utilisation :

```
JS var checkedValues = $('input[name="shipping"]:checked').val();  
var selectedState = $('#state :selected').val();
```

Lire et modifier les valeurs

On peut bien sûr accéder aux valeurs des champs d'un formulaire, dans le but de valider les données, ou de faire un calcul, etc... D'un autre côté on peut aussi modifier les valeurs des champs : imaginons un cas où si vous cochez la case « Mon adresse de facturation est la même que celle de livraison », que les données déjà rentrées soit automatiquement dupliquées dans d'autres champs.

Pour lire et modifier des valeurs, il suffit d'utiliser la méthode *.val()*. Sans paramètre, celle-ci renvoie la valeur du champ sur laquelle elle est appelée, et si on lui donne une valeur en paramètre, cela aura pour effet de modifier le contenu du champ.

```
JS var orderField = $("#inputOrder").val();  
//s'il est vide, on rajoute un texte d'introduction  
if(!orderField) {  
    orderField = 'Je voudrais commander:';  
}  
$("#inputOrder").val(orderField);
```

Dans le cas des boutons radio et des cases à cocher, on peut s'y prendre autrement.

Pour lire la valeur, il faut vérifier si le radio/case à cocher est coché ou pas, et pour ce faire on peut utiliser le [filtre is\(\)](#)

```
JS var rememberMe= $("#rememberMeInput").is(':checked') ;
```

Et pour cocher/décocher une case à cocher ou un radio, on peut utiliser la méthode *attr()* sur l'attribut *checked*

JS

```
$("#rememberMeInput").attr('checked', true);
```

Les événements

De manière similaire aux événements déjà vus, comme un clic de souris sur un lien ou une touche du clavier qui est pressée, il existe des événements spécifiques aux formulaires. Ils sont au nombre de 5 : *submit*, *focus*, *blur*, *click* et *change*.

submit

L'événement *submit* est déclenché lors de l'envoi d'un formulaire, que ça soit en cliquant sur le bouton *submit* ou en appuyant sur *Enter* quand vous tapez dans un champ texte du formulaire.

C'est lors de cet événement que vous pouvez vérifier si le formulaire est valide ou non, en lui associant du code via la méthode *.submit()*. Dans le cas où le formulaire n'est pas valide, il est possible d'annuler son envoi via un *return false*;

JS

```
$("#registration").submit(function() {  
    console.log($("#name").val());  
    //empêcher l'envoi du formulaire  
    return false;  
});
```

focus

Quand vous cliquez dans un champ d'un formulaire, ce champ reçoit ce qu'on appelle le *focus*. Il est possible d'effectuer une action lors du *focus*, en sélectionnant le champ et en lui appliquant la méthode *.focus()*.

JS

```
$("#name").focus(function() {  
    console.log($(this).val());  
});
```

Pour rappel, un événement peut être appelé manuellement si on ne lui fournit pas de paramètre. Dans le cas de *.focus()* le focus sera mis sur l'élément sélectionné. Ce qui peut être pratique pour déjà positionner le curseur de texte dans le premier champ d'un formulaire par exemple.

blur

C'est un peu le contraire de *focus*, *blur* est déclenché lorsque vous quittez un champ d'un formulaire. Cela peut être utile pour vérifier la validité des champs au fur et à mesure du remplissage du

formulaire et non tout à la fin, lors du *submit*. Pour effectuer une action lors du *blur*, il faut sélectionner le champ et lui appliquer la méthode *.blur()*

JS

```
$("#name").blur(function() {  
    console.log($(this).val());  
});
```

click

Il est évidemment possible d'intercepter un clic sur un élément d'un formulaire, par exemple pour effectuer une action lorsqu'une case à cocher est cochée. Cela fonctionne de manière similaire au *click* vu précédemment

JS

```
$("#rules").click(function() {  
    console.log($(this).is(":checked"));  
});
```

change

L'événement *change* peut être utilisé dans le cas où vous voulez appliquer une action lorsque la valeur d'une liste déroulante change .

JS

```
$("#level").change(function() {  
    console.log($(this).val());  
});
```

Activer/désactiver des champs

En HTML, l'attribut *disabled* permet de désactiver un champ de formulaire ou un bouton, dans le cas où vous ne voulez pas que l'utilisateur rentre du contenu dans le champ ou clique sur le bouton (pour éviter un double envoi de formulaire par exemple).

Via jQuery il est bien sûr possible de contrôler l'activation d'un champ ou bouton en lui appliquant/retirant cet attribut de la manière suivante

JS

```
//activer  
$("#sendFormButton").attr('disabled',false);  
//désactiver  
$("#sendFormButton").attr('disabled',true);
```

Introduction

La validation côté client (en JavaScript donc) est une étape essentielle lors de la mise en place d'un formulaire. Elle permet de s'assurer que les données rentrées par l'utilisateur sont bien présentes (par exemple une adresse de livraison dans le cas d'une commande) et qu'elles correspondent au format attendu (un format d'e-mail pour une adresse mail, un texte sous forme de date pour une date, un chiffre pour une quantité, etc...)

En JavaScript pur, cela peut vite devenir long compliqué de valider un formulaire, il faut notamment connaître les [expressions régulières](#). Heureusement, en jQuery, ces vérifications sont très simples à mettre en place grâce au plugin [jQuery Validate](#).

Il y a 2 manières de rajouter de la validation sur un formulaire avec jQuery Validate ; soit une méthode « simple » qui consiste à attribuer des classes et attributs spécifiques dans l'HTML, sur les éléments à valider (indiquant s'ils sont requis, leur type attendu, le message d'erreur etc...) ; soit la manière « avancée » qui consiste à configurer la validation en JavaScript. Cette approche est à préférer car elle permet d'aller beaucoup plus loin.

Configurer la validation

Pour configurer la validation, il faut d'abord donner une valeur à l'attribut « name » de vos champs. Ensuite, il faut appeler la méthode `validate()` sur le formulaire et lui passer un objet littéral qui va définir des **règles** et, pour ces règles des **messages d'erreur**.

La syntaxe est :

```
$('#myform').validate({  
    rules : {  
        nomDuChamp : 'typeDeValidation'  
    },  
    messages : {  
        nomDuChamp : {  
            typeDeValidation : 'Message d'erreur'  
        }  
    }  
});
```

Par exemple

JS

```
$("#orderForm").validate({
    rules: {
        inputMail: {
            required: true,
            email: true
        }
    },
    messages: {
        inputMail: {
            required : "L'e-mail est requis",
            email: "L'e-mail n'a pas un format correct"
        },
    }
});
```

Voici les types de validation les plus utilisés qu'offre jQuery Validate :

Validation	But
required	Le champ est requis
date	La valeur du champ doit être une date
email	La valeur du champ doit être un e-mail
url	La valeur du champ doit être un URL
number	La valeur du champ doit être un nombre décimal
digits	La valeur du champ doit être un nombre entier
equalTo	La valeur du champ doit être égale à la valeur d'un autre champ
minlength	La longueur de la valeur du champ doit être supérieure à une certaine valeur
maxlength	La longueur de la valeur du champ doit être inférieure à une certaine valeur
min	La valeur du champ doit être supérieure à une certaine valeur
max	La valeur du champ doit être inférieure à une certaine valeur

Il est également possible de définir ses propres méthodes de validation via la méthode [*addMethod\(\)*](#) qui demande : un nom, une fonction de validation et un message d'erreur par défaut.

Si le formulaire n'est pas valide, le formulaire ne sera pas envoyé vers le serveur.



A noter que si le formulaire envoie ses données via une requête [*AJAX*](#), il faut définir l'action à faire dans le cas où le formulaire est valide dans l'option ***submitHandler*** de la méthode *validate()*.

Styler les messages d'erreur

Quand le plug-in vérifie un formulaire et trouve un champ invalide, il fait 2 choses : premièrement, il ajoute une classe « error » au champ invalide ; deuxièmement il rajoute un label à la suite du champ qui contient le message d'erreur défini.

Par exemple, si vous avez un champ comme ceci dans votre HTML :

```
</> <input class="left" type="text" name="inputEmail" >
```

S'il n'est pas valide, le nouvel HTML sera :

```
</> <input class="left error" type="text" name="inputEmail" >  
<label class="error" for="inputEmail " generated="true">This field  
is required.</label>
```

Il suffit dès lors d'ajouter dans votre CSS de quoi styler la classe « error ». Mais si vous voulez faire plus lors de l'affichage des messages d'erreur (comme styler un autre élément), vous pouvez utiliser d'autres options de `.validate()` comme `errorPlacement`, `highlight`, etc...



Plus sur les formulaires : <http://api.jquery.com/category/forms/>
Plus sur jQuery Validate : <http://docs.jquery.com/Plugins/Validation>
Options de la méthode `.validate()` :
<http://docs.jquery.com/Plugins/Validation/validate#toptions>

Requêtes AJAX avec jQuery

Introduction

Les applications web ressemblent de plus en plus à des applications « desktop », le temps où votre page web se rafraichissait complètement lorsque vous effectuiez une action est presque révolu grâce à AJAX.

AJAX (acronyme pour *Asynchronous JavaScript And XML*) permet à une page web de contacter un serveur web et de se rafraichir elle-même avec les données renvoyées par le serveur. Cela ouvre les portes à des centaines de possibilités, comme :

- Afficher du nouveau contenu HTML sans devoir recharger la page
- Envoyer un formulaire et afficher un résultat instantanément
- Se logger sans quitter la page
- Parcourir des longues listes
- Etc...

Des sites comme Google Maps, Google Docs, Facebook, Twitter ou autres utilisent AJAX abondamment.

Fonctionnement

Comme vu dans le [Le B-A-BA d'Internet](#), lors d'un appel de page « classique », le navigateur contacte le serveur web, qui renvoie une nouvelle page au navigateur.

Lors d'un appel AJAX, le navigateur ne demande au serveur que de nouvelles informations. Le serveur renvoie les données demandées, et le contenu et l'apparence de la page sont mis à jour avec l'aide de JavaScript.

Il y a donc 3 acteurs lors d'un appel AJAX :

- **Le navigateur web** : via l'objet *XMLHttpRequest* de votre navigateur, JavaScript peut parler à un serveur web et recevoir de l'information en retour
- **JavaScript** : c'est lui qui fait le gros du boulot. Il envoie une requête au serveur, attend pour une réponse, traite la réponse et met à jour la page en ajoutant du contenu (en manipulant le DOM) ou en adaptant l'affichage de la page.
L'information envoyée peut être aussi bien le détail d'un formulaire, qu'une simple donnée (par ex. un *rating* sur un article). L'information reçue en retour peut aussi bien être une simple phrase (« Merci de votre vote ») qu'une liste de nouveaux enregistrements (dans le cas d'un e-commerce par exemple)
- **Le serveur web** : le serveur reçoit donc des requêtes du navigateur et lui renvoie des informations. Les informations peuvent être aussi bien du texte, qu'un bout de code HTML, de XML ou de données JSON.
Le serveur quant à lui peut aussi bien être un serveur IIS qui fait tourner de l'ASP.NET, qu'un serveur Apache avec PHP.

Il est tout à fait possible d'effectuer des appels AJAX en JavaScript pur via l'objet *XMLHttpRequest*, mais il est bien plus aisé d'utiliser jQuery.

Méthodes jQuery

jQuery offre plusieurs méthodes pour effectuer des appels asynchrones en AJAX, les voici :

`.load()`

`.load()` permet de charger le contenu d'une autre page web et de placer le contenu dans le sélecteur sur lequel la méthode est appelée.

Il est possible de lui préciser un sélecteur de la page cible afin de ne sélectionner qu'une partie de la page à charger.

Cette méthode ne requiert pas d'intervention de la part d'un serveur web.

JS

```
//charge le contenu de l'élément #newsItem de la page today.html  
dans l'élément #headlines de la page courante  
$("#headlines").load('today.html #newsItem');
```

`$.get()` et `$.post()`

`$.get()` et `$.post()` : ces méthodes (autonomes) permettent de charger des données d'un serveur web via une requête GET ou POST.

La syntaxe est celle-ci (c'est la même chose pour `$.post`) :

`$.get(URL, [données sous forme de query string], [callback])` ;

JS

```
//contacte le serveur à l'adresse rate.php, envoie les données (un  
rating de 5) et on traite la réponse  
$.get("rate.php", "rating=5", function(data) {  
    console.log("rating moyen: " +data)});
```

Les données renvoyées peuvent être de différents formats: texte, HTML, XML ou JSON

`$.getJSON()`

`$.getJSON()` permet de charger des données d'un serveur web au format JSON via une requête GET. Cette méthode est identique à `$.get` mais le format renvoyé est d'office du JSON

`$.ajax()`

Cette méthode est la plus complète et flexible de toutes. `$.get`, `$.post` et `$.getJSON` sont des « raccourcis » de `$.ajax()`. C'est donc également celle qui demande le plus de configuration.

Pour la configurer, il suffit de lui passer un objet littéral. Les paramètres de cet objet les plus couramment utilisés sont :

- url : l'adresse serveur vers laquelle faire la requête
- type : le type de requête, « GET » ou « POST »
- data : les données à envoyer au serveur
- dataType : le format de données renvoyées par le serveur (« xml », « html », « json », « text »)
- contentType : le content type des données envoyées
- async : true ou false, défini si la requête doit être synchrone ou asynchrone



Plus sur AJAX avec jQuery : <http://api.jquery.com/category/ajax/>

Récupération des résultats et des erreurs

La méthode \$.ajax() offrait jadis 2 *callbacks*, *success* et *error*, auxquels du code pouvait être associé pour effectuer une action en cas de requête fructueuse ou non. Ce mécanisme entraînait souvent la création de « code pyramidal » avec une multitude de *callbacks* imbriqués qui rendait le code illisible et certaines actions très compliquées.

\$.ajax() et les autres méthodes à l'exception de .load() utilisent ce qu'on appelle les Promise pour gérer le retour d'une requête AJAX. 3 callbacks peuvent être chaînés :

- done() : dans le cas où la requête est fructueuse
- fail() : dans le cas où la requête a renvoyé une erreur
- always() : qui va s'exécuter dans le tous les cas

JS

```
$.get( "example.php", function() {
    alert( "success" );
})
.done(function() {
    alert( "second success" );
})
.fail(function() {
    alert( "error" );
})
.always(function() {
    alert( "finished" );
});
```



Plus sur les Promise en JavaScript :

[What's so great about JavaScript Promises?](#)

[A dumb easy model for promises](#)

[Understanding JQuery.Deferred and Promise](#)

Les plug-ins

Quand on parle de jQuery on entend souvent le terme *plug-in*. Un plug-in c'est du code écrit en JavaScript pour but bien précis : afficher un calendrier, fournir des effets visuels, permettre une gestion des cookies plus aisée, afficher des lightbox, afficher des sliders, afficher des notifications, etc... Beaucoup de plug-ins tirent parti de la puissance de jQuery et se basent donc sur jQuery pour fonctionner.

Un plug-in consiste en 1 ou plusieurs fichiers .js à inclure dans vos pages.

Si le plug-in est un élément visuel, il est accompagné d'un fichier .css qui va « styler » le contrôle utilisateur et vous devrez respecter un certain markup HTML pour le faire fonctionner correctement.

jQuery UI

Il ne faut pas chercher bien loin pour trouver des plug-ins jQuery, en effet il existe un « pack » de contrôles utilisateurs disponibles sur le site de jQuery, [jQuery UI](#). Ces contrôles permettent d'afficher un simple choix de date à fournir quantité d'autres contrôles, effets et interactions. Par exemple :

- un accordéon
- un système *d'autocomplete*
- des boutons déjà stylés
- un choix de date
- une barre de progression
- des tabs
- une gestion du drap&drop
- etc...

Autres plug-ins

Il existe une quantité d'autres plug-ins gratuits disponibles, voici une liste non-exhaustive :

- Effets pour les animations jQuery : <http://gsgd.co.uk/sandbox/jquery/easing/>
- Amélioration et remplacement de <select> : <http://ivaynberg.github.com/select2/>
- Check et suggestion de domaine mail : <https://github.com/Kicksend/mailcheck>
- Adapter la taille de la police en fonction de la taille de la fenêtre : <http://jbrewer.github.com/Responsive-Measure/>
- Formater, transformer et manipuler des dates : <http://momentjs.com/>
- Gestion des cookies : <https://github.com/ScottHamper/Cookies>
- Savoir si l'utilisateur est déjà venu sur le site auparavant : <http://www.ravelrumba.com/blog/firstimpression-js-library-detecting-new-visitors/>
- Utiliser Google Maps de manière simple : <http://hpneo.github.com/gmaps/> - <http://gmap3.net>
- Générer une table des matières automatiquement : <http://projects.jga.me/toc/>
- Organiser le contenu en 'tuiles' : <http://masonry.desandro.com/>

- Faciliter l'exécution d'une fonction quand on scrolle un élément : <http://imakewebthings.com/jquery-waypoints/>
- Parallax, navigation fixe, autoscroll,... : <https://github.com/Prinzhorn/skrollr>
- Pour réaliser des potentiomètres : <http://anthonyterrien.com/knob/>
- Pour réaliser des notifications : <http://needim.github.com/noty/> - <http://richardhsu.github.com/jquery.ambiance/>
- Afficher des infos complémentaires qui apparaissent sur le côté : <http://srobbin.com/jquery-plugins/pageslide/>
- Tooltips : <http://craigsworks.com/projects/qtip2/>
- Rating : <http://wbotelhos.com/raty/>
- Sliders : <http://www.jqueryslidershock.com/> - <http://dev7studios.com/nivo-slider/>
- Carousel : <http://demo.dev7studios.com/caroufredsel/> - <http://als.musings.it/>
- Gestion du scroll : <https://github.com/sxalexander/jquery-scrollspy>
- Gestion des images de background : <http://srobbin.com/jquery-plugins/backstretch/> - <http://static.elliottjaystocks.com/responsive-background-images/examples/solution.html>
- Lightbox : <http://www.jacklmoore.com/colorbox>
- Transformations et transitions CSS : <http://ricostacruz.com/jquery.transit/>

Liens et ressources à propos de jQuery

- jQuery Cheatsheet : <http://oscarotero.com/jquery/>
- Cours en ligne: <http://try.jquery.com/>
- jQuery API documentation : <http://api.jquery.com/>
- jQuery Fundamentals : <http://jqfundamentals.com/>
- Useful jQuery Function Demos For Your Projects: <http://coding.smashingmagazine.com/2012/05/31/50-jquery-function-demos-for-aspiring-web-developers/>
- Tous les liens taggués "jQuery" sur mon Delicious: <https://delicious.com/panzerkunst/search/jquery>

Plus loin avec JavaScript

Le debug et la gestion des erreurs

Le debug

Il est très rare voir presque impossible de programmer sans faire d'erreurs... les fameux bugs qui plantent notre application sont si vite arrivés.

Mais quel type d'erreurs peut-on bien faire ? Cela peut être une erreur de frappe : on a oublié un caractère, on a mal orthographié le nom d'une fonction et du coup JavaScript ne la trouve pas, ou alors on a simplement fait une erreur dans notre logique en faisant une division au lieu d'une multiplication.

Avec les vieux navigateurs, les messages d'erreur JavaScript n'étaient pas des plus parlants, et il fallait user et abuser de la méthode *alert()*, qu'on disséminait un peu partout dans le code en espérant trouver la ligne qui pose problème ! Encore faut-il, une fois la ligne coupable trouvée, savoir le corriger ☺

Cette période de galère est révolue, tous les navigateurs possèdent une console d'erreurs et puis c'est sans compter sur notre fidèle Firebug ! Il affiche des messages d'erreurs assez clairs en indiquant généralement l'endroit exact où l'erreur s'est produite.

Mais on peut faire plus : pour débbugger du code dans Firebug, allez dans l'onglet « Script » et activez-le. Vous pouvez choisir le script à analyser et y insérer des **breakpoints** ou points d'arrêts. Ils servent à faire une pause sur une ligne d'un code JavaScript et à afficher l'état complet des variables, méthodes, objets, etc. La quantité d'informations disponible est tout simplement immense, vous pouvez les retrouver dans le cadre de droite de Firebug dès qu'un point d'arrêt est atteint. Vous pouvez aussi passer d'un point d'arrêt à un autre pour voir « au ralenti » l'exécution de votre script.

La gestion des erreurs

Disons que vous avez un script où vous vous dites « à cet endroit-là, y'a des chances que ça plante » car vous faites un calcul complexe, ou faites une opération à partir de données rentrées par l'utilisateur. Au lieu de croiser les doigts pour que ça n'arrive pas, vous pouvez mettre en place une gestion d'erreurs.

La gestion d'erreurs se fait à l'aide du bloc d'instruction *try catch*. Dans le *try* on va essayer le code, et si une erreur (ou *exception*) survient, on passe dans le bloc *catch*, où l'on va exécuter une action en cas d'erreur (affichage d'un message, log de l'erreur, ou autre...)

Ce bloc d'instruction peut être suivi de l'instruction *finally*, où l'on va exécuter du code quoi qu'il se passe, qu'il marche, ou qu'il plante.

Si vous voulez vous-même dans votre code **renvoyer** une erreur (si un paramètre est incorrect par exemple), vous pouvez le faire à l'aide de l'instruction *throw*.

Le testing

Une manière d'éviter de devoir passer par un debugger serait de pouvoir s'assurer du bon fonctionnement du code rédigé avant toute chose. C'est la philosophie des tests unitaires : s'assurer que les méthodes écrites renvoient le résultat attendu, de manière unitaire (granulaire).

Et quand les tests sont rédigés avant même de coder, on appelle ça le TDD : *test-driven development*.



Plus sur les outils de tests unitaires pour du TDD en JavaScript :

<http://stackoverflow.com/questions/300855/javascript-unit-test-tools-for-tdd>

Introduction aux tests unitaires en JavaScript :

<http://coding.smashingmagazine.com/2012/06/27/introduction-to-javascript-unit-testing/>

Orienté objet

Nous avons abordé les objets plusieurs fois dans ce cours via les types objets, les objets du DOM, les objets littéraux utilisés comme paramètre dans des méthodes jQuery. Mais nous n'avons pas abordé comment créer nos objets nous-même ou ce à quoi ils pourraient servir.

Les objets sont un moyen d'organiser l'information de manière logique : les propriétés et méthodes d'un sujet sont regroupées, « encapsulées » dans un objet. Il y a 2 manières de définir des objets : soit de manière littérale - dans ce cas, l'objet n'est utilisable qu'une seule fois, soit via un constructeur – dans le cas plusieurs instances de cet objet peuvent être créées.

Objet littéral

Déclarer et utiliser un objet littéral

Les objets littéraux permettent de créer une liste de propriétés et valeurs en JavaScript. Ils sont beaucoup utilisés par exemple pour :

- paramétrer les plug-ins jQuery ou configurer les animations jQuery
- envoyer des données en AJAX
- structure du code

La syntaxe pour créer un objet littéral est celle-ci :

```
{  
  'propriété1' : 'valeur1',  
  'propriété2' : 'valeur2'  
}
```

La propriété et la valeur sont séparés par : et chaque paire propriété/valeur doit être séparée par une virgule mais il ne faut pas en mettre après la dernière paire propriété/valeur ! Les apostrophes autour des propriétés sont facultatives sauf si les noms des propriétés contiennent des caractères spéciaux.

On peut créer un objet littéral et l'associer à une variable

JS

```
var data = {  
  firstName : 'Michel',  
  lastName : 'Dupont',  
  city : 'Bruxelles'  
};
```

Et après on peut bien sûr récupérer chaque propriété individuellement :

JS

```
console.log(data.firstName);
```

Voici un exemple d'utilisation d'objet littéral passé en paramètre de la méthode *animate()*

JS

```
$('#menu').animate(  
  {  
    left : '0',  
    fontSize : '15'  
  }  
);
```

Les objets littéraux peuvent être plus complexes : on peut des propriétés plus complexes comme des tableau ou d'autres objets littéraux mais également ajouter des méthodes.

JS

```
var person = {  
  firstName: "Georges",  
  lastName: "Abitbol",  
  age: 57,  
  sentences: ["J'ai la classe", "Adieu monde de merde", "Allons  
dans une bonne auberge"],  
  saySomething: function(){  
    alert(this.sentences[1]);  
  }  
};  
  
person.saySomething();
```

Ici l'objet *person* possède plusieurs propriétés mais aussi une méthode *saySomething()*. Remarquez comment *this* est utilisé pour accéder aux propriétés de l'objet, en effet *this* représente le contexte d'invocation de la méthode *saySomething()*, qui est l'objet dans lequel elle est définie.

Autre utilisation d'un objet littéral : le tableau associatif

Sous le capot, un objet littéral est en fait un [Array](#) (ou tableau) qu'on appelle tableau **associatif**.

Pour rappel, un Array est une variable qui contient plusieurs valeurs, appelées *items*. Chaque item est accessible au moyen d'un indice (*index*) numérique. Dans un tableau associatif, on accède à un

item non pas par un indice numérique mais pas une valeur textuelle appelée clé (ou *key*).

Les **propriétés** d'un objet littéral peuvent être donc vues comme des **clés** qui permettent d'accéder aux **valeurs** du tableau associatif. On pourrait donc écrire :

JS

```
var data = {
  firstName : 'Michel',
  lastName : 'Dupont',
  city : 'Bruxelles'
};

data['firstName'] = 'Jean';
```

Pour rajouter des éléments à ce tableau, il suffit simplement de spécifier une nouvelle propriété/clé en lui donnant une valeur, par exemple :

JS

```
data.country = 'Belgique';
```

Ou sous cette forme

JS

```
data['country'] = 'Belgique';
```

La boucle for in

Il existe une variante de la [boucle for](#) qui s'appelle *for in*. Cette boucle permet de boucler sur un tableau associatif (et donc par définition, sur un objet littéral !). Le fonctionnement est quasiment le même que pour un *Array*, excepté qu'ici il suffit de fournir une « variable clé » qui reçoit un identifiant (au lieu d'un index) et de spécifier l'objet à parcourir :

JS

```
for(var key in data) {
  console.log(data[key]);
}
```

Quel est l'intérêt ? Les objets littéraux se voient souvent attribuer des propriétés ou méthodes supplémentaires par certains navigateurs ou certains scripts tiers utilisés dans la page, ce qui fait que la boucle *for in* va permettre de tous les énumérer.

Constructeur

Si on veut représenter plusieurs fois le même objet, qui a des méthodes similaires mais avec des valeurs différentes pour ses propriétés, l'objet littéral n'est pas l'idéal car il faudra sans cesse copier-coller la structure pour chaque instance... dans ce cas, on peut passer par un constructeur pour créer un « canevas » général et instancier plusieurs entités à partir de ce canevas.

Ce système nous permet aussi de prévoir des valeurs par défaut pour les propriétés.

JS

```
function Person(fname, lname, age){
    this.firstName = fname;
    this.lastName = lname;
    this.age = age;
    this.sentences = ["Hello world"];
    this.saySomething = function(sentence){
        if(!sentence){
            sentence = this.sentences[0];
        }
        alert(sentence);
    }
}

var georges = new Person("Georges", "Abitbol", 57);
georges.saySomething();
var beavis = new Person("Beavis", "", 15);
beavis.sentences.push("That sucks");
beavis.saySomething(beavis.sentences[1]);
```

Structurer son code grâce aux objets

Un des atouts majeurs de l'orienté objet, c'est de pouvoir nous aider à mieux structurer notre code, pour le rendre plus flexible, réutilisable, pouvoir isoler des variables et des méthodes et les rendre « privées », c'est-à-dire les rendre invisible par le code externe à l'objet.

Pour illustrer cette structuration, voici 1 seul et même exercice, codé de manière « classique » : tout dans l'espace global, aucune structuration ; et l'autre codé de manière « objet »



Contrôle « captcha » numérique, manière « classique » : <http://jsfiddle.net/ZP3fR/>
Contrôle « captcha » numérique, manière « objet » : <http://jsfiddle.net/NH7uA/>




Plus sur la programmation orienté-objet en JavaScript :
<http://javascriptissexy.com/javascript-objects-in-detail/>
<http://javascriptissexy.com/oop-in-javascript-what-you-need-to-know/>
<http://eloquentjavascript.net/chapter8.html>

Quelques trucs en plus


Isoler son code grâce au *module pattern*

On y réfléchit pas trop, mais dès qu'on crée une variable globale ou une fonction globale, on fait quelque chose de très dangereux et d'assez « sale » car on induit des risques de conflits avec d'autres variables ou fonctions. Qui plus est, la majorité de vos codes sont autonomes ou *self-contained*, bien souvent propres à une seule page.

Pour isoler votre code du reste et éviter de « polluer le global » vous pouvez utiliser ce qu'on appelle le module pattern, qui est en fait une fonction immédiatement exécutée (*Immediately-Invoked Function Expression*, ou IIFE)

```
 (function(){  
    function doStuff(){  
        console.log("inside the module pattern");  
    }  
  
    doStuff();  
  
})();  
  
doStuff(); //ceci va planter, doStuff n'est pas globale
```

Ici nous avons donc une fonction anonyme, appelée immédiatement. Tout ce qui se trouve dans cette fonction est donc contenu dans un *scope*, une « bulle », isolé du reste du code.


 Plus sur le module pattern et d'autres bonnes pratiques : <http://www.js-attitude.fr/2013/01/21/dix-bonnes-pratiques-javascript/#5-isoler%20son%20code%20avec%20le%20module%20pattern>

Optimiser grâce à la minification

Quand vous vous lancerez à la recherche de bibliothèques JavaScript, vous allez souvent voir une version *.js* et une version *.min.js* du script. Que veut dire ce « min » ? Il signifie que c'est une version du script qui a été minifiée : cela consiste à retirer tous les espaces, les commentaires et les retours à la ligne inutiles du code.

Les intérêts de la minification sont principalement de réduire la taille du fichier et donc d'accélérer le temps de chargement et accessoirement de rendre le code moins lisible pour le protéger contre le vol. Un code minifié est aussi plus difficile à déboguer.

Préférez donc un code « non-minifié » pour développer et puis au-moment de mettre votre site en ligne, changez les références dans vos pages du script « non-minifié » vers le script minifié et mettez la version minifiée en ligne.

 Packer vous permet de minifier vos JavaScripts
<http://dean.edwards.name/packer/>

JSBeautifier vous permet de « déminifier » un script minifié
<http://jsbeautifier.org/>

La gestion des cookies

Les cookies ont été inventés par Netscape afin de donner une "mémoire" aux serveurs et navigateurs Web. Le protocole HTTP, qui gère le transfert des pages Web vers le navigateur ainsi que les demandes de pages du navigateur vers le serveur, est dit *stateless* (sans état) : cela signifie qu'une fois la page envoyée vers le navigateur, il n'a aucun moyen d'en garder une trace. Vous

pourrez donc venir deux, trois, cent fois sur la page, le serveur considérera toujours qu'il s'agit de votre première visite.

Cela peut être gênant à plusieurs titres : le serveur ne peut pas se souvenir si vous êtes authentifié à une page protégée, n'est pas capable de conserver vos préférences utilisateur, etc. En résumé, il ne peut se souvenir de rien ! De plus, lorsque la personnalisation a été créée, cela est vite devenu un problème majeur.

Les cookies ont été inventés pour remédier à ces problèmes. Il existe d'autres solutions pour les contourner, mais les cookies sont très simples à maintenir et très souples d'emploi.

Ils sont stockés sous forme de fichiers textes dans le répertoire de fichiers temporaires de votre navigateur et permettent, par exemple, de retenir un choix de langue effectué par l'utilisateur sur votre site afin de ne pas devoir lui redemander à sa prochaine visite.

jsFiddle lui utilise par exemple les cookies pour se « souvenir » des positions et des grandeurs des différentes fenêtres.

On peut créer et lire des cookies avec du code côté serveur mais c'est également possible de le faire en JavaScript !

Si vous voulez stocker des informations dans un cookie, vous devrez spécifier 3 choses :

- Une clé ; un identifiant de l'information que vous souhaitez stocker, par exemple « userLanguage »
- Une valeur ; par exemple « FR » pour Français
- Une date d'expiration ; le moment auquel vous souhaitez que le cookie expire, c'est-à-dire soit détruit



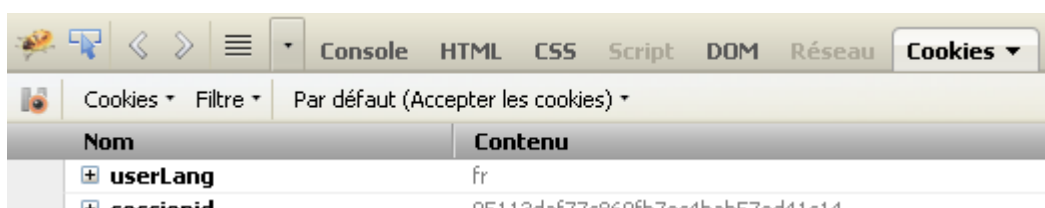
Méthodes de création et d'accès aux cookies + utilisation d'un cookie dans le cas d'un choix de langue FR/NL

<http://jsfiddle.net/BFmdg/>



Une librairie jQuery pour gérer les cookies : <https://github.com/ScottHamper/Cookies>

Pour vérifier les cookies, leur valeur et date d'expiration vous pouvez utiliser l'onglet « Cookies » de Firebug



Cookies	
Filtre ▼ Par défaut (Accepter les cookies) ▼	
Nom	Contenu
+ userLang	fr
+ sessionid	05112daf77c860fb7ad4hsh57ad41c14



A savoir, pour des raisons de sécurité, vous ne pouvez pas modifier la valeur des cookies marqués comme « Http Only » en JavaScript ! Vous pouvez voir ces types de cookies dans Firebug également. Ce sont généralement des cookies qui contiennent des informations « sensibles » comme un identifiant de session

Nom	Contenu	Hôte	Taille	Chemin	Expire	HttpOnly
sessionid	95113daf77c869fb7ec4bab57ed41c14	jsfiddle.net	41 B	/	jeudi 7 mars 2013 10:14:05	HttpOnly

Le futur... et le présent de JavaScript

API HTML5

HTML5 offre de nouvelles API, ce qui étend les possibilités du JavaScript côté client, voici une liste de nouveautés désormais possibles grâce à HTML5 :

- Les attributs data-* : pour stocker des informations dans l'HTML qui ne pourraient pas être sauveés dans des attributs existants
- File API : pour permettre de lire des fichiers (via un champ « file ») sans devoir passer par le serveur, JavaScript peut récupérer le fichier et ses informations
- Géolocalisation : JavaScript peut vous geolocaliser à partir de votre navigateur
- Storage : pour stocker des informations en local, sans devoir passer par un serveur ou par des cookies. Peut s'utiliser de manière temporaire (sessionStorage) ou permanente (localStorage)
- Offline : pour savoir quand la connexion est perdue ou rétablie
- Canvas : pour manipuler des graphiques sur un <canvas>
- Web audio API : pour manipuler les fichiers audio
- Web sockets : pour recevoir des « push » depuis le serveur – le serveur peut envoyer des événements aux clients
- ...

Single page applications et frameworks MVC

JavaScript est désormais considéré comme un langage à part entière, aussi puissant que d'autres langages et vous vous êtes sans doute déjà rendu compte qu'il permet de faire beaucoup plus que de simples effets visuels.

Un nouveau paradigme de développement applicatif est en train de s'établir comme un modèle à suivre : les SPA ou **single-page applications**. Le principe est simple : donner la sensation à l'utilisateur qu'il se trouve plus que sur un site web, mais qu'il se trouve face à une application aussi riche qu'une application de bureau ou d'une application Flash/Flex d'il y a quelques années.

Le principe est simple : pas besoin de naviguer vers d'autres pages pour travailler, tout se trouve dans une seule et même page qui charge et envoie des données via AJAX, rafraichit l'interface en modifiant le DOM via des bibliothèques comme jQuery, change les URLs et gère la navigation « back », etc...

Pour permettre de structurer ce type d'applications en JavaScript, un modèle d'architecture d'applications déjà répandu côté serveur s'est imposé : le **modèle MVC** pour *Model View Controller*.

En quelques mots, ce modèle permet de structurer le code en séparant la logique comme ceci :

- Les « modèles » représentent tout simplement les données. Les données d'un même type sont rassemblées dans ce que l'on appelle des « collections ». Par exemple, dans une application qui gère une liste de « to-do », le modèle est une tâche, et une collection une liste de tâches.
- Les « vues » sont les représentations graphiques des modèles. Pour reprendre l'exemple précédent, il existerait une vue pour « représenter » un to-do : un champ texte avec une case à cocher par exemple
- Les « contrôleurs » sont les chefs d'orchestre, ce sont eux qui vont créer les vues (et les rajouter à la page) et y injecter les données.

Il existe plusieurs *frameworks* qui permettent de travailler avec le modèle MVC en JavaScript, les plus connus étant **Backbone.js**, **Ember.js** et **AngularJS**.



Version de la carte de commande « classique »

<http://jsfiddle.net/HS6jw/>

Même exercice, en utilisant Backbone.js (plus long... mais plus propre !)

<http://jsfiddle.net/8DX7W/>



TodoMVC est un site qui vous permet de comparer les différents frameworks en se basant sur l'exemple d'une application de to-dos : <http://todomvc.com/>

Templating

Travailler avec un *framework* MVC induit inévitablement l'utilisation de *templates*. Les *templates* permettent de séparer le *markup* de la logique dans les vues et donc de maximiser la réutilisabilité de code et la maintenabilité. Une définition d'un *template* pourrait être ceci : « un document qui contient des paramètres, identifiés par une syntaxe spéciale, qui sont remplacés par les vrais arguments par le système de template » ([source](#))

Exemple de template :



```
<h1>{{title}}</h1>
<ul>
  {{#names}}
    <li>{{name}}</li>
  {{/names}}
</ul>
```

Les données à partir desquelles le template va travailler :



```
var data = {
  "title": "Story",
  "names": [
    {"name": "Tarzan"},
    {"name": "Jane"}
  ]
}
```

```
}
```

Et en combinant les 2, voici le rendu final :

```
</> <h1>Story</h1>
      <ul>
        <li>Tarzan</li>
        <li>Jane</li>
      </ul>
```

Il existe plusieurs « moteurs de templating », notamment **Mustache.js** et celui compris dans **Underscore.js** (bibliothèque d'utilitaires utilisée par Backbone.js). A noter que vous pouvez utiliser ces outils en dehors d'un framework MVC.



Plus sur les templates: <http://coding.smashingmagazine.com/2012/12/05/client-side-templating/>

End-to-end JavaScript

JavaScript a bien changé, c'est maintenant un langage robuste, fiable et fonctionnel. Et il est à présent un vif concurrent de langages serveur comme PHP, C# ou Java vu qu'il est désormais possible de créer des applications serveur en JavaScript, grâce à **Node.js**.

Node.js, basé sur l'interpréteur V8, permet de créer rapidement des applications serveurs très légères et très performantes grâce à 2 caractéristiques de JavaScript : l'*event loop* et les *callbacks*. L'*event loop* tourne dans 1 seul thread : dès qu'une requête arrive, elle est prise en compte. Si cette requête doit faire appel à une base de données ou au système de fichier, le thread ne va pas se « bloquer » en attendant la réponse de ce dernier, il va continuer à gérer les autres requêtes et la réponse arrivera par l'intermédiaire d'un callback.

Bref, vous pouvez appliquer toutes vos connaissances acquises côté client pour coder côté serveur. Et n'utiliser qu'un seul langage de programmation, c'est le concept de « full-stack JavaScript » ou de « end-to-end JavaScript ». Qui va encore plus loin que cela vu qu'il est également possible d'utiliser les objets JavaScript (du JSON) pour stocker les données dans une base de données non-relationnelles (ou *NoSQL*) comme par exemple **MongoDB**. JavaScript du début à la fin ! C'est pas beau ça ? ☺



Introduction to full-stack JavaScript :

<http://coding.smashingmagazine.com/2013/11/21/introduction-to-full-stack-javascript/>
Comment marche Node.js: <http://orange-coding.net/2013/06/29/xfiles-part-i-learning-how-to-walk/>

That's all folks !

Cours construit et rédigé par Nicolas Bauwens, conseiller pédagogique et formateur Bruxelles Formation Management & MultimediaTIC – 2013-2014.

Prezi : http://prezi.com/llqt9fa_yloe/module-court-javascript-jquery/

Sources

- [JavaScript & jQuery : the missing manual](#) par David Sawyer McFarland
- [JavaScript](#) sur Codecademy.com
- [Dynamisez vos sites web avec JavaScript !](#) sur le site du Zér0
- [Secrets of the JavaScript ninja](#) par John Resig et Bear Bibeault
- [JavaScript: The Good Parts](#) par Douglas Crockford
- [JavaScript: The Definitive Guide](#) par David Flanagan