

Secteur Tertiaire Informatique
Filière « Etude et développement »

Séquence « Développer des pages Web »

Support 2 JavaScript – JQuery

Apprentissage

Support

Evaluation



Support 2 - JavaScript - JQuery

Afpa © 2019 – Section Tertiaire Informatique – Filière « Etude et développement »

TABLE DES MATIERES

1. OBJETS ET TABLEAUX.....	3
----------------------------	---

1. OBJETS ET TABLEAUX

Pas de classe

Java

```
class Etudiant
{
    public String prenom;
}
...
Etudiant leila=new Etudiant();
leila.prenom="Leïla";
...
```

JavaScript

nouvel objet vide

var leila={};
leila.prenom="Leïla";

propriété ajoutée

En Java:

- on crée une classe
- on instancie un objet de cette classe (**new**)
- on accède à une propriété de l'objet

En JavaScript

- on crée directement un objet (ici, il est vide, mais il pourrait être initialisé)
- on peut ajouter des propriétés simplement

Créer un objet

```
var leila=  
  {  
    prenom : "Leïla",  
    age : 22,  
    specialite : "JavaScript"  
  };
```

Très utilisé !

JSON

Initialisateur d'objet:

```
{  
  propriete1 : valeur1,  
  propriete2 : valeur2,  
  ...  
}
```

créer un objet
en spécifiant
des propriétés
et leurs valeurs

En **JavaScript**, on peut créer un objet en utilisant les accolades "{", "}", et en spécifiant directement des propriétés et leurs valeurs.

De cette façon, on dit qu'on obtient un **objet littéral**.

Cette syntaxe est très utilisée en pratique.

Elle sert aussi comme un format d'échange de données appelé **JSON**.

JSON est utilisé, entre autres, pour la communication entre le navigateur et le serveur. On étudiera JSON ultérieurement.

Propriétés dynamiques

Java

```
class Etudiant
{
    public String prenom;
}
...
Etudiant leila=new Etudiant();
leila.prenom="Leïla";
leila.age=22;
...
```

JavaScript

```
var leila=
{
    prenom: "Leïla"
};
leila.age=22; 😊
```

objet = tableau associatif

leila.age=22; <=> leila['age']=22;

En Java on ne peut pas utiliser une propriété qui n'a pas été déclarée dans la **classe**.

En **JavaScript** on peut ajouter des propriétés dynamiquement en utilisant notamment la propriété **prototype** du langage. Il n'y a **pas de notion de classe**, du moins dans les spécifications ECMAScript 5 (ES5) de **JavaScript**, nous verrons ultérieurement que Microsoft a développé un langage **TypeScript**, qui est un sur-ensemble de **JavaScript** dans sa définition de ES5.

TypeScript permet la création de Classes ... et il est conforme aux nouvelles spécifications ES6.

Pour pouvoir obtenir un script **js** à partir d'un script **ts**, il faut recourir à un « *transpileur* » qui permet une sorte de compilation entre langages de même niveau.

Objets et fonctions

```
var leila=  
  {  
    prenom: "Leïla"  
  };  
leila.age=22;  
leila.afficherNotes=function()  
  {  
    ...  
  };  
leila.afficherNotes();
```

Rappel :
fonction = valeur comme les autres

En JavaScript, les fonctions sont des valeurs comme les autres (**Number**, **String**, **Object**...). On peut les attribuer à des variables. On peut aussi les attribuer aux propriétés d'un objet.

Donc, pour créer une **méthode**, il suffit de créer une propriété dont la valeur est de type **"function"**.

Le JS ne fait rien de spécial pour gérer la méthode. **afficherNotes** est juste une propriété de l'objet, dont la valeur se trouve être de type **"function"**.

Objets et fonctions

```
var leila=  
  {  
    prenom: "Leïla",  
    age: 22,  
    afficherNotes: function()  
    {  
      ...  
    }  
  };  
leila.afficherNotes();
```

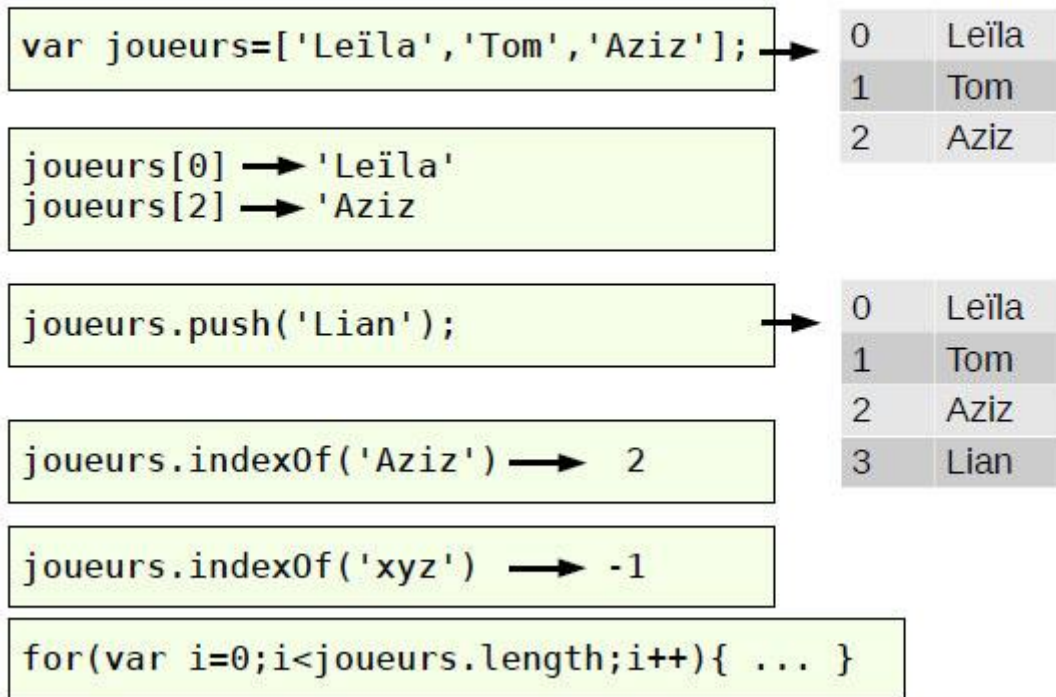
On peut aussi utiliser comme valeur une **fonction anonyme** directement dans l'initialiseur d'objet.

Objets et fonctions

```
var leila=  
  {  
    prenom: "Leïla",  
    age: 22,  
    afficherNotes: xyz  
  };  
function xyz()  
{  
  ...  
}
```

La fonction peut être créée ailleurs. Contrairement au Java, la création d'une méthode en **JavaScript** est simplement une affectation à une propriété d'une valeur de type "**function**".

Tableaux



Les tableaux **JavaScript** se déclarent avec des crochets [...]. Les tableaux sont indexés par un entier et sont numérotés à partir de 0.

De nombreuses méthodes sont associées aux tableaux. Par exemple :

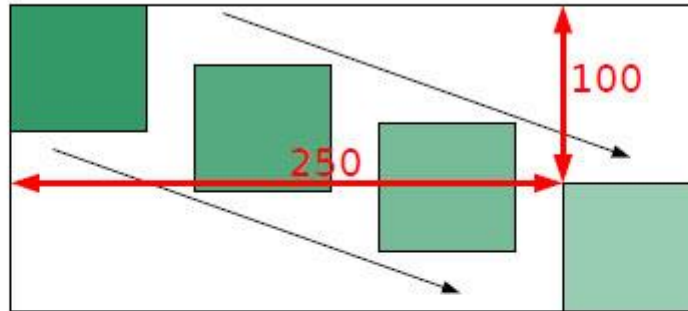
.push(v) permet d'ajouter une valeur à la fin d'un tableau.

.indexOf(v) renvoi l'indice de la première valeur v trouvée dans le tableau

Pour parcourir un tableau, utilisez une boucle numérique.

.length permet d'obtenir la taille d'un tableau.

Exemple : .animate()



```
var options=  
  {  
    top: 100,  
    left: 250,  
    opacity: .5  
  };  
$('#boite').animate(options,1000);
```

millisecondes

Un exemple d'utilisation d'objet:

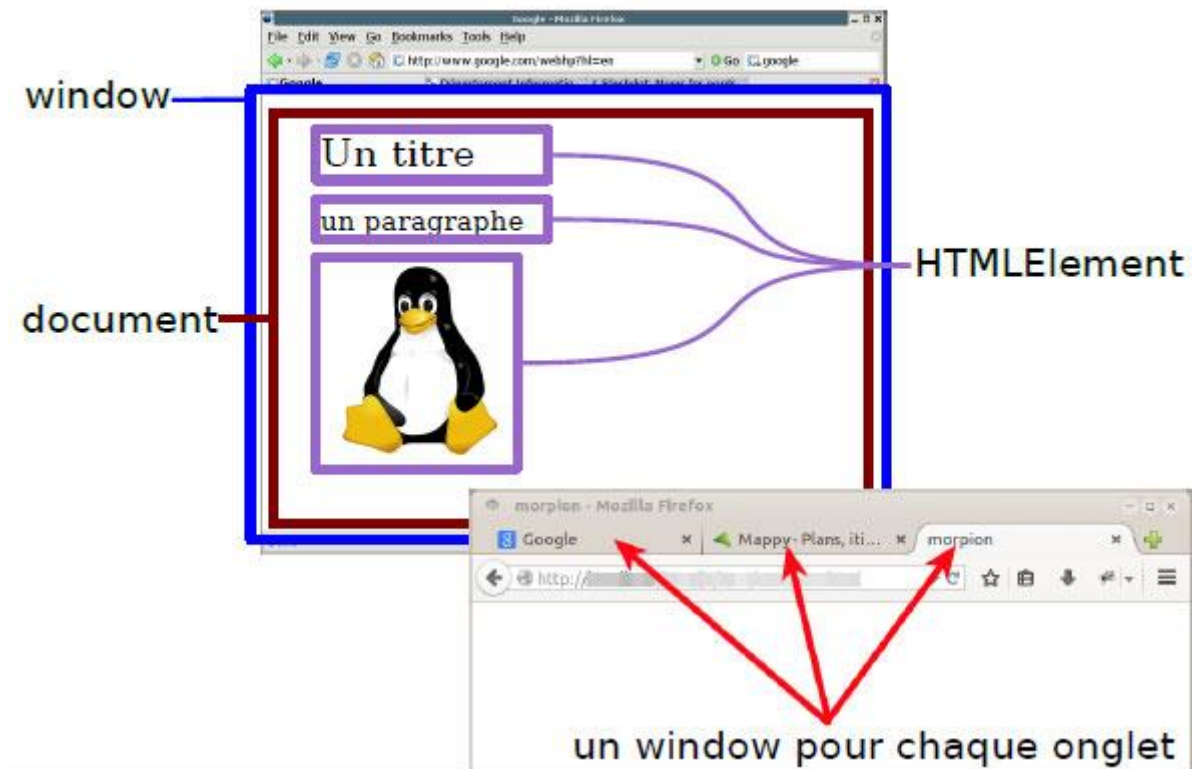
La fonction **jQuery .animate()** permet de changer progressivement les valeurs de propriétés CSS. Le premier argument « options » est un objet indiquant quelles propriétés **CSS** doivent changer progressivement, et vers quelles valeurs elles doivent évoluer.

Ici on passe donc une série d'arguments à une fonction en utilisant un objet.

Cette approche est très utilisée dans **jQuery**. Ici on a décomposé en deux étapes, mais en pratique, on définit souvent l'objet directement dans l'appel de la fonction:

`$('#boite').animate({top: 100, ...},1000);`

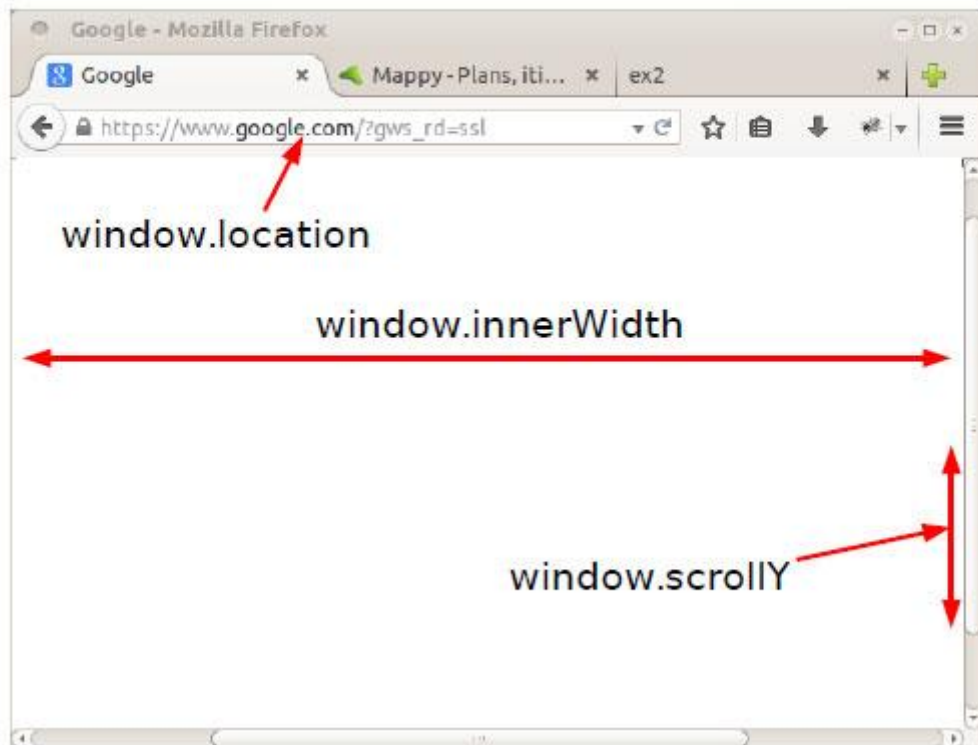
Rappel : objets DOM



Voici 3 types d'objets DOM importants:

- **window** : la fenêtre d'un document. S'il y a plusieurs onglets, chaque onglet a son « window ».
- **document** : à l'intérieur du « window », contient l'**arbre DOM** issu du **HTML**.
- **HTMLElement** : la plupart des nœuds de l'arbre que nous manipulerons sont de type **HTMLElement**

window



window fournit de nombreuses propriétés et fonctions permettant de manipuler la page.

Par exemple, **window.location** permet de connaître l'URL de la page courante. On peut aussi changer **window.location**, dans ce cas le navigateur charge la nouvelle page.

Autres exemples: **window.innerWidth** donne la largeur de la fenêtre. **window.scrollTo** permet de connaître le défilement vertical de la fenêtre (en pixels).

window : objet global



window = objet global

Vos variables globales sont en réalité des propriétés de window.

Les programmes **JavaScript** sont exécutés avec un «**objet global**». Dans un navigateur l'objet global est « window ».

Ça veut dire que si vous n'êtes pas dans une fonction, toutes les variables et fonctions que vous utilisez sont en réalité des propriétés de window.

Par exemple : écrire « `a="bonjour"` » c'est exactement la même chose que d'écrire « `window.a="bonjour"` ».

Donc, quand vous déclarez une variable globale, vous êtes en train d'ajouter une propriété à **window**.

Fonctions window / JS

Méthodes window :

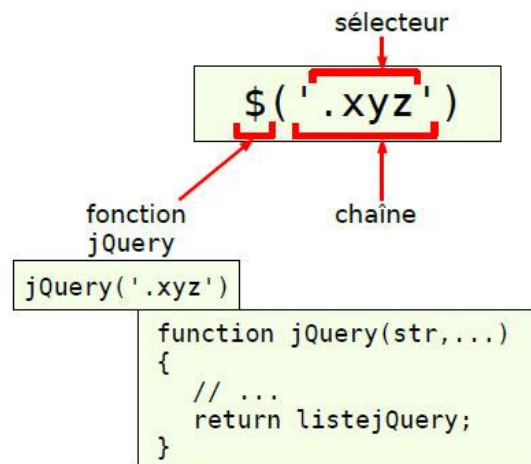
- `window.alert()` -> `alert()`
- `window.console` -> `console`
- `window.setInterval()` -> `setInterval()`
- `window.setTimeout()` -> `setTimeout()`
- ...

Fonctions JavaScript :

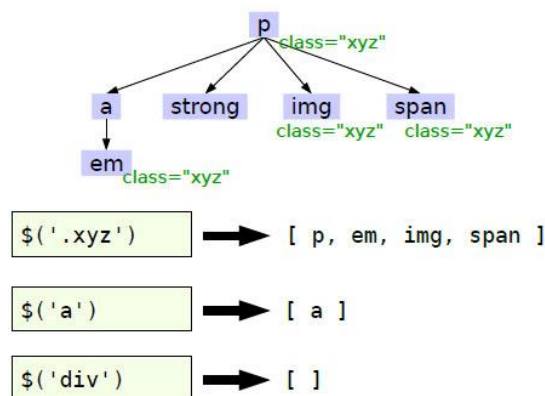
- `parseInt()`
- `eval()`
- ...

- `document` : l'arbre DOM
- **`setInterval()`** : exécuter une fonction régulièrement
- **`setTimeout()`** : exécuter une fonction dans x ms
- ...

Fonction jQuery



jQuery : listes

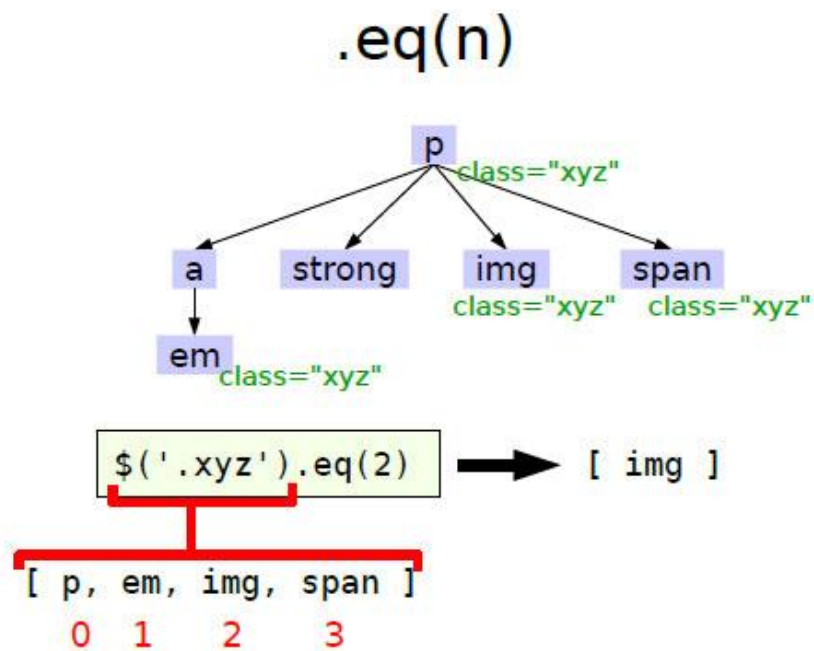


Comme on l'a vu **\$('xyz')** permet de créer une **liste jQuery**. Elle est constituée de tous les éléments désignés par le sélecteur **xyz**.

Cette liste peut contenir de nombreux éléments. Elle peut aussi ne contenir qu'un seul élément, ou parfois même aucun. Sa taille est donnée par « **.length** ».

Ces listes sont le support de la majorité d'opérations faites en jQuery. Comme indiqué au 1er cours, il existe plus de 180 fonctions s'appliquant sur une liste jQuery.

On a souvent besoin de les manipuler ces listes.

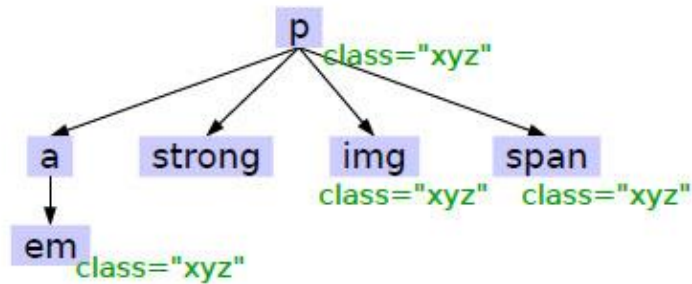


La fonction JQuery **.eq(n)** renvoie une nouvelle **liste JQuery** contenant un seul élément déterminé par son indice **n**.

Remarquez comment on applique une fonction sur une liste pour en obtenir une autre liste.

Cette technique est largement utilisée avec **JQuery**, elle permet, comme nous allons le voir, le **chaînage** des fonctions, puisqu'à chaque fois l'application de ces fonctions renvoie une autre liste qui a son tour peut être encore « *filtrée* » ...

.parent(), .children()



```
$('#img').parent()
```

➡ [p]

```
$('#p').children()
```

➡ [a, strong, img, span]

```
$('#p').children('.xyz')
```

➡ [img, span]

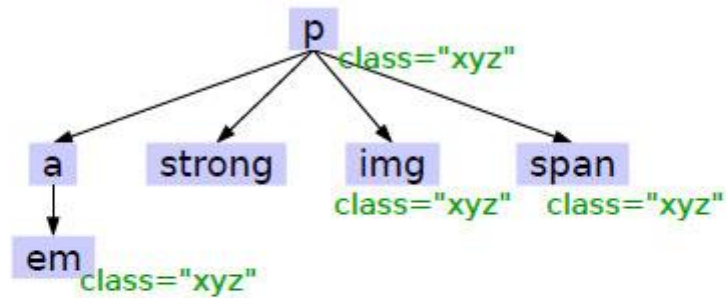
Le **HTML** est représenté par un **arbre**. On a donc souvent besoin de parcourir cet arbre.

Par exemple, avec **.parent()** on obtient l'élément situé **au-dessus** d'un autre dans l'arbre.

.children() permet d'obtenir tous les éléments contenus dans un autre (**descendants directs**).

Si on fournit un sélecteur en argument «**.children('xyz')**», alors seuls *les fils directs* correspondant à ce sélecteur sont gardés dans la liste.

.find()

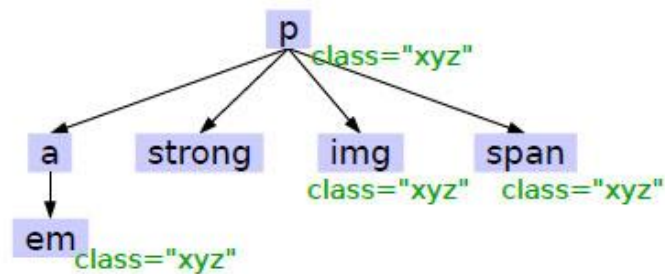


`$('p').children('.xyz')` ➡ `[img, span]`

`$('p').find('.xyz')` ➡ `[em, img, span]`

.children() retourne uniquement les **descendants directs**, alors que **.find()** retourne les **descendants directs et indirects**.

Enchaîner



`$('p').children('.xyz')` ➡ [img, span]

`$('p').children('.xyz').eq(0)` ➡ [img]

`$('a').parent()` ➡ [p]

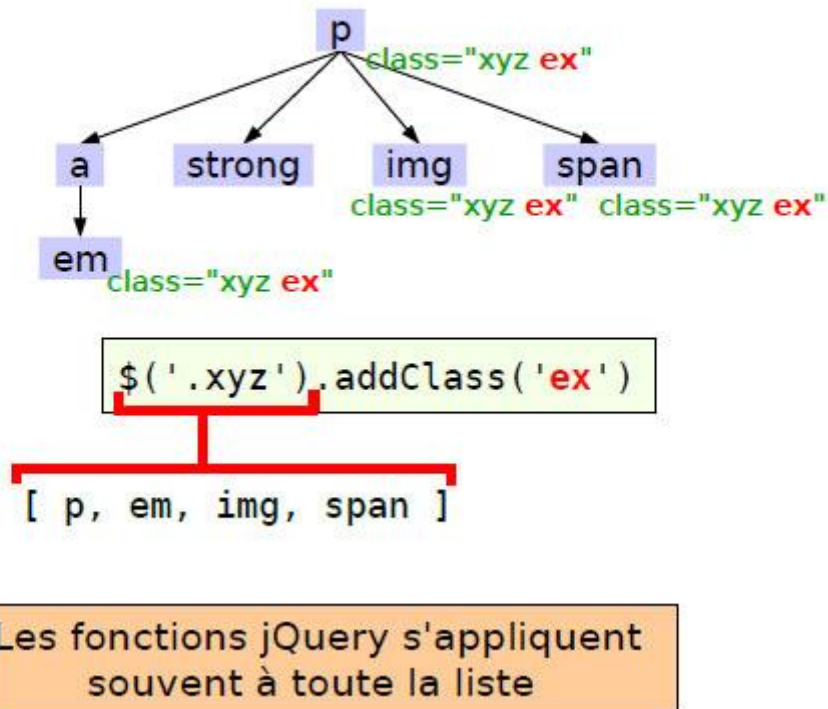
`$('a').parent().find('img')` ➡ [img]

Toutes les fonctions **JQuery** que nous venons de voir s'appliquent à une **liste JQuery** et retournent une liste JQuery.

On peut donc appeler une fonction JQuery sur la nouvelle liste JQuery. Ceci permet d'enchaîner des appels de fonctions JQuery.

C'est très pratique, et c'est ce qui fait la force de la bibliothèque JQuery.

Effet sur toute la liste



En général (mais pas toujours), les fonctions **jQuery** ont un effet sur tous les éléments de la **liste jQuery**.

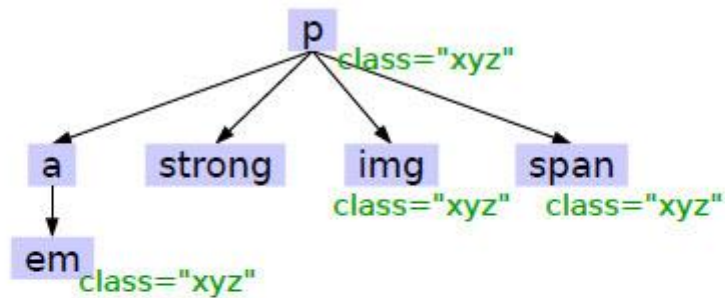
Par exemple `.addClass('ex')` ajoute la classe **'ex'** à **TOUS** les éléments de la liste.

Rappel: en HTML un élément peut avoir plusieurs classes, on les sépare par des espaces:

`<p class="xyz ex">`

Ici il y a bien deux classes : **"xyz"** et **"ex"**.

.each()



```
$('.xyz').each(function()  
{  
    $(this).css('left',100*Math.random());  
});
```

.each() :
parcourir tous les éléments de la liste ... « boucle »

Comme les fonctions **jQuery** s'appliquent sur toute une liste, on peut souvent faire des opérations simples sur toute une liste, sans avoir à traiter chaque élément de la liste séparément.

Pour les opérations plus compliquées, jQuery fournit une sorte de « *boucle* ».

liste.**each**(ma_fonction) appelle **ma_fonction** pour chacun des éléments de la liste. Dans **ma_fonction** l'élément est accessible avec « **this** ».

.attr()

```
<a id="lien5" class="xyz ex" href="page.html">...</a>
```

```
$('#a').attr('id')      ➡ 'lien5'  
$('#a').attr('class') ➡ 'xyz ex'  
$('#a').attr('href') ➡ 'page.html'
```

```
$('#a').attr('href', 'scores.html'),
```

```
<a id="lien5" class="xyz ex" href="scores.html">...</a>
```

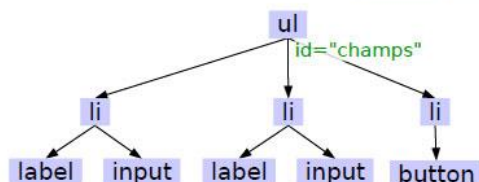
La fonction JQuery **.attr()** permet de **lire** et de **modifier** un attribut HTML.

Exemple : .parent()

```
<ul id="champs">  
  <li><label>Nom : </label><input type="text"/></li>  
  <li><label>Prénom :</label><input type="text"/></li>  
  <li><button>Envoyer</button> </li>  
</ul>
```

Nom :

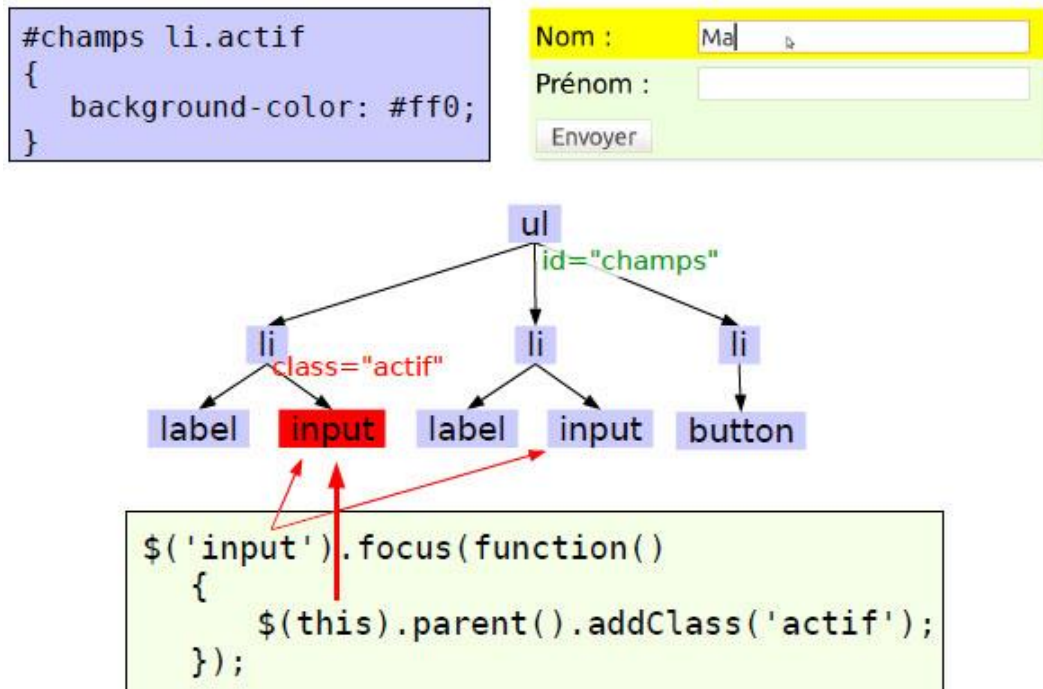
Prénom :



Voici un exemple pour illustrer l'utilisation de **.parent()**

Dans un formulaire constitué de plusieurs lignes, on voudrait changer la couleur de la ligne que l'utilisateur est en train de modifier.

Exemple : .parent()



L'événement « **focus** » permet de savoir quand l'utilisateur commence à éditer un champ.

L'événement « **blur** » permet de savoir quand il le quitte.

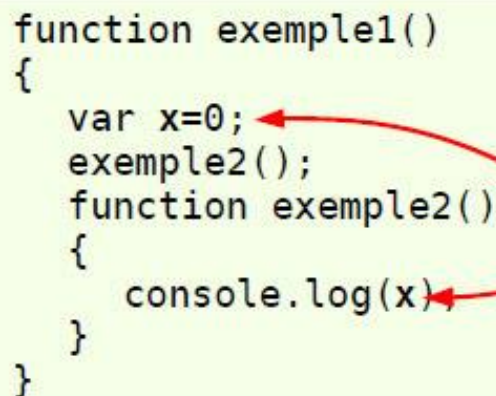
.focus est appelé sur le `<input>` donc "**this**" dans le gestionnaire d'événement correspond au `<input>`.

On voudrait modifier la ligne `` qui contient ce `<input>`.

On utilise donc `$(this).parent()` auquel on ajoute la class `aktif`, ce qui a pour effet de colorer le background de la ligne `` ciblée en jaune !!!

Fermetures

```
function exemple1()
{
    var x=0;
    exemple2();
    function exemple2()
    {
        console.log(x);
    }
}
```



Fermeture :

Une fonction utilisant des variables définies dans une autre fonction contenante.

Les **fermetures** ou **closures** existent dans d'autres langages, mais elles sont **très utilisées en JavaScript**.

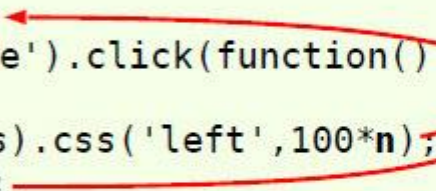
Il s'agit tout simplement du fait de pouvoir utiliser des variables définies dans une fonction parent.

Ici la variable « x » est utilisée dans **exemple2 ()** alors qu'elle est définie dans **exemple1 ()**.

Dans d'autres langages, une variable n'est accessible que dans la fonction qui la déclare.

Fermetures

```
$(document).ready(function()  
{  
  var n=0;  
  $('.boite').click(function()  
  {  
    $(this).css('left',100*n);  
    n=n+1;  
  });  
});
```



Programmation événementielle



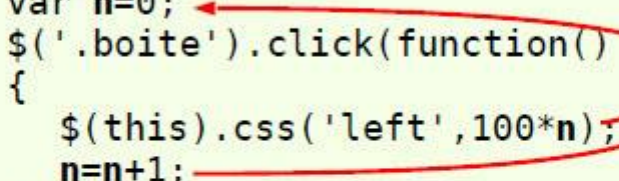
Dans cet exemple la variable `n` est définie dans la 1^{ère} fonction anonyme. La variable `n` est utilisée (lue et modifiée) dans la fonction anonyme appelée par `.click()`.

On a vu qu'en **programmation événementielle** on déclare souvent des fonctions à appeler « **plus tard** », quand un événement survient.

On se retrouve donc avec beaucoup de fonctions et on a besoin de partager des informations entre elles. Les fermetures sont alors très pratiques.

Fermetures : facile ?

```
$(document).ready(function()  
{  
  var n=0;  
  $('.boite').click(function()  
  {  
    $(this).css('left',100*n);  
    n=n+1;  
  });  
});
```



Très utilisé, très intuitif !



Quoi !?



Accéder aux variables locales d'une autre fonction ?!

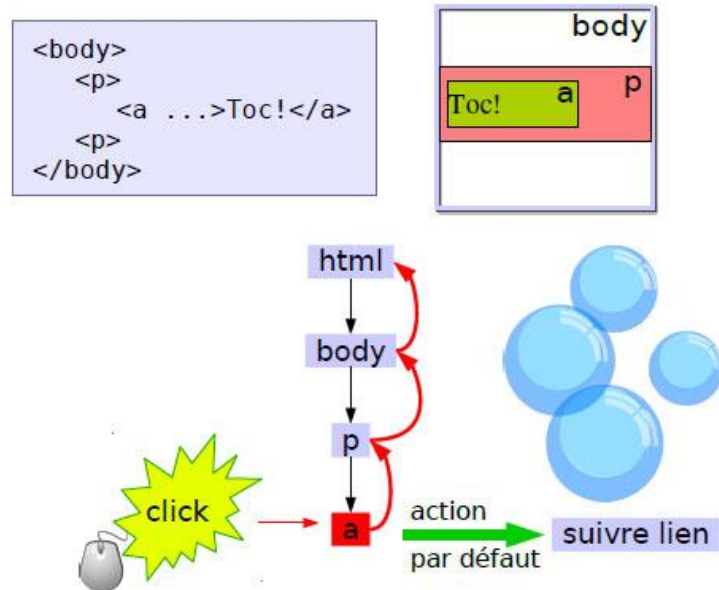
L'autre fonction a déjà fini d'exécuter ?!

Les **fermetures** sont simples et intuitives à utiliser, mais en y regardant de plus près, il se passe des choses un peu bizarres.

Tout d'abord l'idée d'accéder à des variables locales déclarées dans une autre fonction est surprenante.

Ensuite, on se rend compte que ces variables sont utilisées alors que la fonction qui les déclare a déjà fini d'exécuter !

Événements : bubbling



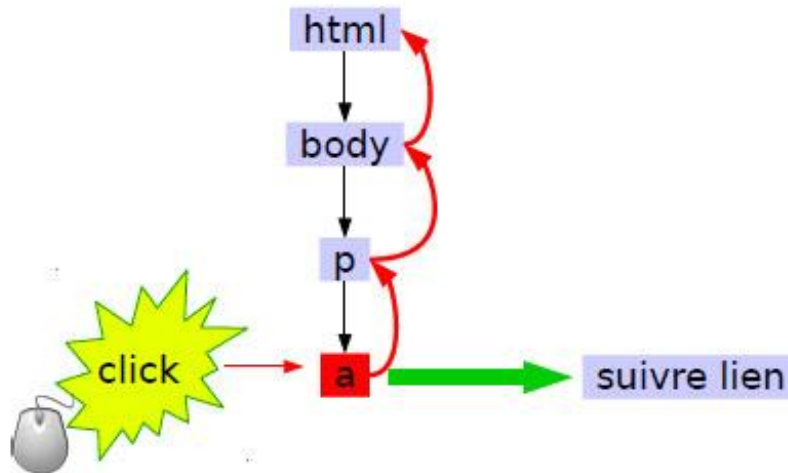
Quand un utilisateur fait une action, un objet événement est créé. Il est successivement envoyé à :

- l'élément où il s'est produit.
- l'élément **parent**
- l'élément **grand parent**
- ...
- ensuite est exécutée l'action par défaut

Exemples d'actions par défaut :

- "**click**" sur un lien : suivre le lien
- "**keypress**" dans un champ texte : afficher la lettre
- "**click**" sur un bouton : envoyer le formulaire
- "**click**" dans une boîte à cocher : cocher
- "**submit**" pour un formulaire : envoyer formulaire
- ...

Événements : bubbling



```
$('html').click(function(e){/* this:html e.target:a */});  
$('body').click(function(e){/* this:body e.target:a */});  
$('p').click(function(e){/* this:p e.target:a */});  
$('a').click(function(e){/* this:a e.target:a */});
```

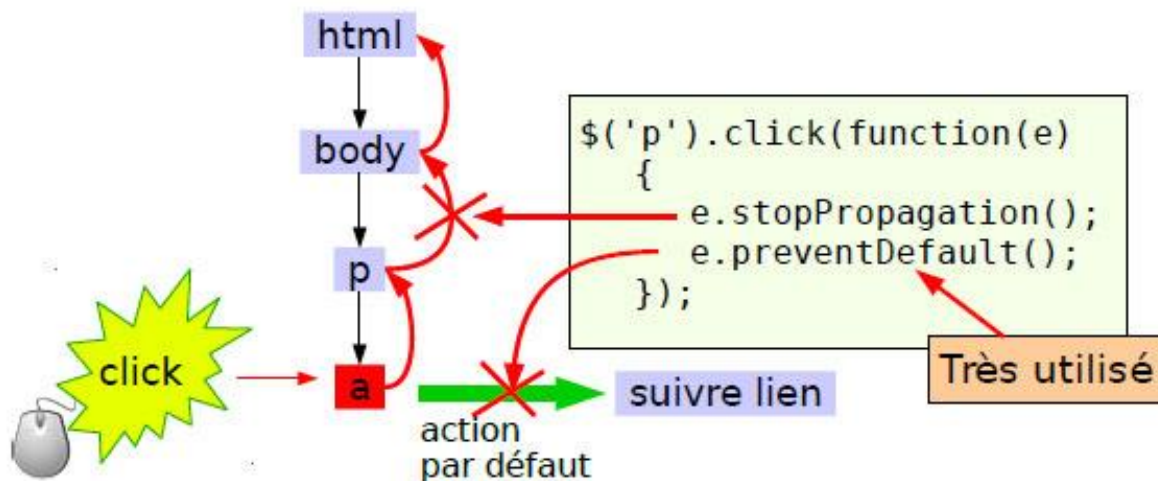
On peut ajouter un gestionnaire d'événement **click** à n'importe quel élément sur toute la chaîne. Ils seront tous appelés.

Par exemple, si on utilise un gestionnaire d'événement **click** sur "body", on récupère les clicks de tous les éléments de la page.

"**this**" est l'objet sur lequel se trouve le gestionnaire d'événement (notre fonction)

"**e.target**" est l'objet sur lequel s'est produit l'événement.

Événements



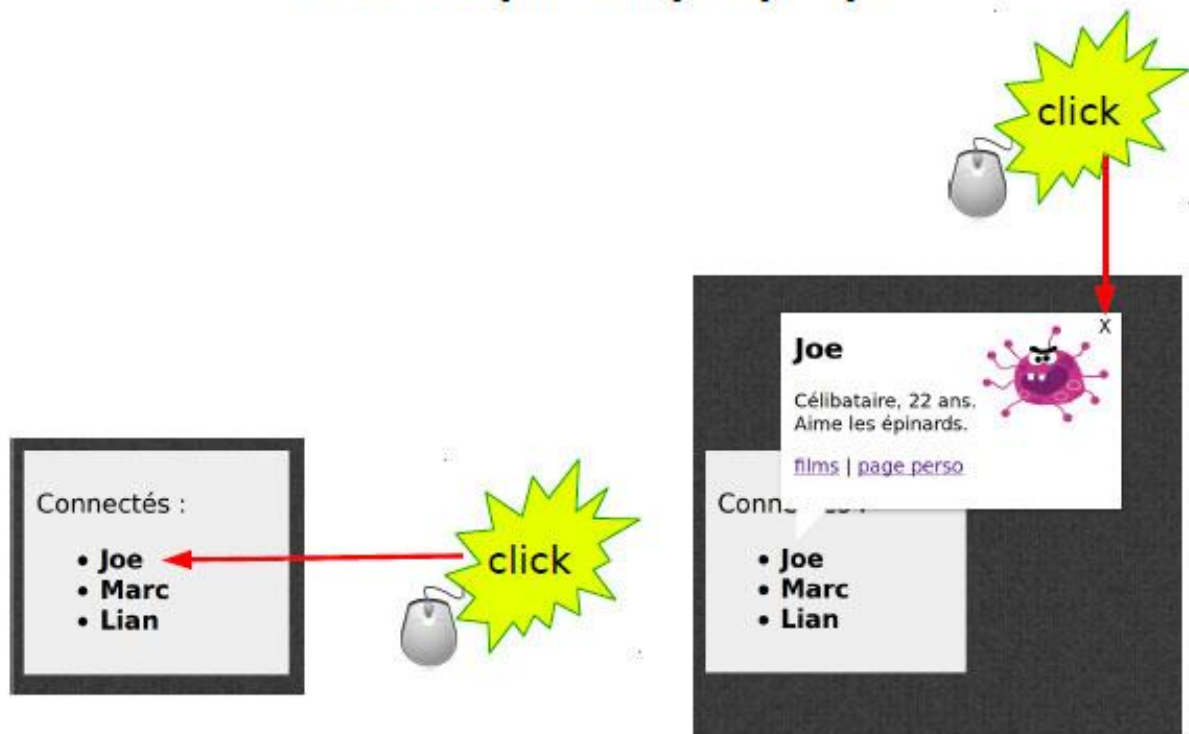
L'action par défaut peut-être annulée avec: **e.preventDefault()**

C'est très utilisé : quand on utilise un gestionnaire d'événement, on veut souvent remplacer la fonctionnalité par défaut par notre fonctionnalité, il faut donc utiliser **preventDefault()**.

Exemples :

- "**click**" sur un lien : ne suit pas le lien
- "keypress" dans un champ texte : n'affiche pas la lettre
- ...

Exemple : popup



Dans cet exemple, l'utilisateur peut cliquer sur un nom pour afficher un **popup** dans lequel sont affichées des informations sur la personne.

Pour fermer le **popup**, il doit cliquer sur la croix située en haut à droite du popup.

Exemple : HTML

```
<div id="connectes">
  <p>Connectés : </p>
  <ul>
    <li><span class="popup-mot" data-popup="joe" >Joe </sp
    <li><span class="popup-mot" data-popup="marc">Marc</sp
    <li><span class="popup-mot" data-popup="lian">Lian</sp
  </ul>
</div>

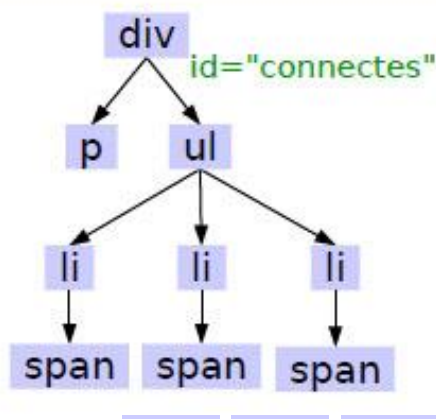
<div id="popup-joe" class="popup">
  <span class="fermer">X</span>
  
  
  <h2>Joe</h2>
  <p>Célibataire, 22 ans.<br/>Aime les épinards.</p>
  <p>
    <a href="...">films</a> | <a href="...">page perso</a>
  </p>
</div>
```

Le HTML est composé de deux parties :

- La liste dans laquelle sont affichés les noms.
- Le popup et son contenu

Exemple : liste

```
<div id="connectes">
  <p>Connectés : </p>
  <ul>
    <li><span class="popup-mot" data-popup="joe" >Joe </sp
    <li><span class="popup-mot" data-popup="marc">Marc</sp
    <li><span class="popup-mot" data-popup="lian">Lian</sp
  </ul>
</div>
```



La liste n'a rien de particulier.

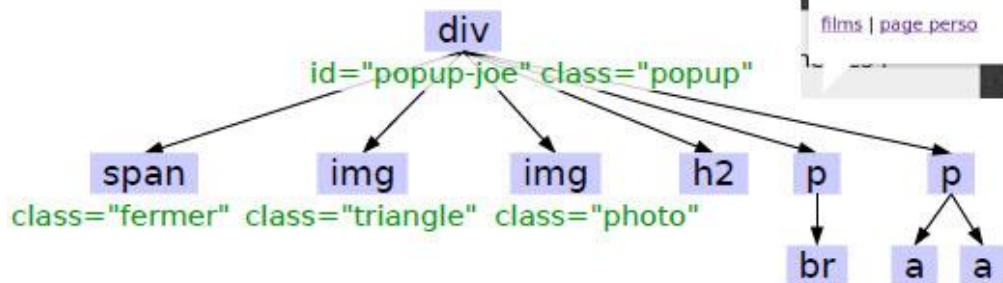
Les noms sont dans des **** de classe **"popup-mot"** et ayant un **attribut "data-popup"** contenant un identifiant associé au nom.

La norme **HTML 5** permet de créer des attributs dont le nom commence par **"data-"**. Cette propriété est largement utilisée pour la construction du **Framework JQuery Mobile**.

On s'en sert ici pour stocker un identifiant qui nous permettra de trouver le popup correspondant à ce nom.

Exemple : popup

```
<div id="popup-joe" class="popup">
  <span class="fermer">X</span>
  
  
  <h2>Joe</h2>
  <p>Célibataire, 22 ans.<br/>Aime les épinards.</p>
  <p>
    <a href="#">films</a> | <a href="#">page perso</a>
  </p>
</div>
```



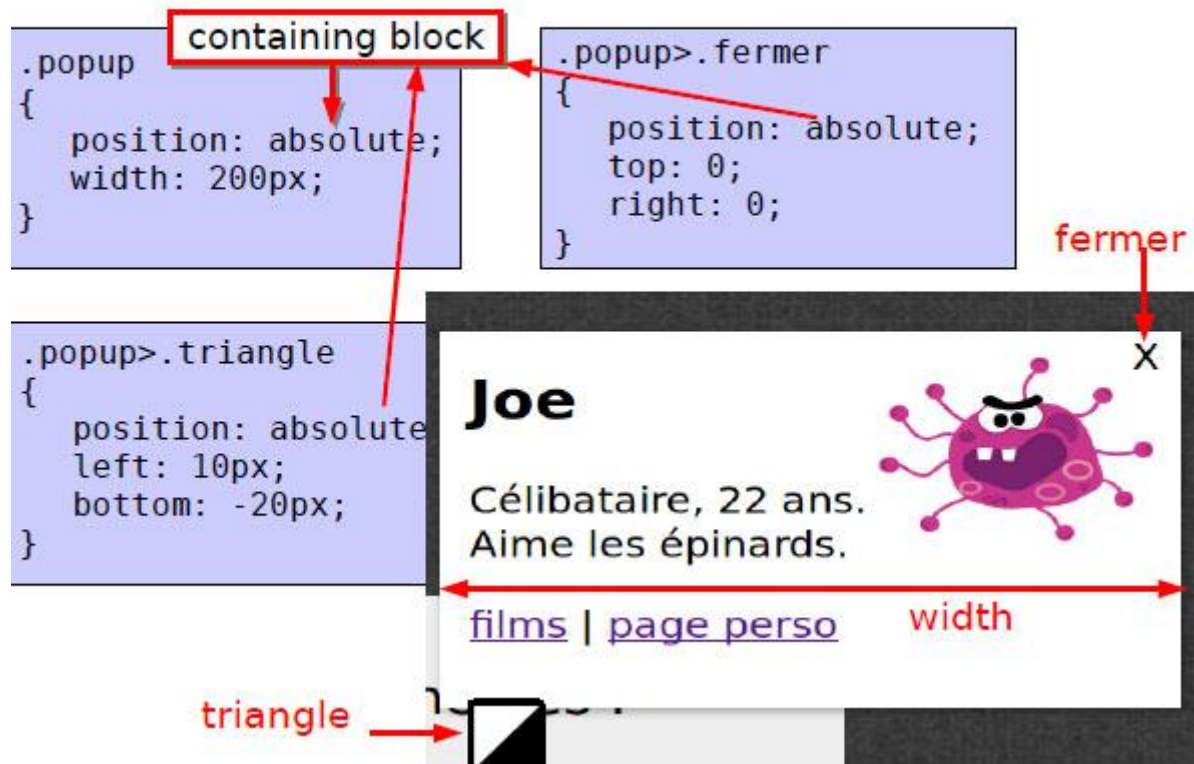
On peut imaginer qu'il existe plusieurs **popup**, tous ayant la même structure.

Tous les popup ont **class="popup"**. Ils contiennent

- un **** avec une croix pour les fermer.
- une **image** "triangle.png" qui représente le petit triangle blanc en bas du popup
- le contenu du popup (photo, titre, texte, liens)

Une classe (fermer, triangle, photo) nous permettra d'agir sur ces éléments dans le CSS et le JS.

Exemple : CSS



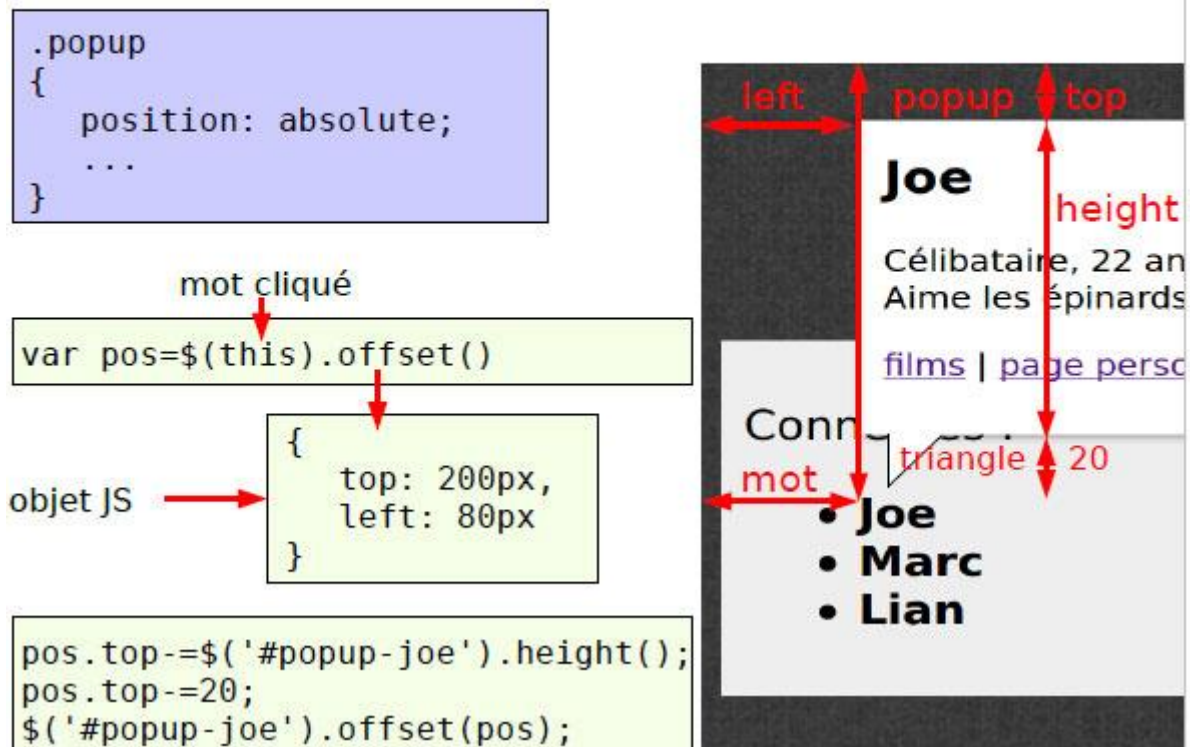
Le popup a "position: absolute", ce qui permet de le déplacer n'importe où à l'aide de "left" et "top" (ce sera fait dans le JS)

Par ailleurs, "**position: absolute**" transforme le popup en un "containing block" : tous les éléments à l'intérieur du popup qui ont "position: absolute" sont positionnés par rapport au popup (et non pas par rapport à la page).

C'est pour ça que "top:0" positionne **.fermer** en haut du popup et pas en haut de la page. De la même manière l'image **.triangle** est positionné en bas du popup (et pas en bas de la page).

La valeur négative de bottom, fait sortir l'image en dehors de la région d'affichage de son parent (le popup). Ce n'est pas un problème, cette situation s'appelle un "**overflow**".

Exemple : .offset()



Notre première tâche en JS est de placer le popup au-dessus du mot qui a été cliqué.

On utilise `.offset()` pour obtenir la position du mot cliqué. `.offset()` renvoie un objet avec deux propriétés (`top`, `left`).

On veut que le bas du triangle se trouve au même niveau que le mot. Pour trouver le haut de la fenêtre, il faut donc soustraire la hauteur du popup et du triangle.

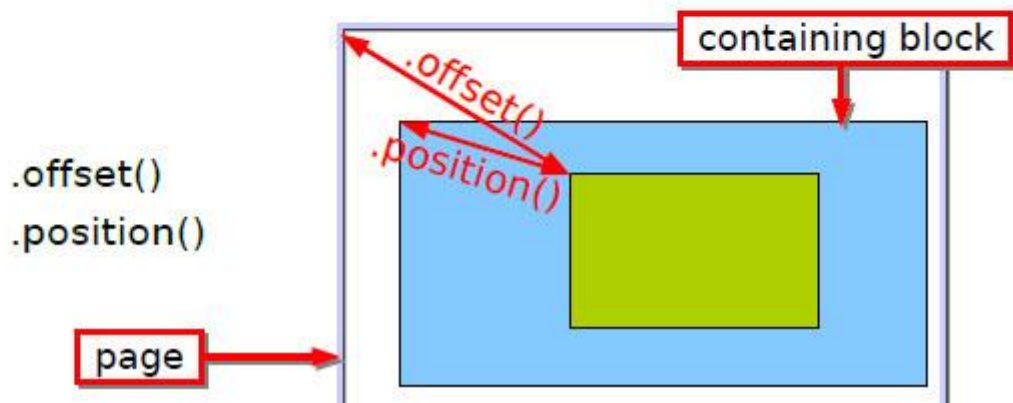
`.offset(pos)` nous permet ensuite de positionner le popup.

Exemple : .offset()

```
$('#popup-joe').offset({left: 100,top: 50});
```

```
<div id="popup-joe" >...</div>
```

```
<div id="popup-joe" style="left: 100; top: 50">...</div>
```



.offset({...}) fonctionne en modifiant la propriété "style" de l'élément HTML.

Il existe deux fonctions pour obtenir une position:

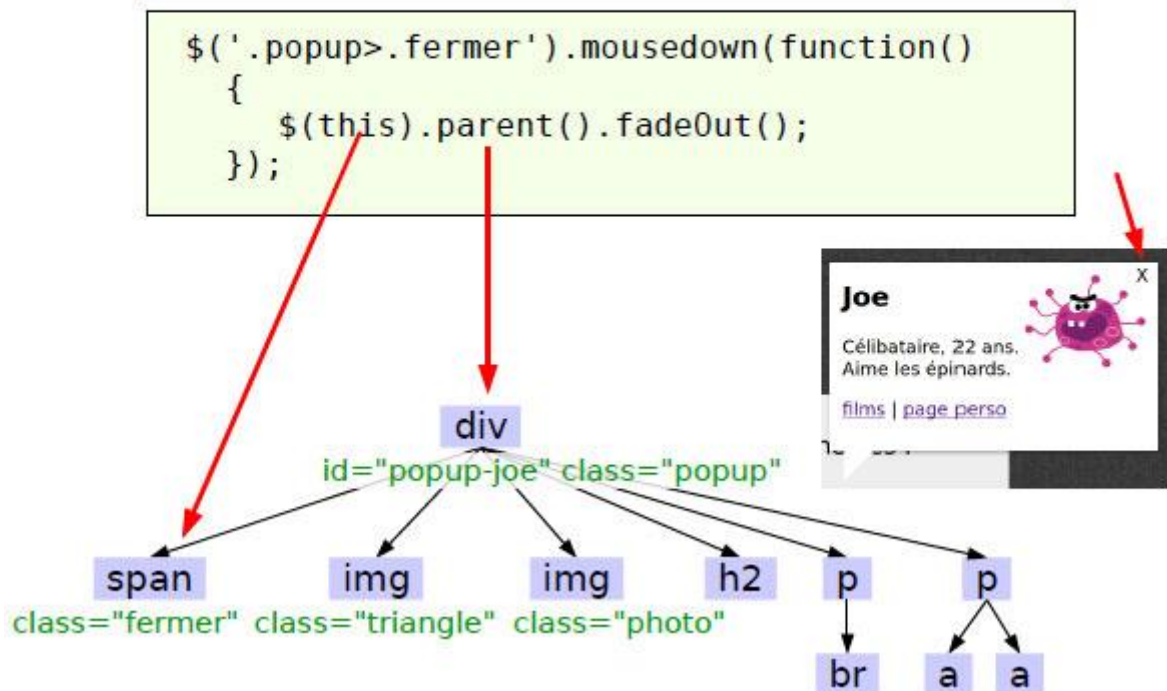
- .offset()** donne la position relative à la page

- .position()** donne la position relative à son "containing block"

- .position()** correspond donc tout simplement aux propriétés CSS left et top.

.offset() fait en interne un calcul, remontant tous les "containing blocks" jusque en haut de l'arbre DOM.

Exemple : fermer



Dans la fonction gérant le click sur le ``, on a accès à "this". Mais "this" correspond au ``, alors qu'on a besoin du `<div class="popup">` pour fermer le popup.

On utilise donc **`$(this).parent()`**

`.fadeOut()` permet de cacher un élément progressivement, avec une animation.

Exemple : JS

```
0  $(' .popup-mot').mousedown(function()  
1      {  
2          var mot=$(this);  
3          var id="popup-"+mot.attr('data-popup');  
4          var popup=$('#'+id);  
5          var pos=mot.offset();  
6          pos.top-=popup.outerHeight()+20;  
7          popup.fadeIn();  
8          popup.offset(pos);  
9      });  
10  
11  $(' .popup>.fermer').mousedown(function()  
12      {  
13          $(this).parent().fadeOut();  
14      });
```

```
<li><span class="popup-mot" data-popup="joe" >Joe </span></li>  
...  
<div id="popup-joe" class="popup">
```

Voici le JS complet.

Remarquez qu'on peut mettre les listes jQuery dans des variables. C'est plus lisible et plus rapide.

À partir de l'attribut "**datapopup**" du mot cliqué on construit le sélecteur '**#popupjoe**'.

Remarquez à la ligne 0 que \$(' .popupmot') contient plusieurs mots.

jQuery ajoute donc notre fonction gestionnaire d'événement à plusieurs éléments.

C'est aussi le cas pour **.popup>.fermer** (on peut imaginer qu'il y a plusieurs popup ...)

CREDITS

ŒUVRE COLLECTIVE DE l'AFPA

**Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services
Et l'appui du professeur M. BOSC de l'université Paris 13.
Document sous License Cnu Fdl.**

Equipe de conception (IF, formateur, mediatiseur)

Formateur : Alexandre RESTOUEIX

Date de mise à jour : 29/01/19

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.

« Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droits ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction par un art ou un procédé quelconque. »