Javascript - DOM et Composants

par JYT (Les expériences Zend de Sekaijin) (Blog)

Date de publication : 17/02/2008

Dernière mise à jour :

Principe de développement d'interface utilisateur enrichie en JavaScript par manipulation directe du DOM. En faisant porter directement aux éléments du DOM les attributs et les méthodes, nous obtenons une grande souplesse et indépendances des composants d'un document HTML.

I - JavaScript et le DOM

1.1 - Composants d'un document HTML :

Un document HTML est une structure hiérarchique qui a été normalisée par le ***W3C***. Cette définition est standard. On la retrouve, quel que soit l'outil utilisé pour l'afficher. Il va sans dire (mais c'est mieux en le disant) qu'il existe bien des disparités quant à l'interprétation de cette structure. Je n'entrerai pas ici dans les détails de cette définition. La littérature est suffisamment vaste. La seule chose qui nous intéresse ici est qu'un document HTML est une structure hiérarchique d'éléments. Chaque noeud est un élément HTML, et en fonction de son type, doit se trouver à un endroit de la hiérarchie et pas à un autre. Les possibilités sont suffisamment larges pour laisser libre cours à son imagination. Un autre point important, dans un document HTML, est que seuls certains des éléments d'un document HTML, et seulement ceux-là, sont susceptibles d'entrer en interaction avec l'utilisateur. Un document HTML est, en soit, relativement statique. C'est pour cette raison qu'a été introduit JavaScript. Ce langage de script embarqué permet de manipuler le document HTML pour un rendu plus dynamique. JavaScript n'introduit pas de nouvel élément dans l'interaction avec l'utilisateur. En clair, il n'est pas possible en JavaScript de créer un élément qui sera vu de l'utilisateur et avec lequel il pourra interagir. La seule chose que permet de faire JavaScript au niveau de l'interface avec l'utilisateur, c'est d'ajouter, supprimer ou modifier des éléments HTML dans le document. IL est donc nécessaire lorsqu'on veut modifier le comportement de l'interface, d'utiliser les composants HTML du document.

1.2 - Éléments scripts dans un document HTML :

Un script embarqué dans le document ou lié au document, où que ce soit dans le code est toujours rattaché au même endroit dans la structure du document. On le retrouvera systématiquement dans **document.scripts**. Vous pouvez éparpiller vos scripts partout dans vos document HTML, l'interprète les placera systématiquement à cet endroit. Il existe cependant un petit détail qui change quelque peut la donne suivant l'endroit où l'on insère son script dans le document. Ce détail est que si le script lors de son interprétation, ou son chargement appelle la méthode **document.write** le résultat sera placé à l'endroit du script. Cela à un côté pratique mais aussi parfois trompeur. Il est peut-être plus complexe dans un premier temps de rechercher l'objet du document que l'on veut modifier et le manipuler. Mais cette approche est beaucoup plus sûre et portable que ne l'est **document.write**. Je ne saurais que trop vous recommander de regrouper le plus possible vos scripts dans l'entête du document. Cela facilite la lecture. Votre code HTML ne contenant que du HTML il n'y a pas de risque de vous perdre. Les scripts JavaScript peuvent servir à bien d'autre chose qu'à la manipulation de l'interface. J'en tiens pour exemple le portage du jeu Awalé (jawale http://myriad-online.com) par la société Myriad en pur JavaScript de la première mouture implémenté dans #Navigator de Netscape#. Une bonne partie de ce code servait à faire jouer l'ordinateur de façon qui était loin d'être bête. Mais ce qui nous intéresse dans cet article c'est l'interaction avec l'utilisateur.

1.3 - Interagir avec l'interface :

Pour agir sur l'interface, les premières approches consistaient à écrire un peu de code pour gérer les évènements dans le Document HTML. Cette écriture a l'avantage d'une grande simplicité, on écrit le code directement sur l'objet concerné. **<a href="" onClick="alert('un peu de code pour un click'); return false;">** C'est simple mais cela montre rapidement ses limites. En effet comment écrire un code qui agit sur plusieurs éléments avec cette méthode. Le ***W3C*** a défini pour cela le ***DOM***. Cet objet qui peut paraître monstrueux dans un premier temps est en fait l'exacte représentation de votre document HTML. Attention toute fois, ce n'est pas votre code HTML ni même la représentation de la structure HTML que vous avez écrite mais c'est la représentation de ce que vous auriez dû écrire pour être conforme à la définition du ***W3C***. C'est tout du moins ce que croit l'interprète HTML de votre navigateur. Souvenez-vous des scripts qui ne se trouvent pas dans la structure à l'endroit où vous avez écrit le code. Il en va de même avec d'autres éléments HTML par exemple si vous ne mettez pas de TBODY dans vos tableaux, vous en trouverez dans le ***DOM***. Ils seront ajoutés automatiquement. Les différents composants du ***DOM*** sont les représentants en mémoire des éléments qu'affiche votre navigateur. Dès ses débuts JavaScript a représenté ces éléments sous forme d'un arbre. Chaque composant correspond à un TAG. C'est un objet muni d'un certain

nombre d'attributs. Au tout début de JavaScript ses attributs ont été rendus accessibles directement. Suivant le navigateur utilisé, les composants n'avaient pas toujours le même nom dans l'arbre, ni les mêmes attributs. Depuis le ***W3C*** y a mis bon ordre avec ***ecmaScript*** et s'il reste des disparités, utiliser les méthodes définies par le ***W3C*** offre certaines garanties de portabilité. (Il reste encore du travail, mais l'essentiel est fait) Je vous conseille donc de lire la doc. (http://www.yoyodesign.com) car même avec peu il est possible de faire les choses proprement. Il suffit pour cela de se donner quelques petits principes. Par exemple : ne pas parcourir l'arbre par les membres des objets (***document.body.myform***). Ce sont des restes de la compatibilité ascendante. Utilisez les méthodes du ***DOM***, prévues à cet effet. La plus simple à utiliser est **document.getElementById** elle permet de retrouver n'importe quel élément de l'arbre pourvu qu'il ait un id. c'est donc une bonne pratique que de mettre des id sur ses objets. Les méthodes **getAttribute** et **setAttribute** permettent sur un objet du ***DOM*** de lire ou fixer la valeur d'un attribut. C'est lourd c'est verbeux mais c'est la norme. Et utiliser les normes, apporte des avantages. Pour ajouter des éléments dans l'arbre, le ***W3C*** nous fournit quelques méthodes. La première est **document.createElement**. Notez que celle-ci porte sur le document. On crée donc un élément dans le document. Mais il n'est pas placé dans l'arbre. Pour cela il est nécessaire de sélectionner un objet du ***DOM*** et d'utiliser les méthodes **appendChild**, **inserBefore**, **insertAfter**. La première insère l'élément comme dernier noud de l'élément sélectionné ; les suivantes avant ou après un noeud donné, fils du noeud sélectionné. De même pour supprimer un noeud il faut sélectionner le père et appeler la méthode **removeChild**. Je ne ferais pas ici un cours sur le ***DOM*** je vous renvoie à la doc pour plus d'informations.

1.4 - Portée des variables et des fonctions :

Dès que l'on veut écrire un code un peu plus complexe qu'une ligne pour la gestion d'un évènement, on se retrouve vite avec une série de fonctions et de variables. En JavaScript, la portée des variables est à la fois simple et complexe. Simple car par défaut une variable est connue hors du bloc qui la définit, il est donc facile de positionner une valeur dans un bloc et de l'utiliser dans un autre. Mais il n'est pas évident du tout lorsque leur nombre se multiplie de ne pas induire des effets indésirables. Particulièrement lorsqu'on charge plusieurs scripts récupérés sur divers sites Internet. JavaScript ne le fait pas pour nous comme d'autres langages, j'ai donc pris pour principe de toujours assurer l'étanchéité des blocs. Pensez au mot-clef var lorsque vous déclarez une variable. Elle sera définie dans le bloc où elle est déclarée. Pour ce qui est des fonctions nous n'avons d'autre choix que de les définir au niveau de l'interprète JavaScript. Du coup toutes les fonctions sont définies au même niveau. Ce qui était valable pour les variables l'est aussi pour les fonctions. Une fonction est définie de façon globale et elle est visible partout. (Je parle ici de fonction et non de méthode des objets.) Le risque d'avoir deux fonctions dans deux scripts qui porte le même nom est sûrement moins courant que pour une variable, mais il n'est pas nul. Je ne saurais que trop vous encourager quitte à être verbeux de bien veiller aux noms de vos fonctions et vos variables. JavaScript offre tout de même la possibilité de définir une fonction locale. Pour y parvenir il suffit de définir une variable locale (avec var) et de lui affecter pour valeur une fonction anonyme. Le mot-clef **function** en JavaScript a plusieurs syntaxes. La plus connue est **function nomDeFonction(parametreList) {/*corps de la fonction*/};** qui définit la fonction globale nomDeFonction ce mot clef agit en fait comme un opérateur de construction qui retourne une référence à une fonction. Un peu comme le mot-clef new le fait avec les objets dans les langages à objets. **var nomDeFonction = function(parametreList) {/*corps de la fonction*/};** est équivalent à la déclaration précédente hormis le **var** qui précède qui rend cette définition locale. Là encore n'hésitez pas à définir des fonctions locales lorsque celles-ci n'ont d'autre utilité que dans un bloc donné. Vous assurerez ainsi une meilleure étanchéité de votre code.

1.5 - Programmation à objet en JavaScript :

JavaScript est un langage à objet qui a pour principe la construction dynamique d'objet et non, comme dans beaucoup, l'utilisation de classes. Ce choix s'explique par la vocation première du langage qui est le scripting. Dans ce contexte, on a bien souvent plus besoin d'objet unique que d'ensembles d'objets ayant la même classe. Il est possible d'introduire une notion de pseudo-classe. Un tableau associatif, ou un objet sont exactement la même chose en JavaScript. Pour implémenter les méthodes d'un objet, JavaScript utilise les membres de cet objet en leur affectant des fonctions anonymes. Elles sont donc définies localement à cet objet. C'est un bon moyen simple et efficace d'éviter les conflits entre scripts. Prenez un script développé à base de variables et de fonctions et encapsulez le tout dans un objet. Toutes les variables et les fonctions deviendront locales à cet objet. C'est un bon moyen à peu de frais d'éviter les conflits entre scripts. Une véritable conception objet de votre développement les rendra plus robustes,

plus évolutifs, plus maintenables. (Mais mieux vaut une bonne conception procédurale, qu'une mauvaise conception objet) la POO permet de définir des unités de programme plus autonomes et plus petites. Les membres (variables) d'un objet sont purement locaux à cet objet. Il en va de même des méthodes. Il devient alors possible d'avoir deux membres dans deux objets qui portent le même nom. Et comme une méthode est un membre de l'objet, il en va ainsi pour les méthodes. Je n'entrerai pas plus ici sur les avantages et les inconvénients de la POO.

II - Utiliser les objets du document HTML comme support

2.1 - Généralités :

Reste dans tout ça un petit problème qui s'est introduit sans qu'on s'en rende compte. Lorsqu'on écrivait notre code dans l'attribut évènement d'un tag HTML celui-ci lui était intimement lié. **<a href="http://localhost/" onClick="alert(this.getAttribute('href')); return false;">** Cette syntaxe définit la méthode onclick (attention sans majuscule) sur l'objet du ***DOM*** qui représente notre tag A. c'est bien une fonction locale à l'objet. Le this dans le corps de la fonction est donc bien l'Objet ***DOM*** lui-même. En passant à la programmation procédurale, ou à objet, nous avons perdu ce lien. Les objets, les variables, les fonctions, que nous définissons ne sont plus attachés à un élément du ***DOM***. Or ce sont ces derniers qui sont les seuls à même de fournir une interaction avec l'utilisateur. Dans nos scripts ainsi définis il va être nécessaire sans cesse de rétablir ce lien. Lorsque par exemple nous définissons un menu déroulant nous avons d'un côté les éléments du ***DOM*** qui permettent l'interaction et de l'autre le code qui permet d'assurer l'effet souhaité. À chaque fois que l'utilisateur déclenche un évènement, nous appelons une fonction ou une méthode qui doit retrouver l'élément du ***DOM*** sur lequel agir. Pour pallier cette difficulté il existe de nombreuses librairies qui permettent d'écrire directement en JavaScript l'ensemble de l'interface utilisateur. La méthode ne consiste plus alors à écrire une page HTML et lui adjoindre des scripts, mais à développer le tout en JavaScript. Ces librairies ont de nombreux avantages lorsqu'on veut développer une interface utilisateur complexe. Elles masquent complètement les notions HTML sous-jacentes. Mais elles sont particulièrement lourdes et complexes lorsque le besoin est plus petit. Reste une autre voie rarement exploitée qui pourtant est particulièrement efficace. Cette voie consiste à utiliser directement les objets du ***DOM*** pour porter les fonctionnalités dont ils ont besoin pour obtenir l'effet souhaité. Un petit exemple : Je veux vérifier qu'un champ de formulaire est de type numérique lorsqu'on le change. (Imaginez une vérification complexe genre c'est un numérique, qui est compris entre 100 et 100 000 et qui est un nombre premier) pour cela sur le **onChange** du champ nous allons appeler une fonction ou une méthode qui prend pour paramètre la valeur du champ affiche une alerte lorsque la valeur est incorrecte et remets le focus sur le champ. Maintenant si j'ai un autre champ qui a le même type de vérification mais avec quelques paramètres différents il faudra que je définisse une autre fonction ou que je passe d'autres paramètres. Utiliser l'élément input du ***DOM*** comme support va permettre de définir une fonction pour chaque input qui lui sera propre mais aussi de faire porter les paramètres à l'input lui-même le **onChange** se réduira à un code genre **this.vérify()**. La méthode n'a pas besoin de paramètre puisqu'elle est attachée à l'objet concerné qui lui est porteur de tout ce dont elle à besoin. Avec cette approche, le TAG représentant un onglet dans le document portera les méthodes nécessaires pour monter ou masquer son contenu. De même le TAG d'un menu déroulant portera les méthodes d'affichage de ses items. Un formulaire portera sa propre méthode de vérification faisant appel aux méthodes individuelles de ses champs. Il n'est alors plus nécessaire de rechercher les éléments concernés. Tout ce que nous faisions avec des fonctions peut être intégré directement dans le ***DOM***. Quels avantages ? Pour le comprendre il suffit de prendre un petit exemple. Je décide de faire un menu déroulant à base de UL LI j'ai deux niveaux

```
<ul>
   <li>menu1
      <ul>
         <li>sous-menu1-1</li>
         <li>sous-menu1-2</li>
      </ul>
   </li>
   <li>menu2
      <ul>
         <li>sous-menu2-1</li>
         <li>sous-menu2-2</li>
         <li>sous-menu2-3</li>
      </ul>
   </li>
   <li>Menu3
      <ul>
         <li>sous-menu3-1</li>
      </ul>
   </li>
```

```
</ul>
```

Lorsque je survole un LI du premier niveau, j'affiche le TAG UL qu'il contient. Mais il me faut effacer celui qui est déjà affiché. Pour cela il me faut le connaître ou alors les effacer tous. Pour le connaître il me faut une variable qui garde la référence au LI concerné. Donc lorsque je passe sur un LI de premier niveau, je dois si j'en ai un, effacer le LI déjà ouvert et afficher celui que je survole. Maintenant je veux mettre deux de ces menus dans mon code. Cela peut paraître inutile, mais c'est bien ce qu'il se passe lorsqu'on fait un menu multi niveau, j'ai bien au niveau trois plusieurs menus qui sont indépendants. J'ai donc besoin de connaître pour chaque niveau, quel est celui qui est actif pour fermer tout le nécessaire. Ce n'est donc plus une variable locale dont j'ai besoin mais d'une structure plus complexe. Or cette structure est déjà disponible c'est la hiérarchie de UL LI, elle-même. Sur chaque UL on garde le LI qui est actif et chaque LI affiche ou masque son sous-menu, je n'ai nullement besoin de garder dans une structure tous ces éléments. Je peux même du coup multiplier les menus dans mon interface sans risque de conflit. Cela simplifie grandement le code. Reste à pouvoir attacher des membres et des méthodes aux éléments HTML du ***DOM***. Pour les membres, ce n'est pas très compliqué. Il suffit de mettre un attribut dans son code HTML pour qu'il devienne un membre de l'objet ***DOM*** correspondant. C'est un bon moyen d'insérer des paramètres. Souvenez-vous de la vérification complexe. **<input type="text" name="test" onChange="this.verify();" minValue="100" maxValue="100000" value="0">** la méthode **verify** aura directement accès à **minValue** et **maxValue** si les paramètre change il ne sera pas nécessaire de modifier le code JavaScript. Ajouter une méthode est moins évident. Et c'est tout l'objet de cet article que de proposer une méthodologie pour y parvenir. Pour cela comme dans beaucoup de scripts il nous faudra une phase de préparation. Donc une fonction ou une méthode pour y parvenir. Pour éviter de retomber dans les mêmes difficultés qu'avec la programmation procédurale, je propose d'attacher systématiquement les fonctions d'un type de composant à un objet unique. Et pour éviter que celui-ci entre en conflit avec des éventuelles variables globales introduites par d'autres script, d'attacher le tout au document. Pour définir des menus déroulants je définirais l'objet **document.menuComponent** qui enfermera de façon hermétique tout le nécessaire pour les menus déroulants de même pour les onglets avec **document.tabComponent** chaque script d'un type de composant commencera par cette définition. Si j'ai deux scripts qui définissent le même nom d'objet il ne sera pas compliqué de trouver et lever l'ambiguïté. Charger le script, donc la définition d'un type de composant, c'est bien mais ce n'est pas suffisant pour transformer des éléments du ***DOM*** encore faut-il exécuter les méthodes concernées. Nous avons vu que les scripts était systématiquement placés dans **document.scripts** et qu'il est préférable de ne pas les éparpiller dans le document HTML. Il est nécessaire donc pour invoquer nos méthodes de passer par l'évènement **onLoad** du corps du document pour être sûr que les éléments du ***DOM*** sur lesquels on veut agir existent. La préparation se fera donc en deux étapes. Chargement du composant, instanciation des éléments dans le ***DOM***.

2.2 - Un exemple simple pour commencer : la gestion d'onglets.

Un onglet n'a de sens que s'il fait partie d'un groupe. Un onglet est un bouton permettant d'afficher ou non un contenu. Nous allons l'implémenter en prenant des DIV pour les contenus et une liste UL LI pour les boutons correspondants et un DIV représentant le groupe lui-même. Tout d'abord penchons nous sur ce que nous voulons obtenir.

```
<div class="tabGroup" id="1">
   <ul class="tabButtons">
      <li class="tabButton">Onglet1</li>
      <li class="tabButton">Onglet2</li>
      <li class="tabButton">Onglet3</li>
   </ul>
   <div class="tab" id="tab1"></div>
   <div class="tab" id="tab2"></div>
   <div class="tab" id="tab3"></div>
</div>
```

Je n'aime pas trop avoir à écrire des quantités de lignes là où juste une ou deux suffisent. Pour définir mon groupe d'onglet les quelques lignes suivantes sont suffisantes

```
<div>
```

```
        <div label="Onglet1"></div>
    <div label="Onglet2"></div>
    <div label="Onglet3"></div>
</div>
```

Nous allons donc définir un composant qui transforme cette définition pour obtenir ce que nous avons vu au-dessus et qui ajoute au passage les méthodes pour rendre cela dynamique. Première étape : définir le composant. Nous avons besoin d'une méthode d'initialisation. **transmute** me parait une bonne candidate. Elle va transmuter mes DIV HTML en groupe d'onglets. Nous allons aussi avoir besoin d'un peu de css. Je prends toujours le parti de mettre en externe mes feuilles de styles. Mais, dans mon composant j'ai besoin de connaitre les noms des classes css.

```
document.tabComponent = {
    defaultStyle: {
        tabGroup:       'tabGroup',
        tabButtonGroup: 'tabButtons',
        tabButton:      'tabButton',
        tabContent:     'tab'
    },
    transmute: function(divId) {
        ...;
    }
}
```

La première chose que doit faire notre méthode transmute c'est de récupérer l'élément ***DOM*** concerné.

```
var tabgroup = document.getElementById(divId);
```

Puis nous allons lui fixer son style et créer la liste

```
tabgroup.className = this.defaultStyle.tabGroup;
var ul = document.createElement('ul');
ul.className= this.defaultStyle.tabButtonGroup;
```

Et enfin l'insérer avant les DIV de contenus.

```
tabgroup.insertBefore(ul, tabgroup.firstChild);
```

À ce stade, nous avons simplement reconstruit dynamiquement le HTML correspondant. Il nous faut maintenant construire un onglet pour chaque DIV de contenu. Pour cela une petite fonction serait la bienvenue car nous allons l'exécuter pour chacun des contenus. Nous pouvons la définir globale avec tous les problèmes que cela implique. Nous pouvons aussi l'attacher à l'objet tabComponent, mais en fait cette fonction n'agit que sur le UL c'est donc sur ce dernier que nous allons l'ajouter.

Créons notre onglet. Il nous faut créer un LI et un texte pour son contenu. Et placer le label dans le LI

```
    ul.createTab = function (div) { //creation d'un onglet
    var li = document.createElement('li');
    var label = document.createTextNode(div.getAttribute('label'));
    li.appendChild(label);
```

Ensuite nous relions le DIV au LI en ajoutant un membre **contentDiv** au LI nous ajoutons la classe css au DIV et nous le masquons.

```
    li.contentDiv = div;
    li.contentDiv.className = document.tabComponent.defaultStyle.tabContent;
    li.contentDiv.style.display = 'none';
```

Et nous ajoutons le LI à la liste.

```
this.appendChild(li);
```

Il nous reste à ajouter les méthodes sur le LI et appeler cette fonction sur tous les DIV de contenus.

```
li.onclick = function() {
   // On masque l'onglet courant
   this.parentNode.currentTab.className = null;
   this.parentNode.currentTab.contentDiv.style.display = 'none';
   // On affiche l'onglet sélectionné
   this.className = 'current';
   this.contentDiv.style.display = 'block';
   // L'onglet sélectionné devient l'onglet courant
   this.parentNode.currentTab=this;
   return false;
} // fin de onclick
```

Notez ici que this est le LI sur lequel a cliqué l'utilisateur. Il nous faut déterminer si l'onglet que l'on crée est actif ou pas par défaut nous avons déjà mis **display** à **'none'**

```
if (!this.currentTab) { //on fixe le premier onglet actif
   li.className = 'current';
   li.contentDiv.style.display = 'block';
   this.currentTab=li;
}
```

Nous avons une méthode pour créer un onglet. Il ne nous reste qu'à l'appliquer.

```
for (var i=0; i<tabgroup.childNodes.length; i++) {
   //transforme tous les div fils en onglet
   if ((tabgroup.childNodes[i].nodeType == 1)&&
       (tabgroup.childNodes[i].tagName.toLowerCase() == 'div')) {
      ul.createTab(tabgroup.childNodes[i]);
   }
}
```

Voilà, nous avons notre composant. Pour pouvoir utiliser ce composant il nous faut le charger dans un document et appeler la méthode transmute sur le **onLoad**. Je suis un peu flegmatique et en plus si j'ai de nombreux composants dans mon document. L'écriture du **OnLoad** va rapidement devenir lourde. Je préfère donc une fois pour toutes, écrire un chargeur. Une méthode **addToStart** qui ajoute un élément à exécuter lors du chargement et une fonction activée sur le **onLoad** qui va les exécuter tous.

```
document.coreComponent = {
   listStart: new Array(),
   load: function() {
      var ls = this.listStart;
      if (ls) {
         for(var i=0; i<ls.length; i++){
            fnc = ls[i];
            if(typeof(fnc) == 'function'){
               fnc();
            } else {
               eval(fnc);
            }
         }
      }
   },
   // ajouter un élément dans onload
   addToStart: function(fnc){
```

```
        this.listStart.push(fnc);
    },
    handlers: {
        onLoad: function() {
            return document.coreComponent.load();
        }
    }
}
window.onload = document.coreComponent.handlers.onLoad;
```

Revenons sur notre composant Onglet. Il serait sympathique de ne pas avoir à appeler la méthode de transformation ainsi un développeur qui veut utiliser le composant, n'aurait qu'à insérer le script et écrire son code HTML. Il faut pour cela trouver un moyen de désigner les DIV à transformer. Je verrais bien quelque chose comme

```
        <div component="Tabs">
    <div label="Onglet1"></div>
    <div label="Onglet2"></div>
    <div label="Onglet3"></div>
</div>
```

Il faut donc maintenant que notre composant retrouve les éléments à transformer de façon automatique.

```
    init: function() {
        var list = document.getElementsByTagName('div');
        for (var i=0; i<list.length; i++) {
            var type = list[i].getAttribute('component');
            if (type == 'Tabs') {
                document.tabComponent.transmute(list[i]);
            }
        }
    },
```

Et une insertion automatique à la liste des chargements au démarrage. La méthode transmute prend alors en paramètre le DIV et non son id

Le composant est donc :

```
document.tabComponent = {
    defaultStyle: {
        tabGroup:       'tabGroup',
        tabButtonGroup: 'tabButtons',
        tabButton:      'tabButton',
        tabContent:     'tab'
    },
    init: function() {
        var list = document.getElementsByTagName('div');
        for (var i=0; i<list.length; i++) {
            var type = list[i].getAttribute('component');
            if (type == 'Tabs') {
                document.tabComponent.transmute(list[i]);
            }
        }
    },
    transmute: function(tabgroup) {
        if (tabgroup) {
            tabgroup.className = this.defaultStyle.tabGroup;
            var ul = document.createElement('ul');
            ul.className= this.defaultStyle.tabButtonGroup;
            tabgroup.insertBefore(ul, tabgroup.firstChild);
            ul.createTab = function(div) { //création d'un onglet
                var li = document.createElement('li');
```

```
            var label = document.createTextNode(div.getAttribute('label'));
            li.appendChild(label);
            li.contentDiv = div;
            li.contentDiv.className = document.tabComponent.defaultStyle.tabContent;
            li.contentDiv.style.display = 'none';
            this.appendChild(li);
            li.onclick = function() {
               // on masque l'onglet courant
               this.parentNode.currentTab.className = null;
               this.parentNode.currentTab.contentDiv.style.display = 'none';
               // on affiche l'onglet sélectionné
               this.className = 'current';
               this.contentDiv.style.display = 'block';
               // l'onglet sélectionné devient l'onglet courant
               this.parentNode.currentTab=this;
               return false;
            } // fin de onclick
            if (!this.currentTab) { //on fixe le premier onglet actif
               li.className = 'current';
               li.contentDiv.style.display = 'block';
               this.currentTab=li;
            }
         }
         for (var i=0; i<tabgroup.childNodes.length; i++) {
            //transforme tout les div fils en onglet
            if ((tabgroup.childNodes[i].nodeType == 1)&&
                (tabgroup.childNodes[i].tagName.toLowerCase() == 'div')) {
               ul.createTab(tabgroup.childNodes[i]);
            }
         }
      }
   }
}
document.coreComponent.addToStart(document.tabComponent.init);
```

Soit donc ces deux scripts dans les fichiers **corecomponent.js** et **tabcomponent.js** voyons comment les utiliser dans un document HTML

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html>
   <head>
      <title>DOM Tabs</title>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
      <link rel="stylesheet" href="tabcomponent.css" type="text/css"/>
      <script src="corecomponent.js" type="text/javascript"></script>
      <script src="tabcomponent.js" type="text/javascript"></script>
   </head>
   <body>
      <div component="Tabs">
         <div label="Onglet1">Contenu du tab 1</div>
         <div label="Onglet2">Le tab 2</div>
         <div label="Onglet3">Et le contenu du troisième</div>
      </div>
   </body>
</html>
```

Si la définition du composant a demandé un peu de réflexion, son code est relativement simple (juste 60 lignes) et son usage est particulièrement aisé. Vous pouvez essayer d'imbriquer les tab dans votre HTML les multiplier, le fait d'avoir attaché le code à l'élément concerné offre une large souplesse. Il peut être intéressant par exemple d'utiliser des styles différents sur les groupes qui se trouvent dans la page. Le composant utilise des classes par défaut pourquoi ne pas ajouter en paramètre dans le HTML le nom de classe à utiliser. Ou un attribut **tabsPosition top, bottom** qui place les onglets avant ou après les **div**. Les possibilités sont multiples.

Certains auront sûrement remarqué quelques appels inattendus dans une programmation à objet. Pourquoi par exemple la méthode **init** de **document.tabComponent** fait-elle appel à **document.tabComponent.transmute** et non **this.transmute**. De même pourquoi la présence de **handlers.onLoad** dans **document.coreComponent**. La raison est la façon don JavaScript gère les évènements.

```
monObjet = {
   func: function() {
      this.a = 45;
   }
}
```

Lorsque j'appelle cette méthode de la façon suivante, **monObjet.func()** la valeur de **monObjet.a** devient **45**. Si l'on passe cette fonction dans un évènement **windows.onload=monObjet.func;** lors de l'exécution **this** est inconnu. En fait la méthode **func** est # détachée # de son objet et exécutée hors du contexte de l'objet. Cela vient du fait que l'on passe la référence au membre **func** de **monObjet** à **windows.onload**. Je vous conseille de bien faire attention à ce genre de situation. Dans les deux composants, j'ai donné deux façons de faire qui permettent de lever la difficulté. La première consiste à créer des **handlers** qui n'ont d'autre but que l'appel sur l'objet. L'autre consiste à remplacer **this** par l'objet lui-même. Les **handlers** ont l'avantage de tout regrouper en un seul endroit dans le composant, mais ils induisent un appel de fonction de plus. Il est plus facile de renommer le composant avec cette méthode. La deuxième est plus simple à écrire.

2.3 - Multiplications des composants :

Un autre petit exemple : pour montrer comment avoir plusieurs composants et éviter les interactions, un bloc avec un ascenseur. Dans un document, je veux pouvoir définir une zone qui possèdera son propre ascenseur si son contenu dépasse les dimensions fixées.

```
<div component="scrollBox" scrollBoxWidth="320px" scrollBoxHeight="125px">
   <p>Lorem ipsum dolor sit amet consectetuer sem sit nascetur ligula tellus.
      Elit eget neque non nunc elit porttitor enim faucibus pede pellentesque.
      Id adipiscing ligula semper interdum mus Donec ac nisl lacus Quisque.
      Nunc vitae tortor Phasellus wisi habitasse tempus pharetra nunc eget
      lobortis. Quam mauris ipsum et nibh ipsum leo pede porttitor nec.</p>
   <p>Penatibus accumsan lorem velit ornare libero ipsum ut Sed enim fermentum.
      Curabitur nulla vel Morbi Aliquam at risus Nam volutpat turpis adipiscing.
      Ante habitasse sagittis In aliquam pretium a Donec fames id facilisi. Leo
      Curabitur laoreet enim dui sapien Curabitur nibh et lacinia faucibus. Dui
      fringilla Sed hendrerit et Curabitur ipsum amet id dui vel. </p>
   <p>Senectus Aenean vitae sociis ante id pede feugiat libero wisi wisi. Ligula
      lobortis congue et tempor urna Nulla Donec Curabitur platea convallis.
      Pretium nascetur tincidunt eu lacinia laoreet ante malesuada amet Sed
      Pellentesque. Fringilla tincidunt interdum Integer justo sed et tellus ac
      dis Suspendisse. Semper molestie leo nunc Vestibulum a ac vitae vel ac
      quis. Consectetuer rutrum laoreet.</p>
   <p>Et wisi amet ante dis urna elit Vestibulum id semper ridiculus. Tellus
      tellus tristique eu justo congue quis Nunc massa porta urna. Curabitur
      porttitor sit elit risus vel Integer Vestibulum velit at turpis. Malesuada
      consequat et Mauris adipiscing lacinia odio Cras sem Ut scelerisque. Dui
      massa Pellentesque wisi consectetuer In.</p>
</div>
```

Je veux afficher ce DIV dans un rectangle de 320x125 pixels. Et si le contenu est trop important faire apparaître un ascenseur.

Nous allons utiliser le même principe

```
// ScrollBox component
document.scrollBoxComponent = {
```

```
    init: function() {
       var list = document.getElementsByTagName('div');
       for (var t=0; t<list.length; t++) {
          var type = list[t].getAttribute('component');
          if (type == 'scrollBox') {
             document.scrollBoxComponent.transmute(list[t]);
          }
       }
    },
    transmute: function(obj) {
       // on récupère les paramètres
       var viewWidth  = parseInt(obj.getAttribute('scrollBoxWidth'))  + 18 + 'px';
       var viewHeight = parseInt(obj.getAttribute('scrollBoxHeight')) + 18 + 'px';
       // On crée un div pour le contenu
       var cntView = document.createElement('div');
       cntView.style.width    = obj.style.width;
       cntView.style.height   = obj.style.height;
       cntView.style.padding  = obj.style.padding;
       // on y place tout le contenu
       for (var i=0; i<obj.childNodes.length; i) {
          cntView.appendChild(obj.childNodes[i]);
       }
       // et on le place dans le div inition.
       obj.appendChild(cntView);
       // on fixe la taille et l'overflow
       obj.style.width = viewWidth;
       obj.style.height = viewHeight;
       obj.style.padding = 0;
       obj.style.overflow ='auto';
    }
}
document.coreComponent.addToStart(document.scrollBoxComponent.init);
```

Voilà un nouveau composant. Qui volontairement agit sur un DIV lui aussi. **scrollBoxComponent.js**

Notez que j'ai aussi appelé les fonctions de la même façon. Ajoutons ce script à notre HTML. On pourrait ajouter un bloc pour montrer le fonctionnement, mais on va utiliser directement sur le premier contenu du groupe d'onglet. Nous avons là un DIV qui est à la fois un contenu d'un onglet et l'objet d'un scroll box.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN">
<html>
   <head>
      <title>DOM Tabs</title>
      <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
      <link rel="stylesheet" href="tabcomponent.css" type="text/css"/>
      <script src="corecomponent.js" type="text/javascript"></script>
      <script src="tabcomponent.js" type="text/javascript"></script>
   </head>
   <body>
      <div component="Tabs">
         <div label="Onglet1"  component="scrollBox" scrollBoxWidth="320px"
              scrollBoxHeight="125px">
            <p>Lorem ipsum dolor sit amet consectetuer sem sit nascetur ligula
               tellus. Elit eget neque non nunc elit porttitor enim faucibus
               pede pellentesque. Id adipiscing ligula semper interdum mus Donec
               ac nisl lacus Quisque. Nunc vitae tortor Phasellus wisi habitasse
               tempus pharetra nunc eget lobortis. Quam mauris ipsum et nibh
               ipsum leo pede porttitor nec.</p>
            <p>Penatibus accumsan lorem velit ornare libero ipsum ut Sed enim
               fermentum. Curabitur nulla vel Morbi Aliquam at risus Nam
               volutpat turpis adipiscing. Ante habitasse sagittis In aliquam
               pretium a Donec fames id facilisi. Leo Curabitur laoreet enim dui
               sapien Curabitur nibh et lacinia faucibus. Dui fringilla Sed
               hendrerit et Curabitur ipsum amet id dui vel. </p>
            <p>Senectus Aenean vitae sociis ante id pede feugiat libero wisi
```

```
                wisi. Ligula lobortis congue et tempor urna Nulla Donec Curabitur
                platea convallis. Pretium nascetur tincidunt eu lacinia laoreet
                ante malesuada amet Sed Pellentesque. Fringilla tincidunt
                interdum Integer justo sed et tellus ac dis Suspendisse. Semper
                molestie leo nunc Vestibulum a ac vitae vel ac quis. Consectetuer
                rutrum laoreet.</p>
            <p>Et wisi amet ante dis urna elit Vestibulum id semper ridiculus.
            Tellus tellus tristique eu justo congue quis Nunc massa porta urna.
            Curabitur porttitor sit elit risus vel Integer Vestibulum velit at
            turpis. Malesuada consequat et Mauris adipiscing lacinia odio Cras
            sem Ut scelerisque. Dui massa Pellentesque wisi consectetuer In.</p>
         </div>
         <div label="Onglet2">Le tab 2</div>
         <div label="Onglet3">Et le contenu du troisième</div>
      </div>
   </body>
</html>
```

Nous avons manipulé deux fois le DIV Onglet1 si ces manipulations avaient ajouté des méthodes et des évènements à cet élément, il aurait fallu se montrer particulièrement vigilant. C'est là que se trouvent le plus de risques. Si la première transformation ajoute une méthode, il ne faut pas que la deuxième ajoute une méthode de même nom. De même avec les attributs. La méthode montre là ses limites.

III - Conclusion :

Cet article visait à démontrer le principe de développement d'interface utilisateur enrichie en JavaScript par manipulation directe du ***DOM***. En faisant porter directement aux éléments du ***DOM*** les attributs et les méthodes, nous obtenons une grande souplesse et indépendances des composants d'un document HTML. La mise en oeuvre par un néophyte est grandement simplifiée, puisqu'il n'est alors pas nécessaire, une fois les composants disponibles de connaître JavaScript. Les exemples donnés ici sont grandement améliorables. Entre autres au niveau conformité XHTML il serait bon pour rester conforme de placer les attributs supplémentaires que l'on a introduits dans le code HTML dans un ***namespace***. Par exemple **xmlns:cpnt="org.ecmascript.dom.cpnt"** il ne reste alors qu'à préfixer les attributs par **cpnt:scrollBoxWidth** et à introduire la récupération du ***namespace*** dans **coreComponent**. J'espère que cette petite intrusion dans le ***DOM*** vous aura ouvert des portes.