Concepteur Développeur en Informatique

Développer des composants d'interface

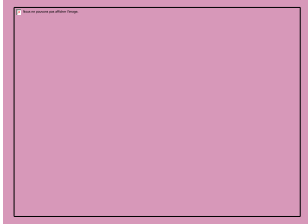
Initiation API DOM

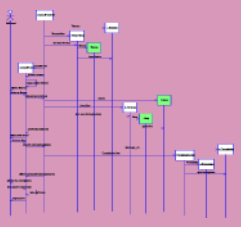
Accueil

Apprentissage

PAE

Evaluation





Localisation : U03-E05-S01

1. Introduction

Le DOM (Document Object Model) n'est pas à proprement parlé un modèle mais l'interface de programmation (API) qui permet d'accéder à l'ensemble des éléments d'un document HTML et de manipuler ces derniers. L'interface DOM est aussi disponible au niveau de XML.

Chaque langage implémente cette API à sa manière. Nous retrouvons cette interface de programmation implémentée sous Javascript, C#, PHP, Java, Python, ….

Il sera possible à partir de ce modèle et du langage Javascript de changer le contenu d'un élément, le style d'un élément ou d'un groupe d'éléments ou même de modifier la structure du document en ajoutant ou supprimant des éléments.

Dans le DOM, les documents ont une structure logique qui ressemble à un arbre, ou plus précisément à un ensemble d'arbres qui contiendraient chacun différentes ramifications.

Chaque élément du DOM est un nœud.

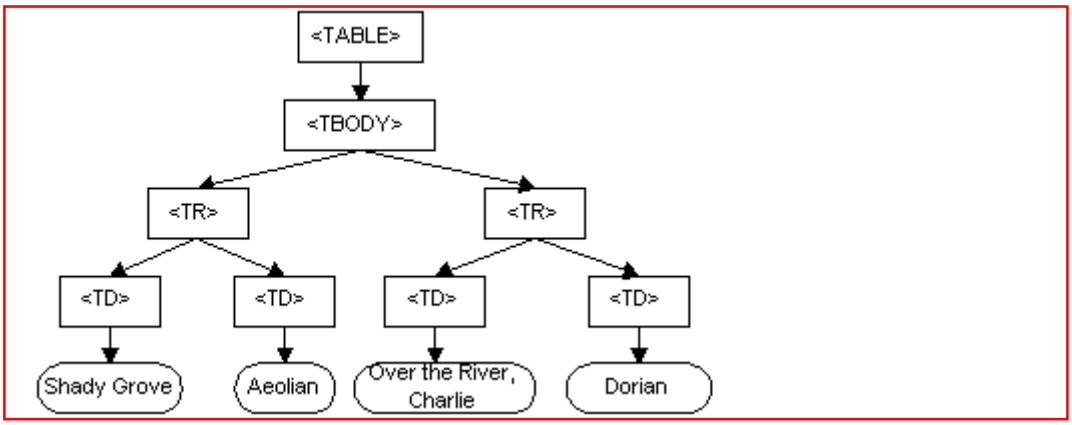
Tout document contient un nœud racine.

Un élément nœud peut être en relation avec des nœuds enfants, des nœuds frères, un nœud parent.

Dans l'exemple suivant :

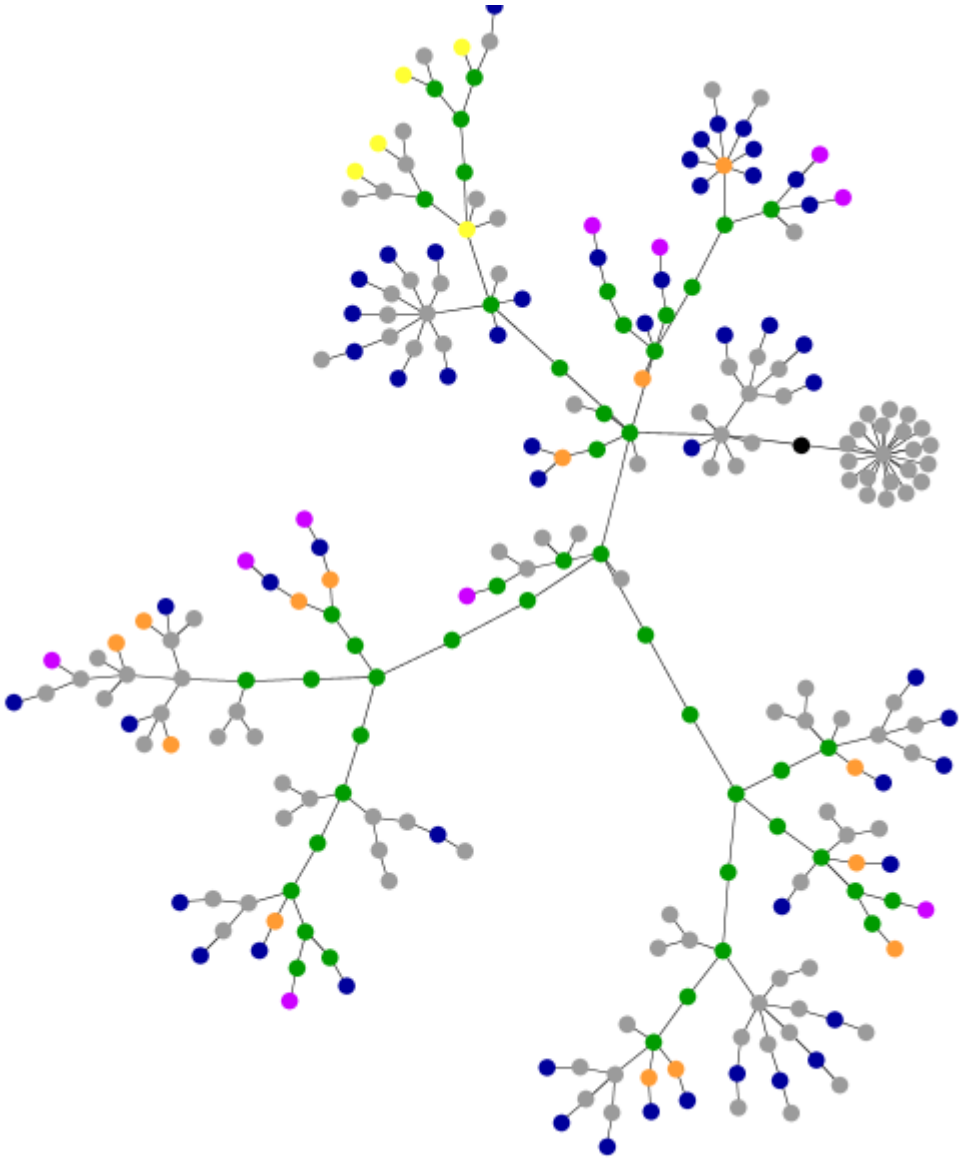
```
<table>
        <rows>
      <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
      </tr>
      <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
      </tr>
      </rows>
</table>
```

La représentation sous forme d'arbres du DOM serait :



Nous manipulons ces nœuds en recourant au langage JavaScript. En fait, par le biais du DOM et de JavaScript nous allons pouvoir modifier tous nos éléments HTML et styles sans nouvel appel au serveur Web : tout se passe côté client

Illustration graphique de l'arbre du DOM de la page afpa.fr. Ce site n'existe plus, dommage…



blue: for links (the A tag)
red: for tables (TABLE, TR and TD tags)
green: for the DIV tag
violet: for images (the IMG tag)
yellow: for forms (FORM, INPUT, TEXTAREA, SELECT and OPTION tags)
orange: for linebreaks and blockquotes (BR, P, and BLOCKQUOTE tags)
black: the HTML tag, the root node
gray: all other tags

2. Les méthodes d'accès aux nœuds du DOM

Le DOM est donc présenté sous un ensemble de nœuds à la manière d'un arbre et n'est pas sans nous rappeler un contrôle, déjà vu dans l'environnement Windows, le treeView.

Il existe un nœud racine à partir duquel nous pourrons accéder à l'ensemble des autres nœuds représenté par l'objet ou node (nœud) **document**..

Pour accéder à un élément de notre document HTML nous utiliserons la méthode de l'objet **document getElementbyId()** .

```
elementCourant = document.getElementById(elementId)
```

S'il n'existe aucun élément porteur de la propriété id=elementID, alors la méthode renverra null.
Il existe aussi des méthodes qui renvoient un tableau de nœuds en faisant référence au nom de la balise (tagName) ou au nom de l'élément (Name).

```
var tNodes = document.getElementsByTagName(NomBalise);
var tNodes = document.getElementsByName(NomElement);
```

Nous pouvons nous déplacer dans l'arbre à partir d'un point de référence à l'aide des méthodes décrites dans le chapitre 2.2.

2.1. Identifier les nœuds enfants

Si l'élément considéré possède des nœuds enfants :

```
if (elementCourant.hasChildNodes() === true)
```

childNodes représente le tableau des enfants. Nous pourrions accéder à chacun de ces éléments pour afficher leurs propriétés ainsi :

```
for (i = 0; i < elementCourant.childNodes.length; i++)
      {  afficherProprietes(elementCourant.childNodes[i]); }
```

2.2. Autres méthodes pour parcourir les nœuds

Obtenir le parent ::	elementCourant.**parentNode**
Obtenir le nœud frère suivant :	elementCourant.**nextSibling**
Obtenir le nœud frère précédent :	elementCourant.**previousSibling**
Obtenir le premier enfant :	elementCourant.**firstChild**
Obtenir le dernier enfant :	elementCourant.**lastChild**

2.3. Les propriétés

Les propriétés considérées restent accessibles depuis tout navigateur quelle que soit sa famille. Parmi les plus importantes permettant de comprendre le modèle de nœud, citons celles-ci :

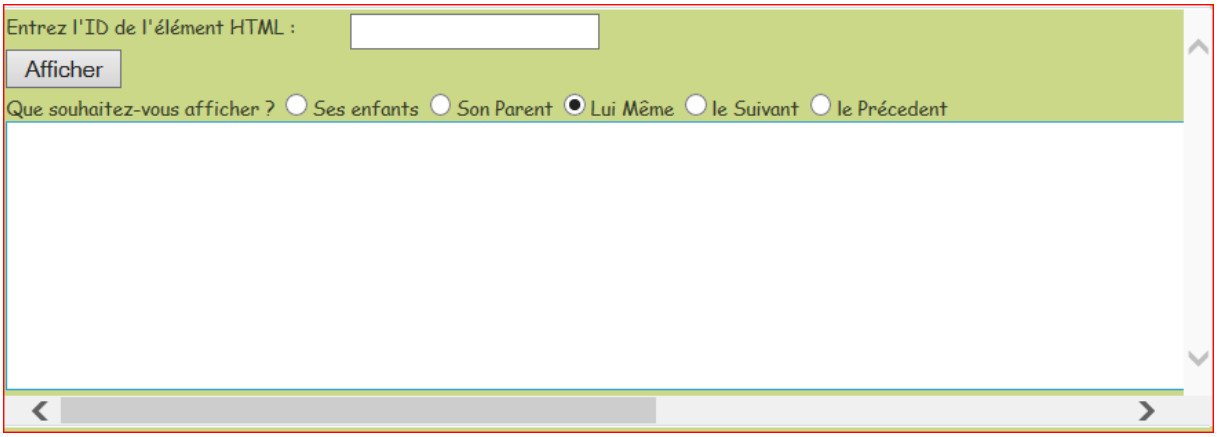
- **nodeType** : Un numéro présentant le type de nœud (élément, texte, document, attribut,
- **nodeValue** : Valeur ou contenu du nœud. Pour un nœud de type élément, null. Pour un nœud de type texte, le texte, pour les nœuds attribut, la valeur de l'attribut (value).
- **innerHTML** : Contenu entre la balise ouvrante et la balise fermante pour un nœud de type Elément, indéfini pour les autres types de nœud.
- **attributs** : Tableau des attributs lorsque le nœud est de type élément.

Pour mieux comprendre, regardons l'exemple suivant portant sur une page HTML composée d'un paragraphe et d'un formulaire sur lequel ont été déposés une balise de regroupement de champs, un label et une boite de texte.

Une méthode permet d'extraire les propriétés de l'élément dont la référence est transmise en argument :

```
// Fonction permettant d'afficher les propriétés de l'élément considéré
function afficherProprietes(choix, element) {
    var description = "";
    description = choix + "\n";
    description += "Propriété nodeType : " + element.nodeType + "\n";
    description += "Propriété id : " + element.id + "\n";
    description += "Propriété nodeName : " + element.nodeName + "\n";
    description += "Propriété nodeValue : " + element.nodeValue + "\n";
    description += "Propriété value : " + element.value + "\n";
    description += "Propriété innerHTML : " + element.innerHTML + "\n";
    description += "Propriété outerHTML : " + element.outerHTML + "\n";
    document.forms['frmAfficher'].elements['txtProprietes'].value += description;

}
```

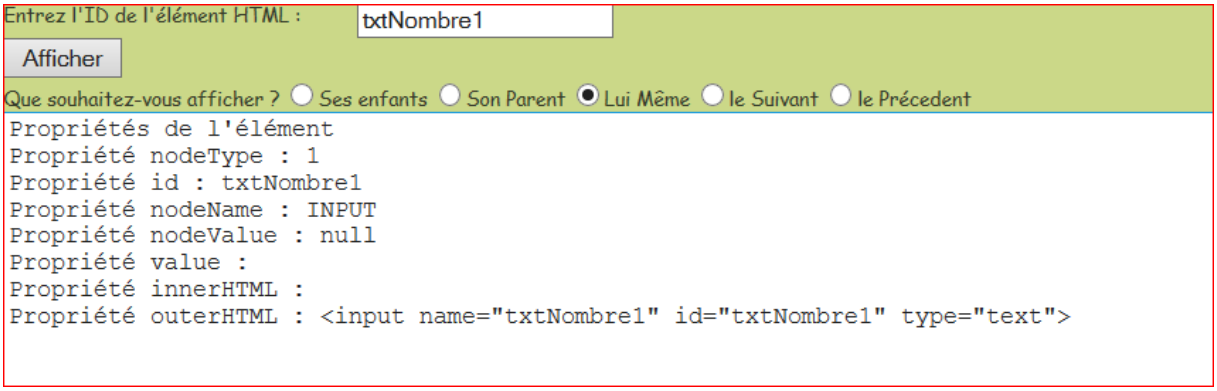
Un formulaire permet de choisir le ou les éléments dont nous souhaitons connaître les propriétés.



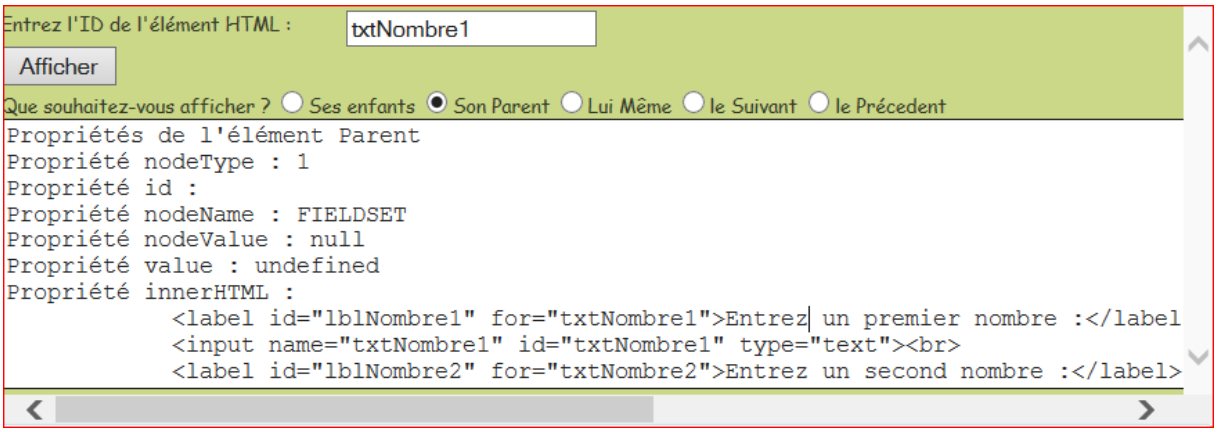
Extrait du choix de l'élément à transmettre en argument de la méthode :

```
if (elementId.toString().length != 0 && document.getElementById(elementId) != null)
 {
    elementCourant = document.getElementById(elementId);

    for (var i = 0; i < choix.length; i++)
        {
        if (choix[i].checked == true)
        {
            choixA = choix[i].value;
            break;
        }
    }

    switch (choixA) {
        case 'L': // lui même
            afficherProprietes("Propriétés de l'élément", elementCourant);
            break;
        case 'P': // Parent
            afficherProprietes("Propriétés de l'élément Parent", elementCourant.parentNode);
            break;
        case 'E': // Ses enfants
            if (elementCourant.hasChildNodes() == true) {
                for (i = 0; i < elementCourant.childNodes.length; i++) {
                    afficherProprietes("Propriétés des enfants", elementCourant.childNodes[i]);
                }
            }
            break;
        case 'S': // Le suivant
            afficherProprietes("Propriétés de l'élément suivant", elementCourant.nextSibling);
```

Un exemple pour les propriétés d'un élément de type input :



Et son parent :



A noter : Lorsque l'on consulte la liste des attributs avec Internet Explorer, celle-ci comporte aussi les événements et les attributs non définis alors que FireFox ne retiendra que les attributs définis.

3. Les méthodes de création dynamique de nœuds

Après nous être familiarisé avec le modèle du DOM, voyons comment nous pouvons ajouter ou supprimer des éléments. Ces mécanismes permettent de créer des structures de pages HTML qui varient selon le contexte.

3.1. Création d'un nouvel élément

Pour créer un nouvel élément, nous avons recours à une méthode de l'objet document
`var element = document.createElement("p");.`

Nous passons à la méthode le nom du type d'élément, ici un paragraphe.

Nous assignons ensuite l'attribut class afin de lier notre élément à un style défini en css.
`element.setAttribute("class", "defaut");`

Puis nous affectons la propriété texte de l'élément.
`element.innerHTML = 'mon beau paragraphe';`

Nous devons ensuite introduire ce nœud dans l'arbre du document à une position choisie.

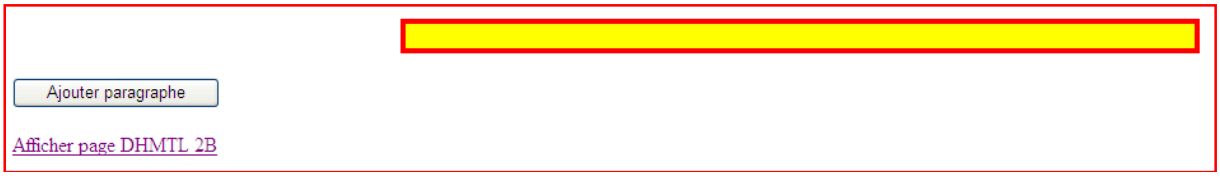
Il existe deux méthodes pour ajouter un élément :

- Comme dernier enfant de l'élément de référence, ici divTexte
  `document.getElementById('divTexte').appendChild(element);`
- Insertion avant un élément de référence
  `document.insertBefore(element, elementReference);`

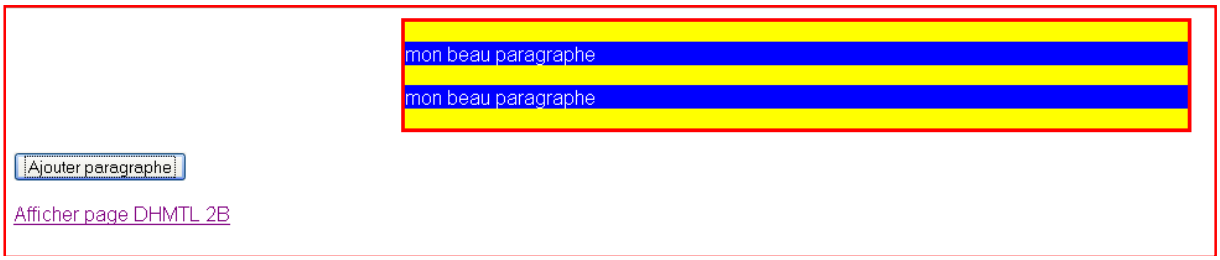
Exemple simple permettant d'ajouter des paragraphes à la demande:

```
    <script type="text/javascript">
<!--
        function ajouterElement() {
            var element = document.createElement("p");
            element.setAttribute("class", "defaut");
            element.innerHTML = 'mon beau paragraphe';
            // Ajout de l'élément après le dernier paragraphe de la division
            document.getElementById('divTexte').appendChild(element);
        }
        function supprimerElement() {
            var elementReference = document.getElementById('divTexte');
            elementReference.removeChild(elementReference.lastChild);
            elementReference.replaceChild(nouveauElement,ancienElement);
        }
//-->
    </script>
```

Affichage initial



Affichage après l'ajout de deux paragraphes



3.2. Suppression ou remplacement d'un élément

Nous ajoutons une méthode qui permet la suppression du dernier élément paragraphe ajouté.
Nous pouvons aussi remplacer un élément par un autre. Ajouter ici uniquement pour illustration.

```
function supprimerElement() {
    var elementReference = document.getElementById('divTexte');
    elementReference.removeChild(elementReference.lastChild);
    elementReference.replaceChild(nouveauElement,ancienElement);
}
```

3.3. Gestion de la persistance

Les éléments ajoutés dynamiquement ne sont pas par défaut persistants.

Il existe plusieurs méthodes pour permettre la persistance des objets ajoutés dynamiquement (ou pour la conservation de données de formulaire) fonction de la famille du navigateur et de la version de ce dernier.
Ces fonctionnalités sont référencées sous le terme de DOM Storage ou Stockage DOM en français. Elles sont prises en charge par les navigateurs de dernière génération.

Ces fonctionnalités sont d'une manière générale plus facile à mettre en œuvre que les cookies qui restent une alternative à la préservation de données côté client. Elles permettent surtout de stocker des volumes beaucoup plus importants.

Nous aborderons les cookies lors de l'apprentissage d'ASP Net.

Les espaces de stockage spécifique au DOM, **SessionStorage** et **LocaleStorage**, permettent de stocker un volume important de données (plusieurs Mo).
Elles sont mises en œuvre sous la forme de dictionnaires associant une clé de référence et une valeur. Ces volumes varient d'un explorateur à un autre.

Microsoft précise : Le stockage DOM offre également beaucoup plus d'espace disque que les cookies. Dans Internet Explorer, les cookies ne peuvent stocker que 4 kilo-octets (Ko) de données. Ce volume d'octets peut être une paire nom/valeur de 4 Ko, ou être constitué de 20 paires nom/valeur d'une taille totale de 4 Ko. En comparaison, le stockage DOM fournit environ 10 mégaoctets (Mo) pour chaque zone de stockage.

Fonctionnellement, les zones de stockage clientes sont assez différentes des cookies.

Le stockage DOM ne transmet pas de valeurs au serveur avec chaque demande, comme le font les cookies, et les données hébergées dans une zone de stockage local n'expirent jamais.

Il est en outre facile d'accéder à des fractions de données à l'aide d'une interface standard dont la prise en charge est croissante parmi des fournisseurs de navigateur, ce que ne permettent pas les cookies.

Vous trouverez une information complète sur ces mécanismes dans l'aide de Microsoft à l'adresse suivante : http://msdn.microsoft.com/fr-fr/library/cc197062%28VS.85%29.aspx

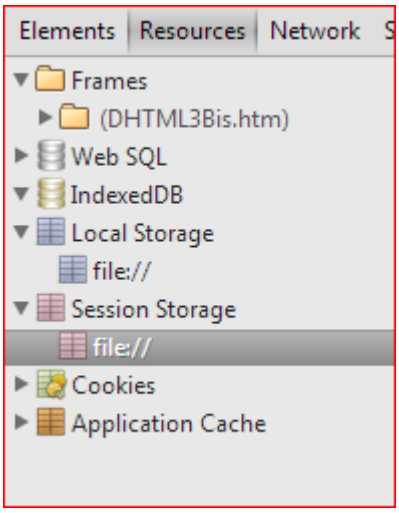
Deux objets de stockage associés à l'objet window existent :

- window.sessionStorage : Persistance durant une session
- window.localStorage : Persistance infinie

D'une manière générale, ces objets ne sont pas sécurisés (il n'existe pas, comme pour les cookies, de mécanismes de cryptage qui permettent d'obtenir des cookies sécurisés. Les chaînes conservées, car se sont toujours des chaines, sont accessibles en clair.

Les objets de stockage, qu'ils soient temporaires à la session ou permanents, implémentent les mêmes propriétés et méthodes.

Les outils du développeur permettent de visualiser ces espaces :



Sauvegarde de la valeur d'une division dans l'espace de la session :

Key	Value
divTexte	mon beau paragraphe mon beau paragraphe

Effacement des zones de stockage

Les utilisateurs peuvent effacer des zones de stockage à tout moment en sélectionnant Supprimer l'historique de navigation.
Cela supprime la session et les zones de stockage local de tous les domaines qui ne figurent pas dans le dossier Favoris et réinitialise les quotas de stockage dans le Registre.
Désactivez la case à cocher Conserver les données des sites Web favoris pour supprimer toutes les zones de stockage, quelle que soit la source.

Pour supprimer des paires clé/valeur dans une liste de stockage, effectuez une itération sur la collection et invoquez la méthode removeItem ou utilisez la méthode clear pour supprimer tous les éléments à la fois. N'oubliez pas que les modifications apportées à une zone de stockage local sont enregistrées de manière asynchrone sur le disque.

Seuls les scripts de pages d'une même origine (domaine) peuvent supprimer les données stockées à partir de ces pages.

Méthodes pour suppression

- clear() : toutes les paires clés/valeurs
- removeItem(Clé) : l'élément désigné par la clé

Méthodes pour stockage et récupération

- setItem(clé,valeur) : définit une paire clé/valeur
- getItem(clé) : récupère la valeur associée à la clé

Ajoutons à l'exemple précédent deux méthodes pour sauvegarder et charger les objets ajoutés dynamiquement.

Je sauvegarde ici l'ensemble du contenu présent entre la balise ouvrante et la balise fermante de la division de texte :

```
function sauvegarderSession() {

 if (window.sessionStorage) {
        window.sessionStorage.setItem('divTexte', document.getElementById('divTexte').innerHTML);
    }
}
function chargerSession() {

if (window.sessionStorage) {
        if (window.sessionStorage.getItem('divTexte') != null) {
            document.getElementById('divTexte').innerHTML = window.sessionStorage.getItem('divTexte');
        }
    }
}
```

Une fois ces méthodes implémentées nous pouvons naviguer d'une page à une autre au cours de la même session et retrouver les éléments paragraphes ajoutés.

Pour savoir si le navigateur implémente l'objet storage, utiliser une structure conditionnelle qui permet de l'existence de la référence objet. Cette technique est fréquemment utilisée en développement Web.

```
if (window.sessionStorage) { // OK }

Pour IE, ces mécanismes ne sont mis en œuvre que depuis IE 8.
```

Pour les navigateurs ne pouvant gérer ces techniques, il est possible d'ajouter des comportements (behaviors) pour stocker les données dans des fragments XML. Ce ne sont toutefois pas des techniques normalisées dans les CSS.

Au niveau de la feuille de style associée à l'élément à sauvegarder

```
  #divTexte
        {
            behavior: url(#default#userdata);
```

```
        }
<div id='divTexte'>
```

La sauvegarde peut être aussi effectuée dans l'espace persistant.

Pour indiquer une date limite de conservation des données (date expiration), il est nécessaire de renseigner la propriété **expires**. La fonction de sauvegarde des données amendée devient alors :

```
function sauvegarderPersistant() {

        var oPersistant = document.getElementById('divTexte');

       oPersistant.setAttribute("sDivTexte", oPersistant.innerHTML);
        var oTimeNow = new Date(); // Start Time
        oTimeNow.setMinutes(oTimeNow.getMinutes() + 1);
        var sExpirationDate = oTimeNow.toUTCString();
        oPersistant.expires = sExpirationDate;
       localStorage.setItem('divTexte', oPersistant);


}
function chargerPersistant() {
     var oPersistant = document.getElementById('divTexte');
     if (localStorage.getItem('divTexte') != null) {
           oPersistant.innerHTML = localStorage.getItem('divTexte');
           }
       }
```