Remarque préliminaire: ce fichier contient une erreur javascript volontaire ligne 927

JavaScript et les formulaires

- Permet de valider les formulaires côté client. Attention : ne dispense pas de la validation côté serveur, qui est indispensable (sécurité).
- On peut accéder aux formulaires et au champs par leurs **id** si on leur en donne.
- On peut accéder aux formulaires par leur **nom** (name); de même pour les champs.
- Principe: on lie des fonctions javascript à certains événements (on quitte un champ, on valide le formulaire, on presse une touche...)

Vérification à l'expédition

- On veut que le formulaire soit vérifié juste avant d'être expédié
- On utilise l'événement `onsubmit` de la balise form
- Quand le formulaire va être envoyé, on appelle le code javascript dans onsubmit
- Si la valeur de onsubmit a la forme `return nomDeFonction()`:
  - Si la fonction renvoie true, l'action est effectuée
  - Sinon, le formulaire n'est pas expédié

Premier exemple

On veut tester que le nom et le prénom fournis par l'utilisateur ne sont pas vides.

nom prénom Envoyer

```
<script type="text/javascript">
       function verifChampNonVideVersion0(id) {
       var result= true;
               field= document.getElementById(id);
               // On accède à la valeur du champ par la propriété javascript value,
               // et pas par l'attribut DOM value !!!
               var value= field.value;
               if (value == "") {
                       result= false;
               }
               return result;
       }

    function verifNomPrenomPasVides() {
               var r= verifChampNonVideVersion0('formEx0Nom');
               if (r == true)
               r= verifChampNonVideVersion0('formEx0Prenom');
               if (! r)
                       alert("erreur: champ non renseigné");
               return r;
       }
       </script>

<form action="javascript:pageSuivante()" onsubmit="return verifNomPrenomPasVides()">
       <p>nom <input name="nom" id="formEx0Nom" type="text">
       prénom <input name="prenom" id="formEx0Prenom" type="text">
       <input type="submit"></p>
</form>
```

Critique de la méthode précédente

- alert() : message d'erreur peu précis
- disparaît quand on veut corriger
- agressif (nouvelle fenêtre)

On utilisera plutôt la manipulation du DOM (voir ci-dessous) pour écrire les messages d'erreur directement sur la page...

nom

prénom

Envoyer

```
       <script type="text/javascript">

       function verifNomPrenomPasVides1() {
               var r0= verifChampNonVideVersion0('formEx1Nom');
               var r1= verifChampNonVideVersion0('formEx1Prenom');
               var msg0= "";
               var msg1= "";
```

```
                if (! r0)
                        msg0= "champ vide";
                if (! r1)
                        msg1= "champ vide";
                document.getElementById('Ex1ErreurNom').innerHTML= msg0;
                document.getElementById('Ex1ErreurPrenom').innerHTML= msg1;
                return r0 && r1;
        }

        </script>

        <form name="formEx1" action="javascript:pageSuivante()" onsubmit="return verifNomPrenomPasVides1()">
                <table>
                <tbody><tr>
                        <td>nom</td>
                        <td><input name="nom" id="formEx1Nom" type="text"> <span id="Ex1ErreurNom" class="error"></span></td
                </tr>
                <tr>
                        <td>prénom</td>
                        <td><input name="prenom" id="formEx1Prenom" type="text"> <span id="Ex1ErreurPrenom" class="error"></
                </tr>
                </tbody></table>
                <br><input type="submit">
        </form>
```

Traitement des champs en cours d'édition

- Le plus souvent, on utilise l'événement `onblur` ou `onchange`
- Avec `onblur`, le code est appelé quand le curseur quitte le champ (il perd le "focus"),
- Avec `onchange`, le code est appelé quand le curseur quitte le champ *et* que la valeur a changé;
- `onchange` s'utilise aussi pour un champ de type `select`, auquel cas il permet tout simplement de savoir que la sélection a été modifiée.
- S'utilise en *complément* de `onsubmit`
- On n'a pas besoin de retourner une valeur

nom

prénom

Envoyer

```
 <script type="text/javascript">

   function verifChampNonVideVersion2(idChamp,idErreur) {
     var result= true;
     var msg="";
     if (document.getElementById(idChamp).value == "") {
       result= false;
       msg= "champ vide";
     }
     document.getElementById(idErreur).innerHTML= msg;
     return result;
   }

   function verifNomPrenomPasVides2() {
     var r0= verifChampNonVideVersion0('formEx2Nom', 'Ex2ErreurNom');
     var r1= verifChampNonVideVersion0('formEx2Prenom', 'Ex1ErreurNom');
     document.getElementById('Ex1ErreurNom').innerHTML= msg0;
     document.getElementById('Ex1ErreurPrenom').innerHTML= msg1;
     return r0 && r1;
   }
 </script>

<form name="formEx2" action="javascript:pageSuivante()" onsubmit="return verifNomPrenomPasVides1()">
  <table>
    <tbody><tr>
      <td>nom</td>
      <td><input name="nom" id="formEx2Nom" onblur="verifChampNonVideVersion2('formEx2Nom', 'Ex2ErreurNom')" type="text">
       <span id="Ex2ErreurNom" class="error"></span></td>
    </tr>
    <tr>
      <td>prénom</td>
      <td><input name="prenom" id="formEx2Prenom" onblur="verifChampNonVideVersion2('formEx2Prenom', 'Ex2ErreurPrenom')" ty
       <span id="Ex2ErreurPrenom" class="error"></span></td>
    </tr>
  </tbody></table>
  <br><input type="submit">
</form>
```

Vérification lors de la frappe

On peut utiliser `onkeydown` pour un traitement avant modification du champ ou `onkeyup` pour un traitement *a posteriori*.

Noter l'utilisation de return (ne fonctionne qu'avec onkeydown)

Tapez du texte (mais pas de chiffre)

```
<script type="text/javascript">
// inspiré de http://www.w3schools.com/jsref/jsref_onkeypress.asp
function filtreNombre(e)
{
var keynum;

if(window.event) // IE
	{
	keynum = e.keyCode;
	}
else if(e.which) // autres
	{
	keynum = e.which;
	}
var s = String.fromCharCode(keynum)

return "0123456789".indexOf(s) == -1;
}
</script>

<p>Tapez du texte (mais pas de chiffre)
<input onkeydown="return filtreNombre(event)" type="text">
</p>
```

Traitement d'un formulaire uniquement en Javascript

- Normalement un formulaire <form> charge une nouvelle page ;
- Si on veut rester sur la même page, plusieurs solutions :
  1. ne pas utiliser de formulaire
  2. que onSubmit retourne false
  3. utiliser des url javascript :

```
<form action="javascript:pageSuivante()" >
```

Le DOM (Document Object Model)

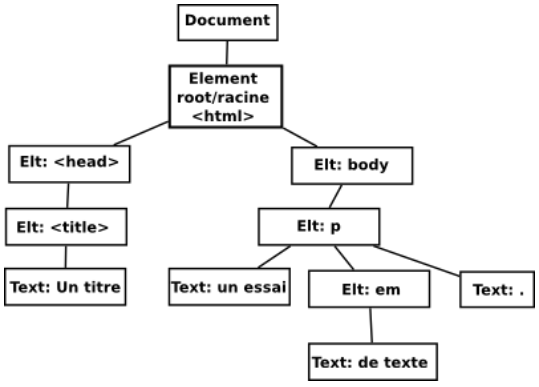
- Représentation *manipulable par javascript* de la page web courante.
- Structure d'arbre
- Accessible à travers l'objet pré-défini `document`
- Compatible avec le DOM XML : spécification pour la manipulation de documents XML dans n'importe quel langage de programmation.
- en théorie, contrôle total sur le document
- en pratique, certains navigateurs ne supportent pas certaines manipulations.

Représentation d'un document

Un document HTML, aux yeux du DOM, est un arbre composé de noeuds, qui représentent les *balises* (appelées *éléments*), le *texte*, et les *attributs*. En pratique, on traitera les attributs à part.

Soit donc le document (simplifié):

```
<html>
	<head>
		<title>Un titre</title>
	</head>
	<body>
		<p> Un essai <em>de texte</em>.</p>
	</body>
</html>
```



- On commence avec l'objet `document`
- Celui-ci contient un élément racine, qui représente la balise html
- Par la suite, on mélange des éléments (des balises) et du texte.
- Chaque élément a des *enfants* (`childNodes`) qui représentent son contenu.
- Ces enfants sont, soit du texte, soit des éléments.
- **Attention:** selon les navigateurs, le contenu du DOM peut varier. Par exemple, certains navigateurs insèrent du texte pour représenter les espaces et les retours à la ligne entre `body` et le premier `p`.

Les noeuds (introduction)

- Chaque partie du document (texte ou élément) est représentée par un noeud (`node`)
- Chaque noeud est un *objet*.
- Je peux lire et (parfois) écrire ses *propriétés* pour consulter ou changer ses caractéristiques
- Je peux appeler des *méthodes* sur le noeud pour modifier son contenu, par exemple
- On distingue plusieurs *types* de noeuds, avec des caractéristiques différentes, dont les noeuds texte et les noeuds éléments
- Pour consulter ou modifier le contenu d'une page en javascript, les étapes sont:
  1. récupérer le ou les éléments concernés
  2. lire et/ou modifier leurs caractéristiques

Accès aux éléments

- la méthode normale pour manipuler un élément est de le doter d'un identificateur, et d'utiliser la méthode `document.getElementById(ID)`.

  Exemple: si j'écris

  ```
  <p id='monIdentificateur0'> un paragraphe </p>
  ```

  Je peux récupérer l'élément en javascript en utilisant getElementById:

  ```
  ... dans du code javascript: ...
  var monParagraphe= document.getElementById('monIdentificateur0');
  // Maintenant, la variable monParagraphe contient l'élément qui représente la balise.
  ```

- getElementById() renvoie `null` si l'élément n'est pas trouvé ;

- la méthode document.getElementsByTagName() renvoie le tableau de *tous* les éléments correspondant à une balise donnée :

  ```
  // Exemple de code: compte le nombre de balises div avec comme classe "exemple" dans le document courant:
  // (du type <div class="exemple">...</div;>)
  funtion compterExemples() {
    var nb=0;
    var i= 0;
    var tab= document.getElementsByTagName("div");
    for (i= 0; i < tab.length; i++) {
         // Le champ qui contient la classe est "className" et non "class"
         // (car class est un mot réservé)
         if (tab[i].className == "exemple")
                 nb= nb + 1;
    }
    return nb;
  }
  ```

- La méthode document.getElementsByName() récupère des éléments à partir de leur attribut "name". Son comportement est très (trop) variable selon les navigateurs.
- document.documentElement désigne la racine du document (l'élément HTML)
- document.body désigne l'élément "body" du document

Propriétés et méthodes des éléments

Tout élément `elt` a les propriétés suivantes.

- innerHTML : non standard (mais implémenté partout), code HTML *interne* de l'élément ;
- childNodes : tableaux des noeuds fils. Chaque noeud fils est, soit un noeud élément (de nodeType 1), soit un noeud texte (nodeType 3) ;
- firstChild : équivalent de childNodes[0] ;
- nodeName : nom de la balise ;
- nodeType : 1 pour éléments (balises) ;
- id : identifiant du noeud ;
- className : classe (au sens CSS) de l'élément ;
- style : accès aux propriétés CSS de l'élément ;
- getAttribute(NOM) : valeur d'un de ses attributs. Ne fonctionne pas pour l'attribut "class" sous IE.
- setAttribute(NOM,VALEUR) : pour un noeud/élément, permet de fixer la valeur d'un attribut. Ne fonctionne pas sous IE pour les styles css ni pour l'attribut "class".

Propriétés des noeuds textes

nodeType
: toujours à 3 pour un noeud texte.

nodeValue
: texte contenu dans le noeud. On peut modifier cette valeur.

Exemple:

ce texte sera modifié, *pas celui-ci*

tester

```
<p id="testNoeuds">ce texte sera modifié, <em>pas celui-ci</em></p>
<script type="text/javascript">
function doTestNoeud() {
	// On récupère le paragraphe...
	var elt= document.getElementById("testNoeuds");
	// Le noeud texte est la première partie de ce paragraphe...
	var noeudTexte= elt.childNodes[0];
	// On change son contenu...
	noeudTexte.nodeValue= "On a modifié ce texte, ";
}
</script>
<button onclick="doTestNoeud()">tester</button>
```

Parcours d'un document

Pour parcourir un document, on va typiquement:

1. Récupérer l'élément racine à partir duquel on fera le parcours.
2. Utiliser childNodes pour parcourir les fils des éléments en question.
3. Pour les dit fils, on regardera quel est leur type, et on agira en conséquence.

=> parcours d'**arbre**

Exemple: on veut afficher le *texte* de la liste précédente dans l'élément ci-dessous. copier

texte quelconque

```
<ol id="maListeOrdonnee">
<li> Récupérer l'élément racine à partir duquel on fera le parcours.</li>
<li> Utiliser childNodes pour parcourir les fils des éléments en question.</li>
<li> Pour les dit fils, on regardera quel est leur type, et on agira en conséquence.</li>
</ol>
<p>=&gt; parcours d'<strong>arbre</strong></p>

Exemple: on veut afficher le <em>texte</em> de la liste précédente dans l'élément ci-dessous.
<button onclick="copierListe()">copier</button>
<div id="ciblePourCopie" class="framed">
texte quelconque
</div>

<script type="text/javascript">
function copierListe() {
	// On récupère l'élément de base
	var eltDeBase= document.getElementById('maListeOrdonnee');
	var txt= getTexteDans(eltDeBase);
	document.getElementById('ciblePourCopie').firstChild.nodeValue= txt;
}

function getTexteDans(noeud) {
	var result= "";
	var i;
	// Le texte contenu dans un noeud
	// est celui contenu dans ses enfants, si c'est un élément
	// et est nodeValue si c'est un noeud texte.
	switch (noeud.nodeType) {
		case 1: // element
			for (i=0; i< noeud.childNodes.length; i++) {
				result+= getTexteDans(noeud.childNodes[i]);
			}
		break;
		case 3: // texte
			result= noeud.nodeValue;
		break;
	}
	return result;
}
</script>
```

Remarques sur les propriétés

En théorie, on peut aussi manipuler des noeuds qui représentent les attributs. Mais en pratique, un bon nombre de navigateurs internets ont des interprétations trop personnelles de ceux-ci. On utilisera donc les propriétés des éléments :

style
: pour accéder aux informations CSS (en lecture et en écriture)

className
: pour lire ou écrire la classe CSS d'un élément

id
: pour accéder à l'identifiant d'un élément

Propriétés spécifiques à certains éléments

value
: Pour les champs de formulaire uniquement. Contient la valeur *actuelle* du champ. *L'attribut* value, accessible par `getAttribute()`, quant à lui, contient la valeur par défaut.

name
: pour les éléments qui ont un attribut name, et en particulier les champs de formulaire, la valeur de l'attribut en question.

disabled
: un booléen. Permet de désactiver/activer un champ

readOnly
: un booléen. Permet de placer un champ en lecture seule

checked
: pour les cases à cocher (CheckBox). Permet de les cocher/décocher

selectedIndex
: pour des listes de type Select, permet de fixer ou consulter l'élément sélectionné

selected
: S'utilise sur un élément `option` dans une liste. Permet de le sélectionner/déselectionner.

Propriétés des éléments de formulaire: input (texte),textarea

name
: nom de l'élément (et variable qui sera envoyée au serveur)

value
: valeur de l'élément. en lecture et en écriture.

Propriétés de <input type="checkbox"> et <input type="radio">

name
: nom de l'élément (et variable qui sera envoyée au serveur)

checked
: booléen (true ou false) permettant de savoir si l'élément est coché. en lecture et en écriture.

value
: valeur de l'élément.

Propriétés des éléments de formulaire: select

Un élément `select` contient des éléments `option`.

Si le select ne permet qu'une sélection, on utilisera la propriété `selectedIndex`. Sinon, on récupèrera le tableau des options (options[]) ou on attaquera directement l'option désirée.

Le code suivant donne la première option sélectionnée:

```
var monOpt= monSelect.options[monSelect.selectedIndex].value;
```

Dans le cas où il y a plusieurs options sélectionnables, le code suivant les range dans un tableau:

```
var o= monSelect.options;
var resultat= new Array();
var i;
for (i= 0; i < o.length; i++) {
        if (o[i].selected) {
                resultat.push(o[i].value);
        }
}
```

Notez que selectedIndex (pour les select) et selected pour les options peuvent être modifiés.

Modification des éléments: 1) innerHTML

Principe

innerHtml est une propriété des éléments DOM, et représente le code HTML compris *à l'intérieur* d'une balise. Par exemple, si j'ai la balise :

```
<ul id='maliste'>
        <li> un</li>
        <li> deux</li>
</ul>
```

Alors, `document.getElementById("maliste").innerHTML` vaut

```
        <li> un</li>
        <li> deux</li>
```

L'intérêt de cette propriété est qu'on peut l'utiliser pour modifier le contenu d'une balise.

Exemple

Du texte à remplacer....

remplacer texte

Le code correspondant:

```
<p><span id="testInnerHTML">
    Du texte à remplacer....
</span></p>

<p><button onclick="javascript:remplacerInnerHTML()">remplacer texte</button></p>

<script type="text/javascript">
  <!--
  function remplacerInnerHTML() {
    //alert("ancienne valeur "+ document.getElementById('testInnerHTML').innerHTML);
    document.getElementById('testInnerHTML').innerHTML=
      "le <b>texte de remplacement</b>";
  }
  //-->
</script>
<!-- Remarque: ci-dessus, nous avons un commentaire HTML utilisé
     pour que le navigateur ne tente pas d'analyser le javascript
     comme du HTML. Le code un peu curieux sur la dernière ligne est
     un commentaire javascript suivi de la fermeture du commentaire
     HTML.
  -->
```

Intérêt et limitations de innerHTML

- Très simple à utiliser
- Propriétaire, mais existe maintenant sur tous les navigateurs
- Pas forcément moins portable que l'utilisation "orthodoxe" du DOM
- Ne fonctionne pas avec tous les éléments, en particulier sous IE (ne permet pas de modifier une case d'une table).
- Remplace tout l'élément ; pas forcément indiqué pour construire des arbres dynamiques, des tables, etc...
- Parfois dangereux ou imprévisible, si le texte de remplacement est du HTML mal formé...

```
<button onclick="document.getElementById('inner2').innerHTML= document.getElementById('inner1').value" >remplacer</button>
```

x<y remplacer

Ce texte sera remplacé...

Solution au dernier problème: écrire une fonction qui remplace les caractères <, > et & par <, > et &, et l'utiliser lors du remplacement... ou passer par le DOM.

Création en passant par le DOM

- Théoriquement, c'est la manière normale. Elle permet, entre autres, d'ajouter et d'insérer des éléments.
- Deux étapes : créer un élément, et d'insérer dans l'arbre.

Création de noeud

Deux méthodes de l'objet document :

- document.createElement('nomDuTag') : crée un élément (balise) pour un type donné de balise. exemple

```
titre= document.createElement('h1');
```

- document.createTextNode('contenu de l'élément')

Les éléments sont simplement créés, et pas insérés dans l'arbre.

Insertion d'élément

On insère les éléments dans un élément parent, en utilisant les méthodes suivantes (n étant un noeud) :

- n.appendChild(nouveauNoeud) : ajoute un noeud comme dernier fils
- n.removeChild(n) : enlève un noeud
- n.replaceChild(old,new) : remplace un noeud par un autre.
- n.insertBefore(a,b) : insère a juste avant le noeud b.

Premier exemple

On reprend l'exemple utilisé pour innerHTML, mais en l'écrivant de manière sûre

x<y remplacer

Ce texte sera *remplacé...*

```
<script type="text/javascript">
  function remplacerAvecDom() {
    // On supprime d'abord tous les noeuds contenus dans le div...
    var i;
    var cible= document.getElementById('domText2');
    var champ= document.getElementById('domText1');
    // Parcours du dernier enfant au premier (exercice: pourquoi??)
    for (i= cible.childNodes.length-1 ; i >=0; i--) {
      cible.removeChild(cible.childNodes[i]);
    }
    // Maintenant, on crée un noeud texte...
    noeudTexte= document.createTextNode(champ.value);
    // On l'insère
    cible.appendChild(noeudTexte);
  }
</script>

<input id="domText1" value="x&lt;y">
<button onclick="remplacerAvecDom()">remplacer</button>

<div class="framed" id="domText2">Ce texte sera <em>remplacé...</em></div>
```

Exemple de modification au travers du DOM...

On crée une liste dont la taille augmente à chaque click sur un bouton. Utilisation pratique: formulaire avec nombre variable d'entrées...

ajouter une entrée

```
<ul id="listeQuiGrandit" style="width: 50%;"></ul>

<script type="text/javascript">
  var numeroDEntree= 0;
  function ajouteEntree() {
    liste= document.getElementById('listeQuiGrandit');
    // On crée le noeud pour la balise li
    nouveauLi= document.createElement('li');
    numeroDEntree++;
    // on crée le noeud texte à insérer dans la balise.
    nouveauTexte= document.createTextNode('entree ' + numeroDEntree);
    // On insère le texte dans la balise
    nouveauLi.appendChild(nouveauTexte);
    // On insère le tout dans la liste:
    liste.appendChild(nouveauLi);
  }
</script>

<button onclick="ajouteEntree()">ajouter une entrée</button>
```

Remarque: ce code n'est pas du HTML valide, car une liste n'a pas le droit d'être vide (en voilà une idée qu'elle est étrange, car quand le code est créé automatiquement, on peut très bien se retrouver avec des "listes" vides. Supposez une requête qui ne renvoie rien, par exemple). Pour bien faire, il faudrait, soit ajouter un élément <li> caché (display="hidden"), soit ne créer la liste qu'avec le premier élément.

Version ancienne du DOM (Pré-XML)

- Les premières versions de javascript ne permettaient pas de manipuler le document de manière souple.

- Seuls étaient accessibles les formulaires (forms), les images, les layers, et les liens.
- Chacun de ces éléments est accessible à travers un tableau : `document.forms[]` , `document.images[]` , `document.layers[]` et `document.links[]` . Ces tableaux sont utilisables même en XHTML.
- pour les éléments en question, s'ils avaient un *name*, ils étaient accessibles par `document.NOM`. Cependant, en XHTML, les `form`, par exemple, ne peuvent plus avoir d'attribut `name`.
- les éléments de formulaire étaient alors accessibles par : document.*nomFormulaire.nomChamp*

Exemple:

```
<form name="monFormulaire2" action="uneAction" onsubmit="return prixPositif()">
<input type="text" name="prix">
........
</form>

<script type="text/javascript">
function prixPositif() {
       var p= document.monFormulaire2.prix.value;
       p= parseFloat(p);
       // si p non numérique, retourne la valeur NaN (not a number)
       // On peut tester avec isNaN
       return ! isNaN(p) && p > 0;
}
</script>
```

système simple, mais qui rend difficile l'écriture de code réutilisable.

Ne fonctionne pas en XHTML strict: name n'est plus défini pour les <form>.

Chargement des pages

L'intéraction entre javascript et la page web dépend de l'existence de certains objets DOM. Il importe donc de savoir comment se passe le chargement des pages:

- Le code est lu en même temps que la page
- Le code HTML crée des objets qui représentent les balises
- Les instructions javascript sont exécutées à la lecture
- pour qu'une instruction soit correctement exécutée, elle doit faire référence à des objets qui existent.

Exemple où l'ordre est respecté

Ce paragraphe sera en rouge sur fond jaune

```
<p id='par1'>Ce paragraphe sera en rouge sur fond jaune </p>

<script type="text/javascript">
       document.getElementById('par1').style.backgroundColor='yellow';
       document.getElementById('par1').style.color='red';
</script>
```

Exemple où l'ordre n'est pas respecté

Ce paragraphe ne va pas changer de couleur

```
<script type="text/javascript">
       document.getElementById('par2').style.backgroundColor='yellow';
       document.getElementById('par2').style.color='red';
</script>

<p id='par2'>Ce paragraphe ne va pas changer de couleur </p>
```

En revanche, une *fonction* peut référencer un élément qui n'existe pas encore. Ce qui importe, c'est qu'il existe *quand la fonction est appelée*.

Ce paragraphe sera aussi en rouge sur fond jaune

```
<script type="text/javascript">
       function changerCouleurPar3() {
              document.getElementById('par3').style.backgroundColor='yellow';
              document.getElementById('par3').style.color='red';
       }
</script>

<p id='par3'>Ce paragraphe sera aussi en rouge sur fond jaune </p>

<script type="text/javascript">
       changerCouleurPar3();
</script>
```

L'ÉVÉNEMENT ONLOAD

- événement associé au body:

```
<body onload="fonctionDeMiseEnplace()">
```

- cette fonction est appelée quand le corps de la page a été chargé, et que les objets DOM existent tous.
- endroit normal pour placer tous les traitements qui doivent se faire quand la page a été chargée.

Étude du fichier presentation.js

Principes:

- Chaque page est dans un div, et a un identifiant (créé par programme au chargement)
- Au départ, toutes les pages on l'attribut "display" à "none"
- Quand on change de page, on cache l'ancienne et on place l'attribut "display" de la nouvelle à "block"