Construction d'applications web

Chapitre 6

Programmation côté client Javascript

Plan

- Le langage javascript
  - Introduction, historique
  - Éléments du langage
    - types
    - fonctions
    - objets
    - constructions syntaxiques
- Le DOM (API permettant au code js d'agir sur la page)
  - Structure du modèle (arbre), navigation, modification
  - Événements et event-handlers/event-listeners
- Intégration de code javascript dans une page web
  - comment (et où) l'ajouter dans un document
  - pratiques à éviter, pratiques conseillées

Javascript : historique

- Langage créé par Netscape pour son navigateur en 1996
- Microsoft crée un langage du même type pour internet explorer : JScript
- Pour résoudre les disparités entre navigateurs : standard ECMAScript
- ECMAScript 5 est dans tous les navigateurs raisonnablement récents.
- La spécification ECMAScript 6 est sortie en 2015.
- Depuis 2015, une nouvelle version chaque année avec quelques ajouts.
- Chaque navigateur a ses propres extensions au standard, incompatibles avec les autres navigateurs.
- Tableau de compatibilité sur ce site
- Pour écrire du code qui marche à peu près chez presque tous les visiteurs potentiels d'un site :
  - n'utiliser que du ECMAScript 5, ou bien une bibliothèque offrant des fonctions de compatibilité comme JQuery (teste les fonctions disponibles et s'adapte)
  - ou écrire son code dans un langage plus adapté au développement (comme TypeScript) et le « transcompiler » en javascript portable.

Javascript

- Javascript est un langage de programmation complet mais avec quelques particularités inhabituelles :

```
var x; //on déclare une variable sans indiquer de type
console.log(typeof x); // x contient initialement la valeur undefined
x = 3; //x contient maintenant un nombre
console.log(typeof x);
x = "hello"; //x contient maintenant une chaîne
console.log(typeof x);
x = function (y) { return x + y; }; //x contient maintenant une fonction
console.log(typeof x); //pas de type de retour ni de paramètre déclaré
x = x(); //toujours pas d'erreur : l'argument absent vaut undefined
//le résultat de x() est x + undefined
//pour pouvoir faire cette opération, x et undefined sont automatiquement
//convertis en chaînes --> concaténation
console.log(typeof x); console.log(x);
x = x(); //enfin une erreur !
//une chaîne ne peut pas être convertie en fonction.
//remarque : l'erreur n'apparaît qu'à l'exécution (pas de compilation)
```

Langage très permissif : une erreur d'inattention a toutes les chances de passer inaperçue dans toute une partie du code avant de provoquer un bug difficile à comprendre...

Types

- Les variables n'ont pas de type fixe, seules les valeurs en ont un
- Toutes les fonctions ont le même type, tous les objets aussi (et les tableaux sont des objets, de même que la constante null)
- Liste des types : undefined, boolean, number, string, function, object. typeof(x) renvoie une chaîne représentant le type de x.
- number signifie float. Il n'y a pas de type int.

- Conversions automatiques : tout est convertible automatiquement en chaîne, en nombre, en booléen et même en objet... (pas en fonction)
- Conversion manuelle chaîne --> flottant : fonctions parseFloat et parseInt (parseInt fait un arrondi)
  - Si la chaîne ne représente pas un nombre valide, ces opérations renvoient le flottant NaN (Not a Number) — donc parseInt ne renvoie pas forcément un entier

Conversions de types automatiques

- Conversion vers une chaîne : dès que l'on utilise l'opérateur + sur des valeurs qui ne sont pas toutes des nombres ("12" + 5 --> "125")
- Conversion vers un nombre : dès qu'on utilise un autre opérateur arith. (p.ex. -) entre des valeurs quelconques
  - "12" - 5 --> 7
  - Tout ce qui ne représente pas un nombre devient le flottant NaN
- Conversion vers un booléen : dans if()
  - deviennent false : les constantes undefined et null, les flottants 0 et NaN, la chaîne vide "" (et la constante false)
  - deviennent true toutes les autres valeurs (y compris l'objet vide {})
- Conversion vers un objet : si on utilise .
  - true.hop = "coucou" --> un objet représentant la constante true est créé, puis un attribut appelé hop, de valeur "coucou", est ajouté à cet objet (la constante elle-même n'est pas modifiée, seul l'objet temporaire créé l'est)
  - parfois utile car l'objet créé automatiquement possède déjà certains attributs et méthodes. Ex : "coucou".length --> 6

Fonctions

- syntaxe : function (args séparés par des virgules) {corps}
- syntaxe alternative (ECMAScript 6) : (args) => {corps}
- déclarer une variable f contenant une fonction :

```
function f (args) {corps}  // équivalent à :
var f = function (args) {corps}
```

- application d'une fonction : pas de vérification du nombre d'arguments
  - si on fournit trop d'arguments, ceux en trop sont ignorés
  - si on n'en fournit pas assez, ceux qu'on n'a pas mis ont la valeur undefined
- conséquence : pas de surcharge (si f est redéfinie avec un nombre différent d'arguments, l'ancienne version est perdue)
- si la fonction ne contient pas return, elle renvoie undefined

Objets

- Un objet est un ens. de propriétés accessibles par leur nom
- Pas de vraie distinction attribut/méthode (une méthode est juste une propriété dont la valeur est une fonction)
  - mais dans une méthode on peut se référer à l'objet qui la contient par this
- On peut ajouter/supprimer des propriétés dynamiquement

```
var objet = {                              // définition directe d'un objet
  attribut1 : "hello",                     // déclaration d'un attribut
  methode1() {return this.attribut1;}      // déclaration d'une méthode
};

objet.methode1();                          // renvoie "hello"
objet.attribut2 = 53;                      // ajout d'un nouvel attribut
objet.methode2 = function (x) {this.attribut3 = x;}; // ajout de méthode
console.log(objet);
objet.methode2(12);                        // crée l'attribut 3
delete objet.attribut1;                    // suppression d'un attribut
objet.methode1();                          // renvoie undefined
console.log(objet);
```

Objets

- Parcours de la liste des propriétés d'un objet :

ATTENTION cette syntaxe ne fait pas ce que vous croyez !

```
for (var i in objet) {console.log(i + " : " + typeof(objet[i]));}
```

- i parcourt les **noms** de toutes les propriétés de objet (ce sont des chaînes)
- ne permet **pas** de parcourir un tableau (utiliser le classique `for (var i=0; i<t.length; i++)`)
- objet["machin"] éq. à objet.machin --> permet l'accès à une propriété dont le nom est dans une variable

- Pas de notion de classe, tous les objets sont du même type
- Mais : il existe des notions d'héritage et de constructeurs utilisant un système de prototypes (très différent de ce qu'on a en java).
- Inutile de connaître les détails pour faire du javascript de base, mais cela permet un opérateur instanceof qui peut être utile quand on manipule des pages web

Tableaux

- Les tableaux n'ont pas une taille fixe
- Ils peuvent contenir des données de différents types
- Ce sont en fait des objets particuliers
- Exemple de code légal (à éviter ! mais on peut le faire par erreur...) :

```
var tableau = [1, "hop", function(){}]; // tableau est de taille 3
console.log(tableau.length); // length est une propriété de l'objet
tableau[2843] = {};          // aucune erreur ici
console.log(tableau.length); // mais le tableau est de taille 2844
tableau[120];                // undefined
delete tableau[2843];        // ne change pas la taille
tableau[-4] = 12;            // légal et ne change pas la taille
```

- Le type d'un tableau est object, mais on peut les reconnaître avec tableau instanceof Array

Comparaisons

- Comparaison usuelle : ===
- Différence : !==
- L'opérateur == existe mais fait une conversion automatique de type avant de comparer les valeurs
- ainsi 5 == "5" est vrai
- idem pour !=
- --> utiliser === et !== sauf si on est vraiment sûr qu'on veut la conversion (rare)

Constructions syntaxiques

- if/else, for, while, do/while --> idem java
- différence : une variable déclarée avec var (même dans un bloc) est toujours globale à la fonction
- déclaration locale « à la java » : let x; (ECMAScript 6)
- switch/case : idem java, les case sont testés avec ===
- throw : lève une exception (prend en argument n'importe quoi, un objet, une chaîne...)
- try/catch/finally : presque comme java, mais le catch attrape forcément **toutes** les exceptions (car pas de type). Ensuite on peut trier avec un switch/case par exemple.

```
try {
  throw "ouille ouille ouille";
} catch (e) {
  console.log(e); // écrit "ouille ouille ouille"
}
```

Chaînes

- Peuvent être délimitées aussi bien par ' que par "
- si par ', les " dans la chaîne n'ont pas besoin d'être échappés et inversement
- --> pratique pour mettre du HTML dans une chaîne javascript ou du javascript dans un attribut HTML
- en js : var lien = 'retour'
- en HTML :

```
<button type="button" onclick="alert('Hello');">Hello</button>
```

Mode strict

- Mode permettant de détecter davantage d'erreurs (p.ex. multiples déclaration d'une même variable, ou utilisation d'une variable non déclarée)
- pour l'activer : `"use strict";` en **tête de fichier**
- à taper avec les guillemets (pour compatibilité avec de vieilles versions de javascript)
- recommandé !

- sans le mode strict, utiliser une variable non déclarée crée une variable globale (même si on est dans une fonction) : rarement ce qu'on veut
- redéclarer une variable ne fait rien du tout

Le DOM : principe

- L'intérêt du javascript dans un navigateur est de pouvoir agir sur la page web
- Pour permettre cela, le navigateur crée une structure de données standardisée représentant le contenu de la page : le DOM (Document Object Model)
- le DOM est standardisé par le W3C et est indépendant de ECMAScript --> utilise des notions comme « interface » étrangères au javascript
- le DOM est orienté-objet (comme son nom l'indique) et le document (= la page web) est représenté par un arbre
- chaque élément HTML est un nœud de l'arbre. Les nœuds sont des objets avec des attributs et méthodes permettant de naviguer dans l'arbre, plus des propriétés spécifiques en fonction du type de nœud

Le DOM : structure

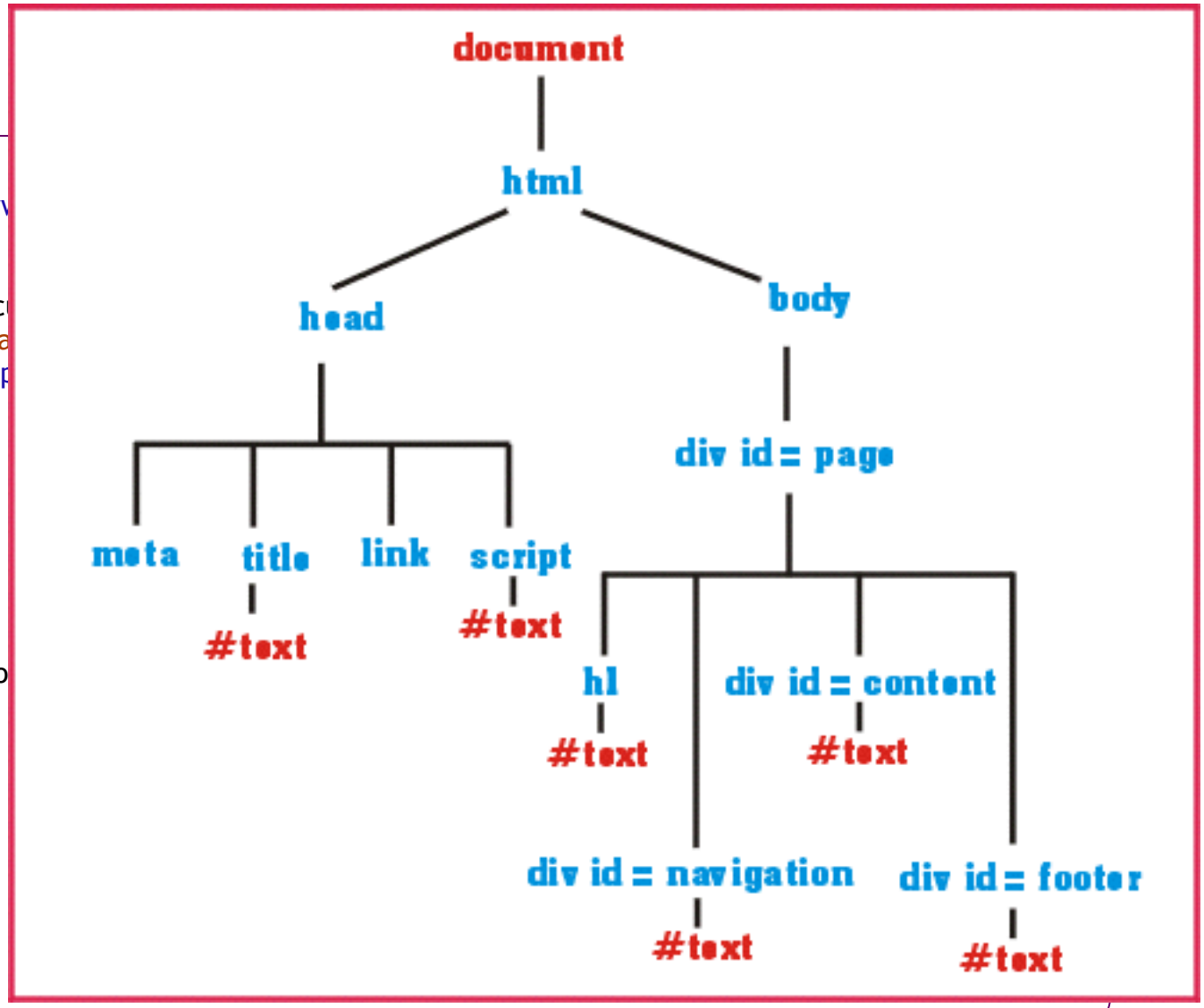
Exemple

```
<!DOCTYPE html>
<html lang="fr" xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <meta  charset="utf-8" />
      <title>Le modèle objet document (DOM)</title>
      <link rel="stylesheet" media="screen" type="text/css" href="/tutorial/style/screen.css" />
      <script type="text/javascript" src="/tutorial/script/script.js"></script>
   </head>
   <body>
      <div id="page">
         <h1>voici un titre</h1>
         <div id="content">
              ici il y a du contenu
         </div>
         <div id="navigation">
              la barre de navigation se trouve ici
        </div>
        <div id="footer">
               un peu de texte
        </div>
     </div>
  </body>
</html>
```

Le DOM : structure

Exemple

```
<!DOCTYPE html>
<html lang="fr" xmlns="http://ww
   <head>
      <meta  charset="utf-8" />
      <title>Le modèle objet doc
      <link rel="stylesheet" media
      <script type="text/javascrip
   </head>
   <body>
      <div id="page">
         <h1>voici un titre</h1>
         <div id="content">
            ici il y a du contenu
         </div>
         <div id="navigation">
            la barre de navigatio
         </div>
         <div id="footer">
            un peu de texte
         </div>
      </div>
  </body>
</html>
```



Le DOM : navigation

- Les nœuds de l'arbre sont les éléments HTML mais aussi :
  - des nœuds de texte. Attention : les pages HTML contiennent de nombreux nœuds textes ne comprenant que des espaces/retours à la ligne, bien que ces caractères soient ignorés pour l'affichage.
  - des nœuds de commentaire (représentant les commentaires dans le code de la page)
  - si un élément (par exemple un lien) apparaît au milieu du texte, il est donc précédé et suivi par des nœuds texte
- API de navigation dans le DOM : l'interface Node

si n est un nœud, il possède les propriétés suivantes :

- `n.parentNode`/`n.childNodes` nœud parent/liste des enfants
- `n.firstChild`/`n.lastChild` premier/dernier nœud enfant
- `n.previousSibling`/`n.nextSibling` nœud frère (= de même parent) précédent/suivant
- En cas d'inexistence : null. Il existe aussi la méthode hasChildNodes()
- Attention aux nœuds texte non significatifs.

Le DOM : navigation et modification

- Navigation restreinte aux éléments :
  - propriétés `parentElement`, `children`, `firstElementChild`, `lastElementChild`, `previousElementSibling`, `nextElementSibling`. À privilégier pour éviter les nœuds texte.
- Modification de l'arbre :
  - insertion d'un élément (avec son sous-arbre)
    - parent.appendChild(enfant) -> ajoute enfant à parent comme dernier fils
    - parent.insertBefore(nouveau, ancien) -> ajoute nouveau aux fils de parent en le plaçant devant ancien (qui doit déjà être un fils de parent)
  - ces méthodes **déplacent** le nœud inséré s'il était déjà ailleurs dans le document. Si on veut le dupliquer :
    - n.cloneNode(true) renvoie une copie de n et de tout son contenu (son sous-arbre)
    - n.cloneNode(false) renvoie une copie de n sans son contenu (élément vide)
    - Attention : si n a un attribut id, sa copie aura la même valeur d'attribut id (interdit dans un même document --> changer la valeur avant de l'insérer)

Le DOM : modification

- Modification de l'arbre (suite)
  - Suppression d'un nœud : parent.removeChild(enfant)
  - Remplacement par un nouveau nœud : parent.replaceChild(nouveau, ancien)
- Création de nœuds :
  - document.createElement(balise) balise est un nom d'élément html ex. "p", "h1"
  - document.createTextNode(texte) texte sera le contenu du nœud
  - Ces nœuds ne sont pas automatiquement insérés dans le document
- Récupérer ou remplacer tout le contenu d'un élément :
  - n.innerHTML, attribut en lecture/écriture, est une chaîne représentant tout le contenu de n sous forme de code source HTML
  - beaucoup moins fastidieux pour ajouter du contenu qu'une série de createElement (mais ne permet pas d'ajouter un nouvel enfant à un nœud qui en a déjà sans supprimer les autres)
  - n.textContent contient tout le texte se trouvant dans le sous-arbre n, sans les balises HTML. En écriture, remplace le contenu de n par un seul nœud texte (plus rapide que innerHTML car pas besoin de parser)

Le DOM : recherche d'éléments

- Pour récupérer directement certains éléments : méthodes de l'objet prédéfini document
  - document.querySelector(sélecteur CSS) : le premier élément correspondant au sélecteur CSS spécifié (ex. "div#principal h1" pour le premier élément h1 de l'élément div d'identifiant "principal")
  - document.querySelectorAll(sélecteur CSS) : liste de tous les éléments correspondant au sélecteur
  - document.getElementById(id) : l'unique élément d'identifiant id (le premier si le document est mal formé et qu'il y en a plusieurs)
  - document.getElementsByTagName(balise) : tous les éléments balise
  - document.getElementsByClassName(classe) : tous les éléments dont l'attribut class vaut classe
  - raccourci : document.head et document.body pour accéder directement à ces deux éléments
  - la recherche par nom de balise ou de classe existe aussi sur les éléments, mais ne cherche que parmi les fils **directs**, et non récursivement en profondeur

Le DOM : attributs HTML

- Les attributs HTML (ex. le href d'un a, le src d'un img, le action ou le method d'un form) sont accessibles (en lecture et écriture) comme des attributs javascript de l'objet correspondant

- La plupart du temps, la valeur est une chaîne
- Certains attributs comme style sont des objets structurés
  - cet attribut correspond au style spécifique de l'élément (pas le style appliqué par des CSS externes), par défaut il est vide si style="..." n'était pas présent dans le code HTML
  - une propriété CSS de nom nom-de-la-propriété correspond à l'attribut nomDeLaPropriété de cet objet style
  - exemple : paragraphe.style.fontWeight = "bold"; met en gras le contenu de l'élément paragraphe

Le DOM : événements

- Le code javascript doit généralement être déclenché en réponse à des actions utilisateur
- Ces actions sont représentées par des événements, qui font partie du standard DOM
- Voir la documentation pour une liste exhaustive. Le plus utilisé : click (clic gauche)
- À la plupart des événements correspond une propriété des éléments appelée "on"+nomdelévénement ex.: onclick
- Cette propriété doit être soit une fonction à un paramètre, soit null. La fonction est appelée lorsque l'événement est déclenché avec pour cible l'élément (dans le cas de click, quand l'utilisateur clique à gauche avec le pointeur sur l'élément), avec pour paramètre l'événement
- La fonction peut accéder à l'élément par this (car c'est une méthode de cet élément) et à l'événement par son paramètre
- Une telle fonction est appelée « event handler »

Le DOM : événements

- Les event handlers peuvent être définis ou remplacés dynamiquement :

```
var bouton = document.createElement("button");
bouton.type = "button";
bouton.textContent = "vous n'avez pas cliqué";
// handler pour l'appui sur un bouton quelconque de la souris
bouton.onmousedown = function (event) {
  var lesboutons = ["à gauche","au milieu","à droite"];
  this.textContent = "vous avez cliqué " + // event.button contient
                     lesboutons[event.button]; // le no du bouton
  this.onmousedown = null; // ce handler ne s'exécutera qu'une fois
 }
// on insère le bouton en fin de page
document.body.appendChild(bouton);
```

Le DOM : événements

- Un event handler peut aussi être défini directement en attribut dans le code HTML
- Particularité : dans le code HTML on ne doit mettre que le corps de la fonction. Exemple :

```
<button type="button" onmousedown="this.textContent = event.button;">blabla</button>
```

ici le event handler est en fait : function (event) {this.textContent = event.button;}

- Inconvénient : du code javascript caché dans un attribut quelque part au milieu du HTML... très pénible à maintenir
- Peut être utile quand même, mais à utiliser avec parcimonie, et en tout état de cause limiter le contenu du handler à un appel de fonction maximum
- Remarque : dans onclick="maFonction();" la fonction maFonction n'est pas une méthode de l'objet --> ne peut accéder à this --> le passer en paramètre si nécessaire
- Meilleure technique en général : ajouter les event handlers après chargement de la page, en récupérant les éléments concernés via getElementById ou querySelector

Le DOM : listeners

- Autre moyen de réagir aux événements : event listeners
- Intérêt :
  - plusieurs listeners possibles pour un même élément et un même événement
  - certains événements n'ont pas d'attribut correspondant
  - globalement plus flexible et à peine plus lourd
- Ajout :

```
element.addEventListener("nom de l'événement", function (event) {...});
```

ou : `function f (event) {...}; element.addEventListener("...", f);`

- Suppression :

```
element.removeEventListener("nom de l'événement", f)
```

Remarque : pour pouvoir supprimer un listener, il faut avoir gardé une référence vers la fonction correspondante

- "nom de l'événement" ne contient pas "on", ex. : "click"
- this peut aussi être utilisé dans les listeners

Ajout de scripts à une page web

Du code javascript peut apparaître de plusieurs façons :

- Cas général : l'élément HTML <script>
- Cas particuliers :
  - attributs onevent pour les handlers
  - **obsolète** mais on peut encore le voir : dans une pseudo-URL dans un lien : <a href="javascript:ducode">
- L'élément <script> peut :
  - soit contenir directement du code

    <script type="text/javascript">ici du code</script>
  - soit contenir une référence à un fichier externe

    <script type="text/javascript" src="monfichier.js">ici un commentaire décrivant le script qui sera ignoré par le navigateur mais qu'on a le droit de mettre si on veut</script>
  - ne pas utiliser <script src... /> : toujours mettre la balise fermante
  - préférer les scripts dans un fichier externe : plus facile à maintenir, et le navigateur n'a pas besoin de re-télécharger le script chaque fois que la page est modifiée. Le script peut aussi être commun à plusieurs pages

Ajout de scripts à une page web

- L'élément <script> peut apparaître (à peu près) n'importe où
- Le code sera exécuté immédiatement quand le parseur HTML rencontre l'élément. Placer cet élément en plein milieu de la page a très rarement un intérêt --> au début dans <head> ou bien tout à la fin
- Dans <head> :
  - intérêt : cohérent avec la structure des documents : même endroit que les références aux feuilles de style etc.
  - inconvénient : document pas encore parsé, le DOM n'est pas encore construit donc on ne peut pas accéder aux éléments
  - solution : une fois le DOM construit un événement est activé sur l'objet document. On peut mettre tout le code de mise en place dans un event listener correspondant.
  - autre inconvénient : si on charge de gros fichiers js (bibliothèques...) le navigateur n'affiche rien tant qu'il n'a pas tout téléchargé et exécuté.
- à la fin (juste avant </body>) :
  - intérêt : c'est une solution simple aux deux inconvénients
  - inconvénient : incompatible avec onclick ou autre dans le code HTML

Éléments d'interface interactifs

- Certains éléments d'une page sont naturellement interactifs : éléments de formulaires, boutons, liens
- L'utilisateur sait qu'ils vont réagir, peut leur donner le focus clavier avec Tab (et cliquer au clavier avec la barre d'espace)
- Les privilégier pour mettre du javascript (sauf à vouloir cacher des fonctionnalités) ou faire en sorte qu'on voie ce qui est interactif
- Le bouton est l'objet de base pour déclencher un script :

```
<button type="button">texte</button>
```

- toujours préciser le type (sinon : bouton de soumission d'un formulaire)
- attention : le texte du bouton est dans l'élément (≠ de l'élément <input type="submit" value="texte" /> où il est dans un attribut)

Actions par défaut

- Certains événements ont une action par défaut (clic sur un lien, soumission d'un formulaire)
- On peut annuler cette action dans un listener via la méthode preventDefault() de event
- Par exemple pour un lien : si on appelle cette méthode, on reste sur la page, sinon le lien est suivi.
- Toujours considérer que le script peut ne pas s'exécuter et que l'action par défaut doit avoir un sens.
- Exemple de mauvaise pratique commune : lien ne servant qu'à exécuter du javascript et ayant une fausse URL : « ouvrir dans un nouvel onglet » n'exécute pas le script et ouvre une page blanche, une erreur 404...
- S'il n'y a pas d'URL pertinente à mettre, ça ne devrait pas être un lien mais par exemple un bouton. (Remarque : un bouton peut ressembler à un lien si on emploie des styles CSS pour cela, mais le navigateur ne proposera pas « ouvrir dans un nouvel onglet »)

Javascript : bonnes pratiques

- Tous les navigateurs ne supportent pas javascript (ou des versions obsolètes...)
- javascript enrichit la page et l'interface, mais l'application doit rester utilisable sans (« graceful degradation »)
- éviter de "casser" l'interface usuelle du navigateur (ex. : si on remplace tout le contenu de la page en js, c'est en fait la même page et le bouton "précédent" revient donc à la page d'encore avant --> à éviter)

Test du code

- Par défaut : erreurs javascript invisibles dans les navigateurs (un visiteur d'un site ne veut pas voir des messages d'erreur abscons)
- Pour les voir : console javascript dans les outils de développement (indispensable pour débugguer)
- La console permet aussi de taper du code interactivement et de l'exécuter pour voir immédiatement son effet et la valeur de retour
- écrire dans la console : console.log(...) (non standard. À éviter dans le code final)
- firefox a aussi l'« ardoise javascript » (scratchpad)
- fonction également utile : alert(message) ouvre un popup avec le message et suspend l'exécution jusqu'à clic sur ok