Familiarisation avec


JavaScript

PARTIE **2**

# Interactions avec le document HTML

# Le DOM (Document Object Model)

Pour pouvoir appréhender les éléments d'une page HTML, en récupérer des informations et agir sur eux, JavaScript a besoin que cette page soit en quelque sorte cartographiée, rendue compatible avec sa logique : il faut que le langage puisse comprendre la page comme une collection d'objets, puisque c'est de cette manière que quasiment tout est structuré pour JavaScript.

Les variables, les tableaux, les fonctions, sont tous des **objets**, c'est à dire des ensembles de ***propriétés*** (leurs caractéristiques) et de ***méthodes*** (ce qu'ils peuvent faire, leurs fonctions).

C'est ici qu'intervient le *Document Object Model*. Il s'agit d'une **représentation du document HTML, lisible et accessible par JavaScript, dans laquelle chaque élément (correspondant à chaque balise et contenu) est un objet.** Il se structure comme un arbre, respectant la hiérarchie écrite en HTML.

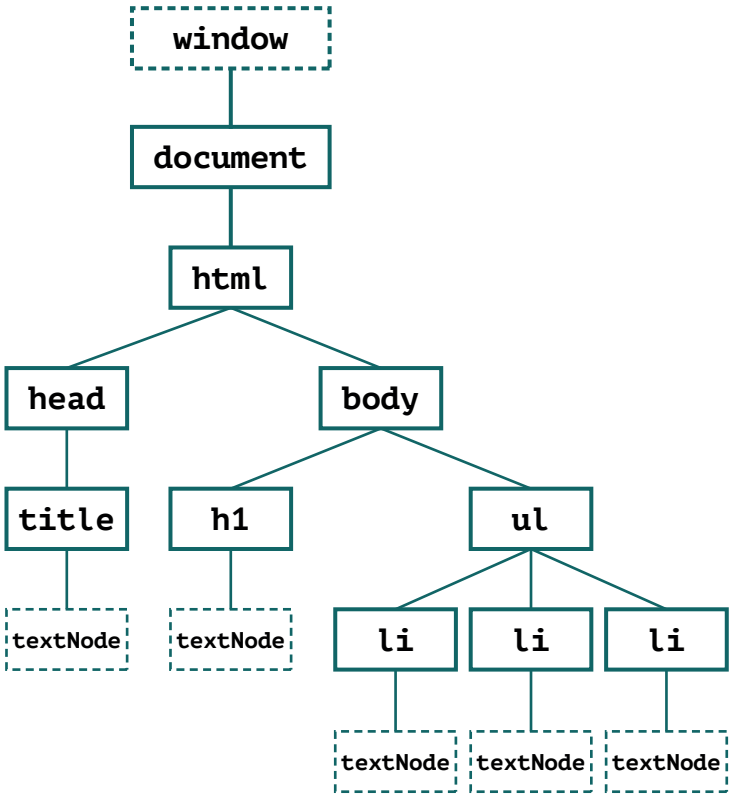
→ Dans cet arbre, l'élément **<html>** est le premier objet. Il est le parent de tous les autres, qui y sont imbriqués. On parle de l'élément racine (*root*).

→ Chaque élément imbriqué est appelé un noeud (*node*). On dit qu'il est un enfant (*child*) de l'élément dans lequel il est imbriqué (son *parent*).

→ Les différents enfants d'un même élément sont appelés frères (*siblings*).

→ Le **Document** lui-même est un objet, contenant la totalité de l'arbre ainsi que les méthodes pour le manipuler. Il est contenu dans l'objet **Window**, représentant la fenêtre du navigateur dans laquelle le Document est affiché.

```
<html>

  <head>
    <title>Une Page</title>
  </head>

  <body>

    <h1>Lorem ipsum</h1>

    <ul>

      <li>dolor</li>

      <li>sit</li>

      <li>amet</li>

    </ul>

  </body>

</html>
```



# Accéder aux éléments de la page

Il faudra tout d'abord **identifier l'élément** auquel on souhaite s'adresser. Plusieurs propriétés des objets du *DOM* peuvent nous aider.

```
<div id = "ma_div" class = "custom_div style1" ></div>
```

Les attributs **id** et **class** de l'élément ci-dessus sont accessibles via les propriétés **id** et **className** de l'objet correspondant. L'objet **Document** possède deux méthodes prévues à cet effet : **getElementById()** et **getElementsByClassName()**.

On parle de *méthode* pour **une fonction appartenant à un objet**.
Pour exécuter une méthode d'un objet, on y fera appel de la manière suivante :

```
mon_objet.Une_de_ses_methodes();
```

Notons que techniquement, toute fonction qui n'est pas attribuée à un objet est implicitement une méthode de l'objet **Window** et pourrait s'écrire **window.Nom_de_la_fonction()**
tout comme **mon_objet.Une_de_ses_methodes()** pourrait s'écrire **window.mon_objet.Une_de_ses_methodes()**.

Mais comme l'objet **Window** englobe le DOM, sa désignation est implicite et il n'est pas nécessaire de l'écrire.

```
var maDiv = document.getElementById("ma_div");
```

Ici on stocke dans **maDiv** l'élément possédant l'identifiant "ma_div". Cette méthode ne peut renvoyer qu'un seul élément, car un identifiant n'est pas censé être partagé par plusieurs éléments sur une même page. Elle renverra **false** si aucun élément ne possède l'identifiant recherché.

```
var mesDivs = document.getElementsByClassName("custom_div");
```

Ici un stocke dans **mesDivs** la liste des éléments possédant la classe "custom_div". Le résultat renvoyé par cette méthode sera toujours une *nodeList* (liste de noeuds, aux caractéristiques similaires à celles d'un tableau), même lorsqu'un seul élément correspond. Si aucun élément ne correspond, la liste sera vide.

Pour désigner précisément notre élément à partir de cette liste il faudra donc utiliser :

```
var maDiv = mesDivs[0];
// mesDivs ne contient qu'un seul élément, celui d'indice 0
```

**Plusieurs classes peuvent être cumulées** dans le paramètre passé à cette méthode, par exemple pour affiner le résultat renvoyé. Les classes doivent être **séparées par des espaces**. L'ordre dans lequel elles sont écrites n'a pas besoin d'être le même dans le paramètre et dans l'attribut **class** de la balise.

```
<div class = "custom_div style1" ></div>
<div class = "custom_div style1 special" ></div>
<div class = "custom_div style2" ></div>
<div class = "custom_div style3" ></div>
<div class = "custom_div" ></div>

<script>

console.log(
document.getElementsByClassName("custom_div").length()
); // affichera 5

console.log(
document.getElementsByClassName("custom_div style1").length()
); // affichera 2

console.log(
document.getElementsByClassName("custom_div style2").length()
); // affichera 1

console.log(
document.getElementsByClassName("custom_div style2 special").length()
); // affichera 0

console.log(
document.getElementsByClassName("special style1").length()
); // affichera 1

</script>
```

**L'ordre des éléments** dans la liste renvoyée correspond à celui dans lequel ils sont rencontrés dans la page HTML.

Une troisième méthode, fonctionnant de manière analogue à **getElementsByClassName()**, permet de désigner une liste d'éléments d'après leur **tagname**, c'est à dire la nature des balises HTML correspondantes.

```
<div id = "conteneur" >

  <h1>Un titre</h1>

  <div class = "colonne"></div>
  <div class = "colonne"></div>

</div>
```

D'après cette structure HTML :

```
var lesDiv = document.getElementsByTagName("div");
// listera 3 éléments : le conteneur et les deux colonnes

var lesH1 = document.getElementsByTagName("h1");
// un seul élément dans la liste

var leH1 = lesH1[0];
```

Pour lister tous les éléments du document à l'aide de cette méthode, on peut utiliser une astérisque en paramètre (toujours sous la forme d'une chaîne) :

```
var tous_les_elements = document.getElementsByTagName("*");
```

Dans le *DOM*, tous les éléments (capables de posséder des enfants) héritent des méthodes **`getElementsByClassName()`** et **`getElementsByTagName()`**.

**On peut donc appeler ces méthodes depuis d'autres objets que le document.** La recherche se limitera alors à ce qui est imbriqué (sans limite de niveaux) dans l'élément depuis lequel la méthode a été appelée.

```
<div id = "div1" ></div>

<div id = "div2" >

  <div id = "div3" ></div>
  <div id = "div4" ></div>

</div>

<div id = "div5" ></div>
```

D'après cette structure HTML :

```
var div2 = document.getElementById("div2");

var test = div2.getElementsByTagName("div");
```

La variable **`test`** contiendra uniquement les **`<div> 3 et 4`** (les **`<div> 1 et 5`** se trouvant en-dehors de l'élément depuis lequel la méthode a été appelée, et la **`<div> 2`** étant cet élément).

## • Désigner un élément via un sélecteur CSS

L'objet **Document** possède deux méthodes supplémentaires permettant de sélectionner un ou plusieurs objets selon la syntaxe, plus polyvalente, des sélecteurs CSS. L'avantage est de pouvoir passer en paramètre **des expressions combinant tagname, classes et identifiant.**

```
<div id = "conteneur1" >

  <div class = "colonne"></div>
  <div class = "colonne"></div>

</div>

<div id = "conteneur2" >

  <p class = "colonne"></p>
  <div class = "colonne"></div>
  <div class = "colonne"></div>

</div>
```

D'après cette structure HTML, on peut par exemple récupérer le premier élément ** ** de classe "colonne" imbriqué dans l'élément dont l'identifiant est "conteneur2" :

```
var colonne4 = document.querySelector("#conteneur2 div.colonne");
```

La méthode **querySelector()** renvoie un seul élément : le premier rencontré correspondant au sélecteur passé en paramètre. Pour obtenir la liste de tous les éléments y correspondant, il faudra utiliser sa variante **querySelectorAll()**.

```
var div_cols = document.querySelectorAll(
"#conteneur2 div.colonne"
);
```

Renverra la liste (sous la même forme que celles données par **getElementsByClassName()** et **getElementsByTagName()**, de tous les éléments ** ** de classe "colonne".

Tout comme lors d'une déclaration de style CSS, plusieurs sélecteurs peuvent être concernés simultanément, en les séparant par des virgules.

```
var elems = document.querySelectorAll(
"#conteneur1 .colonne , #conteneur2 p.colonne"
);
```

Cette expression permettra de lister, ensemble, tous les éléments de classe "colonne" imbriqués dans l'élément **#conteneur1** et les éléments à la fois de classe "colonne" et de type paragraphe imbriqués dans **#conteneur2**.

## • Naviguer entre les éléments

Il est possible, à partir d'un premier élément, d'en désigner un autre en fonction de **la relation qu'ont ces deux éléments**. Chaque objet possède parmi ses propriétés des références à ses voisins.

```
<div>
  <h1>Titre</h1>
  <p>Lorem Ipsum...</p>
  <div id = "colonne1" class = "colonne"></div>
  <img src = "image.jpg" />
  <div class = "colonne"></div>
</div>

<script>

var colonne1 = document.getElementById("colonne1");

var conteneur = colonne1.parentNode;
// correspond à la div extérieure, parent de colonne1

var img = colonne1.nextSibling;
// correspond à l'élément image, placé juste après colonne1

var colonne2 = colonne1.nextSibling.nextSibling;

var parag = colonne1.previousSibling;
// correspond au paragraphe, placé juste avant colonne1

var titre = conteneur.firstChild;
// correspond au premier enfant de la div extérieure

// On peut également obtenir une liste de tous les enfants
d'un élément via sa propriété childNodes :

var contenu = conteneur.childNodes;
// similaire à conteneur.getElementsByTagName("*")

</script>
```

# Manipuler les propriétés des éléments

## • Stylisation en ligne et classes CSS

Pour modifier un élément, c'est à ses propriétés qu'il va falloir accéder. Les propriétés les plus couramment manipulées concerneront la stylisation et les classes CSS de l'élément.

```
<div id = "test" class = "classe1"></div>
```

Pour cet élément HTML :

```
var test = document.getElementById("test");


// Récupération de la propriété className, correspondant aux
classes CSS de l'élément :

console.log(test.className) ; // affichera "classe1"


// Modification de cette propriété :

test.className = "classe1 classe2";


// Modification d'une propriété de stylisation en ligne :

test.style.backgroundColor = "blue";
```

Les attributs de style en ligne et ceux définis via CSS sont deux choses entièrement distinctes. Modifier une propriété via **`element.style`** correspond à ajouter un attribut directement dans la balise HTML de l'élément. **Une propriété modifiée de cette manière sera prioritaire par rapport à la même propriété définie dans une déclaration de style CSS.** Par exemple :

```
<style>

#test , #test.classe1 , div#test.classe1 {

background-color:blue;
color:white;

}

</style>

<div id = "test" class = "classe1" style =
"background-color:green" ></div>
```

Ici, l'élément **`#test`** aura un fond vert et non bleu, car **la stylisation en ligne prend le dessus sur la déclaration CSS**. Son contenu textuel sera bien de couleur blanche comme défini via CSS, car aucune règle en ligne ne vient contredire cette déclaration.

L'attribut **`style`** en ligne, lorsque renseigné dans la balise, reprend la syntaxe CSS, acceptant les tirets et insensible à la casse. **Ce n'est pas le cas lorsqu'on modifie une sous-propriété de cet attribut style via JavaScript** : le tiret n'est plus admis (car considéré comme un opérateur **-**) et la casse *camelCase*, (mots accolés, avec majuscule sur les premières lettres sauf celle du premier mot) omniprésente dans les noms d'objets JavaScript, doit être respectée. Il est par contre possible de réécrire l'attribut **`style`** dans sa totalité via JavaScript, dans quel cas la syntaxe à utiliser sera celle des déclarations CSS.

```
test.style.backgroundColor = "green";
// correspondra à "background-color:green" dans l'attribut
style de la balise

test.style = "color:blue"; // ici la syntaxe CSS est utilisée

console.log(test.style.backgroundColor);
// affichera une chaîne de caractères vide
```

Lors de la seconde instruction, comme la totalité de l'attribut **`style`** est redéfinie, **la modification faite à la ligne précédente sera annulée** ; plus rien n'est défini concernant **`style.backgroundColor`**.

S'adresser précisément et séparément aux sous-propriétés de **`test.style`** permet plus de flexibilité.

## • Modification du contenu d'un élément

Pour la plupart des éléments (tous ceux capables d'en contenir d'autres), leur contenu est accessible, au format HTML, via leur propriété **innerHTML**.

```
<div id = "conteneur" ><p>Contenu dans un paragraphe</p></div>

<script>
var C = document.getElementById("conteneur");

console.log(C.innerHTML);
// affichera "<p>Contenu dans un paragraphe</p>"

// Pour ajouter du contenu (après l'élément paragraphe) :

C.innerHTML += "<p>Et voici un second paragraphe, ajouté via
JavaScript</p>";

</script>
```

Deux manières d'obtenir le contenu d'un élément ont déjà été abordées précédemment : la méthode **element.getElementsByTagName("*")** et la propriété **element.childNodes**. Leur différence avec cette nouvelle propriété **element.innerHTML** est le format dans lequel le contenu est récupéré : ici il s'agit d'une chaîne unique contenant la structure HTML, alors que les précédentes manières donnent accès à la liste des objets du *DOM* correspondant à cette structure HTML.

Pour certains autres éléments, une propriété importante sera **value**. C'est typiquement le cas pour les éléments de formulaires (destinés pour la plupart à recevoir des entrées de la part de l'utilisateur de la page HTML) :

```
<input id = "champ1" type = "text" value = "Contenu par
défaut" />
```

Pour ce champ texte, on accède à son contenu (éventuellement modifié par l'utilisateur) via **document.getElementById("champ1").value**.

Pour un champ multi-lignes **<textarea></textarea>**, bien que la mise en place d'un attribut **value** directement dans la balise HTML ne modifie pas le contenu de ce champ, c'est bien via la propriété **.value** que JavaScript peut y accéder. Il existe bien pour ce type de champs une propriété **.innerHTML** se comportant en apparence comme une valeur, mais elle est indépendante et ne se mettra pas à jour en fonction de ce qui est saisi dans le champ par l'utilisateur.

Ainsi pour donner une valeur par défaut à un champ multi-lignes :

```
<textarea id = "champ2" value = "Il ne faut pas utiliser cet
attribut..." > Ni écrire ici. </textarea>
```

Mais en passant par JavaScript :

```
document.getElementById("champ2").value = "Ici une valeur peut
être donnée correctement.";
```

Pour d'autres éléments la propriété **value** correspondra à diverses caractéristiques :

→ Pour un bouton il s'agira du texte qui y est affiché.

```
<input type = "button" value =  "OK" />
```

→ Dans un menu déroulant, cette propriété contient, pour chaque option, une valeur "cachée", différente de ce qui est lisible par l'utilisateur.

```
<select id = "menu1" >

<option value = "1" >Le premier choix</option>

<option value = "2" >Le second choix</option>

</select>
```

Lorsque l'utilisateur choisit "Le premier choix", c'est la propriété **value** de l'option correspondante, donc **1**, qui est transmise comme propriété value à l'élément **select**.

Dans le cas où le texte effectivement affiché pour l'option doit être manipulé, celui-ci est accessible via **document.getElementById("menu1").options[0].innerHTML**.

→ Pour un bouton radio ou une case à cocher, l'attribut **value** est utilisé de manière similaire :

```
<input id = "choix1" name = "choix1_et_choix2"
type = "radio" value = "1" />
<label for = "choix1">Le premier choix</label>

<input id = "choix2" name = "choix1_et_choix2"
type = "radio" value = "2" />
<label for = "choix2">Le second choix</label>
```

La propriété **value** est cachée à l'utilisateur et permettra surtout d'identifier l'option sélectionnée :

```
var choix_final = document.querySelector(
'input[name="choix1_et_choix2"]:checked'
).value;
```

On remarque ici un attribut **name**, utilisé dans le cas des boutons radio pour établir un groupe de boutons ; dans ce groupe un seul bouton à la fois pourra être sélectionné par l'utilisateur. L'expression passée ci-dessus à **document.querySelector()** cible tous les éléments **input** dont l'attribut **name** est "choix1_et_choix2" et possédant l'attribut **checked** (ce qui est le cas pour un bouton radio sélectionné ou une case cochée).

L'intitulé effectivement lisible par l'utilisateur est mis en place hors de l'élément **<input>** (idéalement dans un élément **<label>**, "lié" au bouton correspondant via un attribut **for**).

Lorsqu'on manipule une chaîne de caractères, comme c'est le cas pour modifier les attributs **`value`** et **`innerHTML`**, celle-ci est délimitée par des guillemets. Or il arrive que cette chaîne doive également contenir des guillemets. **Cela créera par défaut un conflit, puisque le premier guillemet rencontré par JavaScript dans la chaîne sera compris comme la fermeture de la chaîne.**

```
var chaine1 = "Il dit alors "Bonjour" à tout le monde";
// Occasionnera une erreur.
```

Ici pour JavaScript on définit bien une chaîne "Il dit alors ", mais cette définition est suivie par l'instruction Bonjour" à tout le monde" qui est incompréhensible.

→ Une solution serait d'utiliser la seconde manière admise de délimiter une chaîne : les guillemets simples.

```
var chaine1 = 'Il dit alors "Bonjour" à tout le monde';
```

→ Mais le problème reste entier si la chaîne risque de contenir des guillemets simples (souvent présents en guise d'apostrophes). Pour éviter tout conflit entre les délimiteurs de la chaîne et des caractères qu'elle contient, il s'agira alors d'échapper ces caractères, **en plaçant avant eux un antislash** , signalant qu'ils doivent être **traités comme faisant partie de la chaîne** et non comme délimiteurs :

```
var chaine1 = "Il dit alors \"Bonjour\" à toute l'assemblée";

var chaine2 = 'Il dit alors "Bonjour" à toute l\'assemblée';
```

## • Les mesures effectives d'un élément

Lorsqu'on a besoin de récupérer les dimensions d'un élément (sa hauteur ou sa largeur), il est fortement déconseillé d'utiliser ses attributs **`style.width`** ou **`style.height`**, premièrement parce que ceux-ci ne sont la plupart du temps **pas renseignés**, deuxièmement parce que **les mesures finales de l'élément peuvent dépendre d'autres propriétés** (épaisseur des marges et bordures par exemple) et troisièmement car ils se présentent comme des chaînes de caractères, incluant une unité (ex : "100px"), au lieu de nombres directement manipulables.

Pour obtenir des mesures exactes et exploitables, chaque élément possède un ensemble de propriétés qui correspondront aux mesures désirées, en pixels, de **l'élément tel qu'affiché dans le navigateur**. Celles-ci sont en lecture seule, on ne peut pas les remplacer par d'autres valeurs.

→ **`element.offsetWidth`** : donne la largeur de l'élément

→ **`element.offsetHeight`** : donne la hauteur de l'élément

→ **`element.offsetTop`** : donne la différence entre le bord haut de l'élément et celui de son parent

→ **`element.offsetLeft`** : donne la différence entre le bord gauche de l'élément et celui de son parent

Pour obtenir **les mesures de la fenêtre** dans laquelle est affiché le document, c'est vers l'objet **`Window`** qu'il faut se tourner. Il possède notamment les propriétés **`.innerWidth`** et **`.innerHeight`**, également en lecture seule, donnant respectivement la largeur et la hauteur en pixels de la zone d'affichage de la page (ou *viewport*) dans la fenêtre.

```
var hauteur_de_fenetre = window.innerHeight;

var largeur_de_fenetre = window.innerWidth;
```

# Les événements

Si un script ne pouvait s'exécuter qu'à l'affichage du document HTML, l'intérêt en serait très limité. JavaScript intervient pour rendre la page interactive, or **pour pouvoir parler d'interactivité, il faut être capable de réagir à des actions de l'utilisateur ou des changements dans le document**. En JavaScript, ce sont ces actions et changements que l'on appelle événements (***events***). Ils peuvent être surveillés (***écoutés***) et interceptés ; les instructions à exécuter lorsque c'est le cas peuvent être mises en place de plusieurs manières.

## • En utilisant les méthodes d'événements

Chaque objet du *DOM* possède une série de méthodes, prévues pour se déclencher lorsqu'un événement spécifique est détecté concernant l'objet. On dira alors que cet objet est la cible (***target***) de l'événement.

Par défaut ces méthodes ne contiennent aucune instruction ; lorsqu'un événement nécessite une réaction, il s'agira en quelque sorte de redéfinir la méthode correspondante pour y placer les instructions voulues.

Prenons par exemple un bouton, et déterminons ce qui se passe lorsque l'utilisateur clique dessus. Cela correspondra à l'événement **onclick** de ce bouton, qu'il est possible de placer comme attribut en ligne dans sa balise HTML pour y renseigner des instructions :

```
<input type = "button" id = "btn1" value = "Cliquez-moi"
onclick = "alert(&quot;Merci pour ce clic.&quot;);" />
```

Cet attribut peut également recevoir comme valeur une (ou plusieurs) fonction(s) :

```
<input type = "button" id = "btn1" value = "Cliquez-moi"
onclick = "alert(&quot;Merci pour ce clic.&quot;);
une_fonction(); une_autre_fonction();" />
```

Les instructions mises en place de cette manière s'exécuteront à chaque clic sur le bouton.

Dans les instructions renseignées dans les attributs **onclick** ci-dessus, """ est l'entité HTML correspondant aux guillemets. Elle est nécessaire ici pour échapper ces guillemets, car **en HTML les antislash ne fonctionnent pas comme caractère d'échappement**.

De manière plus propre et afin de désencombrer la structure HTML, une méthode d'événement peut aussi être mise en place via JavaScript :

```
document.getElementById("btn1").onclick = function() {
  alert("Merci pour ce clic.");
  une_fonction();
  une_autre_fonction();
}
```

Comme **document.getElementById("btn1").onclick** doit rester une méthode, la syntaxe acceptée pour y placer des instructions consiste à présenter celles-ci sous forme de fonction. Dans l'exemple ci-dessus on utilise une fonction anonyme.

Il est évidemment possible de faire intervenir une fonction qui a été définie et nommée :

```
function changer_valeur() {
   document.getElementById("btn1").value = "Merci pour ce clic";
}

document.getElementById("btn1").onclick = changer_valeur;
```

Notons l'absence de parenthèses après **changer_valeur** : Il s'agit ici d'assigner la fonction et non de l'exécuter immédiatement. Cette syntaxe empêche néanmoins de passer des paramètres à la fonction **changer_valeur()**. On peut contourner ceci au moyen d'une fonction anonyme pour encapsuler le tout :

```
function changer_valeur(elem) {
   elem.value = "Merci pour ce clic";
}

document.getElementById("btn1").onclick = function() {
   changer_valeur(this);
}
```

Au sein d'une méthode d'un objet, on peut utiliser le mot-clé **this** pour faire référence à l'objet auquel la méthode appartient.

## • En utilisant les écouteurs d'événements

Un écouteur d'événement (*EventListener*) fonctionne presque de la même manière que les méthodes d'événements. La différence est que les écouteurs d'événement peuvent se cumuler et se manipuler un par un même pour un événement commun ; **ils peuvent être ajoutés ou supprimés dynamiquement au besoin**.

En reprenant l'exemple précédent, l'ajout d'instructions à exécuter lors du clic sur le bouton peut s'écrire :

```
document.getElementById("btn1").addEventListener(
   "click" , function() { changer_valeur(this); }
);
```

La méthode **addEventListener** attend en paramètres le type d'événement à écouter (on remarque que le préfixe "on" n'est plus utilisé ici) suivi de la fonction à exécuter. Comme dans l'exemple précédent, si cette fonction ne prévoit pas de paramètres, son nom peut être utilisé, sans parenthèses.

```
document.getElementById("btn1").addEventListener(
   "click" , changer_valeur
);
```

Il est ensuite possible d'ajouter d'autres écouteurs au même élément pour le même événement, ce que ne permettent pas les méthodes d'événements vues précédemment (pour un élément, une seule méthode est disponible et peut être définie par événement).

```
document.getElementById("btn1").addEventListener(
   "click" , une_autre_fonction
);

document.getElementById("btn1").addEventListener(
   "click" , function() { encore_une_autre_fonction(param) }
);
```

Pour retirer à l'élément l'un de ses écouteurs, il faut utiliser **elem.removeEventListener()**, avec des paramètres identiques à ceux passés lors de l'ajout de cet écouteur. Ainsi, pour retirer de notre bouton le second écouteur qui y avait été ajouté :

```
document.getElementById("btn1").removeEventListener(
   "click" , une_autre_fonction
);
```

## • Autre exemple d'événement : réagir au chargement de la page

Un événement assez couramment utilisé est **onload**, le plus souvent appelé pour l'élément **body**. Les instructions placées dans **document.body.onload()** seront exécutées lorsque la page et ses contenus liés auront terminé de se charger.

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

ou

```
document.body.onload = fonction_pour_quand_chargement_ok;
```

ou

```
document.body.addEventListener(
   "load", fonction_pour_quand_chargement_ok
);
```

Au moment du déclenchement de **onload** (pour **body**, **document** ou **window**), il est garanti que toutes les images (souvent les éléments les plus lourds dans la page) seront en place, ainsi que les styles CSS et scripts liés. Tout le *DOM* sera également accessible à JavaScript.

## • Récupérer des données liées à un événement

Pour certains types d'événements, on peut avoir besoin de quelques détails sur l'action détectée. C'est par exemple le cas de **onkeypress**, **onkeydown** et **onkeyup**, déclenchés respectivement lorsqu'une touche du clavier est pressée, tant qu'elle le reste, et lorsqu'elle est relachée : la plupart du temps il sera nécessaire de savoir quelle touche exactement a été actionnée.

Lorsqu'il est passé en paramètre à une fonction que l'on associe à un événement, le mot-clé **event** pourra y être utilisé pour faire référence à l'action détectée et renverra un objet listant les propriétés et valeurs que nous cherchons. Analysons ce qui compose un objet événement pour **onkeypress** :

```
document.body.onkeypress = function(event) {
  console.log(event);
}
```

ou

```
function afficher_evt(e) { console.log(e) }
// Lors du déclenchement de l'événement, l'objet sera
automatiquement passé en paramètre à afficher_evt()

document.body.onkeypress = afficher_evt;
// Ce passage de paramètre est implicite, il n'a pas besoin de
figurer ici
```

La console affiche quelques propriétés intéressantes, comme **target** (contenant ici une référence à l'élément **body** puisqu'il est la cible de l'événement) et surtout **key**, indiquant précisément quelle touche a déclenché l'événement.

On peut en déduire comment, par exemple, exécuter des instructions uniquement si la barre d'espace est pressée. Pour cette touche la console nous indique que l'attribut **event.key** correspond à " ", donc :

```
document.body.onkeypress = function(e) {
  if (e.key == " ") {
     // C'est bien la barre d'espace qui est pressée
  }
}
```

Le même principe s'applique à d'autres types d'événements :

→ **onscroll** est déclenché au défilement dans un élément (*scrollbar*, molette de la souris…). On trouve dans l'objet événement correspondant les attributs **pageX** et **pageY**, déterminant de combien de pixels l'élément a défilé horizontalement et verticalement. On peut par exemple contrôler où en est le défilement de la page et agir en conséquence :

```
document.body.onscroll = function(e) {
  if (e.pageY >= window.innerHeight) {
    // La page a défilé d'une hauteur de fenêtre ou plus
  } else {
    // La page a défilé de moins d'une hauteur de fenêtre
  }
}
```

→ **onmousemove** est déclenché au mouvement du curseur au-dessus d'un élément. On trouve dans l'objet événement correspondant les attributs **clientX** et **clientY**, déterminant les coordonnées de la position du curseur par rapport au coin supérieur gauche de la fenêtre. Dans l'exemple suivant, l'élément **div** suivra les mouvements du curseur :

```
<style>

body {
position:absolute; top:0; left:0; width:100%; height:100%
}

#stalker {
position:absolute;
width:40px; height:40px;
margin-top:-20px; margin-left:-20px;
background-color:grey; border-radius:50%
}

</style>

<div id = "stalker"></div>

<script>

var stalker = document.getElementById("stalker");

function stalk(e) {

stalker.style.top = e.clientY;
stalker.style.left = e.clientX;

}

</script>
```