Avec les événements et la manipulation du DOM, on peut aller plus loin pour rendre une page web plus dynamique.

Les événements

Un **événement** est une sorte de notifications qui se lance lors d'une action sur la page. Parmi ces actions, on peut par exemple retrouver :

- l'utilisateur clique quelque part ;
- l'utilisateur appuie sur une touche du clavier ;
- l'utilisateur survole un élément de la page ;
- la page s'est chargée...

Gestion événement dans le HTML : gérer les événements en ligne

On l'a utilisé sans l'expliquer dans les TPs, mais il existe des attributs qu'on peut donner aux éléments HTML auxquels on donne le code qui doit se lancer quand un événement précis s'est déclenché sur cet élément. C'est parce que ces événements sont définis dans le HTML directement qu'on les appelle des **événements en ligne**.

Par exemple, dans le code suivant j'ai un bouton et je lui donne l'attribut `onclick` et je lui donne du code :

```
<html>
	<head>
		<title>Exemple onclick sur un bouton=</title>
	</head>
	<body>
		<button onclick="console.log('CLICK');">Cliquez pour
afficher CLICK</button>
	</body>
</html>
```

Cela veut dire que quand l'utilisateur clique sur le bouton, dans la console va s'afficher CLICK.

Un des autres événements qu'on a déjà croisés, on retrouve aussi `onload` à qui on donne le code à exécuter au chargement de l'élément.

Généralement, on l'utilise sur la balise `<body>` pour déterminer toutes les choses à faire quand l'utilisateur arrive sur la page.

Par exemple, dans le TP3.2, on avait initialisé le tableau avec des valeurs aléatoires au chargement de la page :

```
<html>
  <head>
        <title>Exemple onload</title>
  </head>
  <body onload="init_tableau();">
    <div id="tableau-div"></div>
    <button onclick="trier_tableau();">Cliquez-ici pour trier le
tableau</button>
  </body>
</html>
```

> **Remarque sur les événements en ligne**
>
> Avec cette méthode, le HTML peut vite devenir illisible, et difficile à maintenir. Leur utilisation est donc déconseillée, à part pour effectuer des tests rapides par exemple.

Gestion des événements dans le JS

On a déjà vu dans le cours précédent comment récupérer un élément dans le DOM. À chacun de ces éléments, il se trouve que l'on peut accéder son attribut `onclick` à qui on peut donner une fonction avec le code qu'on veut exécuté au déclenchement de l'événement sur l'élément en question.

On peut reprendre le premier exemple et le réécrire avec cette nouvelle méthode.
On a ce fichier HTML un peu plus minimaliste :

```
<html>
        <head>
                <title>Exemple onclick sur un bouton</title>
                <script src="code.js"></script>
        </head>
        <body>
                <button id="bouton">Cliquez pour afficher CLICK</button>
        </body>
</html>
```

et dans un fichier JS :

```
let bouton = document.getElementById("bouton");
```

```
bouton.onclick = () => {
        console.log("CLICK");
}
```

Si la notation de cet exemple vous embrouille, retournez voir la partie Fonction anonyme du cours précédent.

On pourrait aussi très bien donner une fonction qui a été définie avec un nom, comme par exemple :

```
let bouton = document.getElementById("bouton");

function afficherCLICK() {
        console.log("CLICK");
}

bouton.onclick = afficherCLICK;
```

Remarque importante

Vous noterez que dans l'exemple ci-dessus, on a donné `afficherCLICK` et pas `afficherCLICK()` avec les parenthèses. La différence est :

- `afficherCLICK` : c'est une *référence* vers la fonction, le code à l'intérieur n'est pas exécuté ;
- `afficherCLICK()` : c'est un appel à la fonction, on donnerait donc à la variable la valeur de retour de la fonction.

La propriété `onclick` attend donc une valeur de type **fonction**.

La fonction `addEventListener`

Quand on donne une action à faire lors d'un événement, on dit qu'on *écoute* les événements sur cet élément. C'est pour ça qu'on a introduit le principe de *Event Listener* (littéralement *écouteur d'événement* en français).

Le principe est le même avec cette fonction que les méthodes précédents mais sous une autre forme.

La fonction `addEventListener()` prend 2 arguments :

1. le nom de l'événement à écouter ;
2. la fonction à exécuter au déclenchement de l'événement.

Si on reprend l'exemple du bouton, on peut réécrire avec cette nouvelle méthode :

```
let bouton = document.getElementById("bouton");

bouton.addEventListener("click", () => {
        console.log("CLICK");
});
```

L'avantage de cette méthode est que on peut, si besoin, se débarrasser de ce *listener* avec la fonction `removeEventListener`.

Disons par exemple qu'au bout de 3 clics sur le bouton on ne veut plus afficher "CLICK" sur la console, alors on compte les clics et quand on arrive à 3, on se débarasse du *listener* :

```
let bouton = document.getElementById("bouton");
let count = 0;

function afficher_3_clicks() {
        if (count < 3) {
                console.log("CLICK");
                count++;
        } else {
                bouton.removeEventListener("click", afficher_3_clicks);
        }
}

bouton.addEventListener("click", afficher_3_clicks);
```

Remarque

Pour utiliser `removeEventListener`, j'ai besoin de lui donner la fonction qui correspond à l'écouteur que j'ai donné avec `addEventListener`. Dans ce cas, on ne peut donc pas utiliser de fonction anonyme si on a prévu de s'en débarrasser.

Ceci s'explique parce qu'en réalité on peut appeler `addEventListener` plusieurs fois sur le même élément et le même événement mais avec des fonctions différentes. Il faut préciser donc quel écouteur on veut enlever.

Les événements qui existent

Pour les curieux et les curieuses, vous pouvez jeter un oeil aux événements qu'on peut prendre en charge dans la page de documentation suivante :

https://developer.mozilla.org/fr/docs/Web/Events

Manipuler le DOM

Jusqu'à maintenant, pour ajouter ou enlever des éléments dans le HTML on a simplement ajouté du code HTML à injecter dans le fichier original. Néanmoins, Javascript nous propose un petit ensemble de fonction pour manipuler le HTML en touchant directement au DOM.

Pour toucher au DOM, il faut cependant comprendre un peu mieux sa structure, c'est-à-dire étudier de plus près la structure d'arbre.

Définitions

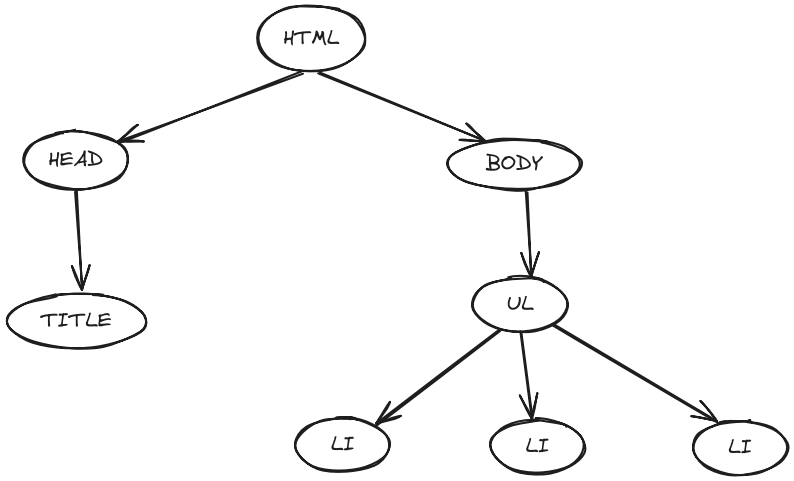
Un arbre est une structure de donnée hiérarchique dont chaque élément est appelé un **noeud** (ou **node** en anglais).

On définit le lien entre des noeuds avec des arêtes, ce sont donc ces arêtes qui définissent la hiérarchie dans l'arbre.

Soit la liste en HTML suivante :

```
<html>
        <head>
                <title>Exemple arbre DOM</title>
        </head>
        <body>
                <ul>
                        <li>Premier item</li>
                        <li>Deuxième item</li>
                        <li>Troisième item</li>
                </ul>
        </body>
</html>
```

Sous forme d'arbre, on obtient :



Racine d'un arbre (ou *root*)

La racine d'un arbre est le noeud le plus haut dans l'arbre.

Dans notre exemple, la racine de l'arbre est le noeud de la balise `<html>`.

Enfant (ou *child*)

S'il y a une ou plusieurs arêtes qui sortent d'un noeud, alors les noeuds à l'autre bout de chaque arête sont ses enfants.

Ici :

- les enfants de `<html>` : `<head>` et `<body>` ;
- l'enfant de `<head>` : `<title>` ;
- l'enfant de `<body>` : `<ul>`;
- les enfants de `<ul>` : les trois `<li>` ;
- il n'y a pas d'arêtes qui sortent des `<li>` ils n'ont donc pas d'enfants.

Parent

À l'inverse, si un noeud A a un enfant B, alors on dit que le parent de B est A.

Quelques exemples :

- le parent de tous les `<li>` est `<ul>` ;
- le parent de `<ul>` est `<body>` ;

Frères et soeurs (*sibling*)

Si plusieurs noeuds ont le même parent, alors on dit qu'il sont frères et soeurs, ou *sibling* en anglais.

Par ex :

- les trois `<li>` sont siblings ;
- `<head>` et `<body>` sont siblings.

> **Remarque**
>
> Ça peut paraître inintéressant de s'intéresser à ces termes parce que ça serait que de la théorie... Néanmoins, ce sont des termes qui reviennent souvent quand vous travaillez sur le DOM, c'est donc indispensable de se familiariser avec ces termes pour comprendre la documentation.

Les fonctions de manipulation du DOM

Pour expliquer les fonctions disponibles pour manipuler le DOM, on va reprendre l'exercice 2 du TP3 où on voulait écrire la table de 2.

Initialement, on avait juste une page avec une `<div>` où insérer la table:

```
<html>
  <head>
    <script type="text/javascript" src="table-de-2.js"></script>
    <title>La table de deux</title>
  </head>
  <body onload="creer_table()">
    <div id="contenu"></div>
  </body>
</html>
```

et on avait écrit le script suivant pour créer la table :

```
function creer_table() {
        let div = document.getElementById("contenu");

        let str = "<table id='t1'>";

        for (let i=1; i <= 10; i++) {
                str += "<tr><td>" + i + "</td><td>" + 2*i + "</td>
</tr>";
        }

        str += "</table>";
```

```
        div.innerHTML = str;
}
```

Lors de cet exercice, vous avez sûrement trouvé ça pénible de devoir écrire des chaînes de caractères avec du HTML... On va voir une méthode moins pénible à implémenter, mais dans laquelle il faut un peu réfléchir.

Créer un élément : `document.createElement`

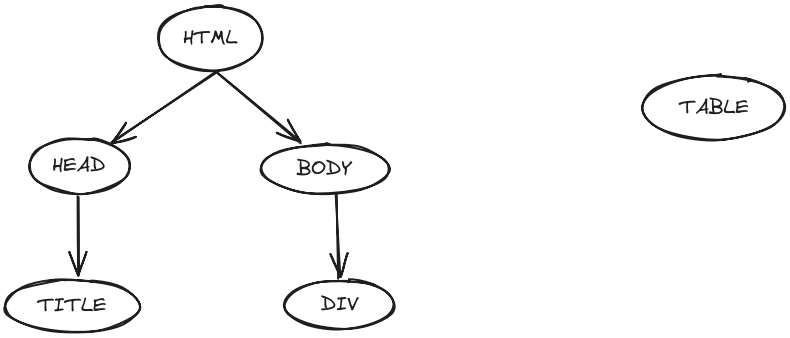
Pour créer un noeud, on utilise la fonction `document.createElement` qui prend en argument le nom de la balise qu'on veut créer, et qui renvoie le noeud créé.

Par exemple, pour créer le noeud table :

```
let table = document.createElement("table");
table.id = "t1";
```

Insérer un élément dans le DOM : `appendChild`

Pour l'instant j'ai, dans mon DOM, l'arbre avec ce qu'on a défini dans le fichier HTML, et un noeud `<table>` qui n'est relié à rien :



La prochaine étape est donc d'insérer ce noeud dans le DOM.

Pour ce faire, pour chaque élément, on a la fonction `appendChild` qui prend un élément en argument et qui l'ajoute comme enfant.

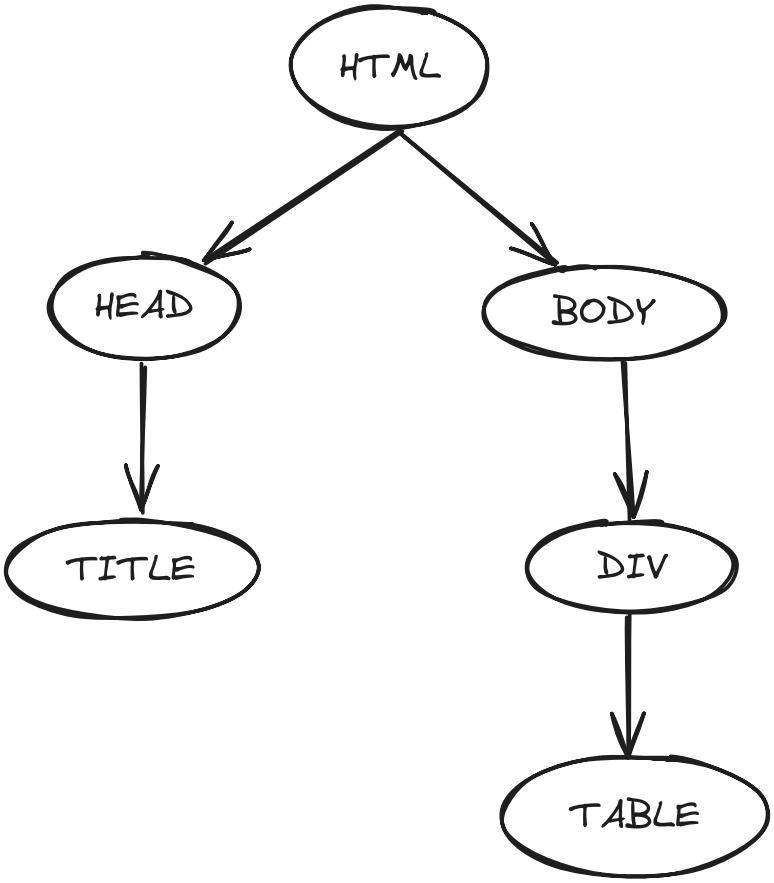
La question à se poser est donc, où est-ce que je veux placer mon noeud ?

Dans notre exemple, on veut que la table soit un enfant de `<div>`. Il faut donc d'abord récupérer l'élément qui lui correspond, et on applique `appendChild` avec en argument l'élément qu'on a conservé dans la variable `table` :

```
let div = document.getElementById("contenu");
```

```
div.appendChild(table);
```

et maintenant on a :



Créer un noeud de texte : `document.createTextNode`

Pour les noeuds contenant juste du texte, on doit créer des noeuds spécifiques avec `document.createTextNode` qui prend pour argument le texte à mettre dedans.

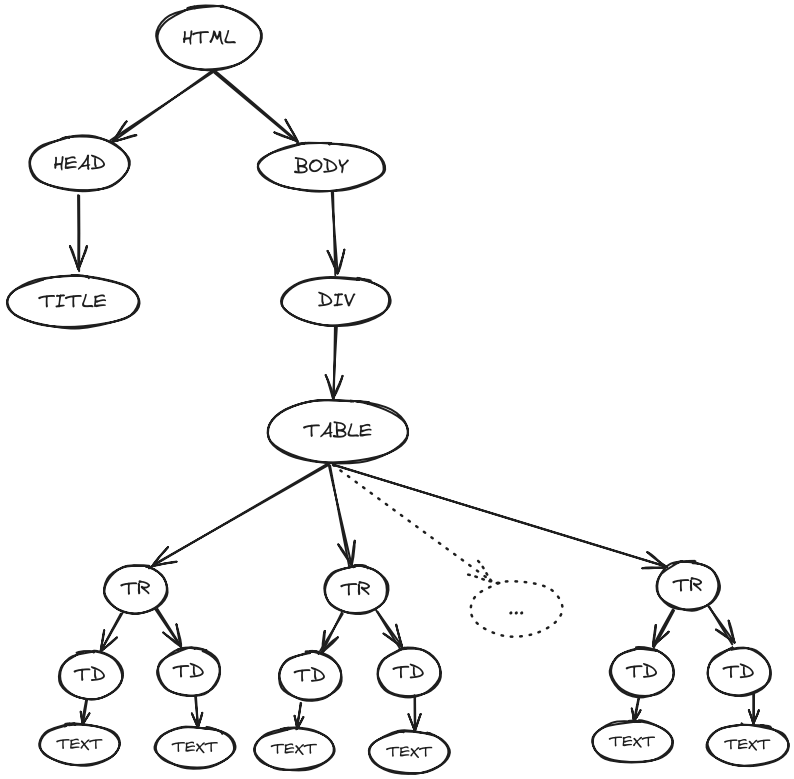
Par exemple, dans notre exemple dans quel noeud se trouve le texte ? Dans les élément `<td>`. Donc il faut créer un enfant de type texte à chaque `<td>`.

On a tout ce qu'il faut maintenant pour continuer d'écrire la table :

```
for (let i=1; i <= 10; i++) {
        // à chaque itération, je crée un tr puis je l'ajoute à table
        let tr = document.createElement("tr");
        table.appendChild(tr);

        // je crée aussi 2 td
        let td_gauche = document.createElement("td");
        let td_droite = document.createElement("td");

        // et j'ajoute les deux td à tr, en commençant par la cellule de
```

```
gauche :
        tr.appendChild(td_gauche);
        tr.appendChild(td_droite);

        // pour finir, on n'oublie pas d'ajouter les noeuds de texte
        // on ne va pas les réutiliser, donc pas besoin de les stocker
dans une variable
        td_gauche.appendChild(document.createTexteNode(i));
        td_droite.appendChild(document.createTexteNode(2*i));
}
```

On a construit l'arbre suivant :



et on retrouve bien la même structure que dans notre ancienne méthode.

Récupérer tous les enfants : `childNodes`

On peut avoir besoin de parcourir tous les enfants d'un même noeud. Pour cela, on les récupère avec la propriété `childNodes` qui renvoie la liste des enfants du noeud.

Par exemple, on pourrait parcourir tous les éléments `<tr>` :

```
let list_tr = table.childNodes;

for (let i=0; i < list_tr.length; i++) {
```

```
        console.log(list_tr[i]);
}
```

Supprimer un enfant : `removeChild`

Si on décide qu'on veut enlever un enfant, on utilise la fonction `removeChild` sur l'élément parent de l'enfant qu'on veut supprimer.

Par exemple, disons qu'on veut supprimer la dernière ligne. Comme en CSS, il existe des propriétés `lastChild` et `firstChild` qui peuvent nous éviter de parcourir les enfants pour retrouver le dernier. On a donc :

```
table.removeChild(table.lastChild);
```

Retrouver le parent d'un noeud : `parentNode`

Disons que l'on veut que quand on clique sur une ligne du tableau, alors cette ligne est modifiée.

Pour faire cela, je vais créer un événement sur tous les éléments de type `<tr>`. Néanmoins, pour enlever une ligne d'une table, je dois agir sur la table.

Par chance, si je connais le noeud qui contient le `<tr>`, je peux récupérer son parent avec la propriété `parentNode`.