

Développement Web

Julien Provillard

PROGRAMMATION ASYNCHRONE


Tempus fugit

- ❑ En programmation Web, il faut prendre en compte la dimension temporelle des applications, par exemple, pour :
 - Délayer des actions,
 - Réaliser des animations,
 - Attendre la réponse d'un serveur distant pour traiter les données reçues,
 - ...
- ❑ On parle de programmation asynchrone quand des éléments du programme s'exécutent indépendamment du flux d'exécution principal.

Fonctions de rappel (callback)

- La manière la plus simple d'introduire de l'asynchronie est d'utiliser une fonction qui :
 - Ne bloque pas le programme,
 - S'exécute en tâche de fond,
 - Passe son résultat à une fonction de rappel pour poursuivre le calcul.

setTimeout et setInterval

- ❑ Ce sont les deux fonctions les plus simples pour réaliser des appels asynchrones.
- ❑ `setTimeout(f, delay, [arg1, ..., argN])` appelle la fonction `f` avec les arguments `arg1, ..., argN` après un temps de `delay` ms.
- ❑ `setInterval(f, delay, [arg1, ..., argN])` appelle la fonction `f` avec les arguments `arg1, ..., argN` après tous les `delay` ms.
- ❑ Les deux fonctions renvoient un identifiant qui peut servir à annuler l'action programmée via `clearTimeout` ou `clearInterval` respectivement.
- ❑ Exemple: 
alarm.html

setTimeout et setInterval

- ❑ L'appel programmé ne s'exécute qu'après le flot d'exécution principale.

Exemple :

```
setTimeout(() => alert("foo")); // délais = 0ms  
alert("bar"); // affiche bar puis foo
```

Autre exemple :

```
setTimeout(() => alert("foo")); // délais = 0ms  
while (true); // boucle et n'affiche rien
```

- ❑ Autrement dit, tout code en cours d'exécution doit se terminer avant que les appels en attente ne se déclenchent, même si cela retarde les délais théoriques.

Fonctions de rappel

- ❑ Nous disposons de données stockées dans une variable data.

```
const data = { /* ... */ };
```

- ❑ On cherche à mimer un accès distant. On interdit donc d'écrire data[key]. À la place, nous allons définir une fonction qui crée volontairement de la latence.

```
function getData(key) {  
  const latence = 5 + Math.floor(Math.random() * 10); // entre 5 et 15 ms  
  setTimeout(???, latence); // mime un appel distant  
}
```

- ❑ La fonction ne peut pas renvoyer la valeur immédiatement.
- ❑ Que faut-il faire ?

Fonctions de rappel

- ❑ Nous disposons de données stockées dans une variable data.

```
const data = { /* ... */ };
```

- ❑ On cherche à mimer un accès distant. On interdit donc d'écrire data[key]. À la place, nous allons définir une fonction qui crée volontairement de la latence.

```
function getData(key, callback) { // callback = calcul à faire avec data[key]
  const latence = 5 + Math.floor(Math.random() * 10); // entre 5 et 15 ms
  setTimeout(callback, latence, data[key]); // mime un appel distant
}
```

- ❑ Les fonctions de rappels permettent la poursuite du calcul quand toutes les conditions nécessaires sont présentes.

Problème des fonctions de rappel

❑ Nous souhaitons réaliser un traitement f sur les données de clés "a", "b" et "c".

❑ Que pensez-vous du code suivant ?

```
getData("a", // accès à la donnée "a"  
  (a) => getData("b", // puis accès à la donnée "b"  
    (b) => getData("c", // puis accès à la donnée "c"  
      (c) => f(a, b, c))))); // traitement véritable
```

❑ Vous allez voir, ça peut être encore pire !

Fonctions de rappel et erreurs

```
try { setTimeout(() => null.a) } // produit une erreur  
catch (err) { alert("Erreur détectée"); }
```

- ❑ L'erreur n'est pas attrapée car `setTimeout` s'exécute sans problème. L'erreur survient plus tard dans un flot d'exécution différent.

- ❑ Il aurait fallut écrire :

```
setTimeout(() => { try { null.a; }  
                  catch (err) { alert("Erreur détectée"); }});
```

- ❑ Usuellement, on définit les fonctions de rappel avec deux arguments.
 - On appelle `callback(error)` si `error` se produit.
 - Sinon on appelle `callback(null, result)`.

Fonctions de rappel et erreurs

❑ La fonction `getData` devrait donc s'écrire :

```
function getData(key, callback) {  
  const latence = 5 + Math.floor(Math.random() * 10);  
  setTimeout(() => {  
    if (key in data) { // test de l'existence de la donnée  
      callback(null, data[key]); // rappel donnée présente  
    } else {  
      callback(new Error(`${key} not found`)); // rappel donnée absente  
    }  
  }, latence);  
}
```

Fonctions de rappel et erreurs

❏ Et le test :

```
getData("a", (error, a) => {  
  if (error) {  
    alert("error on a");  
  } else {  
    getData("b", (error, b) => {  
      if (error) {  
        alert("error on b");  
      } else {  
        getData("c", (error, c) => {  
          if (error) {  
            alert("error on c");  
          } else {  
            f(a, b, c);  
          }  
        });  
      }  
    });  
  }  
}); // le compte est bon
```

Fonctions de rappel et erreurs

❑ Variante en décomposant les appels :

```
const args = [];  
const push = (result) => args.push(result);  
getData("a", step1);
```

```
function step1(error, a) {  
  if (error) { alert("error on a"); } else { push(a); getData("b", step2); }  
function step2(error, b) {  
  if (error) { alert("error on b"); } else { push(b); getData("c", step3); }  
function step3(error, c) {  
  if (error) { alert("error on c"); } else { push(c); f(...args); }  
}
```

❑ Y a-t-on vraiment gagné ?

Fonctions de rappel et erreurs

❑ Autre solution :

```
function getDataExt(keys, callback) {  
  const args = [];  
  process();  
  function process() {  
    if (args.length == keys.length) { callback(null, args); } else {  
      getData(keys[args.length], (err, res) => {  
        if (err) { callback(err); } else {  
          args.push(res);  
          process();  
        }  
      });  
    }  
  }  
}
```

❑ Trop spécialisée.

Promesses

- ❑ Une promesse est un objet qui encapsule un calcul asynchrone et rend son résultat disponible quand celui-ci s'achève.
- ❑ Une promesse peut-être dans trois états :
 - En cours : le calcul n'est pas terminé,
 - Réalisée : le calcul est un succès et le résultat est disponible,
 - Rejetée : le calcul a rencontré une erreur que l'on peut essayer de récupérer.
- ❑ Une fois dans l'un des états réalisé ou rejeté, une promesse est fixe.

Promesses : création

- ❑ Pour créer une nouvelle promesse, on utilise le constructeur `Promise` avec une fonction en argument.

```
const promesse = new Promise((resolve, reject) => { /* ... */})
```

- ❑ Les paramètres `resolve` et `reject` sont automatiquement liés à des fonctions internes de gestion des promesses.
- ❑ Dans le corps de la fonction, tout appel de la forme `resolve(result)` passe la promesse dans l'état réalisé avec pour résultat `result`.
- ❑ Dans le corps de la fonction, tout appel de la forme `reject(error)` passe la promesse dans l'état rejeté avec pour erreur `error`.

Promesses : création

```
function getData(key) {  
  return new Promise((resolve, reject) => {  
    if (key in data) { resolve(data[key]); }  
    else { reject(new Error("key not found")); }  
  });  
}
```

❑ Et avec la latence :

```
function getData(key) {  
  const latence = 5 + Math.floor(Math.random() * 10);  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      if (key in data) { resolve(data[key]); }  
      else { reject(new Error("key not found")); }  
    }, latence);  
  });  
}
```

Promesses : utilisation

- ❑ On peut utiliser le résultat d'une promesse à l'aide de la méthode then.

promesse.then(
 (result) => { /* si la promesse est réalisée, exploite le résultat */ },
 (error) => { /* si la promesse est rejetée, rattrape l'erreur */ })

- ❑ Les fonctions de rappel sont exécutées dès que possible selon l'état de la promesse.

- ❑ Si les fonctions de rappel

- renvoient une promesse, c'est la valeur de retour du then ;
- renvoient une valeur, then renvoie une promesse réalisée sur cette valeur ;
- lancent une erreur, then renvoie une promesse rejetée sur cette erreur.

Promesses : utilisation

```
getData("foo") // promesse initiale
  .then(
    (result) => `La valeur de "foo" est : ${result}`,
    (error) => error.message
  ) // nouvelle promesse sur une valeur chaîne
  .then(alert); // on ignore le cas d'erreur
```

- ❑ La méthode `then` peut être appelée avec des arguments non définis. On ignore alors la valeur ou l'erreur selon les cas.
- ❑ En particulier `p.catch(f)` est un raccourci pour `p.then(null, f)`.
- ❑ Il existe aussi une méthode `finally` dont le sens devrait être clair.

Promesses : reprise de l'exemple

```
function errorOn(name) { throw new Error(`error on ${name}`); }
```

```
const args = [];
```

```
const push = (result) => args.push(result);
```

```
Promise.resolve() // <=> new Promise((resolve) => resolve())
```

```
  .then(() => getData("a").then(push, () => errorOn("a")))
```

```
  .then(() => getData("b").then(push, () => errorOn("b")))
```

```
  .then(() => getData("c").then(push, () => errorOn("c")))
```

```
  .then(() => f(...args))
```

```
  .catch((err) => alert(err.message));
```

Promesses : reprise de l'exemple

❑ C'est en fait un cas d'usage courant. L'API permet de faire mieux.

```
function errorOn(name) { throw new Error(`error on ${name}`); }
```

```
const promises = [  
  getData("a").catch(() => errorOn("a")),  
  getData("b").catch(() => errorOn("b")),  
  getData("c").catch(() => errorOn("c")),  
];
```

```
Promise.all(promises) // les promesses sont gérées en parallèle  
  .then((args) => f(...args))  
  .catch((err) => alert(err.message));
```

Fonctions asynchrones !

- ❑ On peut déclarer une fonction asynchrone via le mot-clé `async`.
- ❑ Une fonction asynchrone renvoie toujours une promesse quitte à encapsuler son résultat (de manière similaire à la méthode `then`).

```
async function f() {  
    return 21;  
}  
const n = f();  
alert(typeof n); // object  
n.then((val) => alert(2 * val)); // on vérifie que n est bien une promesse
```

Mot clé `await`

- ❑ L'expression `await` ne peut être utilisée que dans le corps d'une fonction asynchrone.
- ❑ Passer une promesse à `await` permet d'en attendre la résolution et d'en extraire la valeur.

```
async function f() {  
    return 21;  
}  
  
(async () => { // placement artificiel dans un environnement async  
    const n = await f();  
    alert(typeof n); // number  
    alert(2 * n); // on vérifie que n est bien un entier  
})();
```

Fonctions asynchrones

❏ Reprise de l'exemple fleuve

```
async function process() {  
  try {  
    const a = await getData("a").catch(() => errorOn("a"));  
    const b = await getData("b").catch(() => errorOn("b"));  
    const c = await getData("c").catch(() => errorOn("c"));  
    f(a, b, c);  
  } catch (err) {  
    alert(err.message);  
  }  
}  
process();
```


Fonctions asynchrones

❑ Dans un module, `async` peut même être utilisé au top-level.

```
try {  
  const a = await getData("a").catch(() => errorOn("a"));  
  const b = await getData("b").catch(() => errorOn("b"));  
  const c = await getData("c").catch(() => errorOn("c"));  
  f(a, b, c);  
} catch (err) {  
  alert(err.message);  
}
```

❑ Le code obtenu s'écrit alors de manière très similaire à du code synchrone.

LA PAGE WEB

Environnement

□ On suppose ici que :

- Vous avez les connaissances de base pour la création statique de document html.
- Vous savez modifier le rendu d'un tel document via l'utilisation de feuilles de style en cascade (css) et de classes dans les balises.

□ On travaillera exclusivement depuis une page Web.

Modèles

- ❑ Un fichier html définit un arbre dans une grammaire normalisée.
- ❑ Javascript modélise cet arbre pour pouvoir le manipuler. Cette représentation constitue le DOM (Document Object Model).
- ❑ L'arbre s'appelle `document` et sa racine est `document.documentElement`.
- ❑ On travaille plus souvent avec le sous-arbre qui commence à la balise `<body>`. La variable associée est `document.body`.
- ❑ Pour travailler dans l'entête, on utilise `document.head`.
- ❑ En plus du DOM, il existe le BOM (Browser Object Model) qui donne des méta-informations (navigateur, URL, ...).
- ❑ On peut visualiser le DOM de la page courante dans les navigateurs.

Navigation dans le DOM

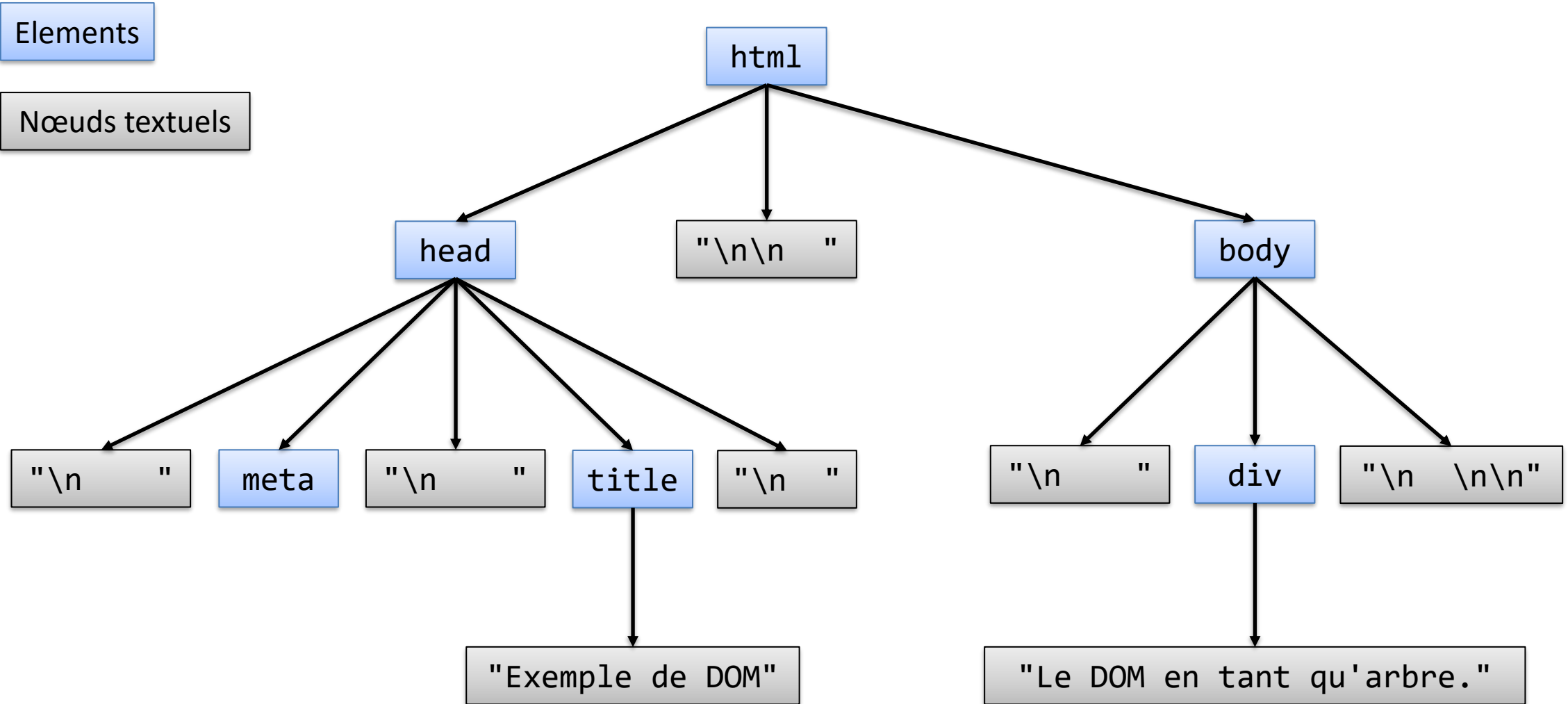
- ❑ Le DOM est composé de nœuds :
 - des nœuds éléments correspondant aux balises html (<html>, <div>, , ...),
 - des nœuds textuels (texte brut ou commentaires).
- ❑ On liste ici les propriétés de navigation disponibles, leur nom est souvent suffisant pour comprendre leur fonctionnalité :
 - Pour tous les nœuds : parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.
 - Pour les éléments uniquement : parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.

Navigation dans le DOM

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Exemple de DOM</title>
  </head>

  <body>
    <div>Le DOM en tant qu'arbre.</div>
  </body>
</html>
```

Navigation dans le DOM



Navigation dans le DOM

☐ Exemples

```
const root = document.documentElement; // le tag html
const level2 = Array.from(root.children); // from iterable to array

alert(level2.length == 2); // true
alert(level2[0] == document.head); // true
alert(level2[1] == document.body); // true
alert(document.head.nextSibling == document.body); // false
alert(document.head.nextElementSibling == document.body); // true
```


Navigation dans le DOM

- ❑ Si une des propriétés vaut `null`, le parent/frère/enfant n'existe pas.
- ❑ Les propriétés `childNodes` et `children` renvoient des collections :
 - itérables (propriété `length`, accès par indice, support des boucles `for of`),
 - vivantes (l'ajout/retrait d'enfants se répercute dynamiquement),
 - en lecture seule.
- ❑ On peut transformer n'importe quel itérable en tableau. On perd cependant l'aspect vivant (snapshot).

```
const childrenArray = Array.from(element.children);
```

Cas particulier des tables

- ❑ Les tables sont utilisées pour présenter des données sous forme matricielle. Pour faciliter leur manipulation,
 - chaque table a une propriété `rows` qui renvoie la collection des lignes,
 - chaque ligne a une propriété `cells` qui renvoie la collection des cellules sur la ligne et une propriété `rowIndex` qui donne son numéro de ligne,
 - chaque cellule a une propriété `cellIndex` qui donne sa position dans la ligne.

❑ Exemple : 
table.html

Accès direct

- ❑ Pour accéder à un élément d'identifiant `"someId"`, on peut appeler la méthode `document.getElementById("someId")`.
- ❑ L'appel `elem.querySelectorAll(selector)` renvoie la collection (statique) des éléments du sous-arbre de racine `elem` vérifiant le sélecteur CSS `selector`.
- ❑ L'appel `elem.querySelector(selector)` renvoie le premier élément de cette collection. En particulier,
 - `elem.querySelector(selector) === elem.querySelectorAll(selector)[0]`
 - `document.querySelector("#someId") === document.getElementById("someId")`
- ❑ L'appel `elem.closest(selector)` renvoie le plus proche parent de `elem` vérifiant le sélecteur CSS `selector`.

Propriétés importantes

- ❑ `innerHTML` (éléments) : le contenu de l'élément en incluant les balises. En modifiant cette propriété, les balises sont interprétées pour créer de nouveaux nœuds.
- ❑ `outerHtml` (élément) : comme `innerHTML` mais inclus l'élément lui-même. En modifiant cette propriété, un nouveau nœud est créé pour remplacer l'élément courant dans le DOM. L'élément courant est détaché du document mais est inchangé pour tous les autres aspects.
- ❑ `data` (nœud textuel) : le contenu du nœud.
- ❑ `textContent` (tous les nœuds) : le contenu textuel du nœud, ignore les balises.

Propriétés du DOM vs attributs HTML

- ❑ Les nœuds du DOM sont des objets Javascript, on peut leur associer n'importe quelle propriété.
- ❑ Les attributs **standards** HTML ont une propriété associée (même nom) dans la représentation DOM de l'élément qui les contient. La valeur de la propriété et de l'attribut sont synchronisés dans la plupart des cas.
- ❑ Dans le DOM, on manipule les propriétés **non-standards** via les méthodes :
 - `elem.hasAttribute(name)`
 - `elem.getAttribute(name)`
 - `elem.setAttribute(name, value)`
 - `elem.removeAttribute(name)`

Attributs non-standards réservés

- ❑ Tout attribut commençant par `"data-"` est réservé pour les besoins de programmation. Il est assuré qu'il ne sera jamais utilisé par le standard.
- ❑ De plus, si un nœud `elem` du DOM représente une balise dotée d'un attribut `"data-someId"`, on peut y accéder par `elem.dataset.someId`.

```
<div id="myData" data-important="My awesome data">Rien</div>  
<script>  
  const elem = document.querySelector("#myData");  
  alert(elem.dataset.important); // My awesome data  
</script>
```

Création, ajout, suppression

- ❑ On peut créer un nouvel **élément** à l'aide de la méthode `document.createElement(tag)`.

```
const div = document.createElement("div");
```

- ❑ On peut créer un nouveau **nœud textuel** à l'aide de la méthode `document.createTextNode(text)`.

```
const textNode = document.createTextNode("Rarement utilisé");
```

- ❑ En règle général, on utilise rarement la création de nœud textuel. Ceux-ci sont implicitement créés par d'autres méthodes.

Création, ajout, suppression

- ❑ On peut ajouter des nœuds dans le DOM par rapport à un nœud via les appels suivants :
 - `elem.append(...values)` // ajout à la fin des fils de elem
 - `elem.prepend(...values)` // ajout au début des fils de elem
 - `node.before(...values)` // ajout dans le parent de node avant node
 - `node.after(...values)` // ajout dans le parent de node après node
 - `node.replaceWith(...values)` // ajout dans le parent de node à la place de node
- ❑ Les arguments `values` sont soit des nœuds soit des chaînes automatiquement converties en nœuds textuels.
- ❑ Pour supprimer un nœud `node`, on appelle `node.remove()`.

CSS : Classes et styles

- ❑ Il est possible de modifier l'apparence d'un élément en lui ajoutant un style CSS.
- ❑ On peut ajouter un style en modifiant l'attribut `style` d'un élément.
- ❑ On peut définir le style d'une classe et ajouter la classe à l'élément.
- ❑ Sauf cas particulier (ex: positionnement), cette deuxième façon de faire est à privilégier.

CSS : Les classes

- ❑ La propriété associée à l'attribut `class` s'appelle `className` pour des raisons historiques.
- ❑ Cette propriété contient toutes les classes de l'élément séparées par des espaces.
- ❑ On manipule plus souvent les classes via la propriété `classList` dont l'interface est plus facile d'usage.
 - `elem.classList.add("class");` // ajoute une classe à elem
 - `elem.classList.remove("class");` // retire une classe de elem
 - `elem.classList.toggle("class");` // ajoute une classe à elem si non présente ou la retire si présente
 - `elem.classList.contains("class");` // appartenance d'une classe
- ❑ La propriété `classList` est également énumérable.

CSS : Les styles

- ❑ La propriété `style` d'un élément est un objet qui contient tous les styles appliqués à l'objet. On y accède par le nom du style à lire/modifier.

```
document.body.style.backgroundColor = "gray";
```

```
button.style.width = "100px";
```

- ❑ Attention aux propriétés géométriques, elles sont toujours accompagnées d'une unité.
- ❑ Pour réinitialiser un style, il faut lui passer une chaîne vide ou utiliser la méthode `removeProperty` de l'objet `elem.style`. Cela permet de préserver l'effet en cascade du CSS.
- ❑ On peut réécrire le style en utilisant la propriété `elem.style.cssText`. Le style actuel est complètement remplacé.

Données locales

- ❑ La page Web dispose de deux dictionnaires permettant de stocker des données : `localStorage` et `sessionStorage`.
- ❑ Ces deux objets disposent des méthodes:
 - `setItem(key, data)`
 - `getItem(key)`
 - `removeItem(key)`
 - `clear()`
- ❑ Les clés et les données sont des chaînes de caractère. Il faut donc encoder/décoder les objets pour les stocker (ex: `JSON.stringify` et `JSON.parse`).
- ❑ Les données du `localStorage` sont persistantes, celles du `sessionStorage` sont vidées à la fermeture du navigateur.

LES ÉVÉNEMENTS

Présentation

- ❑ Lorsqu'un utilisateur interagit avec la page Web, il génère un certain nombre d'évènements.
- ❑ Ces évènements peuvent être écoutés par les éléments de la page Web pour exécuter dynamiquement du code.
- ❑ Le nombre d'évènements possibles est très important. Seule une partie d'entre eux sera présentée et étudiée.
- ❑ Par exemple, les supports tactiles disposent d'évènements dédiés plus compliqués que ceux générés par la combinaison souris/clavier.

Événements courants

Nom de l'évènement	Description
click / contextmenu	Clic gauche ou clic droit de la souris sur un élément.
dblclick	Double clic.
mouseover / mouseout	Souris entrant ou sortant d'un élément.
mousedown / mouseup	Souris appuyée ou relâchée sur un élément.
mousemove	Déplacement de la souris
keydown / keyup	Touche appuyée ou relâchée.
submit	Formulaire soumis.
DOMContentLoaded	DOM entièrement construit.

Écouteurs

- ❑ On peut placer un écouteur directement dans une balise html en précisant le code Javascript à exécuter. L'attribut à utiliser est nommé **on**eventname.

```
<button onclick="alert('Got you! ')">Click me!</button>
```

- ❑ On peut aussi référencer une fonction Javascript. C'est à privilégier dès que le code est complexe.

```
<button onclick="onClick()">Click me!</button>
```

```
<script> function onClick() { alert("Got you!"); }</script>
```

- ❑ Il est aussi possible d'assigner la propriété dans le script.

```
<button id="id">Click me!</button>
```

```
<script>
```

```
    document.querySelector("#id").onclick = () => alert("Got you!");
```

```
</script>
```


Écouteurs

- ❑ Ces trois façons de faire ont le même inconvénient, chaque élément ne peut avoir qu'un seul écouteur pour un type d'évènement donné.
- ❑ Si on veut plusieurs écouteurs pour un même évènement, il faut utiliser la méthode `addEventListener`.
- ❑ L'appel `elem.addEventListener(event, handler)` ajoute à `elem` un écouteur de l'évènement `event` qui réagit en exécutant `handler`.

```
<button id="id">Click me!</button>
```

```
<script>
```

```
  const button = document.querySelector("#id");
```

```
  button.addEventListener("click", () => alert("Got you!"));
```

```
</script>
```

Écouteurs

- ❑ L'appel `elem.removeEventListener(event, handler)` retire l'écouteur ajouté par `addEventListener`.
- ❑ La référence `handler` doit être la même pour les deux appels.

```
<button id="id">Click me!</button>
```

```
<script>
```

```
  const button = document.querySelector("#id");
```

```
  const handler = () => alert("Got you!");
```

```
  button.addEventListener("click", handler);
```

```
  // ...
```

```
  button.removeEventListener ("click", () => alert("Got you!")); // non
```

```
  button.removeEventListener ("click", handler); // oui
```

```
</script>
```

Objet évènement

- ❑ Tous les écouteurs reçoivent en argument un objet qui contient des informations sur l'évènement qui les a déclenché.
- ❑ Tous les évènements ont les propriétés :
 - `type` pour le type de l'évènement ("`click`", "`keydown`", ...),
 - `target` pour l'élément qui a généré l'évènement,
 - `currentTarget` pour l'élément qui gère actuellement l'évènement (?).
 - ...
- ❑ Des évènements de certains types ont des propriétés supplémentaires :
 - `clientX` / `clientY` pour les évènements souris,
 - `key` pour un évènement clavier,
 - ...



Bouillonnement

- ❑ Lorsqu'un évènement s'initie dans un élément, on exécute les écouteurs de cet élément dans l'ordre de leur ajout.
- ❑ Ensuite, les écouteurs du parents s'enclenchent sur le même évènement et ainsi de suite pour tous les ancêtres.
- ❑ Ce phénomène est appelé bouillonnement (*bubbling*).
- ❑ La propriété `target` référence toujours l'élément initial, `this` et `currentTarget` référencent celui qui gère actuellement l'évènement.
- ❑ On peut stopper le bouillonnement en appelant la méthode `event.stopPropagation()`.

Bouillonnement : exemple

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Bouillonnement</title>
  </head>

  <body>
    <div id="div">
      <p id="p"><button id="button">Click me!</button></p>
    </div>
  </body>
</html>
```

Bouillonnement : exemple

```
const handler = (event) =>
  console.log(
    `Target = ${event.target.id}, current = ${event.currentTarget.id}`
  );
const elems = ["#div", "#p", "#button"].map((id) =>
  document.querySelector(id)
);
elems.forEach((elem) => elem.addEventListener("click", handler));
button.click();
```

```
Target = button, current = button
Target = button, current = p
Target = button, current = div
```

Bouillonnement : exemple

```
const handler = (event) =>
  console.log(
    `Target = ${event.target.id}, current = ${event.currentTarget.id}`
  );
const elems = ["#div", "#p", "#button"].map((id) =>
  document.querySelector(id)
);
elems.forEach((elem) => elem.addEventListener("click", handler));
const [, p] = elems;
p.addEventListener("click", (event) => event.stopPropagation());
button.click();
```

```
Target = button, current = button
Target = button, current = p
```

Bouillonnement : délégation

- ❑ Le bouillonnement permet de faire de la délégation d'évènements.
- ❑ Au lieu de gérer un évènement directement à la source de son déclenchement, c'est un élément parent qui s'en charge.
- ❑ On peut dès lors centraliser une gestion d'évènements dans un élément parent pour tout un ensemble d'enfants.
- ❑ Par exemple, le gestionnaire se trouve sur un élément `<table>` pour gérer tous les évènements générés par les cellules `<td>`.
- ❑ C'est ce qu'on a fait dans `table.html`.



Capture

- ❑ Avant le bouillonnement, une phase de capture intervient.
- ❑ Alors que le bouillonnement remonte du nœud cible à la racine du DOM. La capture intervient dans l'ordre inverse.
- ❑ On peut déclencher un écouteur durant cette phase si nécessaire. Pour cela, il faut utiliser un paramètre supplémentaire lors de son ajout.

```
elem.addEventListener(event, handler, { capture: true });
```

```
elem.addEventListener(event, handler, true);
```

- ❑ Un appel à `stopPropagation` annulera la descente dans la phase de capture mais aussi toute la phase de bouillonnement.

Capture : exemple

```
const cap = (event) => console.log(`Capture in ${event.currentTarget.id}`);
const bub = (event) => console.log(`Bubbling in ${event.currentTarget.id}`);
const elems = ["#div", "#p", "#button"].map((id) =>
  document.querySelector(id)
);
elems.forEach((elem) => elem.addEventListener("click", cap, true));
elems.forEach((elem) => elem.addEventListener("click", bub));
button.click();
```

```
Capture in div
Capture in p
Capture in button
Bubbling in button
Bubbling in p
Bubbling in div
```

Capture : exemple

```
const cap = (event) => console.log(`Capture in ${event.currentTarget.id}`);
const bub = (event) => console.log(`Bubbling in ${event.currentTarget.id}`);
const elems = ["#div", "#p", "#button"].map((id) =>
  document.querySelector(id)
);
elems.forEach((elem) => elem.addEventListener("click", cap, true));
elems.forEach((elem) => elem.addEventListener("click", bub));
const [, p] = elems;
p.addEventListener("click", (event) => event.stopPropagation(), true);
button.click();
```

```
Capture in div
Capture in p
```

Actions par défaut

❑ Certains éléments ont des actions par défaut lors d'évènements particuliers :

- Un clic droit ouvre le menu contextuel ;
- La molette et les touches PageUp et PageDown activent le scrolling ;
- Un clic sur un lien active la navigation ;
- ...

❑ Pour empêcher ces actions, il existe deux solutions :

- appeler `event.preventDefault()` dans un écouteur ou
- renvoyer faux dans un écouteur lié à la propriété `on<event>` (donc qui n'est pas passé par un `addEventListener`).