DOM

Michel Gagnon et Nikolay Radoev
École Polytechnique de Montréal

Ce qui se passe lorsqu'un document HTML est chargé

1. Déclenchement du processus de construction du DOM (Document Objet Model)

2. Quand une balise <script> est rencontrée, la construction du DOM est interrompue pour charger et exécuter le script (sauf s'il y a présence de l'attribut **async** ou **defer**)
(ex. <script src='script.js' async></script>)

3. Les scripts asynchrones sont téléchargés et exécutés dès que possible, mais en attendant on continue la construction du DOM

4. Quand le DOM est construit, les scripts déclarés avec l'attribut **defer** sont exécutés
(ex. <script src='script.js' defer></script>)

5. Il se peut, à ce stade, qu'il reste encore des éléments à télécharger, comme des images. Quand tout est téléchargé, le navigateur déclenche un événement **load.**

DOM

- Fournit une représentation structurée (sous forme d'arbre d'objets) d'un document HTML en mémoire
- L'API du DOM contient plusieurs interfaces qui sont implémentées par chaque engin de rendu.
- Voici les 2 interfaces de haut niveau les plus utilisées:
  - Nœuds (classe Node): tous les items qui forment l'arborescence du document HTML (chaque nœud, sauf la racine, a donc un parent)
  - Événements (classe Event) : différents éléments du DOM peuvent écouter (*listen)* sur des événements s'ils implémentent l'interface EventTarget (ex: HTMLButtonElement dont la chaîne d'héritage remonte jusqu'à EventTarget)

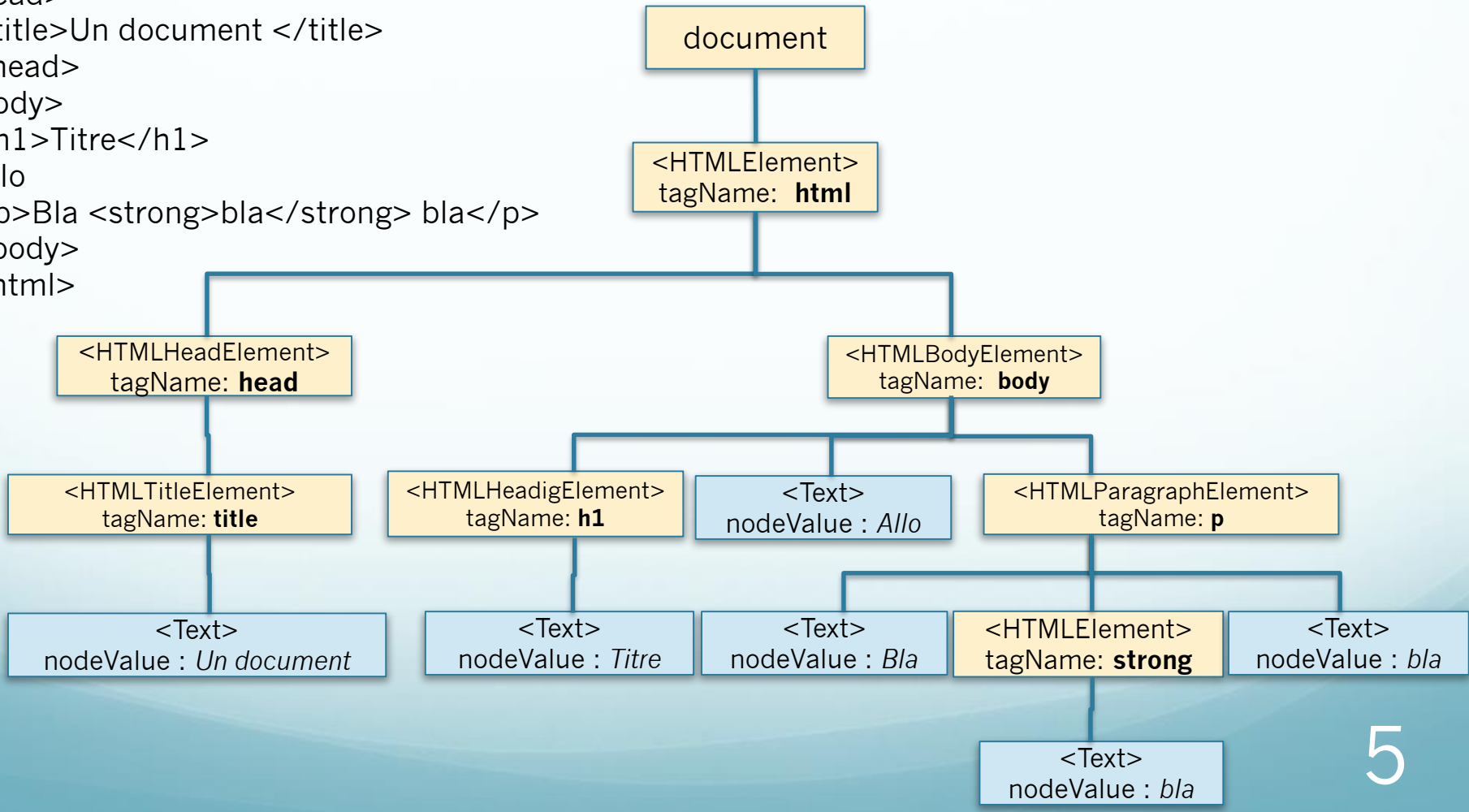
DOM

- Les nœuds qui composent le DOM implémentent une des interfaces suivantes:
  - **Element**: il s'agit d'un élément du document.
    - L'interface HTMLElement en hérite et définit les éléments HTML
    - L'interface SVGElement en hérite, mais pour les éléments SVG
  - **Text**: il s'agit du contenu texte d'un Element et ne peut pas avoir d'autres nœuds comme enfants
    - ex : ** texte en gras ** ici contient 2 noeuds *text* ("Texte en" et "ici") et 1 nœud vs ** texte en gras ici ** qui a 1 seul nœud *text*

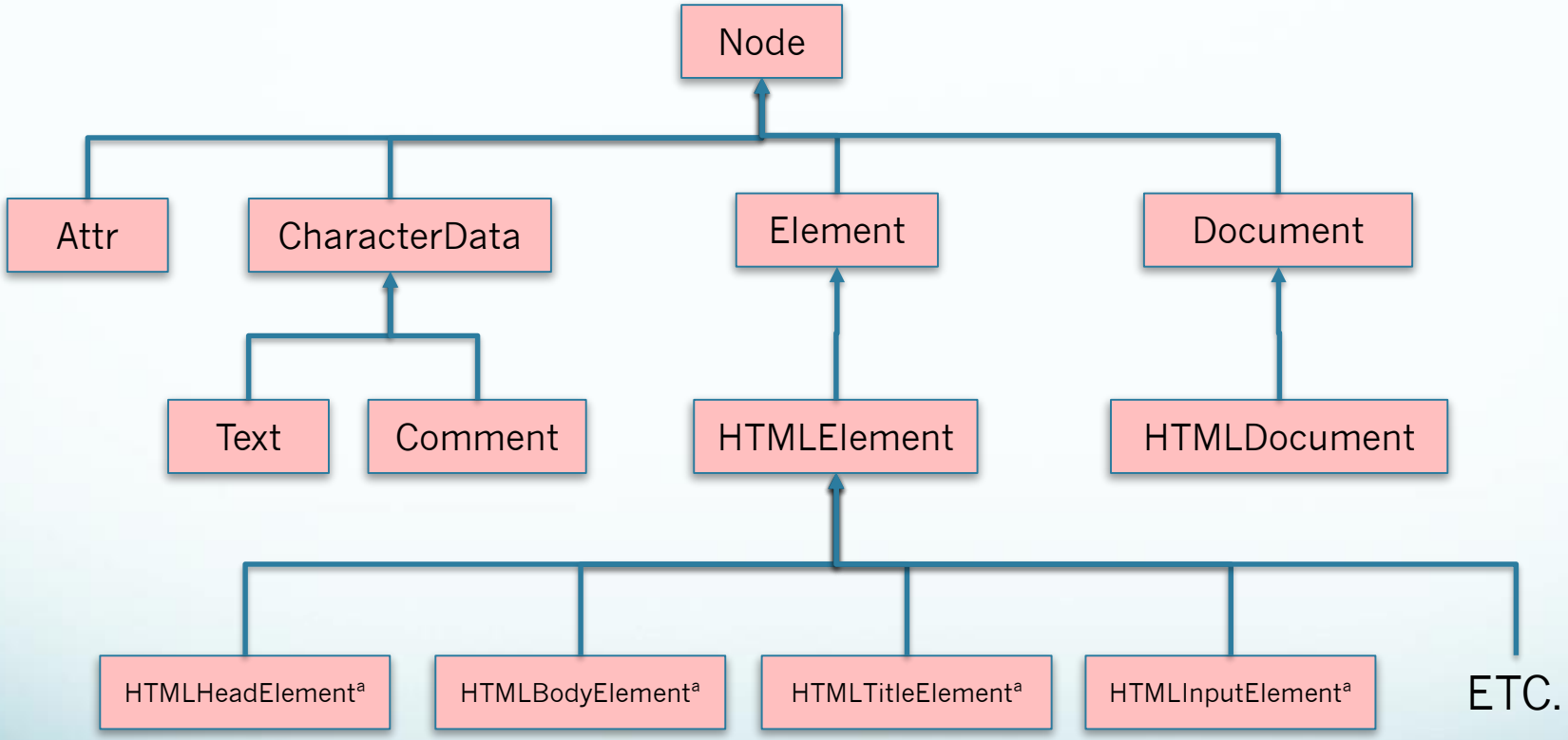
- Les nœuds forment un arbre dont la racine est l'objet **document** . Son parent est **null**

Exemple d'arbre DOM

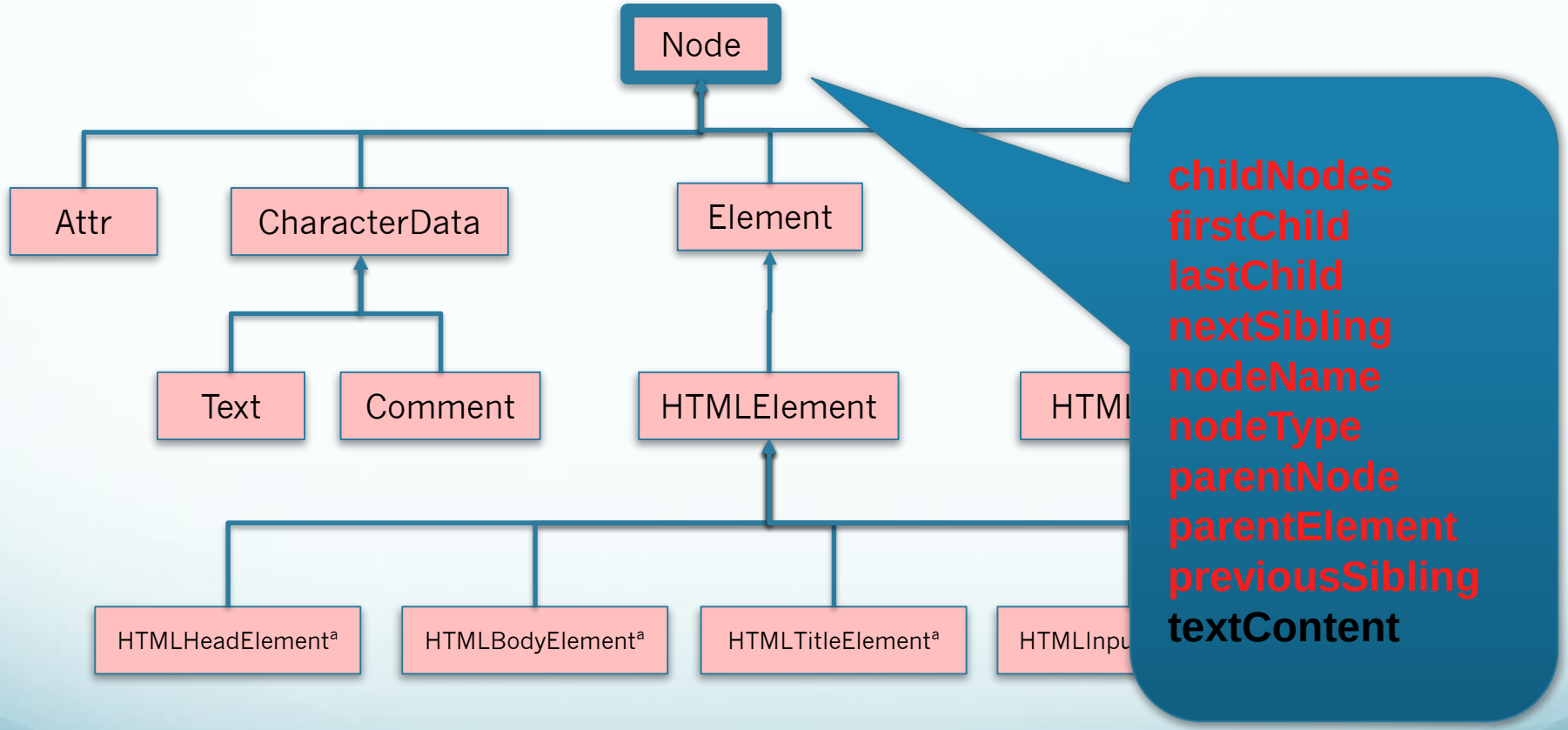
```
<html>
<head>
  <title>Un document </title>
</head>
<body>
  <h1>Titre</h1>
  Allo
  <p>Bla <strong>bla</strong> bla</p>
</body>
</html>
```



Hiérarchie des interfaces du DOM

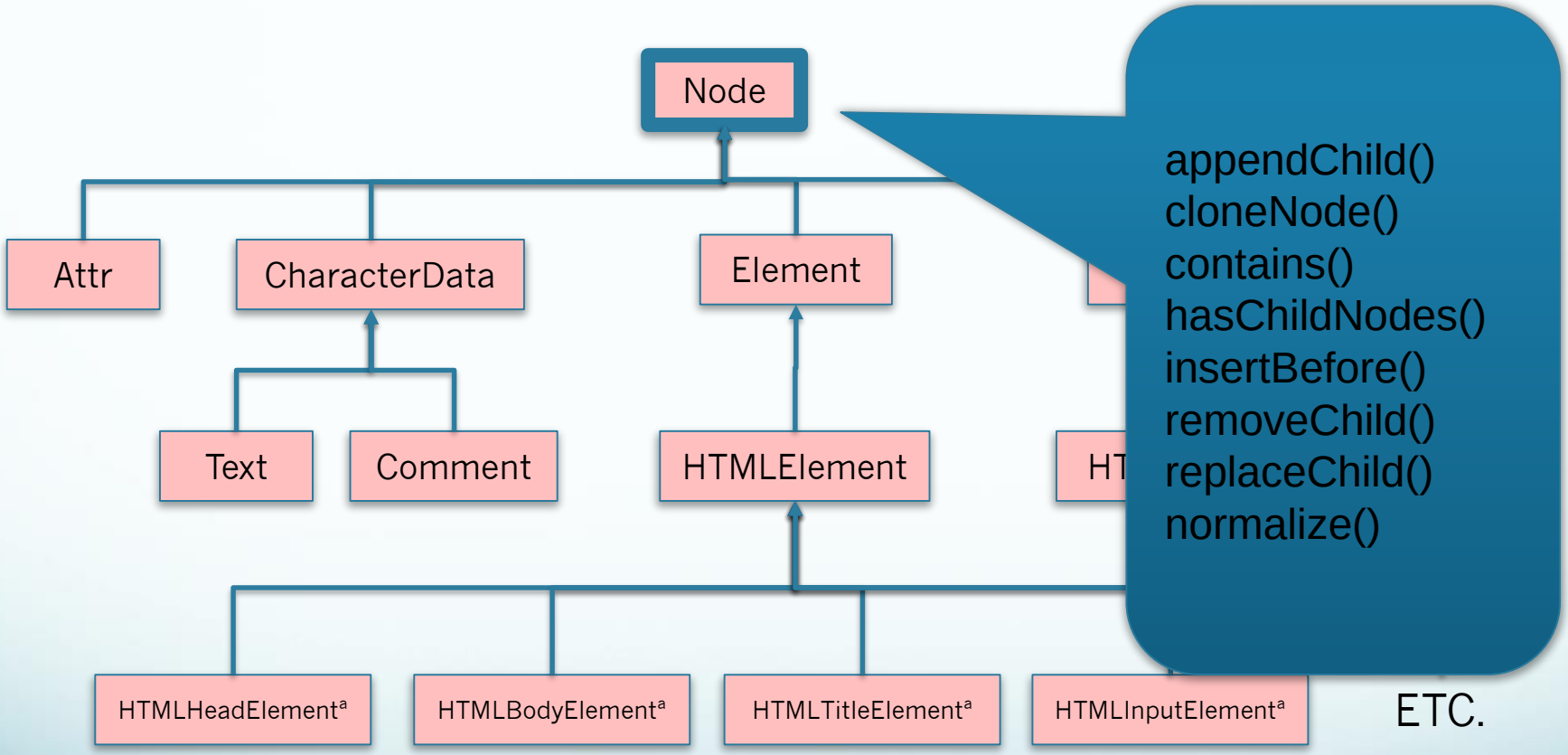


Hiérarchie des interfaces du DOM

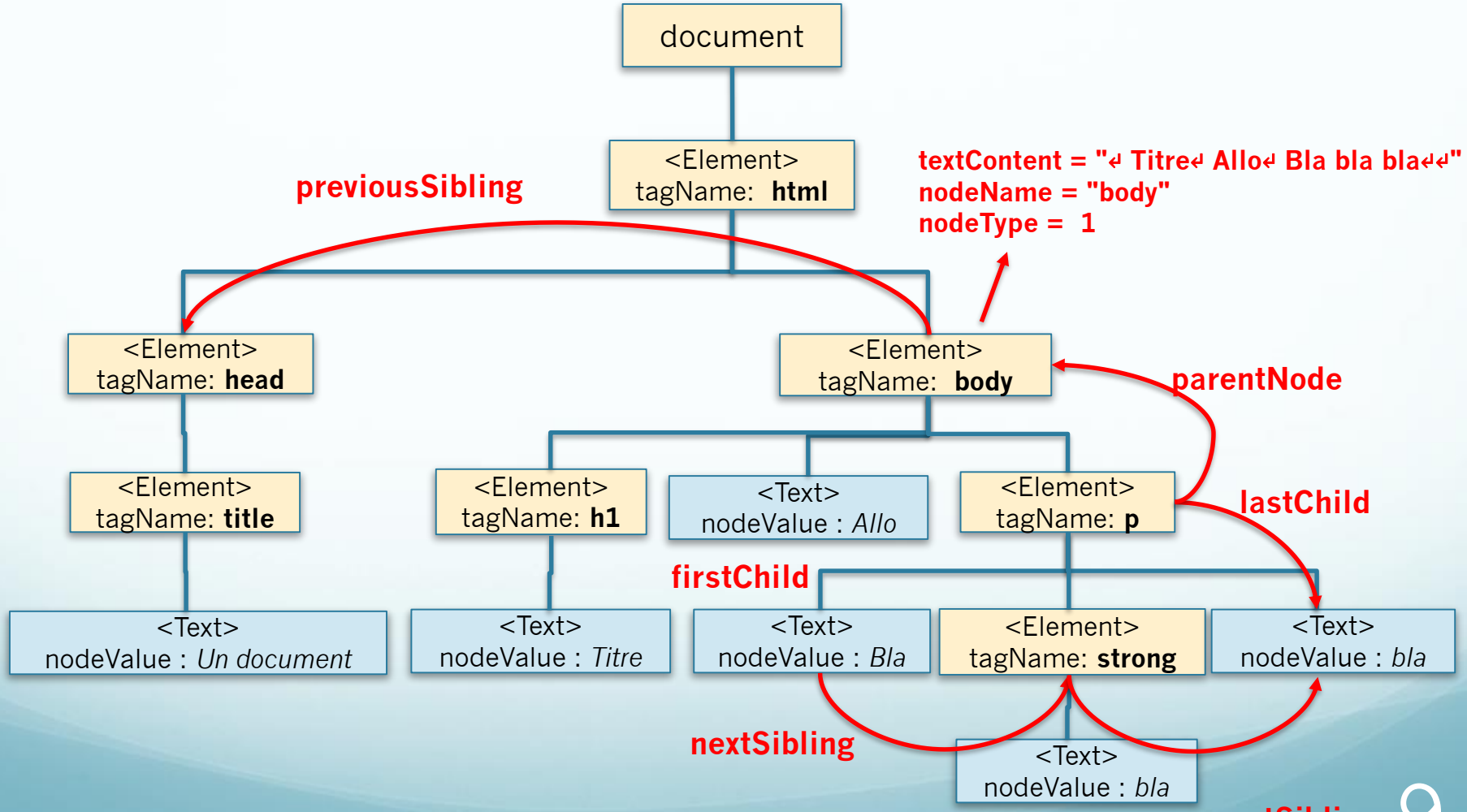


Note: les propriétés en rouge sont accessibles seulement en lecture

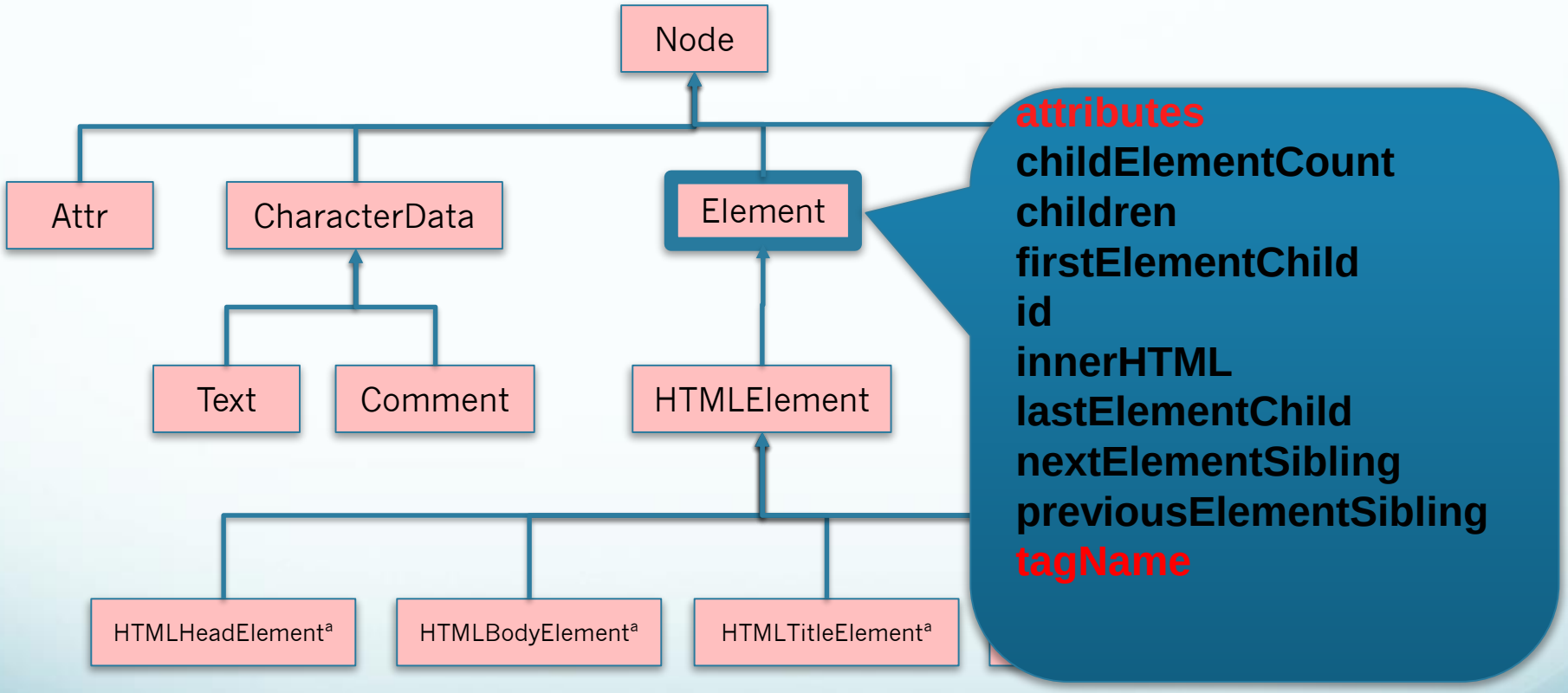
Hiérarchie des interfaces du DOM



DOM (API par nœuds)

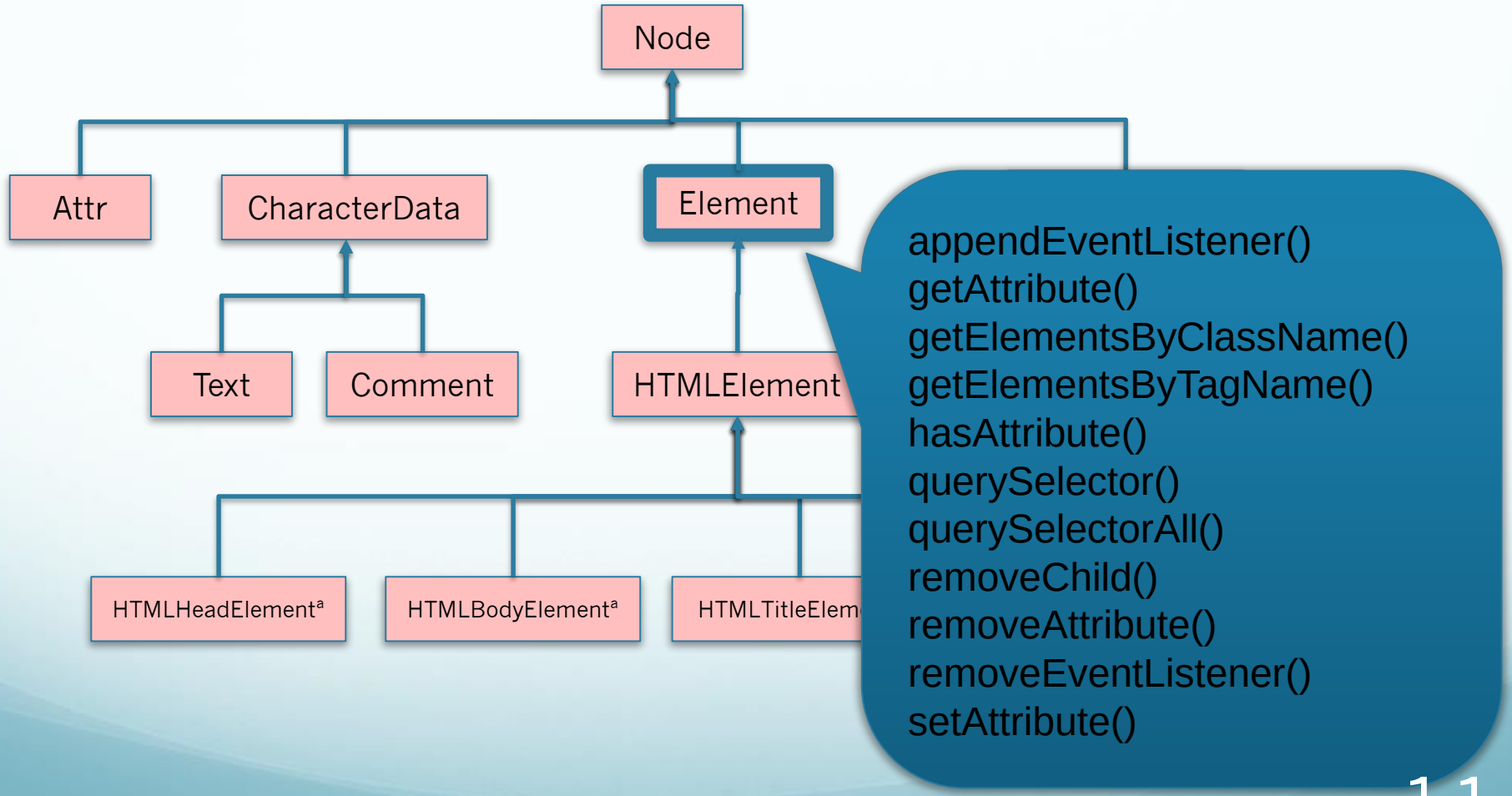


Hiérarchie des interfaces du DOM

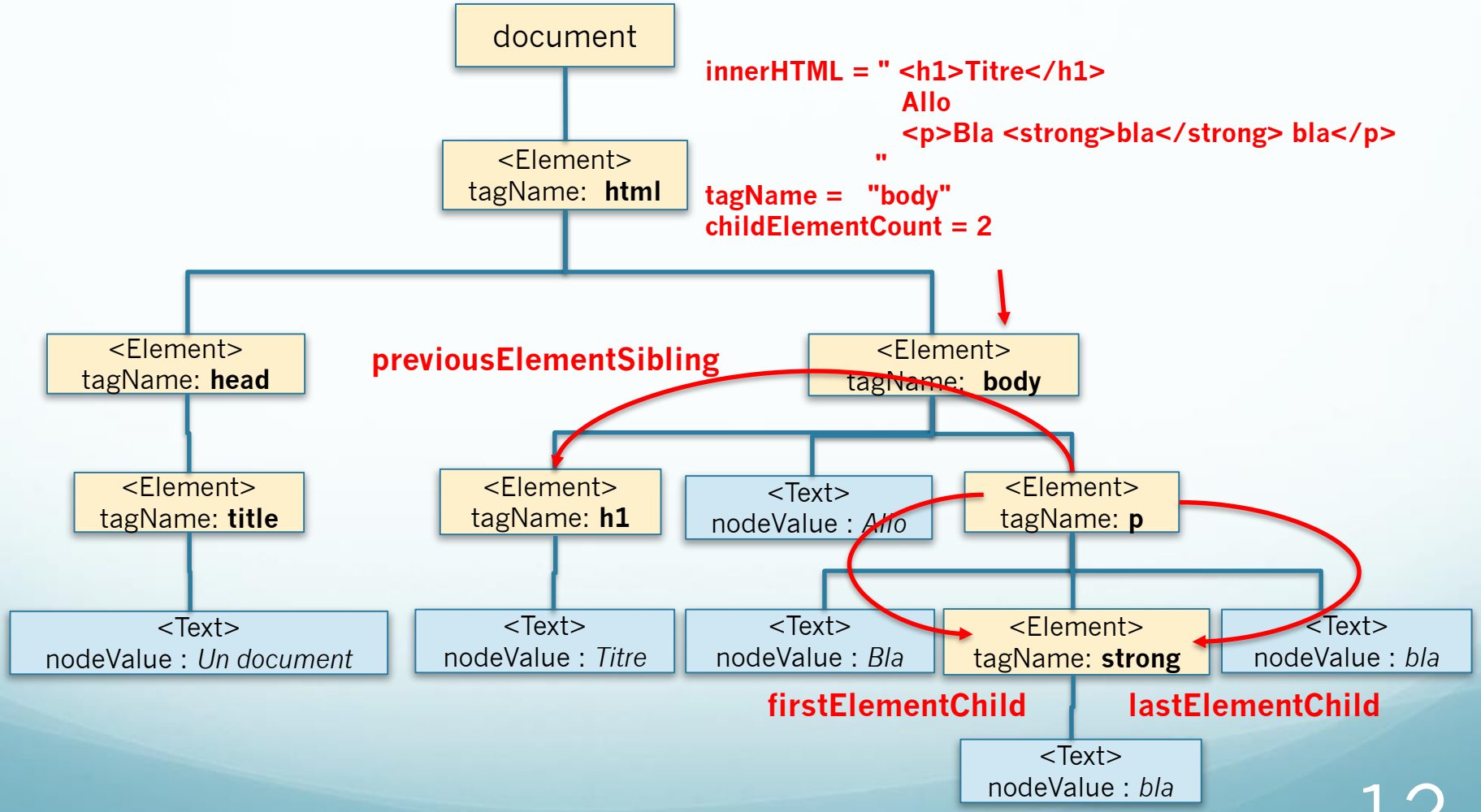


Note: les propriétés en rouge sont accessibles seulement en lecture

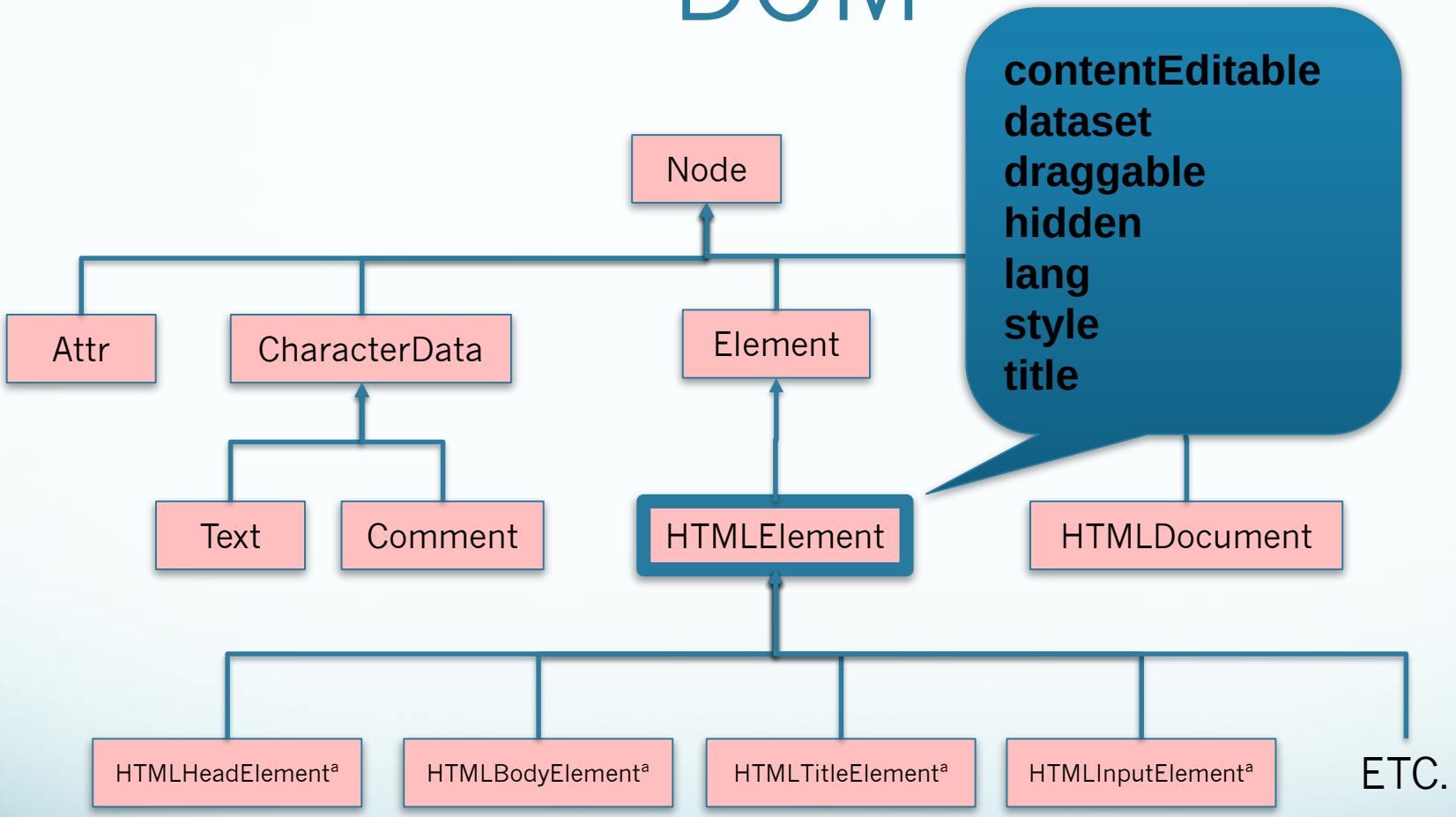
Hiérarchie des interfaces du DOM



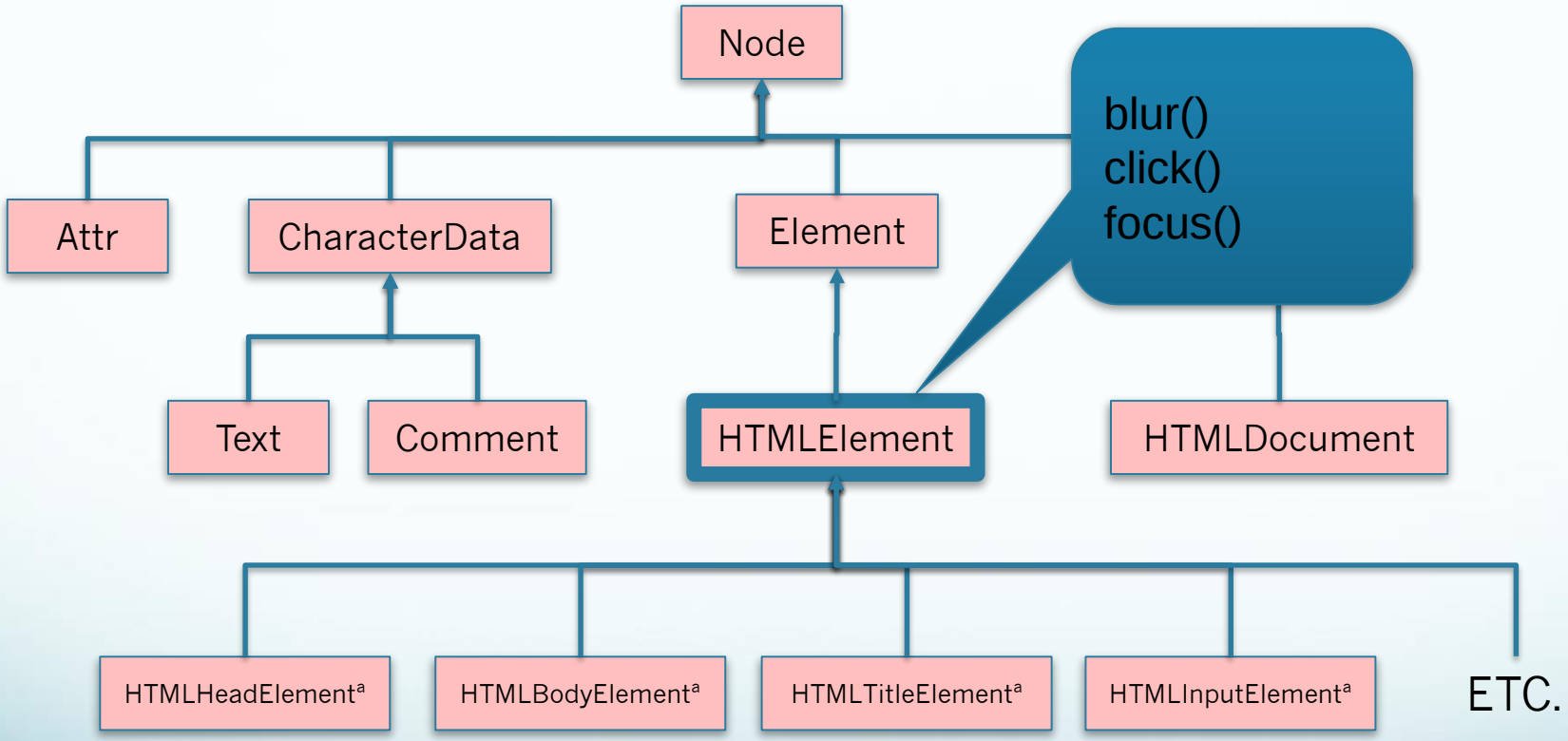
DOM (API par éléments)



Hiérarchie des interfaces du DOM



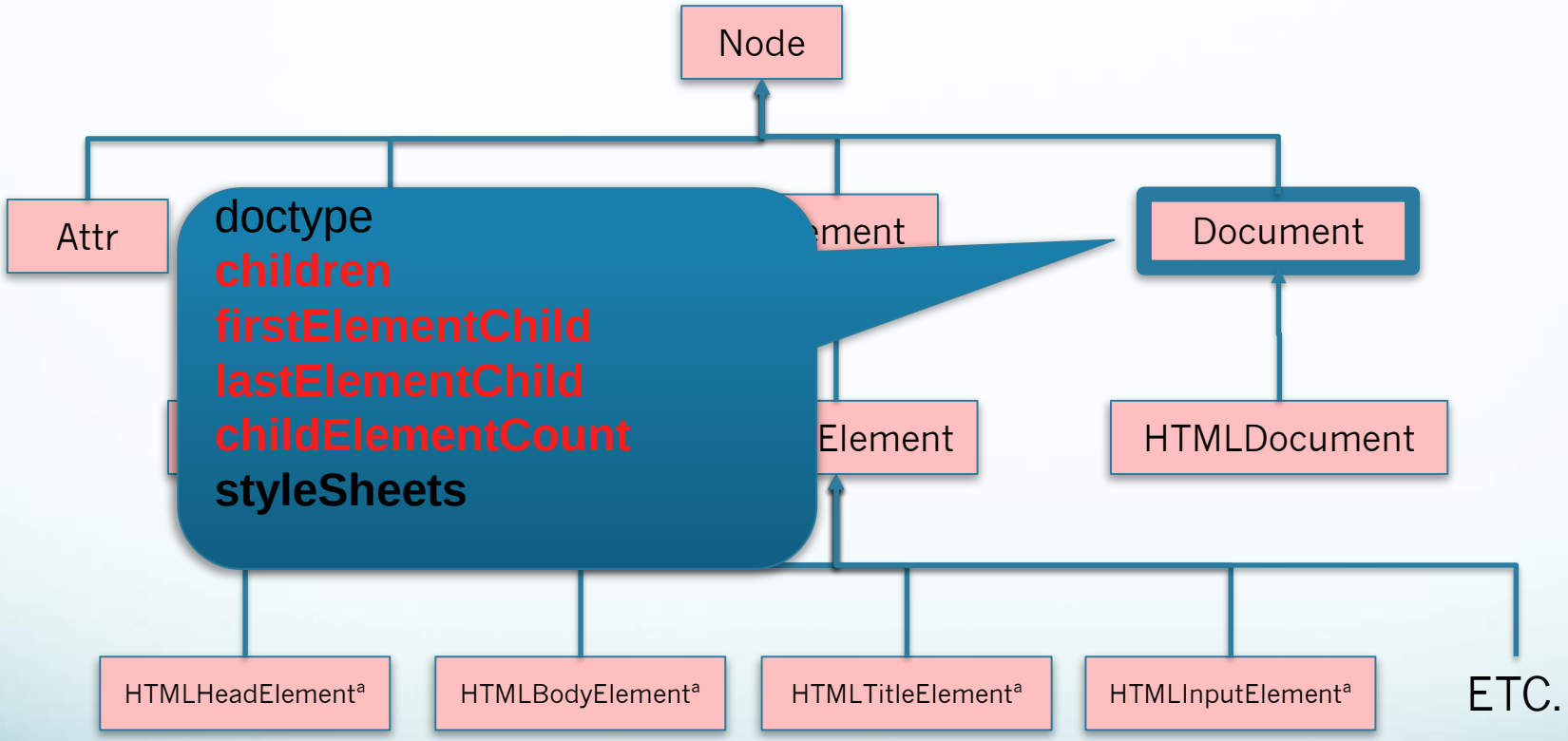
Hiérarchie des interfaces du DOM



Propriété **style**

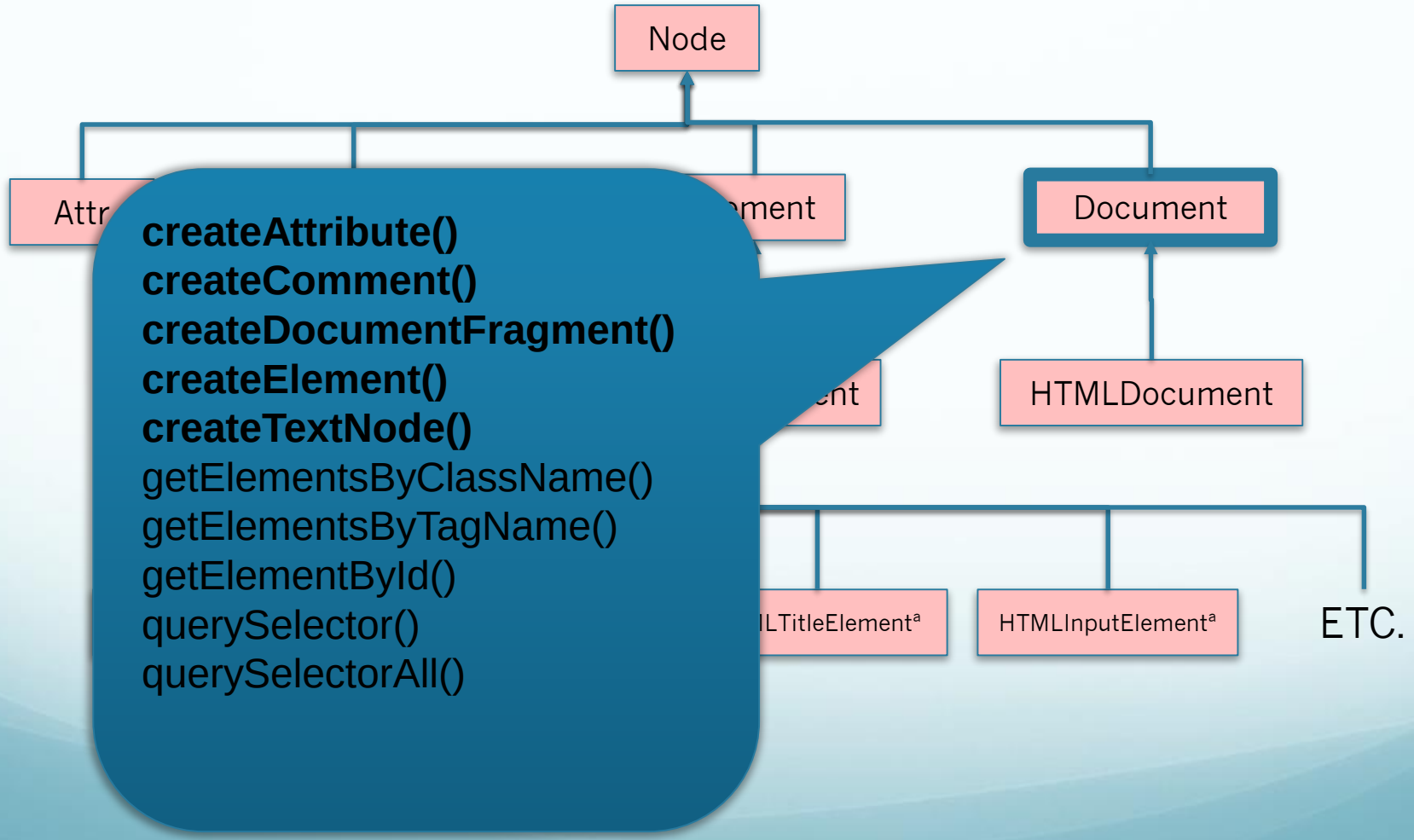
- Cette propriété est associée à des déclarations en-ligne CSS qui indiquent la présentation de l'élément
- Si on utilise cet attribut pour fixer des valeurs de propriété, elles seront au niveau « en-ligne » dans la cascade CSS
- Ce n'est pas cette propriété qu'il faut utiliser pour connaître le style d'un élément, puisqu'elle ne concerne que les déclarations en-ligne (utiliser plutôt la méthode **getComputedStyle(***element***)** de l'objet **window** pour un *element* quelconque)

Hiérarchie des interfaces du DOM

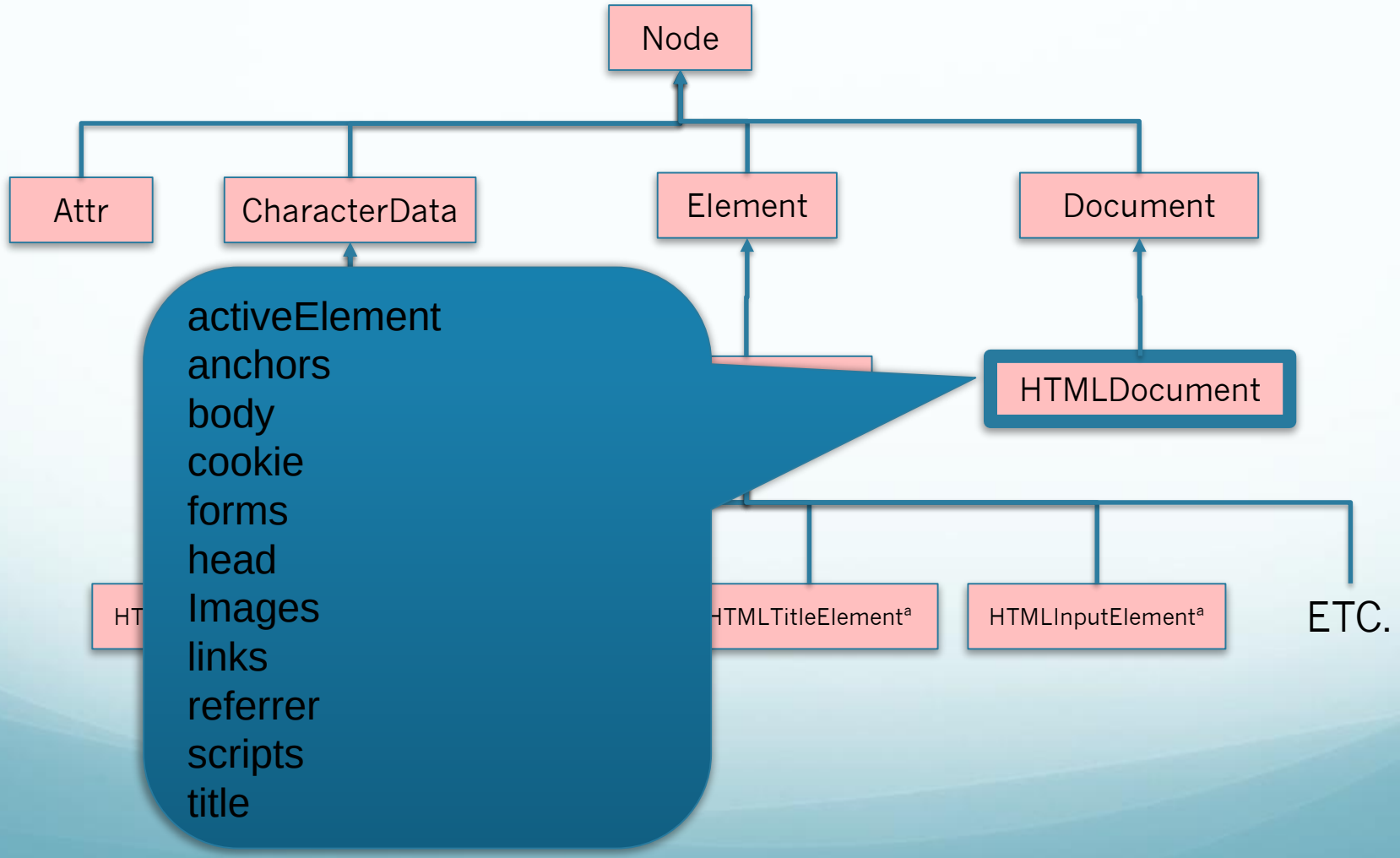


Note: les peropriétés en rouge sont accessibles seulement en lecture

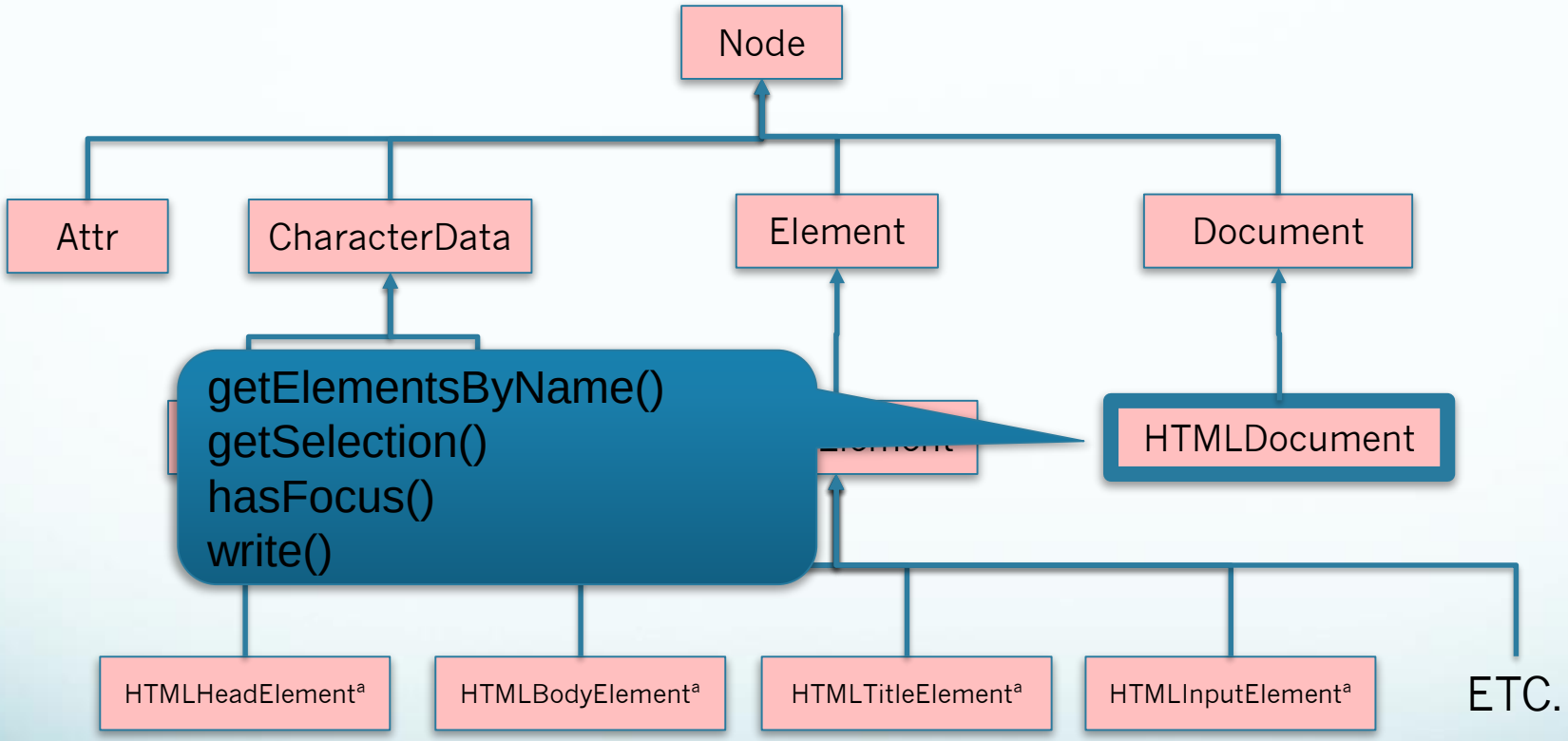
Hiérarchie des interfaces du DOM



Hiérarchie des interfaces du DOM



Hiérarchie des interfaces du DOM



Objet **window**

- Window : interface qui représente la fenêtre qui contient le document DOM.
- La variable globale **window** représente l'espace où roulent les scripts.
  - L'objet "this" par défaut est le **window**
  - Une variable globale devient un attribut de **window**
- Dans le cas de plusieurs onglets, chaque onglet a son propre **window**
  - Certaines propriétés de **window** font référence à la fenêtre au complet et non seulement l'onglet
  - L'implémentation de l'interface dépend du navigateur

Objet **window**

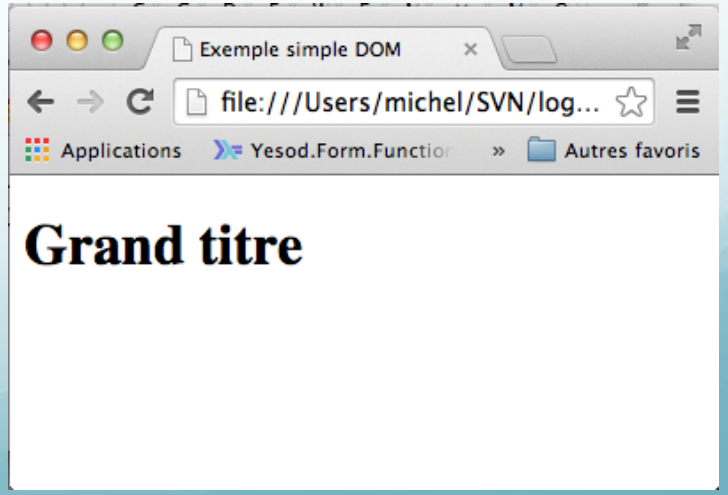
- Propriétés globales:
  - console (retourne une référence vers l'objet Console)
  - fullscreen (seulement sur Firefox)
  - history (retourne une référence vers l'objet History)
  - document (référence vers le Document de la page)
  - variables JS déclarés sans autre environnement lexique
  - etc.
- Méthodes globales:
  - alert()
  - open() / close() (le script doit avoir été la source de l'ouverture d'une fenêtre pour pouvoir la fermer)
  - confirm()
  - getComputedStyle()
  - etc.

Objet **History**

- Contient les URL visités par l'utilisateur durant la session
- Attributs:
  - length (l'historique est un tableau dans lequel on se déplace)
- Méthodes:
  - back() et forward()
    - Equivalents à go(-1) et go(1) respectivement
  - go(?*delta*)
    - delta de 0 recharge simplement la page
  - pushState() et replaceState()

Manipulation du DOM Exemple

```
<html>
<script>
 // lancer cette fonction quand le document est chargé
 window.onload = function() {
   // crée quelques éléments dans
   // une page HTML pour l'instant vide
   heading = document.createElement("h1");
   heading_text = document.createTextNode("Grand titre");
   heading.appendChild(heading_text);
   document.body.appendChild(heading);
 }
</script>
</html>
```

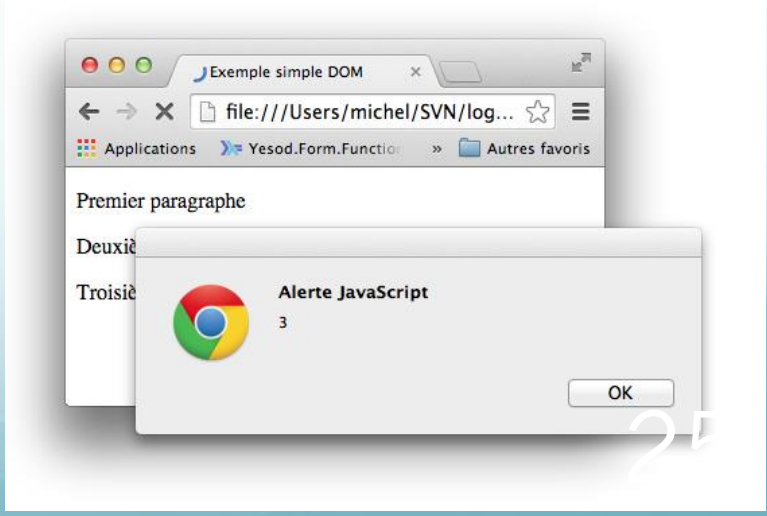


Interface NodeList

- Certain accès à des éléments, comme la méthode *getElementsByClassName()* ou la propriété *childNodes*, retournent une collection d'éléments, qui sont contenus dans un objet de type *HTMLCollection* et *NodeList* respectivement
- On peut accéder à chaque élément contenu individuellement, avec l'opérateur [], comme si c'était un tableau
- Attention: très souvent ce conteneur est « vivant » *(live)*, c'est-à-dire que des changements dans le DOM seront reflétés dans la collection.
  - Une *HTMLCollection* est toujours « vivante ».
  - Le conteneur de type *NodeList* retourné par *querySelectorAll()* n'est pas « vivant », mais plutôt statique (*static*)

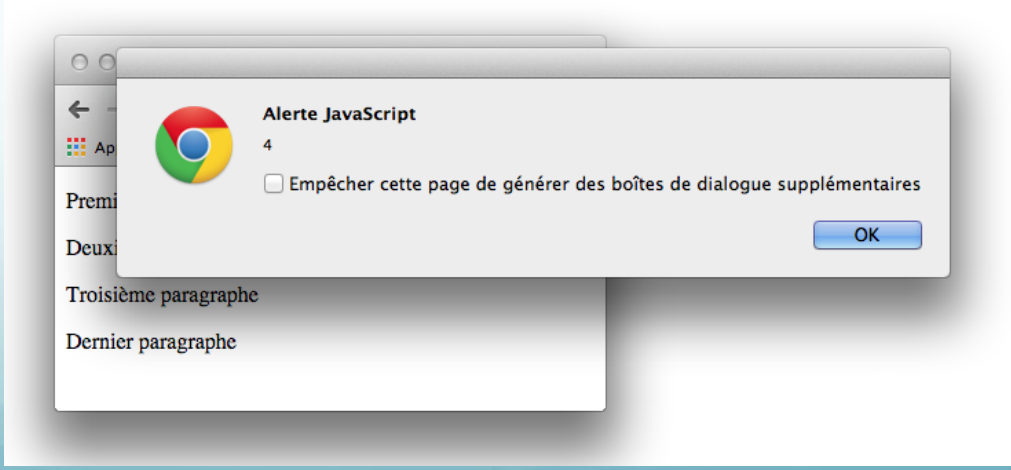
Exemple de NodeList vivant

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple simple DOM</title>
    <script>
      window.onload = function() {
        paragraphes = document.getElementsByTagName('p');
        alert(paragraphes.length);
        nouveauParagraphe = document.createElement('p');
        nouveauParagraphe.textContent = "Dernier paragraphe";
        document.body.appendChild(nouveauParagraphe);
        alert(paragraphes.length);
      }
    </script>
  </head>
  <body>
   <p> Premier paragraphe</p>
   <p> Deuxième paragraphe</p>
   <p> Troisième paragraphe</p>
  </body>
</html>
```



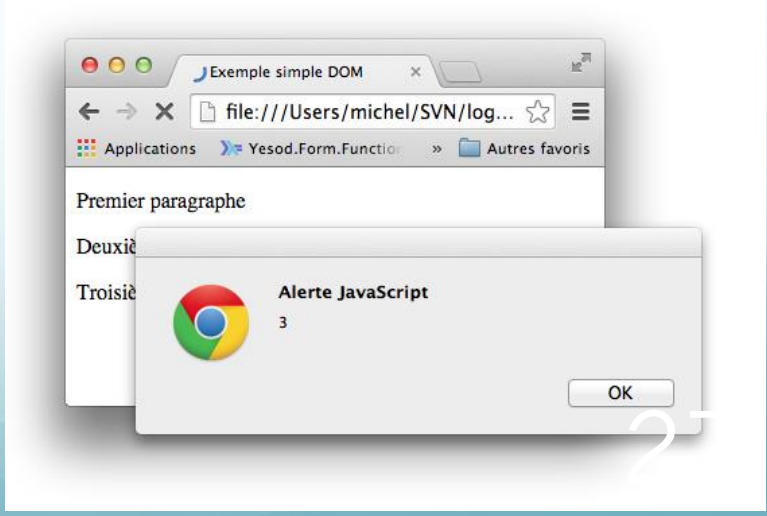
Exemple de NodeList vivant

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple simple DOM</title>
    <script>
      window.onload = function() {
        paragraphes = document.getElementsByTagName('p');
        alert(paragraphes.length);
        nouveauParagraphe = document.createElement('p');
        nouveauParagraphe.textContent = "Dernier paragraphe";
        document.body.appendChild(nouveauParagraphe);
        alert(paragraphes.length);
      }
    </script>
  </head>
  <body>
   <p> Premier paragraphe</p>
   <p> Deuxième paragraphe</p>
   <p> Troisième paragraphe</p>
  </body>
</html>
```



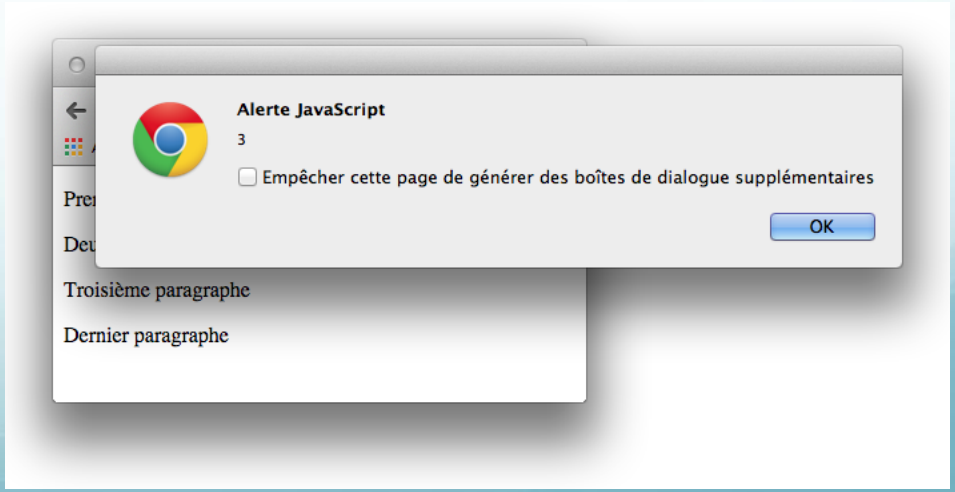
Exemple de NodeList statique

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple simple DOM</title>
    <script>
      window.onload = function() {
        paragraphes = document.querySelectorAll('p');
        alert(paragraphes.length);
        nouveauParagraphe = document.createElement('p');
        nouveauParagraphe.textContent = "Dernier paragraphe";
        document.body.appendChild(nouveauParagraphe);
        alert(paragraphes.length);
      }
    </script>
  </head>
  <body>
   <p> Premier paragraphe</p>
   <p> Deuxième paragraphe</p>
   <p> Troisième paragraphe</p>
  </body>
</html>
```



Exemple de NodeList statique

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple simple DOM</title>
    <script>
      window.onload = function() {
        paragraphes = document.querySelectorAll('p');
        alert(paragraphes.length);
        nouveauParagraphe = document.createElement('p');
        nouveauParagraphe.textContent = "Dernier paragraphe";
        document.body.appendChild(nouveauParagraphe);
        alert(paragraphes.length);
      }
    </script>
  </head>
  <body>
   <p> Premier paragraphe</p>
   <p> Deuxième paragraphe</p>
   <p> Troisième paragraphe</p>
  </body>
</html>
```



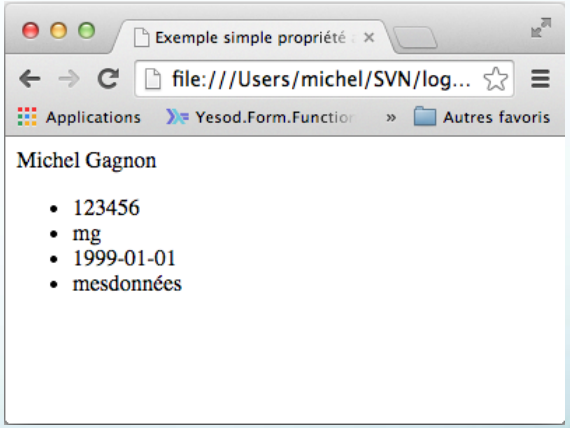
Propriété **dataset**

- Il s'agit d'un objet qui contient tous les attributs fixés par la forme **data-***, dans la balise correspondant à l'élément
- Pour chacune des occurrences, il y a une propriété associée dataset, dont le nom correspond à tout ce qui vient après « data- », sans les tirets, et où tout caractère qui succède à un tiret est mis en majuscule
  - Ex: data-date-of-birth donne dateOfBirth lorsqu'utilisé comme variable en JS
  - On ne peut donc pas avoir des majuscules qui suivent un tiret dans le nom de l'attribut
- Très utile si on veut rajouter de l'information directement dans une balise HTML pour être traité par le code par la suite

Propriété **dataset** - exemple

```
<div id="user" data-ident="123456" data-user="mg" data-date-of-birth>
    Michel Gagnon
</div>

const el = document.querySelector('#user');

// Fixer la date de naissance
el.dataset.dateOfBirth = '1999-01-01';

// On ajoute une nouvelle propriété
el.dataset.autresDonnees = 'mesdonnées';

document.write('<ul>');
document.write('<li>' + el.dataset.ident + '</li>');
document.write('<li>' + el.dataset.user + '</li>');
document.write('<li>' + el.dataset.dateOfBirth + '</li>');
document.write('<li>' + el.dataset.autresDonnees + '</li>');
document.write('</ul>');
```

el.dataset.ident = '123456'
el.dataset.user = 'mg'
el.dataset.dateOfBirth = ''



Événements

- Chaque événement a un nom: *click*, *change*, *load*, *mouseover*, etc.
- Chaque événement a une cible (*target*), soit l'élément sur lequel il a été déclenché.
- Il faut associer à chaque événement une fonction qui sera appelée lorsque celui-ci sera déclenché : gestionnaire d'événement (*event handler*)
- Trois manières d'associer un gestionnaire d'événement à un élément:
  - Le spécifier directement dans la balise (déconseillé)
  - L'associer à un attribut de l'élément (*onclick*, *onchange*, *onload*, *onmouseover*, etc) (un seul *handler* à la fois)
  - Utiliser la méthode ***addEventListener()*** qui, contrairement à la méthode précédente, permet d'ajouter plusieurs gestionnaires à un événement

Événements (suite)

- Le gestionnaire peut recevoir en paramètre un objet de type **Event** qui fournit de l'information sur l'événement (notamment, la cible)
- L'événement est propagé dans les éléments englobant celui qui l'a déclenché (*event bubbling*)
- Certains événements ont une action par défaut qui leur est associée:
  - Clic sur un hyperlien: se déplacer à l'URL indiquée
  - Clic sur bouton de type **submit** dans un formulaire: envoyer avec une requête POST les données d'un formulaire à un URL spécifié

Interface Event

- Propriétés:
  - **bubbles** : indique si l'événement doit être propagé
  - **target** : l'élément sur lequel l'événement a été déclenché
  - **cancelable :** si on peut annuler le comportement par défaut
  - **type** : le nom de l'événement
- Méthodes:
  - **preventDefault()**: annule l'événement
  - **stopPropagation()**: interrompt la propagation de l'événement
- Certains événements ont aussi une propriété **relatedTarget**, qui identifie un élément secondaire (par exemple, pour un événement *mouseover*, ce sera l'élément d'où provient la souris)
- Selon le type spécifique d'événement, d'autres propriétés peuvent apparaître

Événements de formulaires

- **submit** et **reset**
  - Définis en spécifiant le type d'un bouton dans le formulaire comme "submit" ou "reset"
  - Appeler *submit*() directement ne lance pas l'événement. Faut plutôt utiliser *requestSubmit()*
- Différents types d'événement selon le type d'input
  - **input** est lancé chaque fois que le contenu d'un input est modifié

Événements de fenêtre

- Le plus important est **load**, lancé quand toute la page et ses ressources additionnelles sont chargées et affichées
  - Souvent les scripts simples qui manipulent le DOM sont mis dans le gestionnaire de l'événement **load** :
    - window.onload = function() { // notre script }
- Autres événements:
  - **unload**, **beforeunload**
  - **focus**, **blur**
  - **resize**, **scroll**

Événements de souris

- Déclenchés par l'élément le plus imbriqué parmi ceux qui se trouvent sous la souris
  - Ceci n'est pas toujours le cas : mouseenter est déclenché sur chaque élément de l'hierarchie vs mouseover qui est propagé
- Contiennent des propriétés supplémentaires indiquant la position de la souris et l'état des boutons
- Exemples: **mousemove**, **mousedown**, **mouseup**, **click**, **dblclick**, **mouseover**, **mouseout**
- Remarques sur **mouseout** et **mouseover** et événements similaires:
  - L'événement a une propriété **relatedTarget** pour indiquer l'élément d'où on vient (ou celui où l'on va)
  - La propagation nous oblige à vérifier si l'élément en question est réellement celui qui nous intéresse

Événements de clavier

- Ils sont associés à l'élément qui a le focus et ils sont propagés aux objets **document** et **window**
- Événements de bas niveau: **keydown** et **keyup**
- Si la touche de clavier correspond à un caractère, on a en plus un événement **keypress**, qui spécifie le caractère en question
  - **keypress** est obsolète dû à sa dépendance à l'appareil. La propriété **key** de **KeyboardEvent** est à utiliser

Événements du DOM (niveau 3)

- **focusin** et **focusout**, sont des alternatives à **focus** et **blur**, mais qui se propagent
- **mouseenter** et **mouseleave** sont des alternatives à **mouseover** et **mouseout**, mais qui ne se propagent pas
  - Si l'hierarchie est trop profonde, ceci peut avoir un impact sur la performance vu que chaque élément relance l'événement.
- Événement **input**, qui detecte le changement d'un **input, textarea** ou un élément avec l'attribut **contenteditable**
  - Lancé à chaque changement de la valeur versus **change** (malgér son nom), qui est lancé seulement après une confirmation (bouton Enter, sélection dans une liste, etc.)

Événements de HTML5

- Événements associés aux balises <audio> et <video>: **canplay**, **playing**, **pause**, etc.
- Événements associés à l'API drag and drop: **dragstart**, **drag**, **dragenter**, **dragend**, **drop**, **ended**, etc.
- Gestion de l'historique: **hashchange**, **popstate**
- Applications hors-ligne: **cached**, **checking**, **downloading**, **progress**, etc.
- Gestion du stockage local: **storage**

Assignation d'un gestionnaire d'événement

```
element.addEventListener(événement,gestionnaire,capture)
```

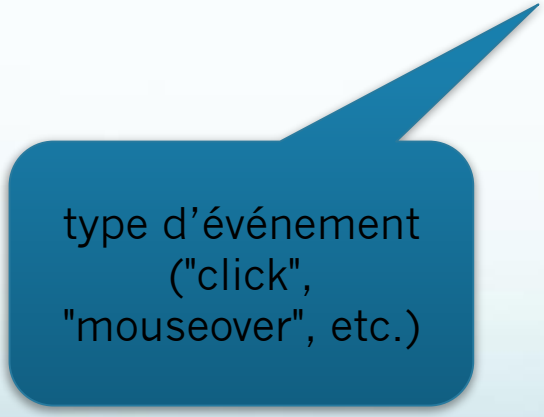
Assignation d'un gestionnaire d'événement

```
element.addEventListener(événement,gestionnaire,capture)
```

Élément du DOM auquel on veut assigner le gestionnnaire

Assignation d'un gestionnaire d'événement

```
element.addEventListener(événement,gestionnaire,capture)
```



Assignation d'un gestionnaire d'événement

```
element.addEventListener(événement,gestionnaire,capture)
```

Une fonction qui représente le gestionnaire d'événement (exécutée chaque fois que l'événement sera déclenché)

Assignation d'un gestionnaire d'événement

```
element.addEventListener(événement,gestionnaire,capture)
```

true si on on veut que le gestionnaire soit exécuté lors de la phase de capture, **false** s'il doit être exécuté lors de la phase de propagation

Assignation d'un gestionnaire d'événement

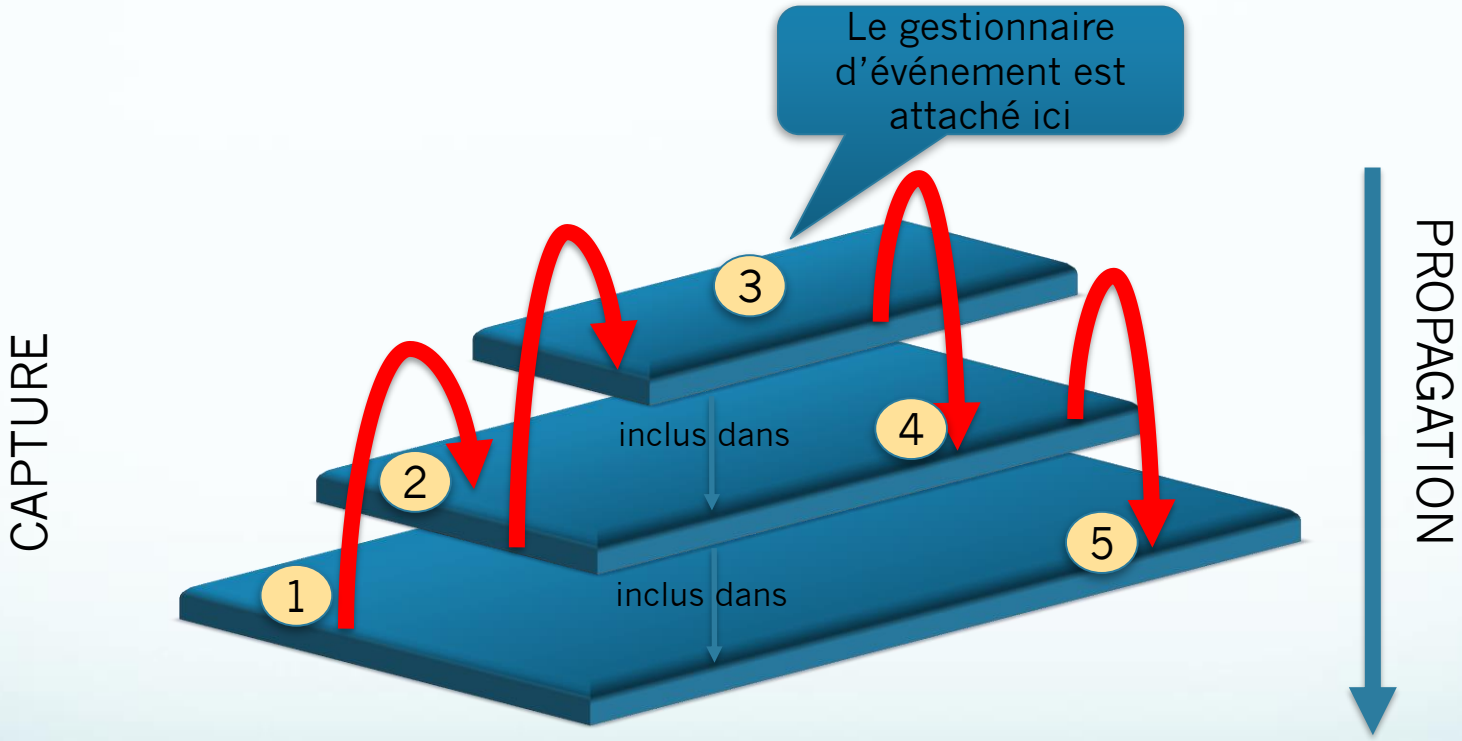
```
element.addEventListener(événement,gestionnaire,options)
```

Un objet qui représente des différentes options autres que juste la capture

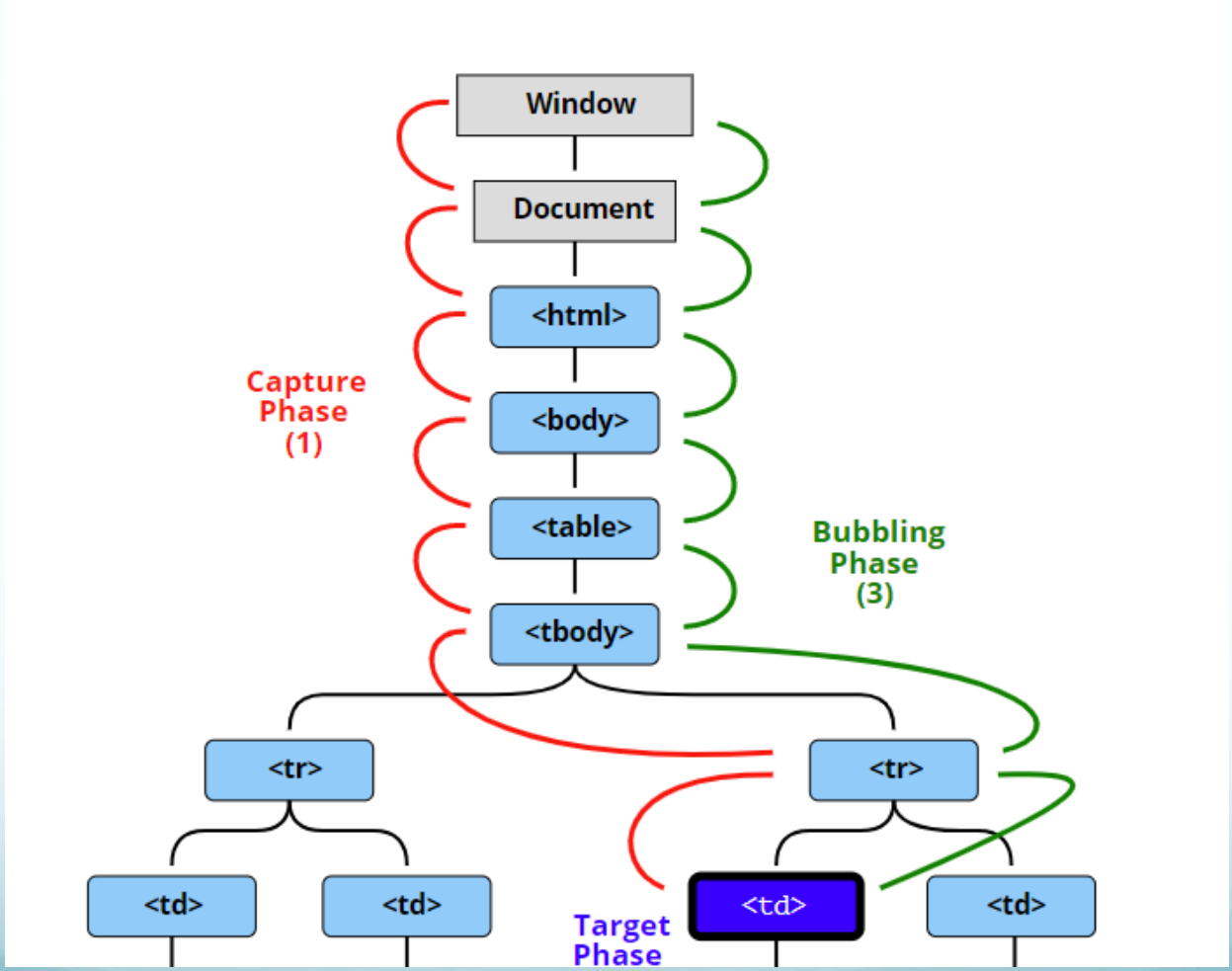
Suppression des gestionnaires d'événements

- Il est possible d'enlever un gestionnaire enregistré avec addEventListener() en utilisant removeEventListener(*type*,*listener[, options]*)
  - Les paramètres suivants doivent être pareils que lors de l'ajout du *listener* : *type*, *listener* et le paramètre *capture* dans *options*, si utilisé.
  - Si listener est défini avec une **arrow function** directement dans *addEventListener()*, il n'est pas possible de l'enlever puisqu'on n'a pas de référence à passer à *removeEventListener()*
- Si on clone un noeud avec *cloneNode()*, les gestionnaires d'événements ne sont pas copiés
  - Pratique si on veut enlever des gestionnaires crées avec des *arrow functions*

Propagation d'événements



Propagation d'événements



Suppression de la propagation

- Il est possible d'arrêter la propagation d'un événement dans l'hiérarchie
  - La méthode *stopPropagation()* bloque la propagation au-dela de l'élément ou elle est invoquée. Ceci arrête un seul événement et ne bloque pas le comportement par défaut.
  - La méthode *preventDefault()* bloque le comportement de gestion par défaut de l'événement, mais ne bloque pas la propagation.
    - Note: l'événement doit être annulable (*cancelable*), sinon rien ne se passe ex: scroll
  - La méthode *stopImmediatePropagation()* annule tous les gestionnaires subséquents du même événement.

Exemple - événement **click**

```
img {
    border: solid red 1px;
}

function fixerLargeur(largeur) {
  document.getElementById("img1").style.borderWidth = largeur + "px";
}
const boutonAgrandir = document.querySelector('#boutonAgrandir’);
const boutonReduire = document.querySelector('#boutonReduire');
boutonAgrandir.addEventListener('click', function(event){ fixerLargeur(20); }, false);
boutonReduire.addEventListener('click', function(event){ fixerLargeur(5); }, false);

<body>
<p>
  <img id="img1"
            src="logoPoly.jpeg"
            width="480" height="240"
            alt="logo Poly">
</p>
<form name="UnFormulaire">
  <input id="boutonAgrandir"
              type="button"
              value="Agrandir la bordure à 20px" />
  <input id="boutonReduire »
              type="button »
              value="Réduire la bordure à 5px" />
</form>
</body>
```

Exemple - événement **click**

```
img {
    border: solid red 1px;
}

function fixerLargeur(largeur) {
  document.getElementById("img1").style.borderWidth = largeur + "px";
}
const boutonAgrandir = document.querySelector('#boutonAgrandir’);
const boutonReduire = document.querySelector('#boutonReduire');
boutonAgrandir.addEventListener('click', function(event){ fixerLargeur(20); }, false);
boutonReduire.addEventListener('click', function(event){ fixerLargeur(5); }, false);

<body>
<p>
  <img id="img1"
            src="logoPoly.jpeg"
            width="480" height="240"
            alt="logo Poly">
</p>
<form name="UnFormulaire">
  <input id="boutonAgrandir"
              type="button"
              value="Agrandir la bordure à 20px" />
  <input id="boutonReduire »
              type="button »
              value="Réduire la bordure à 5px" />
</form>
</body>
```

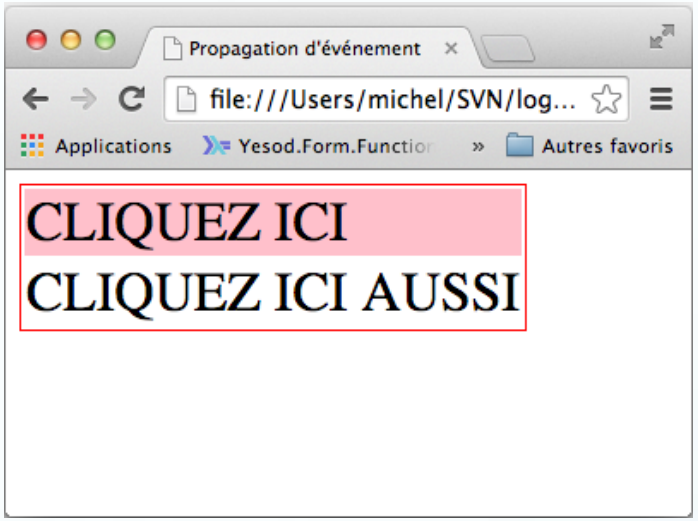
Exemple -propagation

```
table {
   border: 1px solid red;
   font-size: xx-large;
}
#col1 { background-color: pink; }

function gestionnaire(e) {
  col2 = document.getElementById("col2");
  col2.innerHTML = "Allo";
  e.stopPropagation();
  alert("Propagation de l'événement interrompue");
}

window.onload = function () {
  elem = document.getElementById("ligne1");
  elem.addEventListener("click",gestionnaire, false);
  elem = document.getElementById("tableau");
  elem.addEventListener("click",function(e) {alert('Coucou!');}, false);
}
```

```
<body>
  <table id="tableau">
     <tr id="ligne1"><td id="col1">CLIQUEZ ICI</td></tr>
     <tr id="ligne2"><td id="col2">CLIQUEZ ICI AUSSI</td></tr>
  </table>
</body>
```

Exemple -propagation

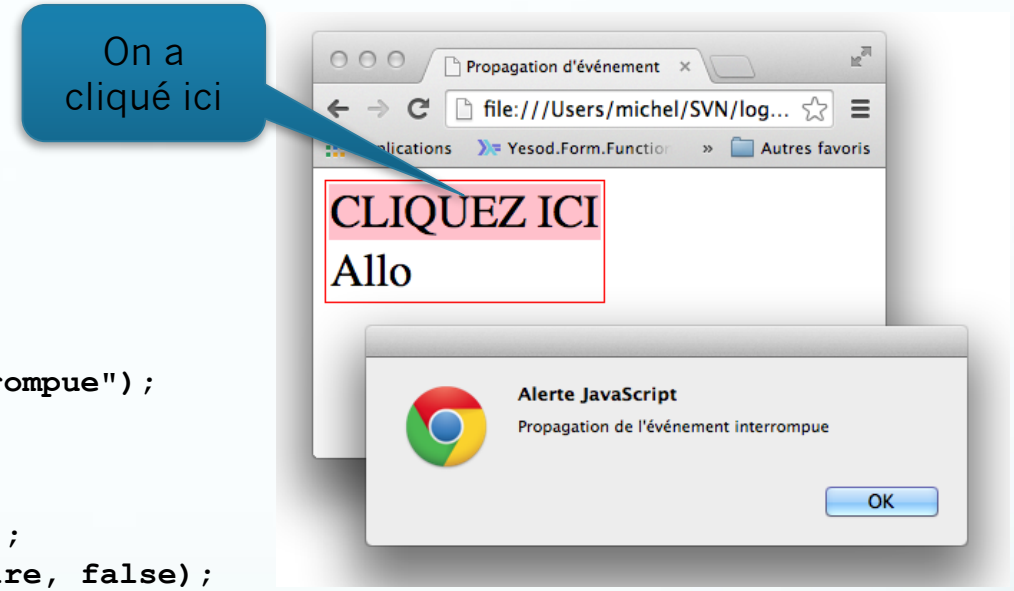
```
table {
   border: 1px solid red;
   font-size: xx-large;
}
#col1 { background-color: pink; }

function gestionnaire(e) {
  col2 = document.getElementById("col2");
  col2.innerHTML = "Allo";
  e.stopPropagation();
  alert("Propagation de l'événement interrompue");
}

window.onload = function () {
  elem = document.getElementById("ligne1");
  elem.addEventListener("click",gestionnaire, false);
  elem = document.getElementById("tableau");
  elem.addEventListener("click",function(e) {alert('Coucou!');}, false);
}
```




```
<body>
  <table id="tableau">
     <tr id="ligne1"><td id="col1">CLIQUEZ ICI</td></tr>
     <tr id="ligne2"><td id="col2">CLIQUEZ ICI AUSSI</td></tr>
  </table>
</body>
```

Exemple -propagation

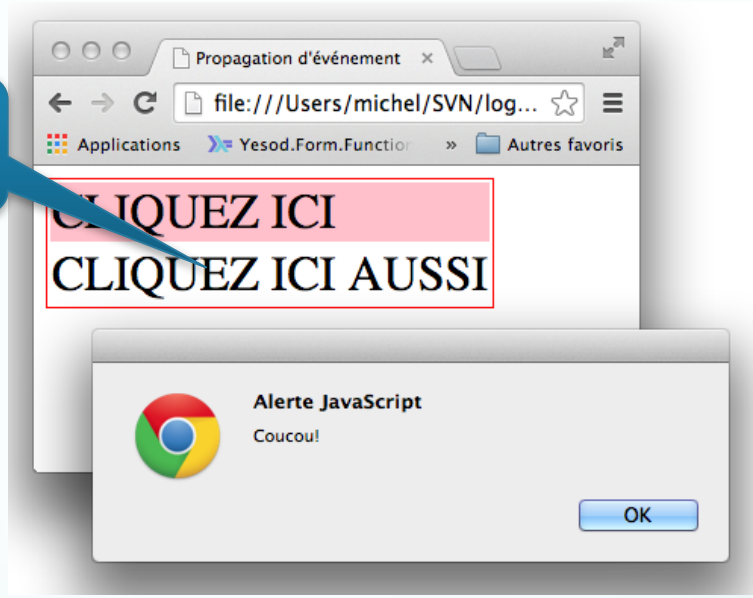
```
table {
   border: 1px solid red;
   font-size: xx-large;
}
#col1 { background-color: pink; }

function gestionnaire(e) {
  col2 = document.getElementById("col2");
  col2.innerHTML = "Allo";
  e.stopPropagation();
  alert("Propagation de l'événement interrompue");
}

window.onload = function () {
  elem = document.getElementById("ligne1");
  elem.addEventListener("click",gestionnaire, false);
  elem = document.getElementById("tableau");
  elem.addEventListener("click",function(e) {alert('Coucou!');}, false);
}
```




```
<body>
  <table id="tableau">
     <tr id="ligne1"><td id="col1">CLIQUEZ ICI</td></tr>
     <tr id="ligne2"><td id="col2">CLIQUEZ ICI AUSSI</td></tr>
  </table>
</body>
```

Exemple -propagation

```
table {
   border: 1px solid red;
   font-size: xx-large;
}
#col1 { background-color: pink; }

function gestionnaire(e) {
  col2 = document.getElementById("col2");
  col2.innerHTML = "Allo";
  e.stopPropagation();
  alert("Propagation de l'événement interrompue");
}

window.onload = function () {
  elem = document.getElementById("ligne1");
  elem.addEventListener("click",gestionnaire, false);
  elem = document.getElementById("tableau");
  elem.addEventListener("click",function(e) {alert('Coucou!');}, false);
}
```




```
<body>
  <table id="tableau">
     <tr id="ligne1"><td id="col1">CLIQUEZ ICI</td></tr>
     <tr id="ligne2"><td id="col2">CLIQUEZ ICI AUSSI</td></tr>
  </table>
</body>
```