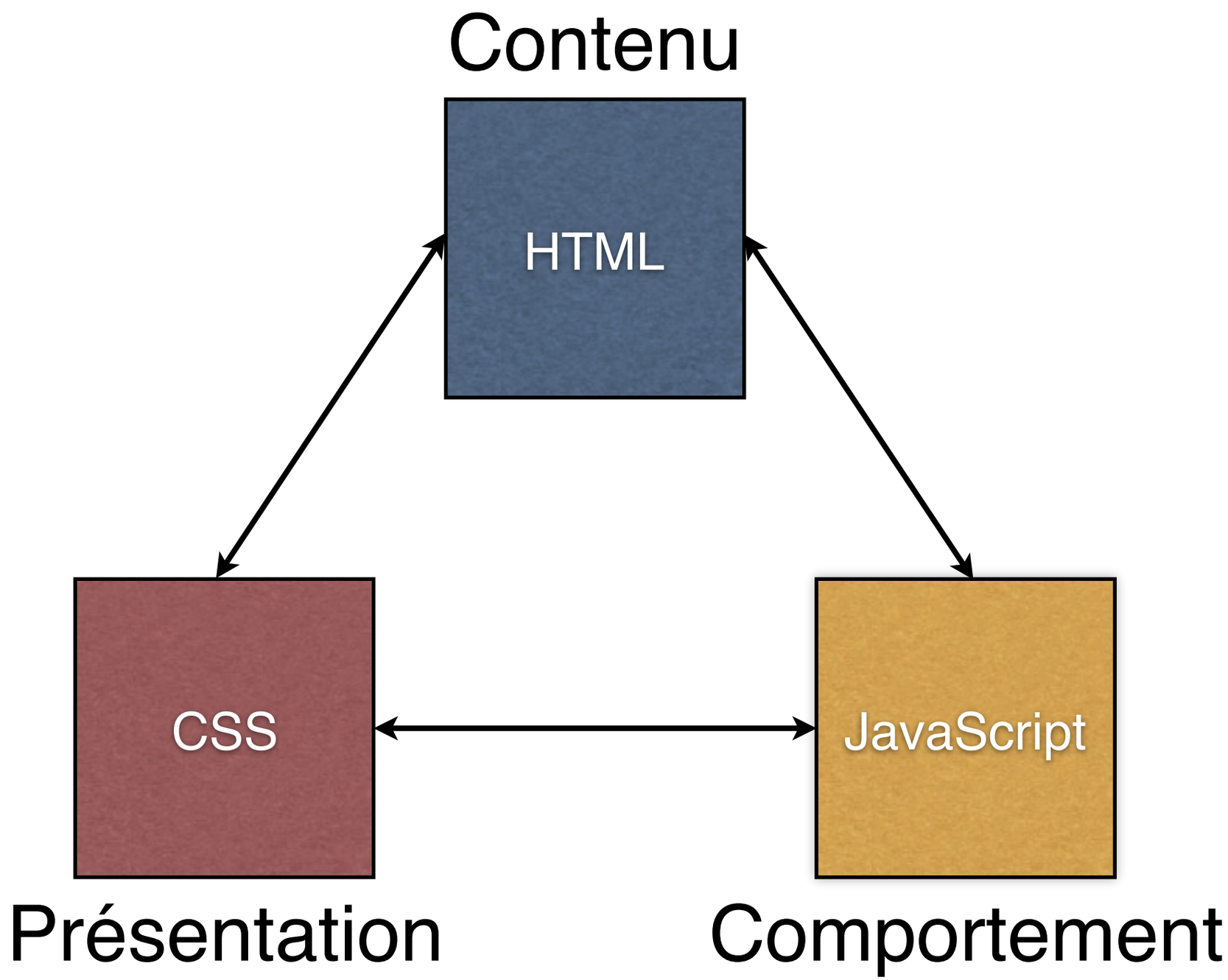
# Javascript dans le navigateur

Matthieu Nicolas
Licence Pro CIASIE
Slides par Christophe Bouthier

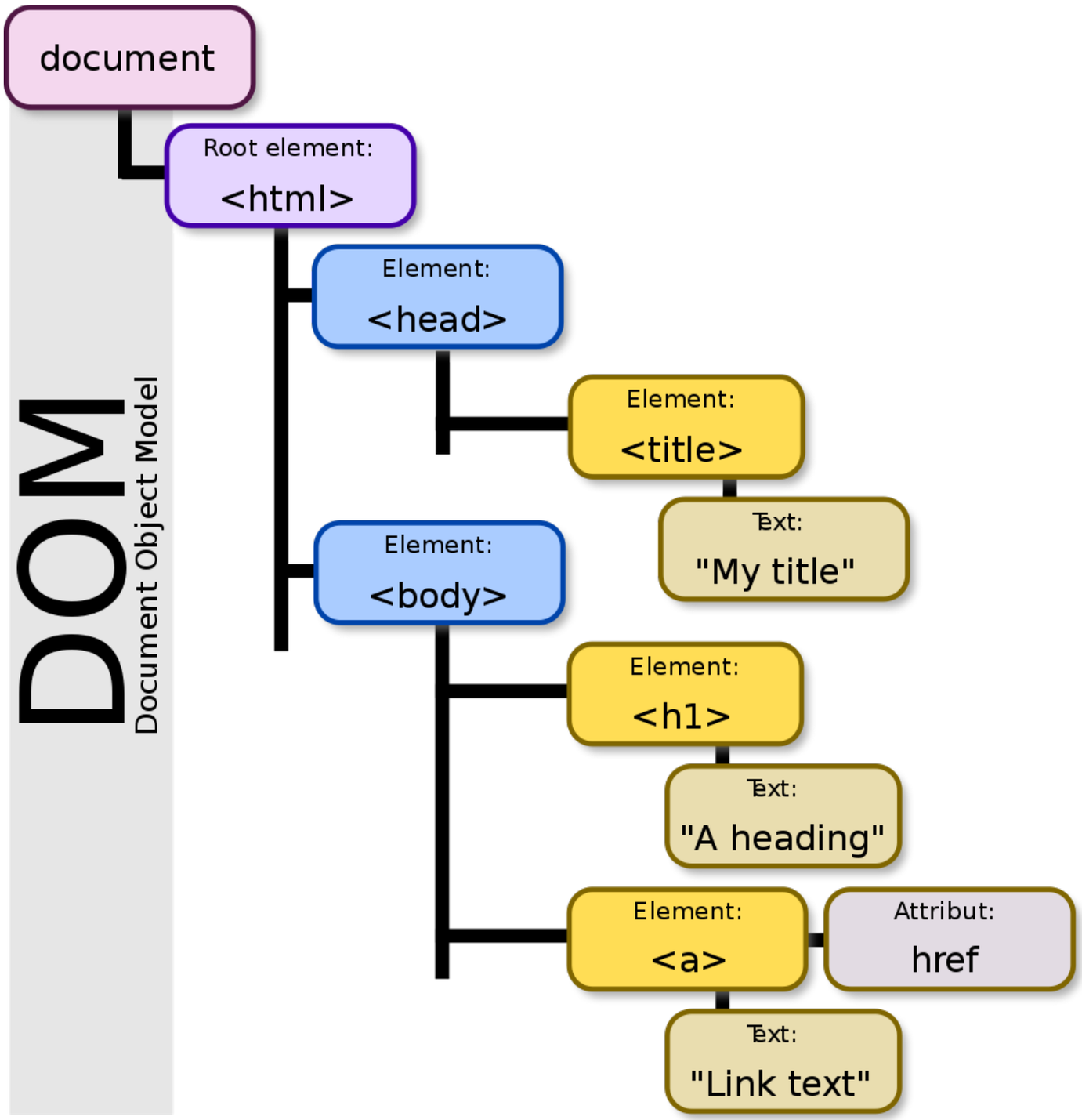
# Architecture Web



# Le DOM

- **D**ocument **O**bject **M**odel
- Hiérarchie de noeuds représentant les éléments HTML de la page web
  - Structure arborescente

# Le DOM


DOM
Document Object Model
document
Root element:
<html>
Element:
<head>
Element:
<title>
Text:
"My title"
Element:
<body>
Element:
<h1>
Text:
"A heading"
Element:
<a>
Attribut:
href
Text:
"Link text"

# Interface avec le DOM

- Le DOM fournit une API langage-agnostique
  - Sélectionner et accéder à des éléments du DOM
  - Créer, insérer, modifier et supprimer des éléments du DOM
  - Réagir à des évènements produits par les éléments du DOM
- Implémentée par JavaScript

# Plan

- L’accès au DOM
- Manipulation du DOM
- Les événements
- Les timers et le thread unique

# L’accès au DOM

Javascript dans le navigateur

# Accès au DOM

- `window.document`
  - Racine du DOM
  - Possède les méthodes “globales”
- Différents types de noeuds
  - Element : balise HTML (aussi appelée tag HTML)
  - Text : texte

# Accès aux éléments

- Plusieurs méthodes existantes
  - Via le tag HTML
  - Via les informations CSS (id, class)
  - Via un sélecteur CSS

# Exemples

- Utilise la page HTML suivante pour les prochains exemples

```html
<!DOCTYPE html>
<head>
    <meta charset="utf8">
    <title>Example</title>
</head>
<body>
    <main>
        <article id="article1" class="article">Article 1</article>
        <article id="article2" class="article">Article 2</article>
    </main>
    <nav>Menu</nav>
</body>
```

# Accès aux elements

- `document.getElementsByTagName("tag")`
  - Retourne une liste d'éléments
  - `"tag"` correspond au nom des éléments (*a*, *img*, *div*…)

```
>> document.getElementsByTagName("article")
<- ▶ HTMLCollection { 0: article#article1 , 1: article#article2 , length: 2, … }
>> document.getElementsByTagName("article")[0]
<- ▶ <article id="article1">
>> document.getElementsByTagName("article")[0] instanceof HTMLElement
<- true
```

  - Retourne TOUS les éléments, pas juste les sous-fils

# Accès aux elements

- Possible d'appeler `getElementsByTagName` sur un élément HTML
  - Même comportement que précédemment
  - Limite juste la recherche au sous-arbre de l'élément

```
>> const mainElt = document.getElementsByTagName("main")[0]
<- undefined
>> mainElt.getElementsByTagName("article")
<- ▶ HTMLCollection { 0: article#article1 , 1: article#article2 , length: 2, … }
```

# Accès par CSS

- `document.getElementByID("id")`
  - Retourne un seul élément
  - Sélectionné à partir de son ID CSS

```
>> document.getElementById("article1")
<- ▶ <article id="article1" class="article">
```

- Les identifiants CSS doivent être uniques !
  - Se contente sinon de retourner le premier élément correspondant

# Accès par CSS

- `document.getElementsByClassName("cl")`
  - Retourne une liste d'éléments
  - Sélectionnés par leur classe CSS

```
>> document.getElementsByClassName("article")
← ▶ HTMLCollection { 0: article#article1.article , 1: article#article2.article , length: 2, … }
```

  - De la même manière que `getElementsByTagName`, peut être appelé à partir d'un élément HTML pour limiter la recherche à son sous-arbre

# Selector API

- Fortement inspirée des idées de jQuery
- Permet d'utiliser un sélecteur CSS pour récupérer un ou plusieurs éléments
- Prend la forme de deux méthodes
  - `querySelector`
  - `querySelectorAll`

# Sélecteur CSS

- `"#myId"` sélectionne l'élément avec `"id=myId"`
- `".myClass"` sélectionne les éléments avec `"class=myClass"`
- `"tag"` sélectionne les éléments `<tag>`

# Combinaison de sélecteurs CSS

- Possible de combiner plusieurs sélecteurs CSS pour être plus précis
  - `"section.main"` sélectionne les éléments `<section>` avec `"class=main"`
- Pour aller plus en détails:
  - https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors

# Selector API

- `document.querySelector("selector")`
  - Retourne le premier élément correspondant

```
» document.querySelector("#article1")
← ▶ <article id="article1" class="article">
```

  - De la même manière que `getElementsByTagName`, peut être appelé à partir d'un élément HTML pour limiter la recherche à son sous-arbre

# Selector API

- `document.querySelectorAll("selector")`
  - Retourne cette fois-ci tous les éléments correspondants

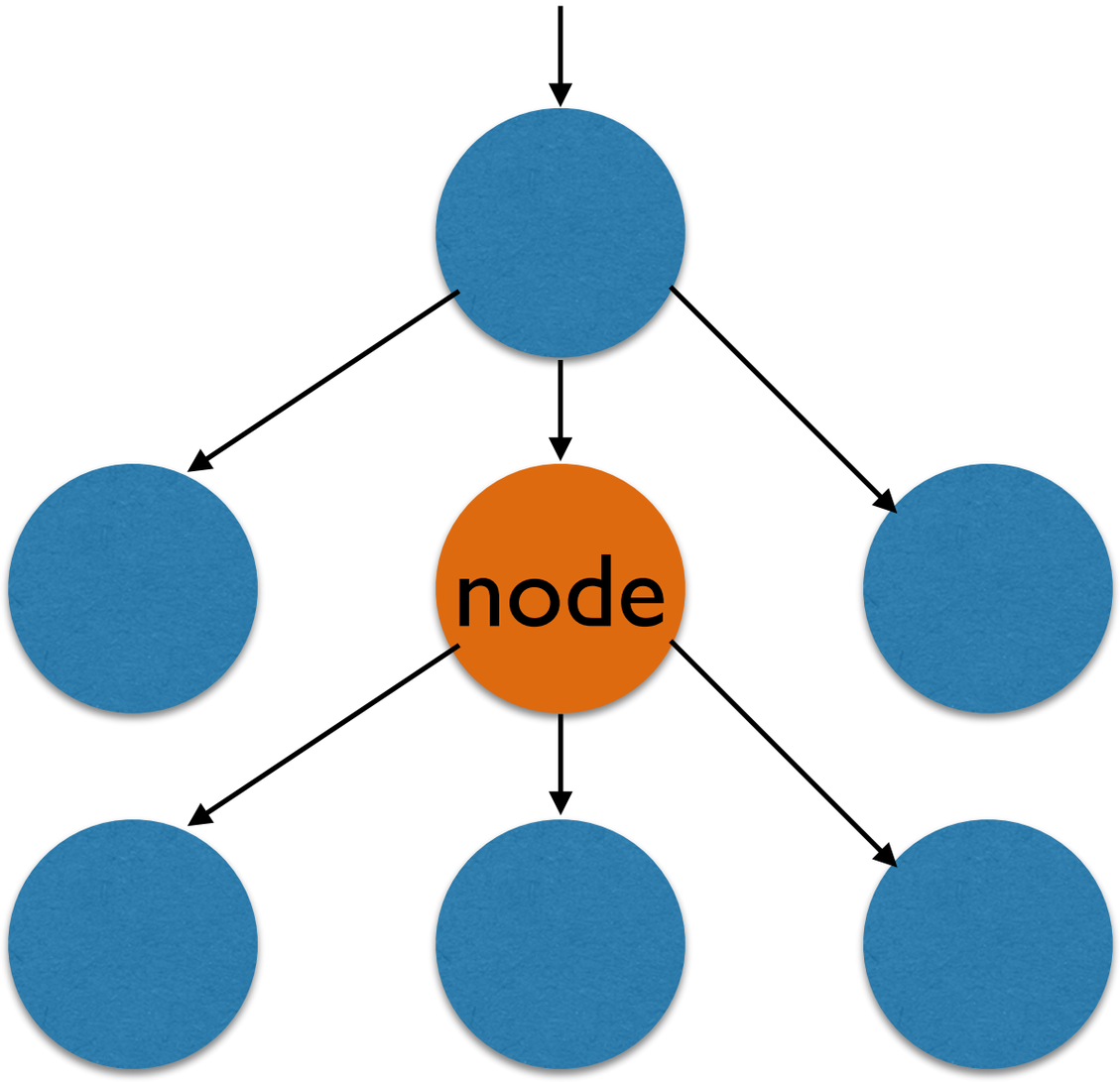
```
>> document.querySelectorAll("article.article")
<- ▶ NodeList [ article#article1.article , article#article2.article ]
```

- De la même manière que `getElementsByTagName`, peut être appelé à partir d'un élément HTML pour limiter la recherche à son sous-arbre

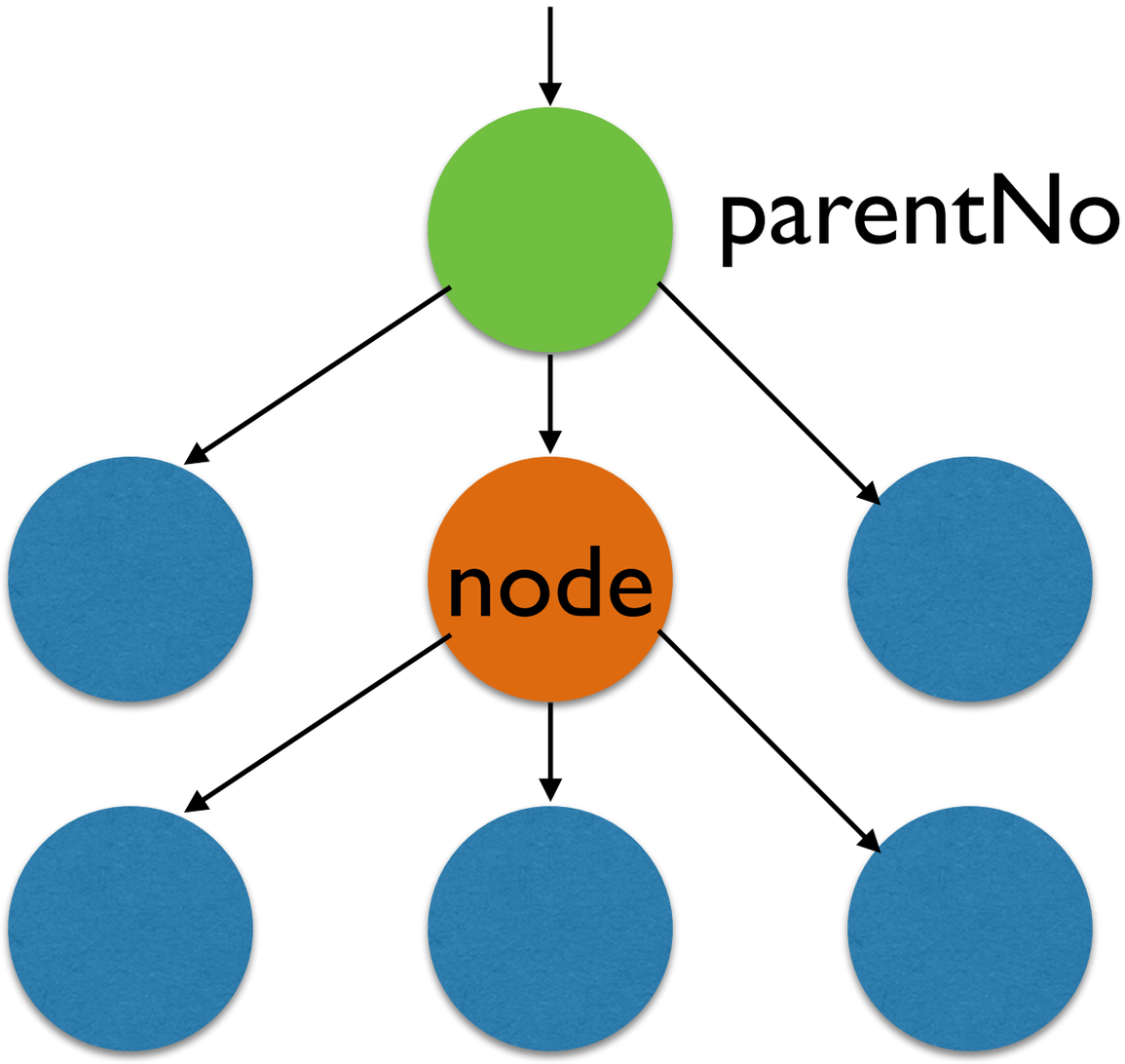
# Navigation à partir d'un noeud

- Différentes propriétés permettent de naviguer dans le DOM depuis un noeud qu'on a récupéré précédemment
  - `parentNode`
  - `childNodes`
  - `firstChild, lastChild`
  - `nextSibling, previousSibling`

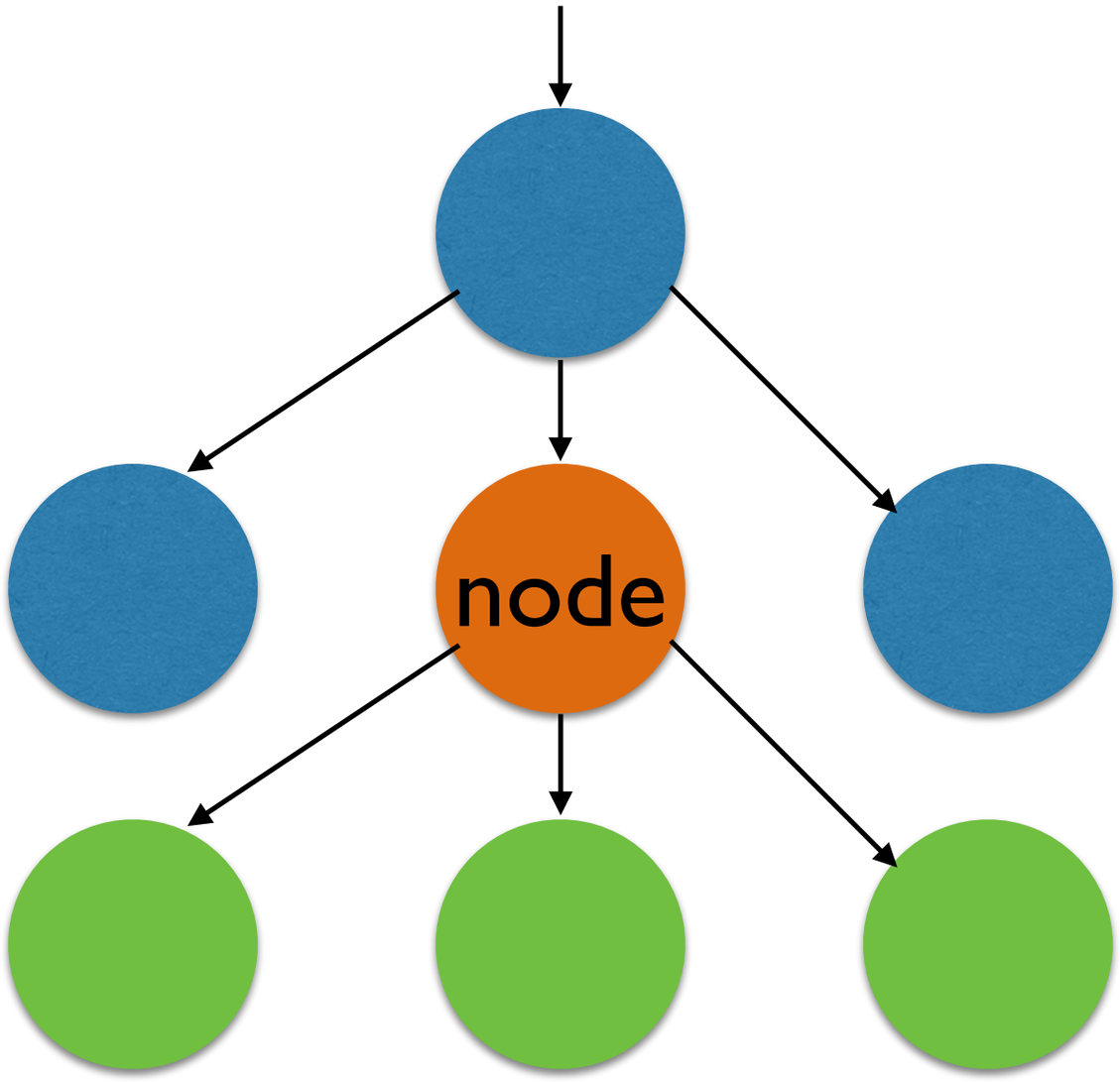
# Navigation



# Navigation

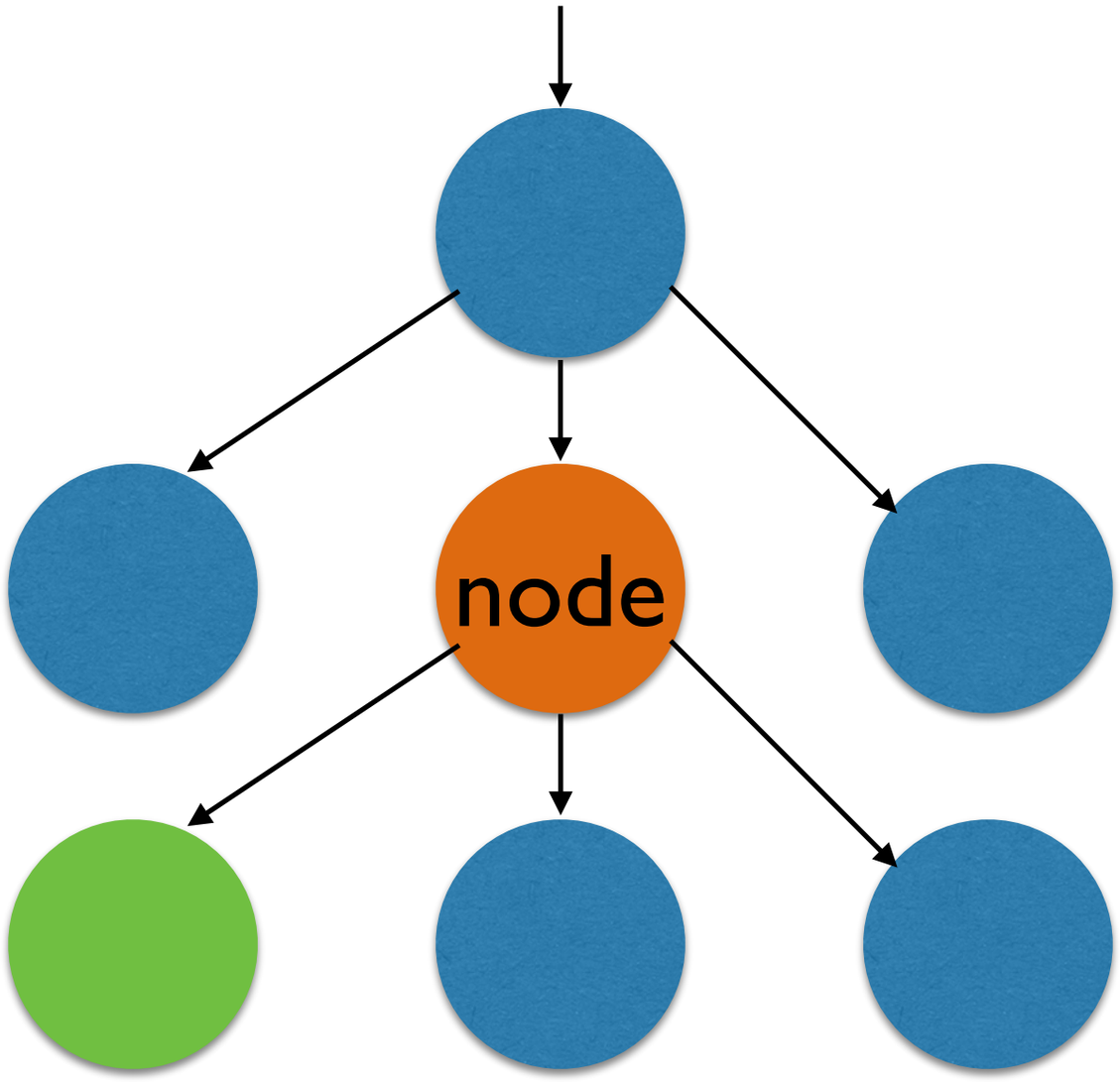


# Navigation



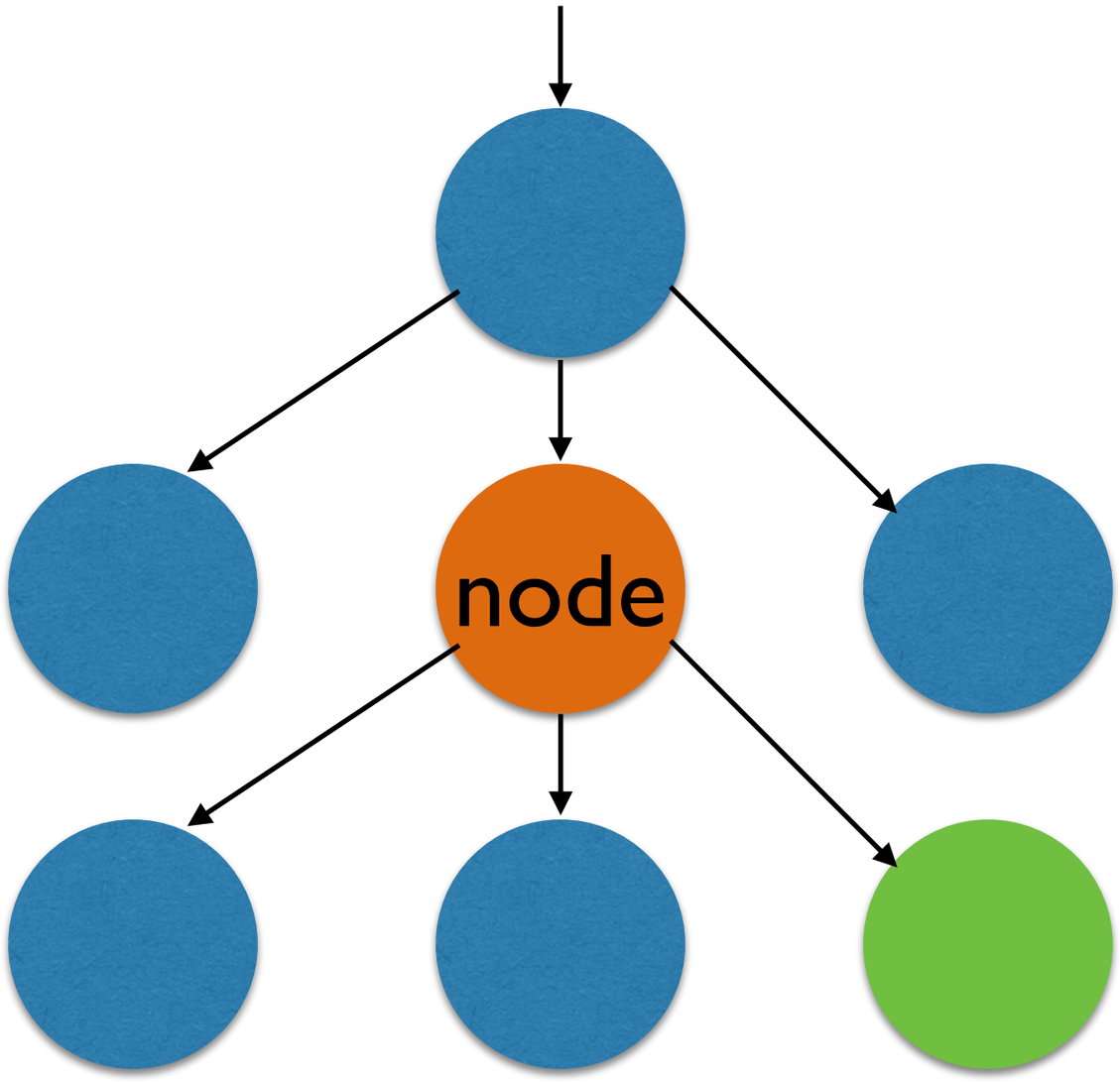

childNodes

# Navigation




firstChild

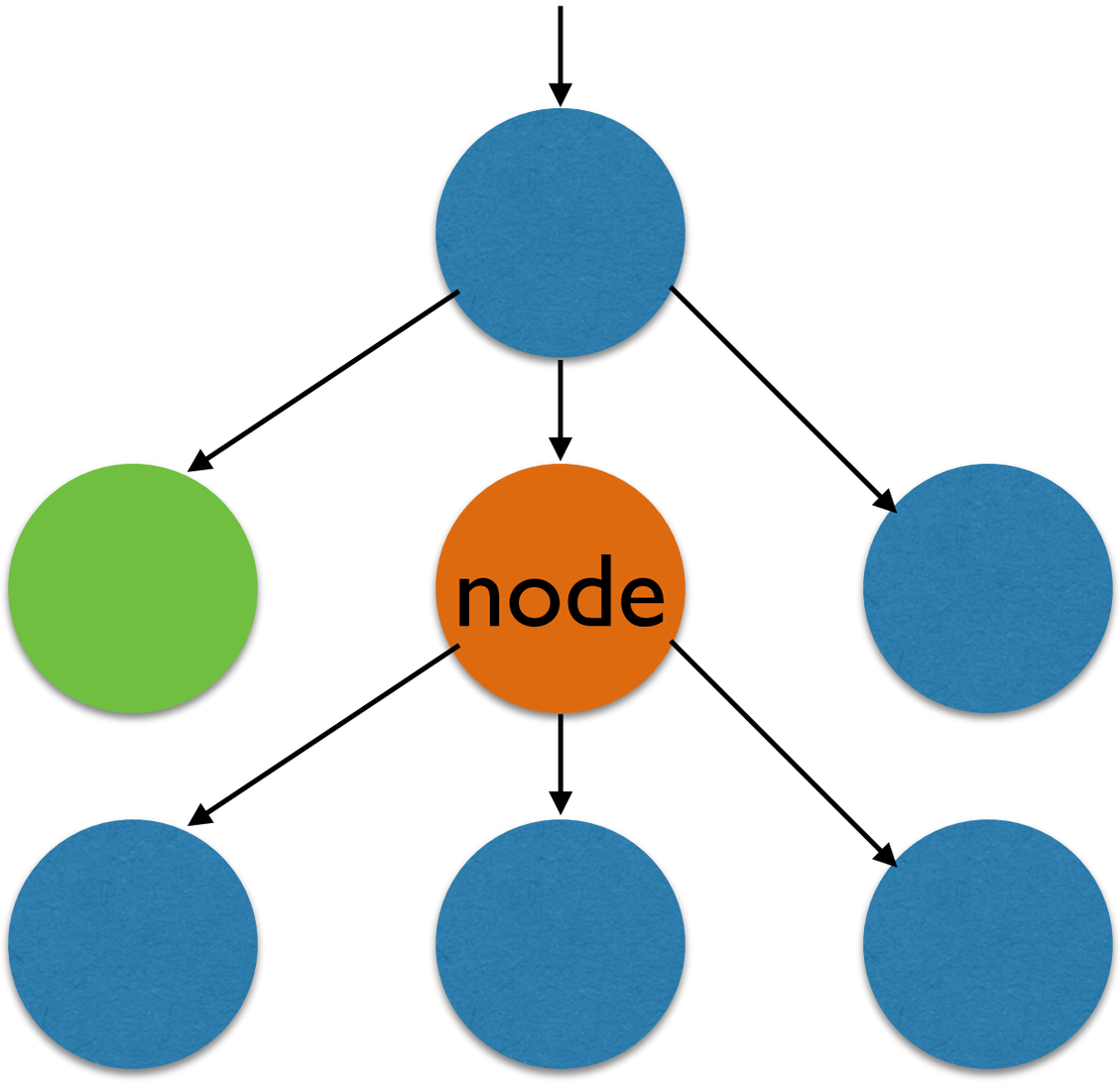
# Navigation



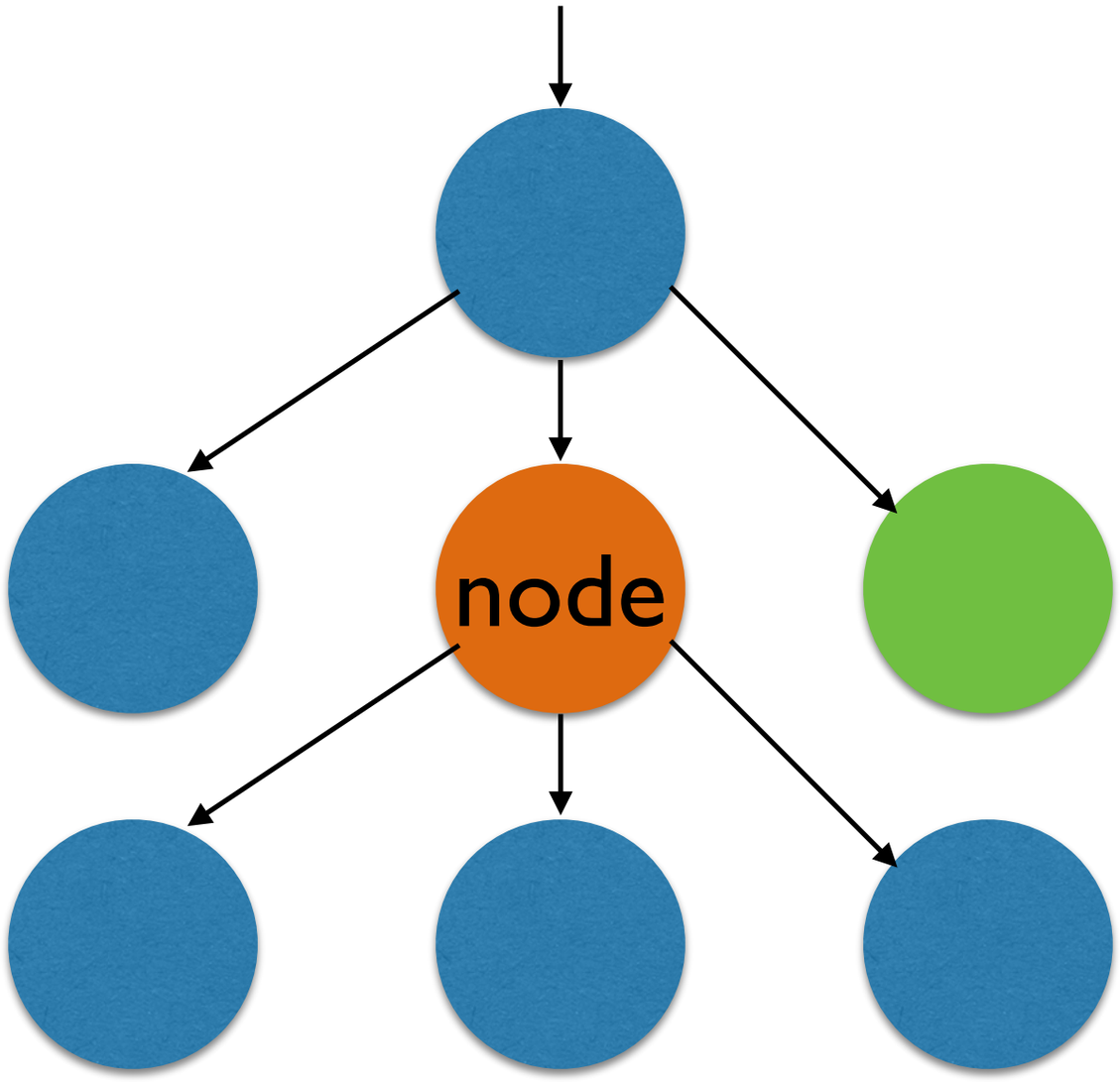

lastChild

# Navigation

previousSibling



# Navigation




nextSibling

# Manipulation du DOM

Javascript dans le navigateur

# Différents types de noeuds

- Deux types de noeuds
  - Noeud Element
  - Noeud Texte
- Chacun avec leurs propriétés

# Noeud Element

- Correspond aux balises HTML

```
>> const mainElt = document.querySelector("main")
<- undefined
>> mainElt.tagName
<- "MAIN"
>> mainElt.innerHTML
<- "
            <article id=\"article1\" class=\"article\">Article 1</article>
            <article id=\"article2\" class=\"article\">Article 2</article>
        "
>> const article1Elt = document.querySelector("#article1")
<- undefined
>> article1Elt.className
<- "article"
>> article1Elt.id
<- "article1"
```

# Noeud Element

- Liste des balises HTML et de leur rôle
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Element
- Liste de leurs attributs
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes

# Noeud Texte

- Correspond à chaque zone de texte entre des balises
  - Même les retours chariots

```
» const node = document.querySelector("main").firstChild
← undefined
» node.nodeName
← "#text"
» node.nodeValue
← "
            "
```

# Accès aux attributs

- Deux types d'accès
  - `elt.attribut`
  - `elt.getAttribute("name")`

```
» const article1Elt = document.querySelector("#article1")
← undefined
» article1Elt.id
← "article1"
» article1Elt.getAttribute("id")
← "article1"
```

# Pourquoi ?

- Noms d’attributs incompatibles avec certains langages

```
» const article1Elt = document.querySelector("#article1")
← undefined
» article1Elt.class
← undefined
» article1Elt.getAttribute("class")
← "article"
» article1Elt.className
← "article"
```

- Plus d’infos:
  - https://developer.mozilla.org/en-US/docs/Web/API/Element/className#Notes

# Pourquoi - bis ?

- Distinction entre le contenu de l'attribut et l'**I**nterface **D**efinition **L**angage (**IDL**) de l'attribut

```
» document.querySelector("input").type
← "text"
» document.querySelector("input").getAttribute("type")
← "foobar"
```

- Plus d'infos:
  - https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes#Content_versus_IDL_attributes

# Autre exemple

```
» document.querySelector("img").src
← "file:///Users/matthieu/Documents/teachings/2019-2020/progweb/session3/td/img/hello%20world.png"
» document.querySelector("img").getAttribute("src")
← "./img/hello world.png"
```

- L'IDL va normaliser la valeur de l'URL, pour éviter la présence de caractères spéciaux
- `getAttribute` va renvoyer la valeur telle quelle

# CSS d’un élément

- Modifiable via l’attribut `"style"`
  - À la fois sous forme d’une chaîne de caractères (mauvaise pratique)
  - À la fois sous forme d’un objet

```
» document.querySelector("#article1").style="color: blue;"
← "color: blue;"
» document.querySelector("#article1").style
← ▸ CSS2Properties { color → "blue" }
» document.querySelector("#article1").style.color = "red"
← "red"
» document.querySelector("#article1").style
← ▸ CSS2Properties { color → "red" }
```

# CSS d’un élément

- Attention: la valeur de l’attribut `"style"` d’un élément est différente du CSS réel de l’élément
  - Le style de l’élément peut être complété ou surchargé par des règles CSS
- Pour obtenir le style réel d’un élément, utiliser la fonction `getComputedStyle()`
  - Voir https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style#Getting_style_information

# Création d'un noeud

- `document.createElement("tag")` permet de créer un élément…
- … mais il n'est pas encore ajouté au DOM

```
» const article3Elt = document.createElement("article")
← undefined
» article3Elt.parentNode
← null
» article3Elt.previousSibling
← null
```

# Modification d’un noeud

- C’est un objet, on peut le modifier !

```
» article3Elt.id = "article3"
← "article3"
» article3Elt.innerHTML = "Article 3"
← "Article 3"
```

# Ajout au DOM

- On dispose de plusieurs méthodes pour ajouter un noeud au DOM
  - `parentNode.appendChild(newNode)`
  - `parentNode.insertBefore(newNode, referenceNode)`

# AppendChild

- `parentNode.appendChild(newNode)`
  - Ajoute `newNode` à la fin de la liste des enfants de `parentNode`

```
>> document.querySelector("main").appendChild(article3Elt)
<- ▶ <article id="article3">
```

# InsertBefore

- `parentNode.insertBefore(newNode, referenceNode)`
  - Ajoute `newNode` en tant qu'enfant de `parentNode`, mais le place avant `referenceNode`

```
>> document.querySelector("main").insertBefore(article3Elt, document.querySelector("#article2"))
← ▶ <article id="article3">
```

# Suppression d’un noeud

- `parentNode.removeChild(node)`
  - Supprime le noeud `node` de la liste des enfants de `parentNode`

```
>> document.querySelector("main").removeChild(article3Elt)
<- ▶ <article id="article3">
```

# Les événements

Javascript dans le navigateur

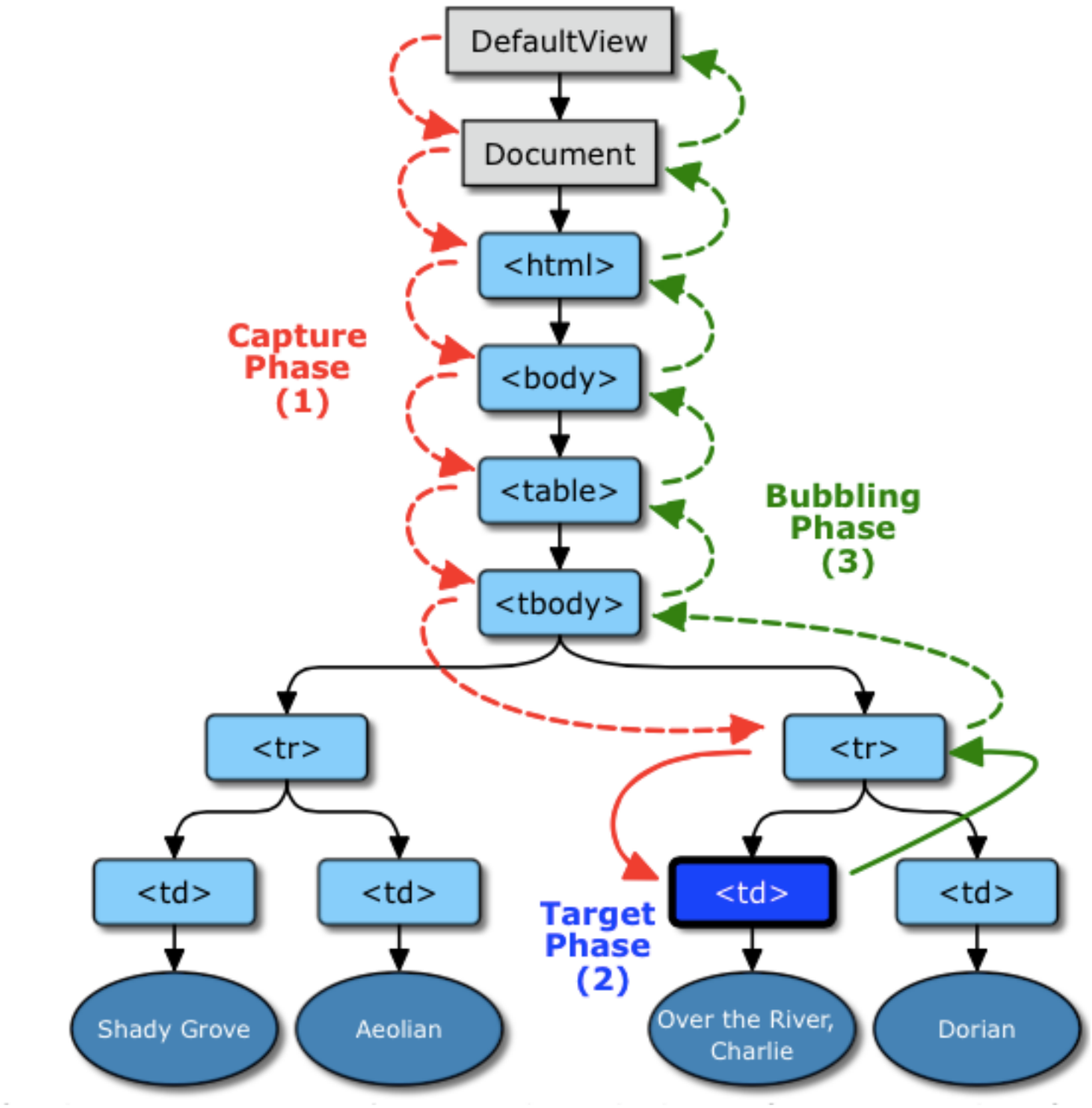
# Les évènements

- Messages, notifications que quelque chose vient de se produire
- Certains sont liés à des éléments du DOM
    - Actions utilisateurs (`click`, `keypress`, ...)
- D’autres au fonctionnement du navigateur
    - État de la connection (`online`, `offline`)
    - Communication avec le serveur (`message`)
- Liste disponible: https://developer.mozilla.org/en-US/docs/Web/Events

# Les évènements du DOM

- Va se concentrer sur les évènements liés aux éléments du DOM dans le cadre de ce cours
- Comments fonctionnent-ils ?
- Comment réagit-on à un évènement ?

# Event propagation




http://www.w3.org/TR/DOM-Level-3-Events/#event-flow

# Capture et Bubbling - 1

- Le mode `Capture` était poussé par NetScape…
- … tandis que le mode `Bubbling` par IE
- Dorénavant, les navigateurs supportent les 2

# Capture et Bubbling - 2

- Mode `Bubbling` par défaut
- + de sens
    - C’est l’élément le plus proche qui se charge de traiter l’évènement
- Pas de différence notable entre les performances

# Réagir aux évènements

- Ajoute un `EventListener` à un élément du DOM
  - S’agit d’une fonction
  - Sera appelée automatiquement par le navigateur
  - Lorsque l’élément du DOM verra passer l’évènement

# addEventListener

- `elt.addEventListener(evtName, callback)`

```
>> const mainElt = document.querySelector("main")
<- undefined
>> mainElt.addEventListener("click", function (event) {
       console.log("IL A CLIQUÉ !")
   })
<- undefined
   IL A CLIQUÉ !
```

# Paramètres de addEventListener - 1

- `evtName`
  - Une chaîne de caractères
  - Le nom de l’évènement ciblé (`"click"`, `"mousedown"`, …)

# Paramètres de addEventListener - 2

- `callback`
  - Une fonction
  - Prend un paramètre, `event`, qui est un objet décrivant l'évènement

# event

- Fournit automatiquement par le navigateur lors de l'appel de `callback`
- Un objet
  - Possède des propriétés
  - Possède des méthodes

# Propriétés de Event

- `target`
  - Élément du DOM où l'event a eu lieu
- `currentTarget`
  - Élément du DOM qui a appelé `callback`
- Mais aussi propriétés spécifiques par types d'évènements
  - `x`, `y` pour `mouseevent`, ...

# Méthodes de Event

- `preventDefault()`
  - Supprime le comportement par défaut du navigateur
  - Par exemple, lors d'un `click` sur `<a>`, bloque l'ouverture de la nouvelle page
- `stopPropagation()`
  - Supprime le reste de la propagation, en `Capture` et/ou en `Bubbling`

# Valeur de retour de callback

- Si `callback` retourne `false`
  - Se comporte comme si `callback` avait déclenché `event.preventDefault()` et `event.stopPropagation()`
- Ne fait rien de particulier dans les autres cas

# Retirer un EventListener

- Peut vouloir supprimer un EventListener attaché à un élément au préalable
- Pour cela, utilise la méthode suivante:
  - `elt.removeEventListener(evtName, callback)`

# Options

- `addEventListener` accepte un 3ème paramètre, qui est optionnel, `options`
- Prend la forme d'un objet avec les propriétés suivantes:
  - `capture`, un booléen, qui indique si le listener est déclenché au cours de la phase `Capture`
  - `once`, un booléen, qui indique que le listener sera supprimé après avoir été exécuté une fois

# Delegation d’event - 1

- Peut vouloir attacher un EventListener aux éléments d’une liste
- Problèmes
  - Lourd
  - Doit gérer les éléments qui sont ajoutés dynamiquement à la liste

# Delegation d’event - 2

- `Bubbling`
  - Si l’évènement n’est pas traité et bloqué au niveau de l’élément, il passe au niveau supérieur
  - On met l’EventListener sur l’élément parent
    - La liste ici
  - On filtre en fonction de `event.target`

# Les timers et le thread unique

Javascript dans le navigateur

# Contexte

- Browser Event Loop
  - Boucle d’évènements
  - Gérée par le navigateur
  - C’est elle qui appelle le script
- Mécanisme de callback
  - Associe aux évènements une ou plusieurs callbacks
- Langage fonctionnel tout à fait pertinent

# Thread unique

- Browser Event Loop
- En charge de tous les évènements
  - Évènements du navigateur (`page load`, ...)
  - Évènements de l'utilisateur (`click`, ...)
  - Évènements du réseau (`AJAX`)
  - Évènements des timers (`timers`)
- Une seule boucle = un seul thread !

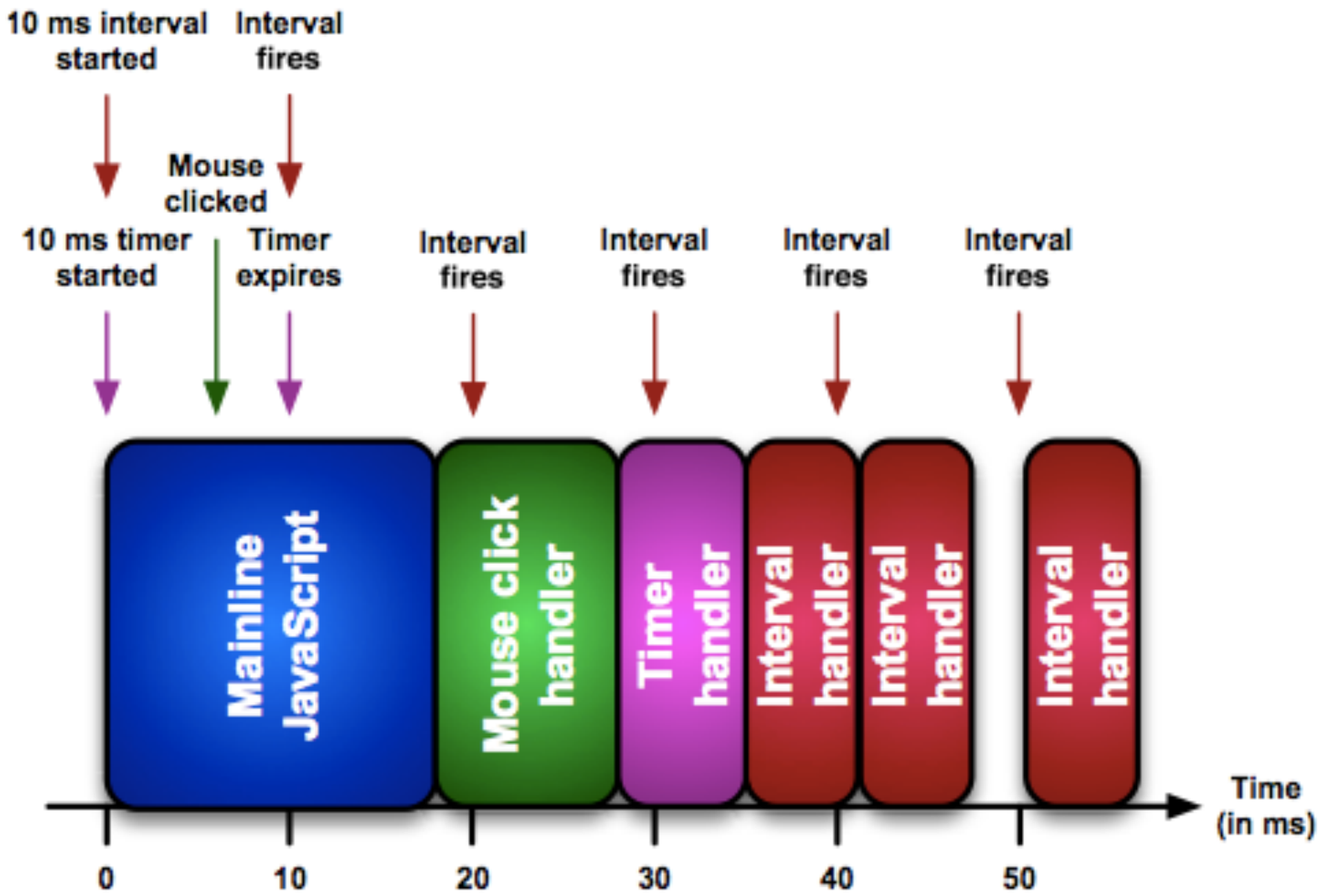
# Timers

- `setTimeout(callback, delay)`
  - Met en place un timer unique
  - La fonction de callback est appelée une seule fois
- `setInterval(callback, delay)`
  - Met en place un timer répétitif
  - La fonction callback est appelée tant que le timer n’est pas annulé
    - À l’aide de la fonction `clearInterval`

# Thread unique !

- Pas de garantie concernant les délais
- Les timers sont enregistrés dans la boucle d’event, et traité les uns après les autres
- Un `setInterval` n’est mis dans la queue que s’il n’est pas DEJA présent dans la queue

# Exemple




source : Secrets of the JavaScript Ninja

# Problèmes

- Pas de garantie sur les temps
- Slow Script Warning
  - Si un script prend trop de temps sans laisser la main à la boucle d’event
  - Message du navigateur demandant à l’utilisateur s’il veut tuer le script
- Plus il y a de timers, plus le navigateur ralenti

# Résumé

- Accès au DOM
  - `getElementByID`, `getElementsByClassName`
  - `querySelector`, `querySelectorAll`
- Event Handlers
  - `addEventListener`, `removeEventListener`
- Timers
  - `setTimeout`, `setInterval`