



Document Object Modeling (DOM)

Asst.Prof. Dr. Umaporn Supasitthimethee

ผศ.ดร.อุมาพร สุภสีทธิเมธี



Document Object Model (DOM)

- Understanding the Document Object Model in JavaScript
- Node Types
- Traversing Nodes
- Manipulating Nodes

DOM -> ไม่ผูกกับภาษาใด
-> เป็นแนวคิดในการสร้างเอกสารเป็น Tree
-> ใช้กับ Markup Language ได้ทุกอัน
-> ถ้าใช้ <script> ตรง Head จะทำให้โหลดนาน
-> ทำ Dynamic Node



Introduction to Document Object Modeling (DOM)

- The Document Object Model (DOM) represents the HTML page so that programs can change the document structure, style, and content.
- The DOM represents a document as a **hierarchical tree of nodes**, allowing developers to select, add, remove, and modify individual parts of the page.
- The DOM is now a truly cross-platform, language-independent way of representing and manipulating pages for markup.
- The DOM is not a programming language, but without it, the JavaScript language wouldn't have any model or notion of web pages, HTML documents, XML documents, and their component parts.



<script> Tag Placement

```
<html>
  <head>
    <title>Example HTML Page</title>
    <script src="script.js"></script>

  </head>
  <body>
    <!-- content here -->
  </body>
</html>
```

- Including all JavaScript files in the `<head>` of a document means that all of the JavaScript code must be downloaded, parsed, and interpreted before the page begins rendering

<script> Tag Placement



```
<html>
  <head>
    <title>Example HTML Page</title>
  </head>
  <body>
    <!-- content here -->
    <script src="script.js"></script>
  </body>
</html>
```

- For pages that **require a lot of JavaScript code**, this can cause a noticeable delay in page rendering, during which time the browser will be completely blank.
- For this reason, modern web applications typically include all JavaScript references in the `<body>` element, after the page content.

ย้าย JS มาอยู่ท้าย body

- จะทำให้ JS เห็น node ทั้งหมดก่อนทำงาน

- มีการโหลด content มาก่อน ทำให้เวลาเป็นหน้าว่างจะโหลดไม่นาน

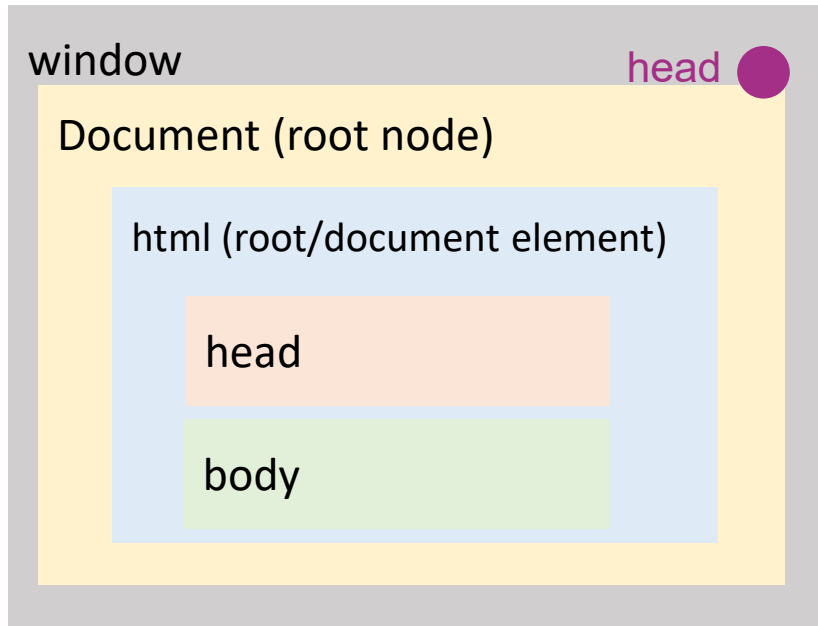


Document Object Modeling (DOM)

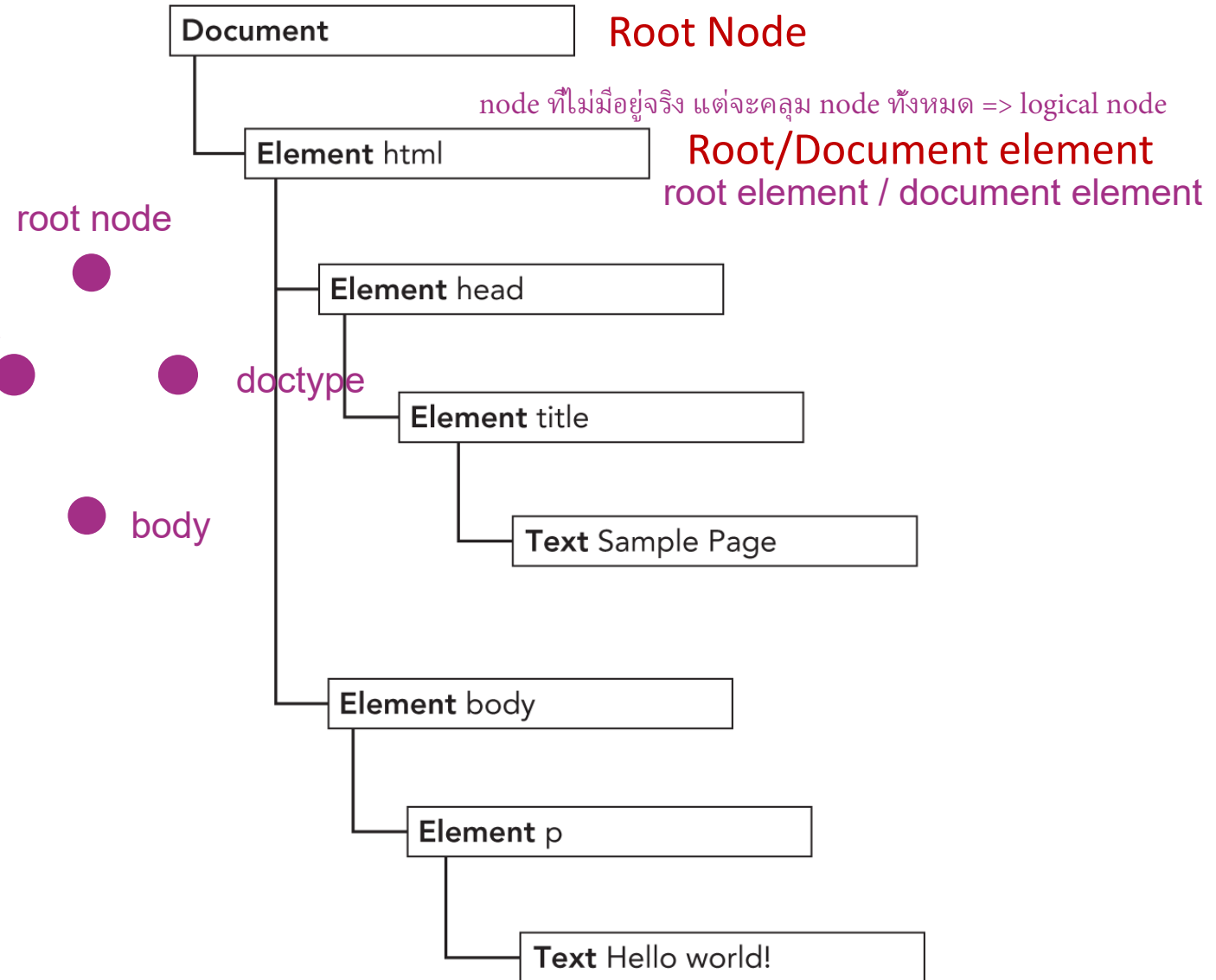
- When an HTML document is loaded into a web browser, it becomes a **document object**, everything is a **node**
- The document object is the root node of the HTML document and all other nodes: element nodes, text nodes, attribute nodes, and comment nodes
- Each node type has different characteristics, data, and methods, and each may have relationships with other nodes.
- These relationships create a hierarchy that allows markup to be represented as a tree, rooted at a particular node.

Dom and JavaScript

```
<html>
  <head>
    <title>Sample Page</title>
  </head>
  <body>
    <p>Hello World!</p>
  </body>
</html>
```



`const rootNode = window.document`



In browsers, the document object is an instance of HTMLDocument (which inherits from Document) and represents the entire HTML page. The document object is a property of window and so is accessible globally. It is read-only.



Nodes Types

Node types are represented by one of the following 11 numeric constants on the

[Node.nodeType](#) Read only

Returns an `unsigned short` representing the type of the node. Possible values are:

Name	Value
ELEMENT_NODE	1
ATTRIBUTE_NODE	2
TEXT_NODE	3
CDATA_SECTION_NODE	4
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11


- The **Node** interface is implemented in JavaScript as the Node type, which is accessible in all browsers
- All node types inherit from Node in JavaScript, so **all node types share the same basic properties and methods.**
- Every node has a **node Type property** that indicates the type of node that it is.

```
if (someNode.nodeType === Node.ELEMENT_NODE) {  
    alert("Node is an element.");  
}
```

The nodeName and nodeValue Properties

- Two properties, nodeName and nodeValue, give specific information about the node.
- The values of these properties are completely dependent on the node type.
- It's always best to test the node type before using one of these values, as the following code shows:

```
if (someNode.nodeType === 1){    //someNode.nodeType===Node.ELEMENT_NODE
    value = someNode.nodeName    // nodeName will be the element's tag name
                                // nodeValue is always null
}
```



Node	Value of nodeValue	nodeName value
CDATASection	Content of the CDATA section	"#cdata-section"
Comment	Content of the comment	"#comment"
Document	null	"#document"
DocumentFragment	null	"#document-fragment"
DocumentType	null	The value of DocumentType.name
Element	null	The value of Element.tagName
ProcessingInstruction	Entire content excluding the target	The value of ProcessingInstruction.target
Text	Content of the text node	"#text"

Document Children

- This example shows that the values of `documentElement`, `firstChild`, and `childNodes[0]` are all the same—all three point to the `<html>` element.
- As an instance of `HTMLDocument`, the `document` object also has a `body` property that points to the `<body>` element directly.

```
<html>
  <body>

  </body>
</html>
```

```
const html = document.documentElement // get reference to <html>
alert(html === document.childNodes[0]) // true
alert(html === document.firstChild) // true
```

```
const body = document.body // get reference to <body>
```



Creating Elements

- New elements can be created by using the `document.createElement()` method.
- This method accepts a single argument, which is the tag name of the element to create.
- In HTML documents, the tag name is case-insensitive.
- To create a `<div>` element, the following code can be used:

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

- The element can be added to the document tree using `appendChild()`, `insertBefore()`, `replaceChild()`, `removeChild()`.
 - The following code adds the newly created element to the document's `<body>` element:
- ```
document.body.appendChild(div);
```
- Once the element has been added to the document tree, the browser renders it immediately. Any changes to the element after this point are immediately reflected by the browser.



# Attr Node

- The Attr node inherits from Node but is not considered a part of the document tree.
- Common Node attributes like parentNode, previousSibling, and nextSibling are `null` for an Attr node. You can, however, get the element to which the attribute belongs with the `ownerElement` property.

```

const pElement = document.createElement('p')

const idAttr = document.createAttribute('id')
idAttr.value = 1001
pElement.setAttributeNode(idAttr)
// pElement.setAttribute('id', 1001)

const pText = document.createTextNode('<h2>Hello,
 JS DOM</h2>')

pElement.appendChild(pText)

const divElement = document.querySelector('div')
divElement.appendChild(pElement)

```

```

▼ <body>
.. ▼ <div> == $0
 <p id="1001"><h2>Hello, JS
 DOM</h2></p>
 </div>

```

```

const rootDocument = divElement.ownerDocument
console.log(rootDocument.nodeName) // #document
console.log(rootDocument.nodeType) // 9
console.log(rootDocument.nodeValue) // null

```

```

console.log(pElement.nodeName) // P
console.log(pElement.nodeType) // 1
console.log(pElement.nodeValue) // null

```

```

console.log(idAttr.nodeName) // id
console.log(idAttr.nodeType) // 2
console.log(idAttr.nodeValue) // 1001

```

```

console.log(pText.nodeName) // #text
console.log(pText.nodeType) // 3
console.log(pText.nodeValue) // '<h2>Hello,
 JS DOM</h2>'

```

```

console.log(pElement.parentElement) // div
console.log(idAttr.parentElement) // null
console.log(pText.parentElement)
// <p id="1001"...</p>

```

# innerHTML, innerText, and textContent

```
const pElement = document.createElement('p')
```

```
pElement.innerHTML =
 '<h2>Hello, JS DOM</h2>'
```

**Hello, JS DOM**

```
*** ▼ <div> == $0
 ▼ <p id="1001">
 ▼ <h2>
 "Hello, "
 JS
 " DOM"
 </h2>
 </p>
 </div>
```

```
const p = document.getElementsByTagName('p')
console.log(p[0].innerText)
console.log(p[0].textContent)
console.log(p[0].innerHTML)
```

Hello, JS DOM

Hello, JS DOM

<h2>Hello, <span style="color:red">JS</span> DOM</h2>

**textContent** has better performance because its value is the raw content not parsed as HTML and also prevent Cross-Site Scripting (XSS) attacks. This includes <script> and <style> elements, whitespace, line breaks, and carriage returns.

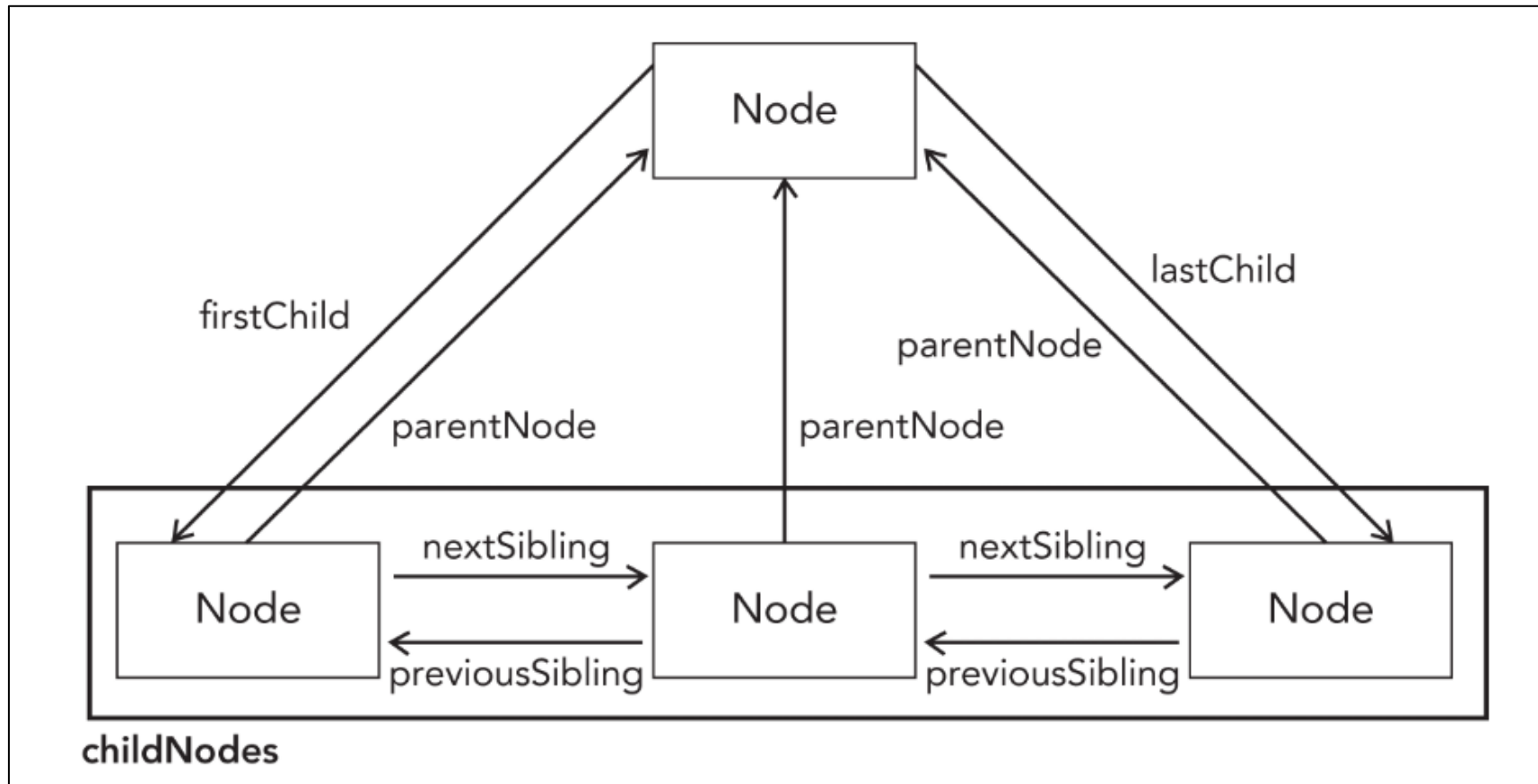




# Traversing Nodes

## Node Relationships

The value of `someNode.firstChild` is always equal to `someNode.childNodes[0]`,  
and the value of `someNode.lastChild` is always equal to `someNode.childNodes[someNode.childNodes.length-1]`.



With all of these relationships, the **childNodes** property is really more of a convenience than a necessity because **it's possible to reach any node in a document tree** by simply using the relationship pointers.



# Element Traversal

**The Element Traversal API adds five new properties to DOM elements:**

- `childElementCount`—Returns the number of child elements (excludes text nodes and comments).
- `firstElementChild`—Points to the first child that is an element. Element-only version of `firstChild`.
- `lastElementChild`—Points to the last child that is an element. Element-only version of `lastChild`.
- `previousElementSibling`—Points to the previous sibling that is an element. Element-only version of `previousSibling`.
- `nextElementSibling`—Points to the next sibling that is an element. Element-only version of `nextSibling`.

# Element Traversal Example

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>Manipulating Elements</title>
 </head>
 <body>
 <h3>SIT@KMUTT Restaurant</h3>
 <div class="menu">
 <ul id="appetizer">
 <li class="vegan">Vegetable Rolls
 <li class="meat">Chicken Wings
 <li class="meat">Tuna Sandwich

 <ul id="soup">
 <li class="meat">Spicy Bacon-Corn Soup
 <li class="vegan">Vegetable Soup
 <li class="meat">Beef Soup
 <li class="vegan">Coconut Soup

 </div>
 <p>***Enjoy Your Meal, Thank you***</p>
 </body>
</html>
```

```
const htmlElement = document.querySelector('html')
console.log(htmlElement.parentNode) // #document root node
console.log(htmlElement.parentElement) // null
```

```
const soupMenu = document.querySelector('#soup')

const firstChildNode = soupMenu.firstChild
const nextFirstChildNode =
 firstChildNode.nextElementSibling
const lastChildNode = soupMenu.lastElementChild
const previousLastChildNode =
 lastChildNode.previousElementSibling

console.log(firstChildNode)
// <li class="meat">Spicy Bacon-Corn Soup
console.log(lastChildNode)
// <li class="vegan">Coconut Soup
console.log(nextFirstChildNode)
// <li class="vegan">Vegetable Soup
console.log(previousLastChildNode)
// <li class="meat">Beef Soup

const meatMenus = document.querySelectorAll('.meat')
meatMenus.forEach((meatMenu) => console.log(meatMenu))
/* <li class="meat">Chicken Wings
<li class="meat">Tuna Sandwich
<li class="meat">Spicy Bacon-Corn Soup
<li class="meat">Beef Soup */
```

# Element Traversal Example

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>Manipulating Elements</title>
 </head>
 <body>
 <h3>SIT@KMUTT Restaurant</h3>
 <div class="menu">
 <ul id="appetizer">
 <li class="vegan">Vegetable Rolls
 <li class="meat">Chicken Wings
 <li class="meat">Tuna Sandwich

 <ul id="soup">
 <li class="meat">Spicy Bacon-Corn Soup
 <li class="vegan">Vegetable Soup
 <li class="meat">Beef Soup
 <li class="vegan">Coconut Soup

 </div>
 <p>***Enjoy Your Meal, Thank you***</p>
 </body>
</html>
```

```
let currentChildNode=soupMenu.firstElementChild
// <li class="meat">Spicy Bacon-Corn Soup

while (currentChildNode !== null) {
 if (currentChildNode.nodeType === 1) {
 console.log(currentChildNode)
 }
 currentChildNode =
currentChildNode.nextElementSibling
}
```

---

```
<li class="meat">Spicy Bacon-Corn Soup
```

---

```
<li class="vegan">Vegetable Soup
```

---

```
<li class="meat">Beef Soup
```

---

```
<li class="vegan">Coconut Soup
```

---

\*The firstChild and lastChild return the first and last child of a node, which can be any node type including text node, comment node, and element node.

\*\*The firstElementChild and lastElementChild return the first and last child Element node.



# Selecting Nodes

# Selecting Elements

The `Document` type provides methods to specific element or sets of elements to perform certain operations.

```
getElementById() // returns the element if found, or null if an element with that ID doesn't exist.
```

**HTMLCollection** `getElementsByTagName()` and `getElementsByClassName()` are live.

When an underlying document is changed, the HTMLCollection will be updated automatically.

- `getElementsByTagName()`

//returns an HTMLCollection of elements with the given tag name.

- `getElementsByClassName()`

//returns A live HTMLCollection an array-like object of all child elements which have all the given class name(s).

## **NodeList**

- `getElementsByName()`

// returns a NodeList, which returns all elements that have a given name attribute.

- `querySelectorAll()`

//the CSS query and returns all matching nodes instead of just one. This method returns a static instance of NodeList.

- `querySelector()`

//a CSS query and returns the first descendant element that matches the pattern or null if there is no matching element.



# HTMLCollection Vs. NodeList

**An HTMLCollection** is always a **live collection of element nodes**.

- For example: If you add a `<li>` element to a list in the DOM, the list in the HTML Collection will also change.
- A HTMLCollection items can be accessed by their name attribute, id attribute, or index number.

**A NodeList** is a **list of all nodes including elements nodes, attribute nodes, and text nodes** extracted from a document.

- A NodeList is most often a static collection. For example: If you add a `<li>` element to a list in the DOM, the list in the NodeList will not change.
  - NodeList items can only be accessed by their index numbers.
- Both an HTMLCollection object and a NodeList object is an array-like collections (lists) of nodes extracted from a document.



# HTMLCollection and NodeList Examples

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>Document</title>
 </head>
 <body>
 <div id="app">
 This is a Text Node
 <div>This is a div Element Node#1</div>
 <div>This is a div Element Node#2</div>
 </div>
 <script src="../main.js" />
 </body>
</html>
```

```
HTMLCollection(2) [div, div]
 <div>This is a div Element Node#1</div>
 <div>This is a div Element Node#2</div>
NodeList(5) [text, div, text, div, text]
 " This is a Text Node "
 <div>This is a div Element Node#1</div>
 ▶ #text
 <div>This is a div Element Node#2</div>
 ▶ #text
 >
```

```
//main.js
const myApp = document.getElementById('app')
const childrensHTML = myApp.children //return HTMLCollection
const childNodesList = myApp.childNodes //return NodeList
Array.from(childrensHTML).forEach((element) => console.log(element))
//HTMLCollection(2) [div, div], cannot use forEach directly and need to create array
childNodesList.forEach((element) => console.log(element))
//NodeList(5) [text, div, text, div, text] can use forEach
```

# HTMLCollection (live) Vs. NodeList (static)

```
<div id="sections">
 <div class="section">Section 1</div>
 <div class="section">Section 2</div>
 <div class="section">Section 3</div>
 <div class="section">Section 4</div>
 <!-- <div class="section">Section 5</div> -->
</div>
```

```
const sectionDiv = document.getElementById('sections')
const sectionsbySelectorAll = document.querySelectorAll('.section')
const sectionsByClassName = document.getElementsByClassName('section')
```

```
//create a new div section
const div = document.createElement('div')
div.classList = 'section'
div.innerHTML = 'section 5'
sectionDiv.appendChild(div)
console.log(sectionsByClassName.length) //5
console.log(sectionsbySelectorAll.length) //4
```

# Selecting Elements

```
<html lang="en">
 <head>
 <title>Document Object Model Sample Page</title>
 </head>
 <body>
 <h1 id="myHeading">Selecting Elements</h1>
 <h2 class="heading">Document Object Model</h2>
 <h2 class="heading">DOM</h2>
 <ul class="all-list-items">
 <li class=list-item>out of class
 <li class="list-item">[1] Single Element by ID:
 <li class="list-item">[2] Single Element by CSS selector:
 <li class="list-item">[3] Multiple Element by CSS selector:
 <li class="list-item">[4] Multiple Element by class (live):
 <li class="list-item">[5] Multiple Element by tag (live):

 <input type="radio" value="red" name="color" id="colorRed">
 <label for="colorRed">Red</label>

 <input type="radio" value="green" name="color" id="colorGreen">
 <label for="colorGreen">Green</label>

 <input type="radio" value="blue" name="color" id="colorBlue">
 <label for="colorBlue">Blue</label>

 <script src="main.js"></script>
 </body>
</html>
```

```
const domElementId = document.getElementById("myHeading")
const domByTagName = document.getElementsByTagName("ul")
const domByName = document.getElementsByName("color")
const domClassName = document.getElementsByClassName("list-item")
const domQuerySelector = document.querySelector(".heading")
const domQuerySelectorAll = document.querySelectorAll(".heading")
console.log(domElementId) //<h1 id="myHeading">Selecting Elements</h1>
console.log(domByTagName) //HTMLCollection(2) [ul.all-list-items,ul]
console.log(domByTagName.length) //2
console.log(domByTagName.item(0)) //<ul class="all-list-items">...
console.log(domByName) //NodeList(3) [input#colorRed, input#colorGreen, input#colorBlue]
console.log(domClassName) //HTMLCollection(6) [li.list-item, li.list-item, li.list-item, li.list-item, li.list-item, li.list-item]
console.log(domQuerySelector) //<h2 class="heading">Document Object Model</h2>
console.log(domQuerySelectorAll) //NodeList(2) [h2.heading, h2.heading]
```

► `h1#myHeading`

► `HTMLCollection(2) [ul.all-list-items, ul]`

2

► `ul.all-list-items`

► `NodeList(3) [input#colorRed, input#colorGreen, input#colorBlue]`

► `HTMLCollection(6) [li.list-item, li.list-item, li.list-item, li.list-item, li.list-item, li.list-item]`

► `h2.heading`

► `NodeList(2) [h2.heading, h2.heading]`



# Manipulating Nodes



# Manipulating Nodes

- `appendChild()` // adds a newly node to the end of the `childNodes` list
- `createElement()` //create a new HTML element
- `insertBefore(newNode, referenceNode)` //The node to insert becomes the previous sibling of the reference node
- `replaceChild(newChild, oldChild)` //replaces a child node within the given (parent) node
- `removeChild(child)` //removes a child node from the DOM and returns the removed node.

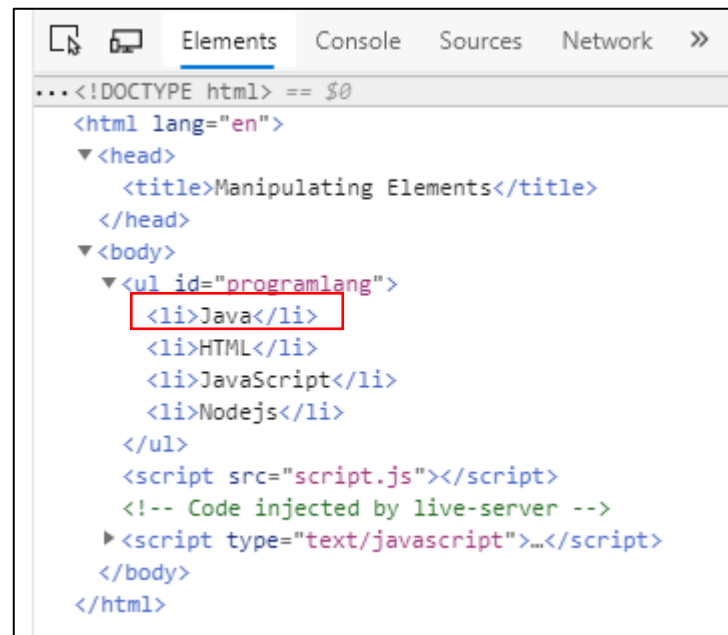
```
<html lang="en">
 <head>
 <title>Manipulating Elements</title>
 </head>
 <body>
 <ul id="programlang">
 </body>
</html>
```



```
const langs = ["HTML", "JavaScript", "Nodejs"]
const langElement = document.querySelector("#programlang")
for (const lang of langs) {
 const li=document.createElement("li")
 li.innerHTML=lang
 langElement.appendChild(li)
}
```

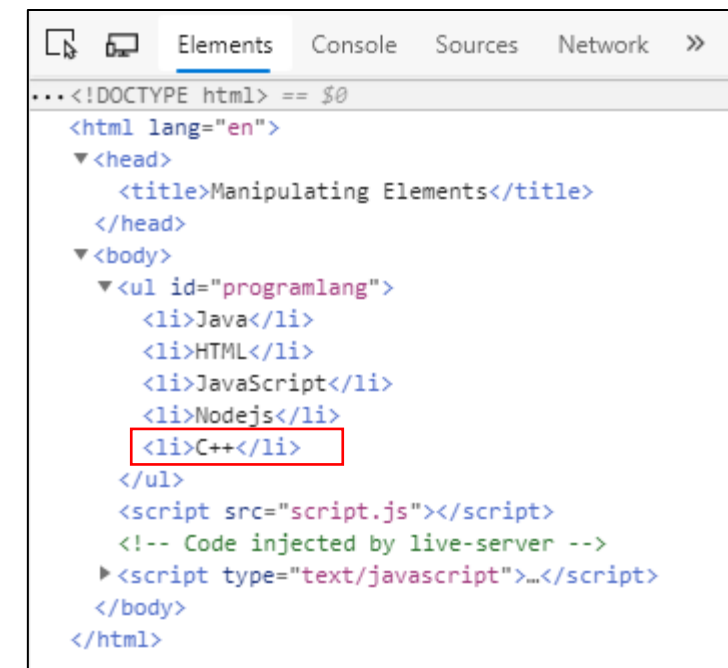
```
const firstLangElement=langElement.firstElementChild
const newLangNode=document.createElement("li")
newLangNode.innerHTML="Java"
langElement.insertBefore(newLangNode, firstLangElement)
```

```
const newLastLangNode=document.createElement("li")
newLastLangNode.innerHTML="C++"
langElement.insertBefore(newLastLangNode, null)
```



The screenshot shows the 'Elements' panel of a browser's developer tools. The DOM tree is expanded to show a list with the id 'programlang'. The first list item, '<li>Java</li>', is highlighted with a red box. The rest of the DOM structure, including the head, body, and other list items, is visible below it.

- Java
- HTML
- JavaScript
- Nodejs



The screenshot shows the 'Elements' panel of a browser's developer tools. The DOM tree is expanded to show a list with the id 'programlang'. The list now contains five items: 'Java', 'HTML', 'JavaScript', 'Nodejs', and 'C++'. The last item, '<li>C++</li>', is highlighted with a red box. The rest of the DOM structure is visible below it.

- Java
- HTML
- JavaScript
- Nodejs
- C++

```
langElement.replaceChild(newLastLangNode, newLangNode)
```



The screenshot shows the 'Elements' panel of a web browser. The DOM tree is expanded to show the following structure:

```
...<!DOCTYPE html> == $0
<html lang="en">
 <head>
 <title>Manipulating Elements</title>
 </head>
 <body>
 <ul id="programlang">
 C++
 HTML
 JavaScript
 Nodejs

 <script src="script.js"></script>
 <!-- Code injected by live-server -->
 <script type="text/javascript">...</script>
 </body>
</html>
```

- C++
- HTML
- JavaScript
- Nodejs

```
langElement.removeChild(langElement.firstChildChild)
```



The screenshot shows the 'Elements' panel after the first child of the `ul` has been removed. The DOM tree is now:

```
...<!DOCTYPE html> == $0
<html lang="en">
 <head>...</head>
 <body>
 <ul id="programlang">
 HTML
 JavaScript
 Nodejs

 <script src="script.js"></script>
 <!-- Code injected by live-server -->
 <script type="text/javascript">...</script>
 </body>
</html>
```

- ~~C++~~
- HTML
- JavaScript
- Nodejs





# System Dialogs

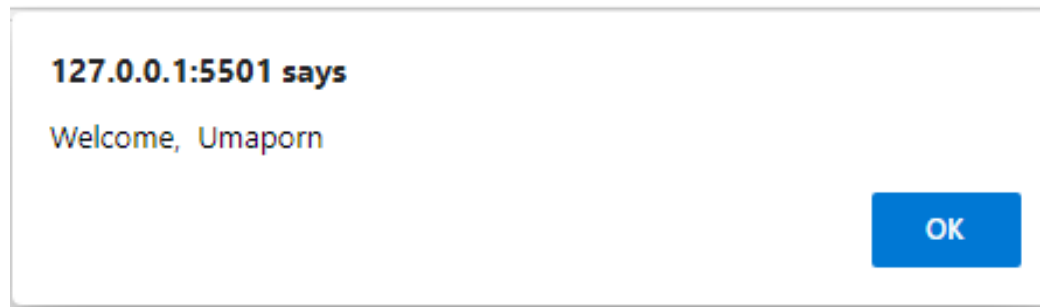


# System Dialogs

- The browser is capable of invoking system dialogs to display to the user through the **alert()**, **confirm()**, and **prompt()** methods.
- These dialogs are not related to the web page being displayed in the browser and do not contain HTML.
- Their appearance is determined by operating system and/or browser settings rather than CSS.
- Additionally, each of these dialogs is synchronous and modal, meaning code execution stops when a dialog is displayed, and resumes after it has been dismissed.

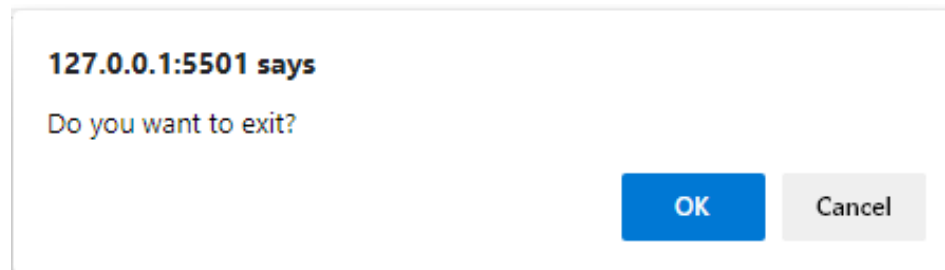
# System Dialogs: alert()

- The `alert()` method simply accepts a string to display to the user.
- When `alert()` is called, a system message box displays the specified text to the user, followed by a single OK button.
- Alert dialogs are typically used when users must be made aware of something that they have no control over, such as an error.



# System Dialogs: confirm()

- A `confirm()` method looks similar to an alert dialog in that it displays a message to the user.
- The main difference between the two is the **presence of a Cancel button along with the OK button**, which allows the user to indicate if a given action should be taken.
- The `confirm()` method returns `true` if the user clicked "OK", and `false` otherwise.



# System Dialogs: prompt()

- The final type of dialog is displayed by calling `prompt()`, which along **with OK and Cancel buttons**, this dialog has a text box where the user may enter some data.
- The `prompt()` method accepts two arguments: the *text to display* to the user, and the *default value* for the text box (which can be an empty string).
- If the OK button is clicked, `prompt()` returns the value in the text box; if Cancel is clicked or the dialog is otherwise closed without clicking OK, the function returns `null`.

