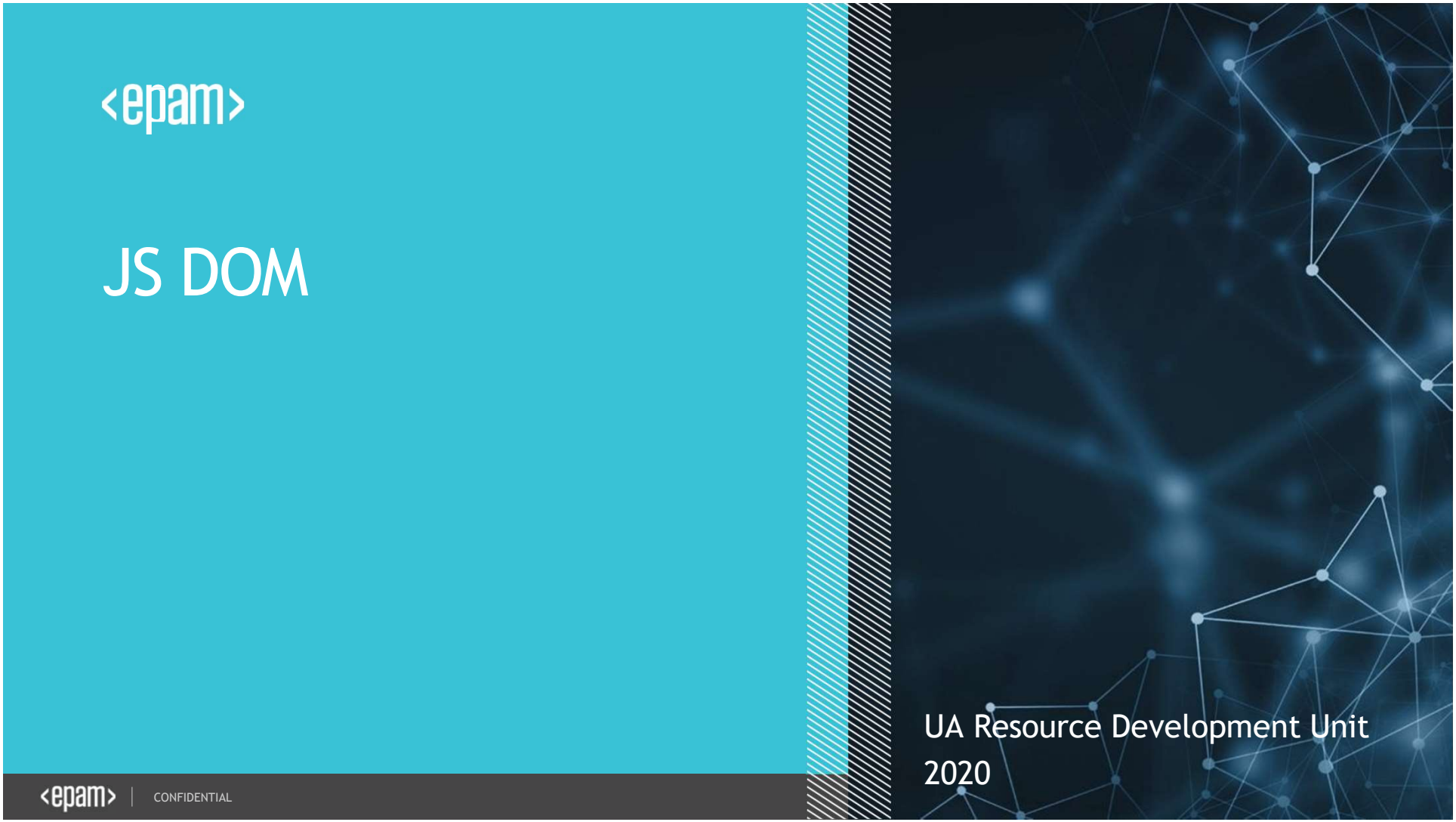

JS DOM
UA Resource Development Unit
2020
CONFIDENTIAL

Agenda

1. **Document Structure**
2. **Select elements**
3. **Modifying the document**
4. Insertion Methods
5. Summary

What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.
The DOM defines a standard for accessing documents:
"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."
The W3C DOM standard is separated into 3 different parts:
Core DOM - standard model for all document types
XML DOM - standard model for XML documents
HTML DOM - standard model for HTML documents

What is the HTML DOM?

The HTML DOM is a standard object model and programming interface for HTML. It defines:

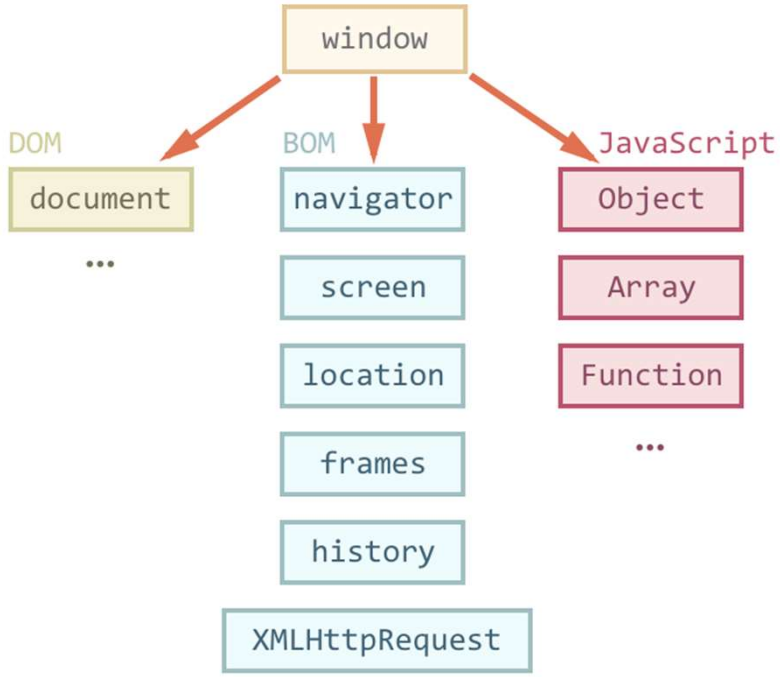
The HTML elements as objects

The properties of all HTML elements

The methods to access all HTML elements

The events for all HTML elements

In other words: The HTML DOM is a standard for how to get, change, add, or delete HTML elements.

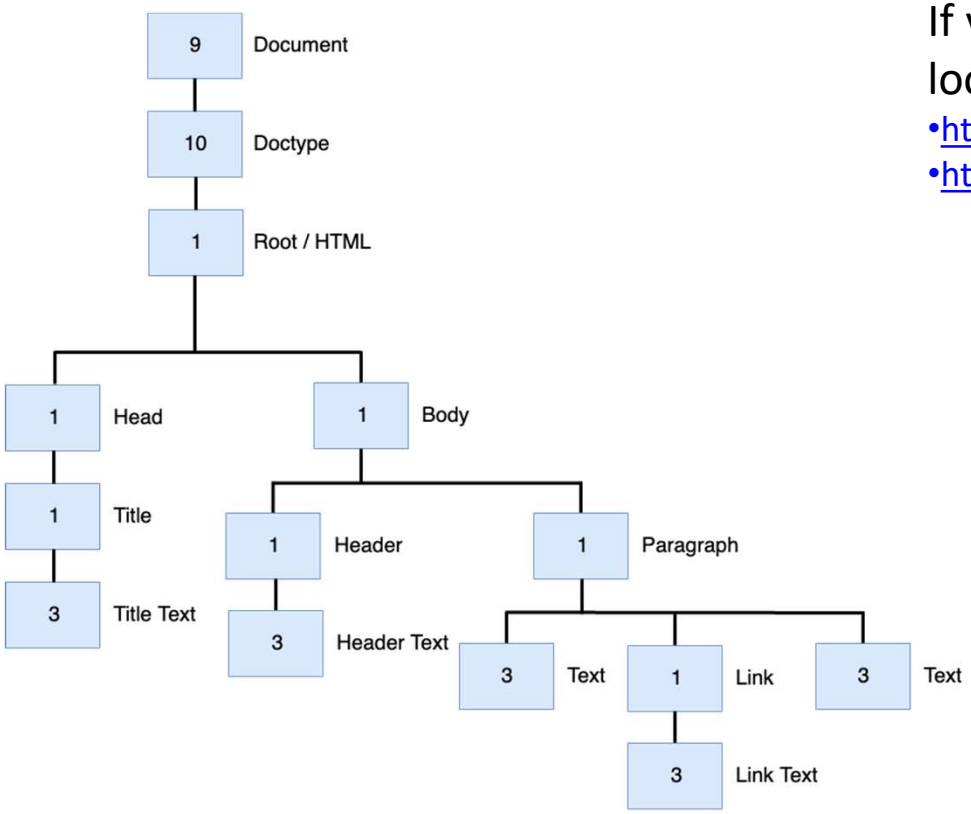


HTML DOM

From this 👇

```
<!DOCTYPE html>
<html>
<head>
  <title>The DOM</title>
</head>
<body>
  <h1>Awesome title</h1>
  <p>Some text with a <a href="https://epam.com">link</a></p>
</body>
</html>
```

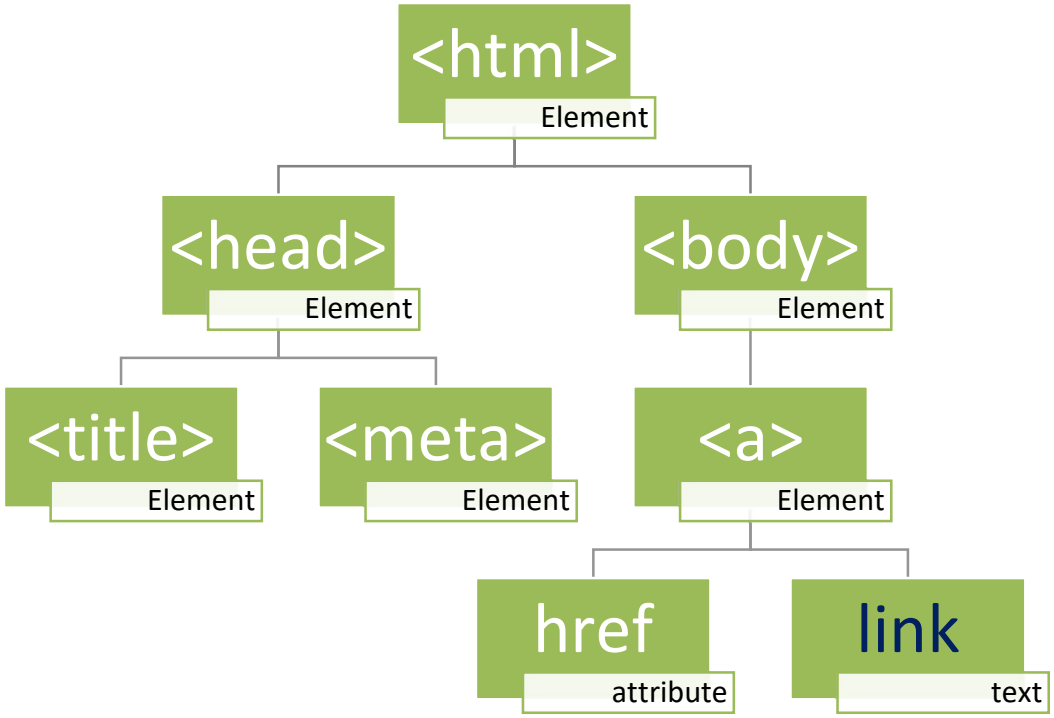
To this 👇



If you 🤔 wondering what's a tree is, take a look at 😉:

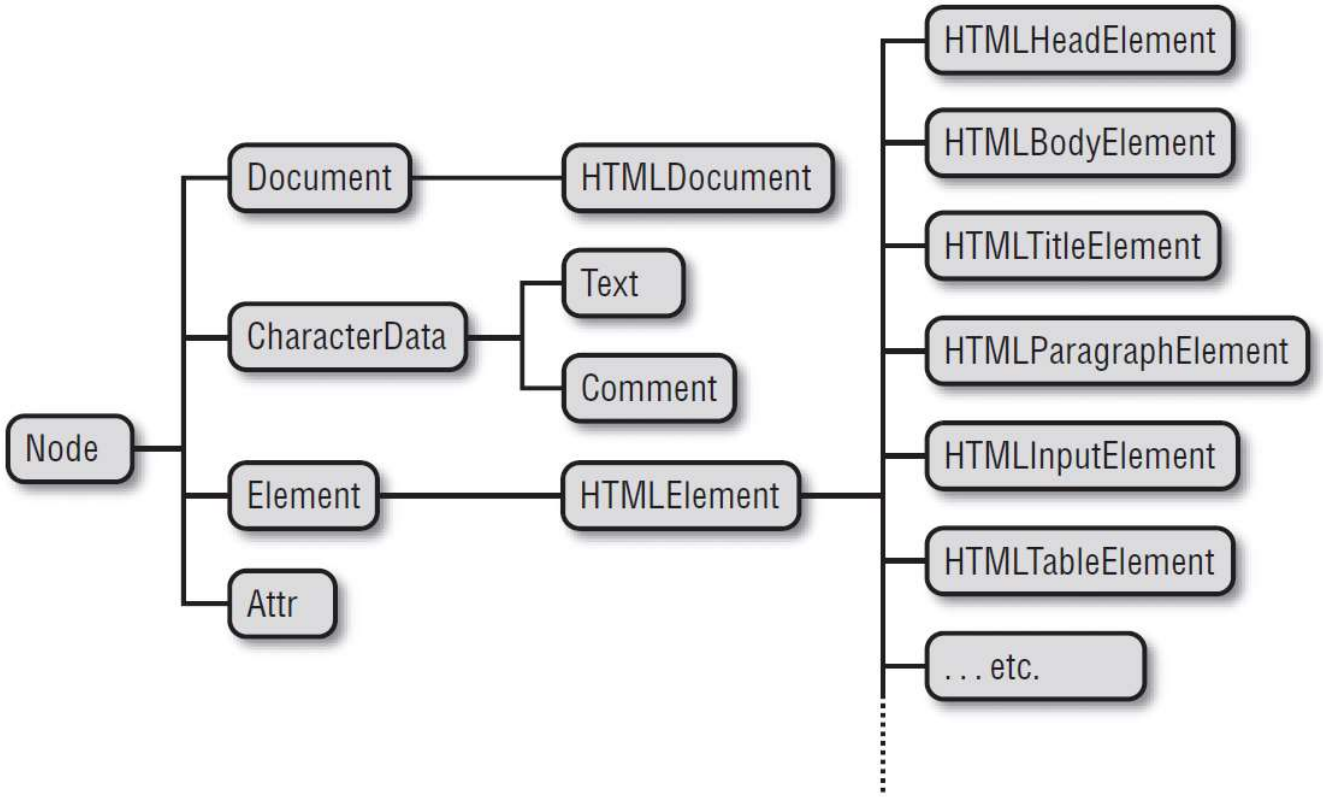
- https://en.wikipedia.org/wiki/Tree_(data_structure)
- https://youtu.be/oSWTXtMglKE

The HTML DOM Tree of Objects



When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page

The HTML Types of Objects



DOM

With the DOM, JavaScript gets all the power it needs to create dynamic HTML:

JavaScript can change all the HTML elements in the page

JavaScript can change all the HTML attributes in the page

JavaScript can change all the CSS styles in the page

JavaScript can remove existing HTML elements and attributes

JavaScript can add new HTML elements and attributes

JavaScript can react to all existing HTML events in the page

JavaScript can create new HTML events in the page

Select elements

The DOM defines a number of ways to select elements

```
with a specified id attribute;
with a specified name attribute;
with the specified tag name;
with the specified CSS class or classes;
or
matching the specified CSS selector
```

Selecting Elements by ID

You can select an element based on this unique ID with the getElement ById() method of the Document object

```
<html>
<head>
   <title>getElementById example</title>
</head>
<body>
   <p id="para">Some text here</p>
   <button onclick="changeColor('blue');">blue</button>
   <button onclick="changeColor('red');">red</button>
</body>
<script>
   var elem = document.getElementById('para');
   elem.style.color = 'red';
</script>
</html>
```

id

Selecting Elements by Name

To select HTML elements based on the value of their name attributes, you can use the getElementsByName() method of the Document object

```
<!DOCTYPE html>
<html lang="en">
<title>Example: using document.getElementsByName</title>

<input type="hidden" name="up">
<input type="hidden" name="down">

<script>
  var up_names = document.getElementsByName("up");
  console.log(up_names[0].tagName); // displays "INPUT"
</script>
</html>
```

name

Selecting Elements by Type

You can select all elements of a specified type (or tag name) using the getElementsByTagName() method of the Document object

```
1    let spans = document.getElementsByTagName("span");
2    var firstpara = document.getElementsByTagName("p")[0];
```

You can obtain a NodeList that represents all elements in a document by passing the wildcard argument “*” to getElementsByTagName().

tag

NodeLists and HTMLCollections

getElementsByName() and getElementsByTagName() return NodeList objects, and properties like document.images and document.forms are HTMLCollection objects

NodeList objects are collections of nodes such as those returned by properties such as Node.childNodes and the document.querySelectorAll() method.

The **HTMLCollection** interface represents a generic collection (array-like object similar to arguments) of elements (in document order) and offers methods and properties for selecting from the list.

Selecting Elements by CSS Class

Like getElementsByTagName(), getElementsByClassName() can be invoked on both HTML documents and HTML elements, and it returns a live NodeList containing all matching descendants of the document or element

```
var testElements = document.getElementsByClassName('test');
var testDivs = Array.prototype.filter.call(testElements, function (testElement) {
  return testElement.nodeName === 'DIV';
});
```

Selecting Elements with CSS Selectors

CSS stylesheets have a very powerful syntax, known as selectors, for describing elements or sets of elements within a document.

```
#nav                              // An element with id="nav"
div                               // Any  element
.warning                          // Any element with class
"warning"
p[lang="fr"]                      // A paragraph in French:
*[name="x"]                       // Any element with name="x"
attribute
span.fatal.error            // with classes "fatal" & "error"
span[lang="fr"].warning     // Any warning in French
#log span                         // Any descendant of the log
#log>span                         // Any child of the log
div, #log                         // All elements plus the log
```

document.querySelector()

Returns the first Element within the document that matches the specified selector, or group of selectors, or null if no matches are found.

```
let el = document.querySelector("div.user-panel.main input[name='login']");
```

Returns **null** if no matches are found; otherwise, it returns the first matching element.
If the selector matches an ID and this ID is erroneously used several times in the document, it returns the first matching element.
Throws a SYNTAX_ERR exception if the specified group of selectors is invalid.

Document.querySelectorAll()

Returns a list of the elements within the document (using depth-first pre-order traversal of the document's nodes) that match the specified group of selectors. The object returned is a NodeList.

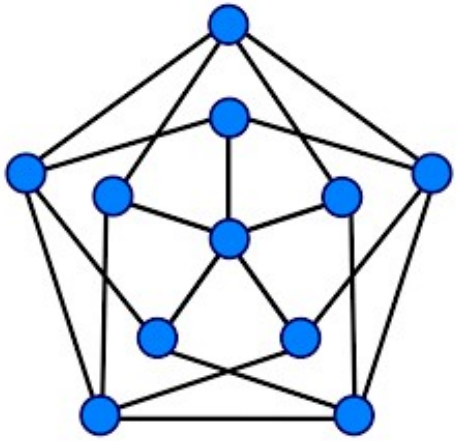
```
let matches = document.querySelectorAll("div.note, div.alert");
```

Returns a non-live `NodeList` of all the matching element nodes.
Throws a `SYNTAX_ERR` exception if the specified group of selectors is invalid.

Document Structure and Traversal

The Document object, its Element objects, and the Text objects that represent runs of text in the document are all Node objects

Node is an interface from which a number of DOM API object types inherit; it allows these various types to be treated similarly, for example inheriting the same set of methods, or being tested in the same way.



Navigating Between Nodes [Properties]

Node.baseURI
Returns
a DOMString representing the base URL.

Node.childNodes
Returns a
live NodeList containing all the children of this node.

Node.firstChild
Returns a Node representing the first direct child node of the node, or null if the node has no child

Node.lastChild
Returns a Node representing the last direct child node of the node, or null if the node has no child.

.
Node.nextSibling
Returns a Node representing the next node in the tree, or null if there isn't such node.

Node.nodeName
Returns
a DOMString containing the name of the Node. The structure of the name will differ with the node type.

Node.parentNode
Returns a Node that is the parent of this node. If there is no such node, like if this node is the top of the tree or if doesn't participate in a tree, this property returns null.

Node.nodeValue
Returns / Sets the value of the current node

.
Node.previousSibling
Returns a Node representing the previous node in the tree, or null if there isn't such node.

Node.nodeType

Read onlyReturns an unsigned short representing the type of the node. Possible values are

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

Attributes and properties

When the browser loads the page, it “reads” HTML text and generates DOM objects from it. For element nodes most standard HTML attributes automatically become properties of DOM objects.

DOM nodes are regular JavaScript objects. We can alter them.
DOM properties and methods behave just like those of regular JavaScript objects:

They can have any value. They are case-sensitive (write elem.nodeType, not elem.NoDeTyPe).

```
Element.prototype.sayHi = function () {
  alert(`Hello, I'm ${this.tagName}`
  );
};

document.documentElement.sayHi();
// Hello, I'm HTML document.body.sayHi(); // Hello, I'm BODY
```

Reading Element Attributes, Node Values and Other Data

```
// Retrieves  the value of the attribute with the name attribute
node.getAttribute('attribute');
// Sets the value of the attribute with the name attribute to value
node.setAttribute('attribute', 'value');
// Reads the type of the node (1 == element, 3 == text node)
node.nodeType;
// Reads the name of the node (either element name or #textNode)
node.nodeName;
// Reads or sets the value of the node (or the text content in the case of text nodes)
node.nodeValue;
```

HTML attributes

When the browser reads HTML text and creates DOM objects for tags, it recognizes standard attributes and creates DOM properties from them.
So when an element has id or another standard attribute, the corresponding property gets created. But that doesn't happen if the attribute is non-standard

```
<body id="test" something="non-standard">
<script>
  alert(document.body.id); // test
  // non-standard attribute does not yield a property
  alert(document.body.something); // undefined
</script>
</body>
```

HTML attribute

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // non-standard

    // All attributes are accessible using following methods
    // elem: any dom element provided
    elem.hasAttribute('name') // checks for existence.
    elem.getAttribute('name') // gets the value.
    elem.setAttribute('name', 'value') // sets the value.
    elem.removeAttribute('name') // removes the attribute.
    elem.attributes // a collection of objects that belong to a built-in Attr class
  </script>
</body>
```

Property-attribute synchronisation

When a standard attribute changes, the corresponding property is auto-updated, and (with some exceptions) vise-versa.

```
<body>
  <input />
  <script>
    let input = document.querySelector('input'); // attribute => property
    input.setAttribute('id', 'id');
    alert(input.id); // id (updated) // property => attribute
    input.id = 'newId';
    alert(input.getAttribute('id')); // newId (updated)
  </script>
</body>
```

closest

The **Element.closest()** method returns the closest ancestor of the current element (or the current element itself) which matches the selectors given in parameter. If there isn't such an ancestor, it returns null.

```
<div id="block" title="I'm block">
   <a href="#"> I'm link</a>
   <a href=http://site.com>link</a>
   <div>
      <div id="too">
      </div>
   </div>
</div>
```

```
<script>
   var div = document.querySelector("#too");
   div.closest("#block");
   div.closest("div");
   div.closest("a");
   div.closest("div[title]");
</script>
```

Creating an element

document.createElement

In an HTML document, the **Document.createElement()** method creates the HTML element specified by tagName, or an HTMLUnknownElement if tagName isn't recognized

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

document.createTextNode(text)

Creates a new *text node* with the given text

```
let textNode = document.createTextNode('Here I am');
```

Insertion methods

parentElem.appendChild(node)

method adds a node to the end of the list of children of a specified parent node. If the given child is a reference to an existing node in the document, appendChild() moves it from its current position to the new position

```
var p = document.createElement("p");
document.body.appendChild(p);
```

Insertion methods

parentElem.insertBefore(node, nextSibling)

The method inserts a node before the reference node as a child of a specified parent node. If the given child is a reference to an existing node in the document, insertBefore() moves it from its current position to the new position.

If the reference node is null, the specified node is added to the end of the list of children of the specified parent node.

```
<ol id="list">
  <li>0</li>
  <li>1</li>
</ol>
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';
  list.insertBefore(newLi, list.children[1]);
</script>
```

Insertion methods

parentElem.replaceChild(node, oldChild)

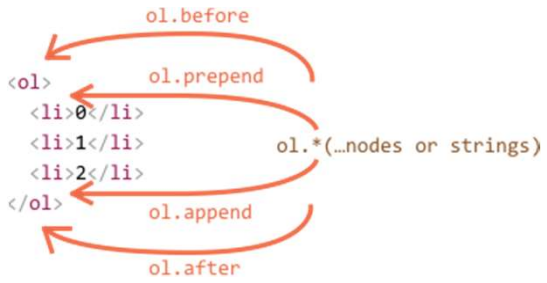
Replaces oldChild with node among children of parentElem.
All these methods return the inserted node. In other words, parentElem.appendChild(node) returns node.

```
var sp1 = document.createElement("span");
sp1.id = "newSpan";
var sp1_content = document.createTextNode("new replacement span element.");
sp1.appendChild(sp1_content);
var sp2 = document.getElementById("childSpan");
var parentDiv = sp2.parentNode;
parentDiv.replaceChild(sp1, sp2);
```

Insertion methods

prepend/append/before/after

node.append(...nodes or strings)
– append nodes or strings at the end of node,
node.prepend(...nodes or strings)
– insert nodes or strings into the beginning of node,
node.before(...nodes or strings)
–- insert nodes or strings before the node,
node.after(...nodes or strings)
–- insert nodes or strings after the node,
node.replaceWith(...nodes or strings)
–- replaces node with the given nodes or strings.




```
<ol id="ol">
  <li>0</li>
  <li>1</li>
</ol>
<script>
  ol.before('before');
  ol.after('after');
  let prepend = document.createElement('li');
  prepend.innerHTML = 'prepend';
  ol.prepend(prepend);
  let append = document.createElement('li');
  append.innerHTML = 'append';
  ol.append(append);
</script>
```

Insertion methods

element.insertAdjacentHTML(position, text);

insertAdjacentHTML() parses the specified text as HTML or XML and inserts the resulting nodes into the DOM tree at a specified position.

position is the position relative to the element, and must be one of the following strings:
‘beforebegin' - Before the element itself.
‘afterbegin' - Just inside the element, before its first child.
‘beforeend' - Just inside the element, after its last child.
‘afterend’ - After the element itself.

```
var d1 = document.getElementById('one');
d1.insertAdjacentHTML('afterend', '<div id="two">two</div>');
```

Cloning nodes

cloneNode

The call elem.cloneNode(true) creates a “deep” clone of the element – with all attributes and subelements. If we call elem.cloneNode(false), then the clone is made without child elements.

```
var p = document.getElementById("para1");
var p_prime = p.cloneNode(true);
```

Removal methods

The **Node.removeChild()** method removes a child node from the DOM. Returns removed node.

All insertion methods automatically remove the node from the old place.

```
var d = document.getElementById("top");
var d_nested = document.getElementById("nested");
var throwawayNode = d.removeChild(d_nested);
```

Removal methods

The **ChildNode.remove()** method removes the object from the tree it belongs to.

```
<div id="div-01">Here is div-01</div>
<div id="div-02">Here is div-02</div>
<div id="div-03">Here is div-03</div>

<script>
   var el = document.getElementById('div-02');
   el.remove(); // Removes the div with the 'div-02' id
</script>
```

innerHTML

The **Element.innerHTML** property sets or gets the HTML syntax describing the element's descendants.

const content = element.innerHTML;
On return, content contains the serialised HTML code describing all of the element's descendants.

element.innerHTML = content;
Removes all of element's children, parses the content string and assigns the resulting nodes as children of the element.

```
const name = "John"; // assuming 'el' is an HTML DOM element
el.innerHTML = name; // harmless in this case
// ...
name = "<script>alert('I am John in an annoying alert!')</script>";
el.innerHTML = name; // harmless in this case
```

Styles and classes

className gets and sets the value of the class attribute of the specified element.

```
let elm = document.getElementById('item');
if (elm.className === 'active') {
  elm.className = 'inactive';
} else {
  elm.className = 'active';
}
```

Styles and classes

The **Element.classList** is a read-only property which returns a live DOMTokenListcollection of the class attributes of the element.
Using classList is a convenient alternative to accessing an element's list of classes as a space-delimited string via element.className.

Methods

add(String [, String])
remove(String [, String])
item(Number)
toggle(String [, force])
contains(String)
replace(oldClass, newClass)

```
div.classList.remove("foo");
div.classList.add("anotherclass");
div.classList.toggle("visible");
div.classList.toggle("visible", i < 10);
```

Element style

The property elem.style is an object that corresponds to what's written in the "style" attribute.

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Resetting the style property

```
document.body.style.display = "none"; // hide

setTimeout(() => document.body.style.display = "", 1000); // back to normal
```

Computed styles

getComputedStyle

The style property operates only on the value of the "style" attribute, without any CSS cascade.

The getComputedStyle(elem[, pseudo]) returns the style-like object with them. Read-only.

```
<script>
  let computedStyle = getComputedStyle(document.body);
  alert(computedStyle.marginTop);
  alert(computedStyle.color);
</script>
```

Q&A

UA Frontend Online LAB