JavaScript and the DOM

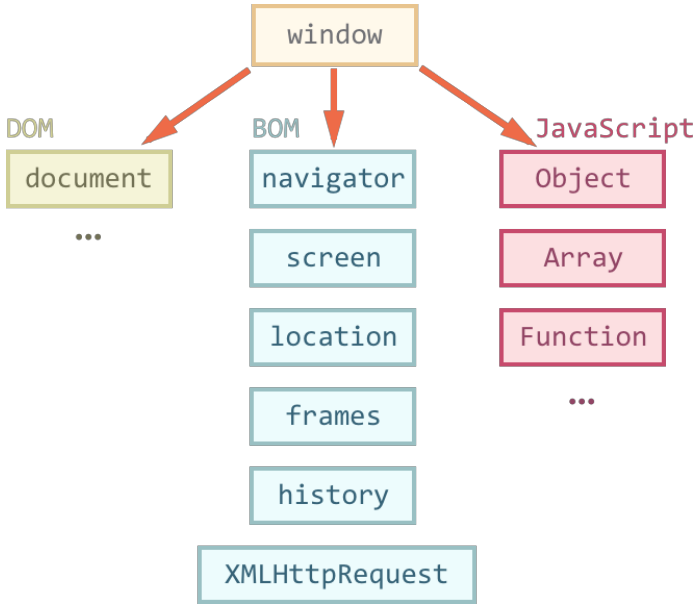
Roi Yehoshua
2018

Browser Environment

- The JavaScript language was initially created for web browsers
- Since then, it has evolved and become a language with many uses and platforms
- A platform may be a browser, a web-server, or a washing machine, or another *host*
- The JavaScript specification calls that a *host environment*.
- A host environment provides platform-specific objects and functions additional to the language core.
  - Web browsers give a means to control web pages
  - Node.JS provides server-side features, and so on

Browser Environment

- Here's a bird's-eye view of what we have when JavaScript runs in a web-browser:



- There's a "root" object called window. It has two roles:
  - First, it is a global object for JavaScript code
  - Second, it represents the "browser window" and provides methods to control it

The window Object

- For instance, here we use the window as a global object:

```
function sayHi() {
    alert("Hello");
}

// global functions are accessible as properties of window
window.sayHi();
```

- And here we use it as a browser window, to see the window height:

```
alert(window.innerHeight); // inner window height
```

Document Object Model (DOM)

- The **document** object gives access to the page content
- We can change or create anything on the page using it
- For instance:

```
// change the background color to red
document.body.style.background = "red";

// change it back after 1 second
setTimeout(() => document.body.style.background = "", 1000);
```

- The **DOM specification** explains the structure of a document and provides objects to manipulate it
- It is being developed by two groups:
  - W3C – the documentation is at https://www.w3.org/TR/dom
  - WhatWG – publishing at https://dom.spec.whatwg.org

Browser Object Model (BOM)

- **Browser Object Model** (BOM) are additional objects provided by the browser (host environment) to work with everything except the document
- For instance:
  - The navigator object provides background information about the browser and the OS
  - The location object allows us to read the current URL and redirect the browser to a new one
- Here's how we can use the location object:

```
alert(location.href); // shows current URL
if (confirm("Go to wikipedia?")) {
    location.href = "https://wikipedia.org"; // redirect the browser to another URL
}
```

- BOM is the part of the general HTML specification

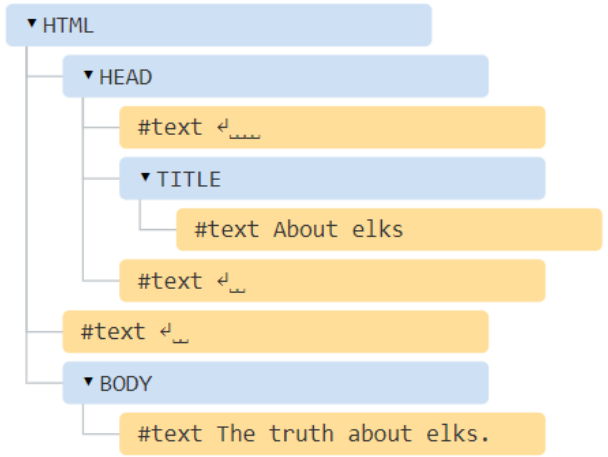
DOM Tree

- The backbone of an HTML document are tags
- According to Document Object Model (DOM), every HTML tag is an object
- Nested tags are called “children” of the enclosing one
- The text inside a tag it is an object as well
- All these objects are accessible using JavaScript

Example of DOM

- For instance, let's explore the DOM for this document:

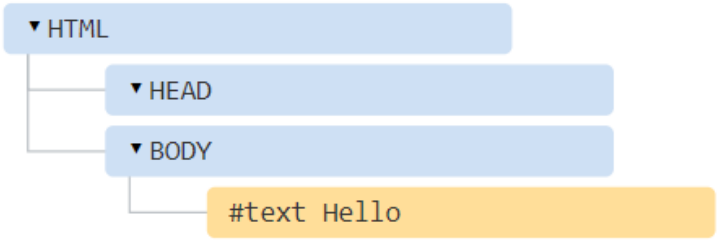
```
<!DOCTYPE HTML>
<html>
<head>
    <title>About elks</title>
</head>
<body>
    The truth about elks.
</body>
</html>
```



- We have a tree of elements: <html> is at the root, then <head> and <body> are its children, etc.
- The text inside elements forms *text nodes*, labeled as #text
  - A text node contains only a string. It may not have children and is always a leaf of the tree.
  - Spaces and newlines – form text nodes and become a part of the DOM

Autocorrection

- If the browser encounters malformed HTML, it automatically corrects it when making DOM
- For instance, the top tag is always <html>
  - Even if it doesn’t exist in the document - the browser will create it
  - The same goes for <body>
- As an example, if the HTML file is a single word "Hello", the browser will wrap it into <html> and <body>, add the required <head>, and the DOM will be:

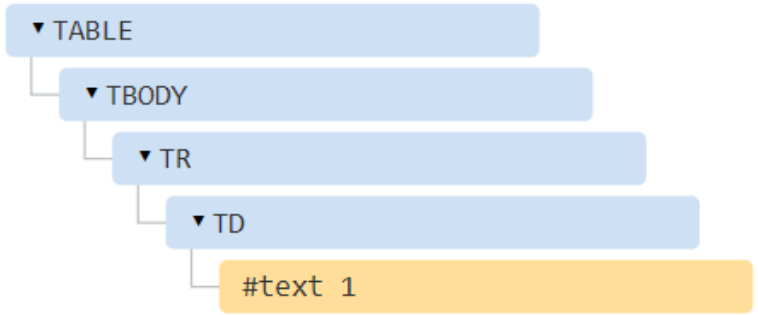


Autocorrection

- An interesting “special case” is tables
- By the DOM specification they must have <tbody>, but HTML text may (officially) omit it. Then the browser creates <tbody> in DOM automatically.
- For the HTML:

```
<table id="table"><tr><td>1</td></tr></table>
```

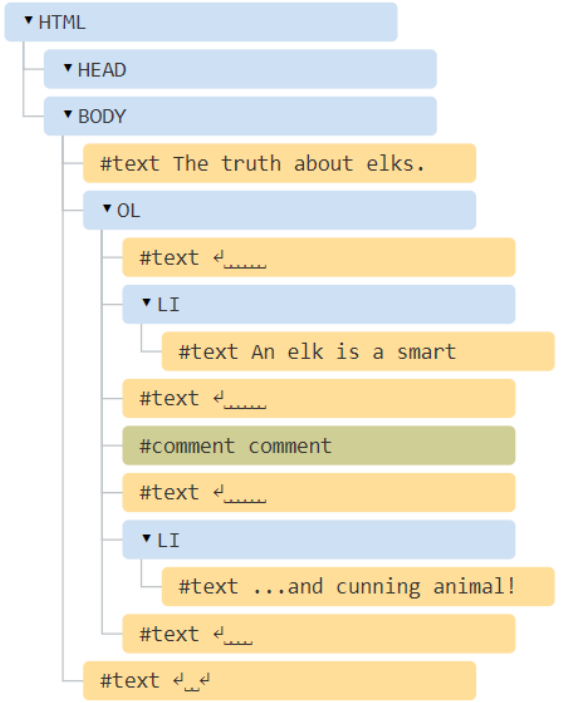
- DOM-structure will be:



Other Node Types

- Let's add more tags and a comment to the page:

```
<!DOCTYPE HTML>
<html>
<body>
    The truth about elks.
    <ol>
        <li>An elk is a smart</li>
        <!-- comment -->
        <li>...and cunning animal!</li>
    </ol>
</body>
</html>
```



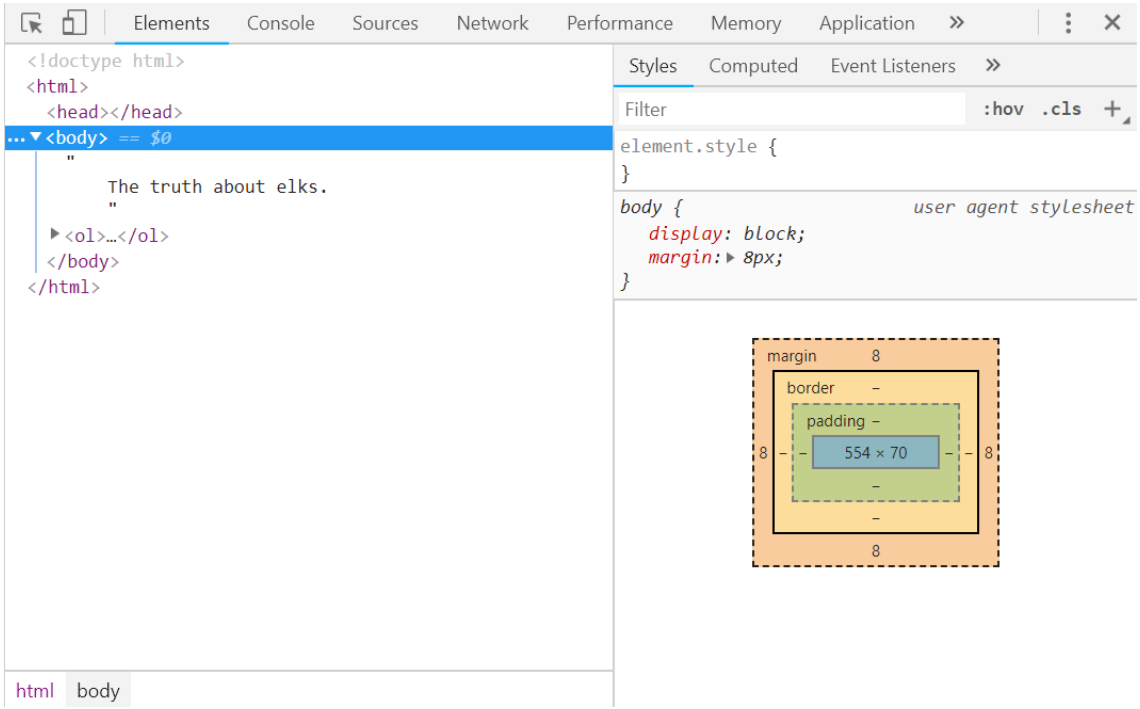
- Everything in HTML, even comments, becomes a part of the DOM

Other Node Types

- There are 12 node types
- In practice we usually work with 4 of them:
  - document – the “entry point” into DOM
  - element nodes – HTML-tags, the tree building blocks
  - text nodes – contain text
  - comments – sometimes we can put the information there, it won’t be shown, but JS can read it from the DOM

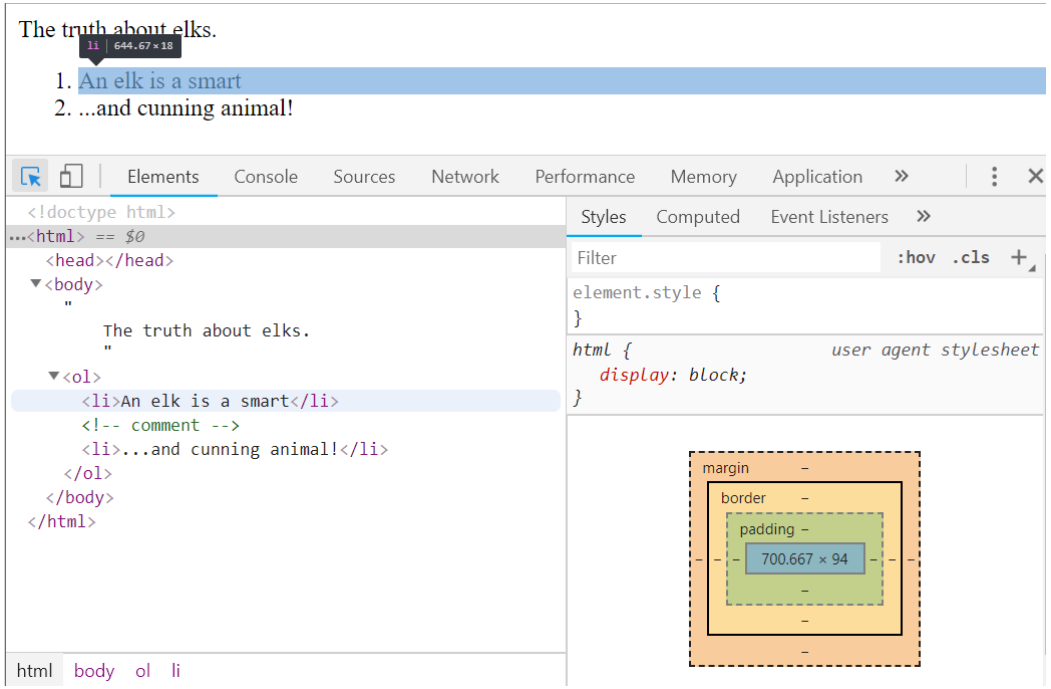
The Browser Inspector

- You can use the browser developer tools to explore the DOM
- To do so, open the web-page, turn on the browser developer tools (F12 in Chrome) and switch to the Elements tab
- You can see the DOM, click on elements, see their details and so on



The Browser Inspector

- Clicking the button in the left-upper corner allows to choose a node from the webpage using a mouse and “inspect” it (scroll to it in the Elements tab)
- Another way to do it would be just right-clicking on a webpage and selecting “Inspect” in the context menu



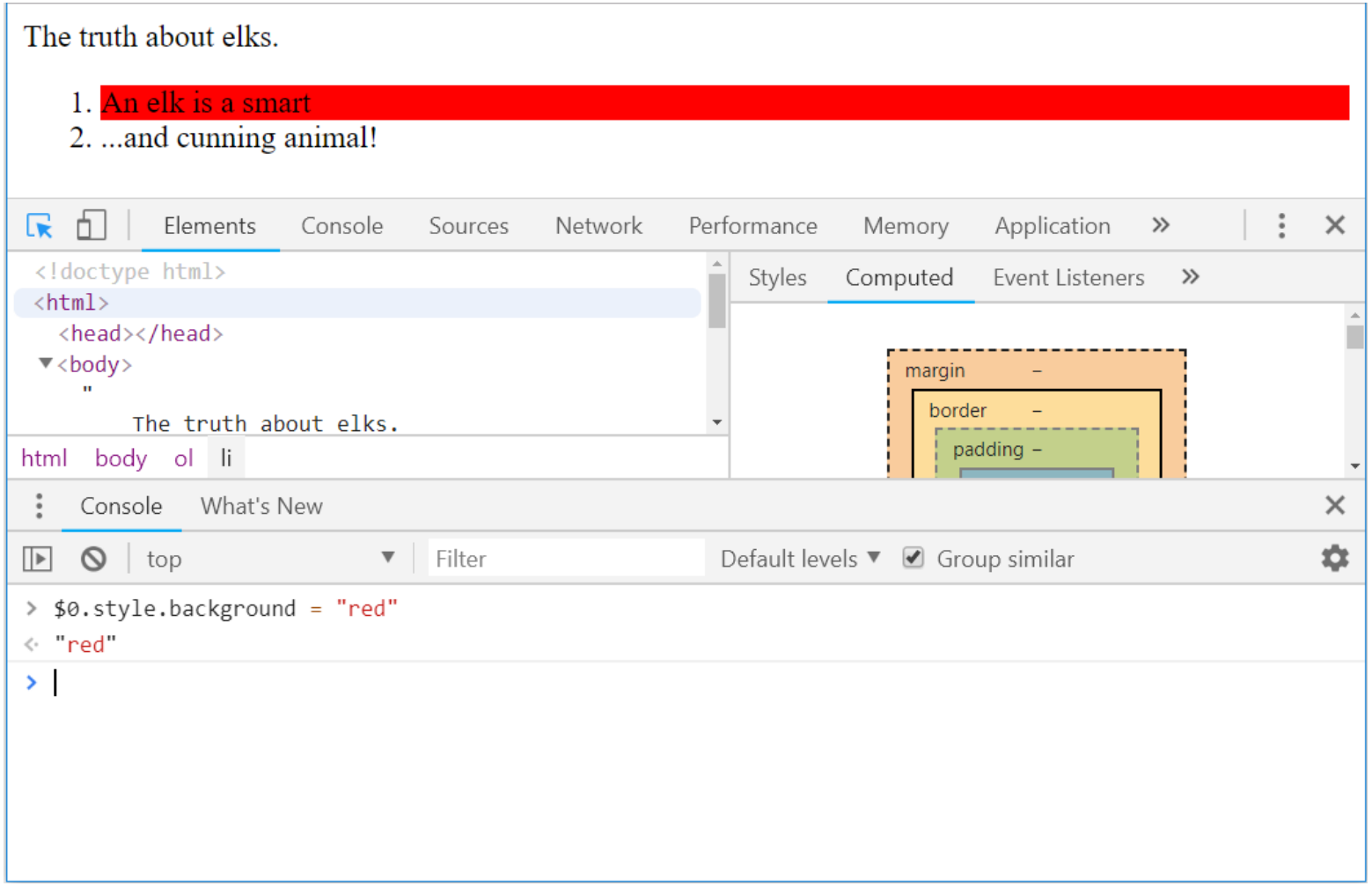
The Browser Inspector

- At the right part of the tools there are the following subtabs:
  - **Styles** – we can see CSS applied to the current element rule by rule, including built-in rules (gray). Almost everything can be edited in-place, including the dimensions/margins/paddings of the box below.
  - **Computed** – to see CSS applied to the element by property: for each property we can see a rule that gives it (including CSS inheritance and such).
  - **Event Listeners** – to see event listeners attached to DOM elements
  - …and more
- The best way to study them is to click around. Most values are editable in-place.

Interaction with Console

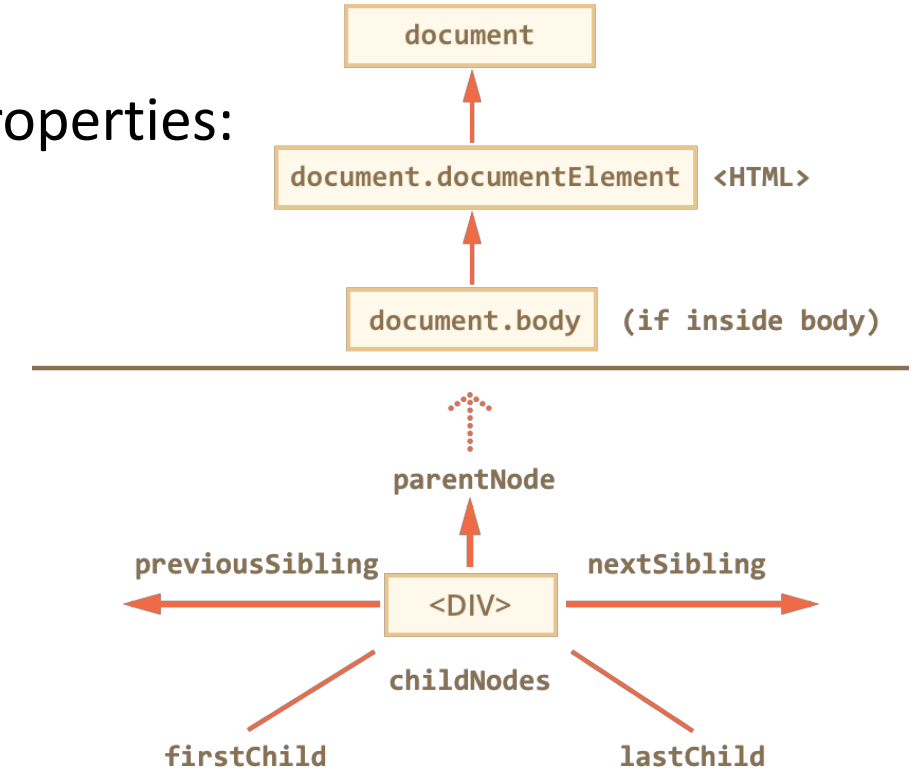
- As we explore the DOM, we also may want to apply JavaScript to it
  - Like: get a node and run some code to modify it, to see how it looks.
- Here are few tips to travel between the Elements tab and the console:
  - Select the first <li> in the Elements tab
  - Press Esc – it will open console right below the Elements tab
  - Now the last selected element is available as $0, the previously selected is $1 etc
  - We can run commands on them
  - For instance, $0.style.background = 'red' makes the selected list item red

Interaction with Console



Walking the DOM

- The DOM allows to do anything with elements, but first we need to reach the corresponding DOM object, get it into a variable, and then we are able to modify it
- All operations on the DOM start with the document object
- From it we can access any node
- The topmost tree nodes are available as document properties:
  - <html> = document.documentElement
  - <body> = document.body
  - <head> = document.head



Walking the DOM

- For example, the following script changes the background color of the body:

```
<html>
<head>
    <title></title>
</head>
<body>
    <script>
        document.body.style.backgroundColor = "red";
    </script>
</body>
</html>
```

Walking the DOM

- Note that a script cannot access an element that doesn't exist at the moment of running
- In particular, if a script is inside <head>, then document.body is unavailable, because the browser did not read it yet

```
<html>
<head>
    <title></title>
    <script>
        document.body.style.backgroundColor = "red";
    </script>
</head>
<body>
</body>
</html>
```

```
Uncaught TypeError: Cannot read property 'style' of null
    at DocumentBody.html:7
```

Child Nodes

- **Child nodes (or children)** – elements that are direct children, i.e., they are nested exactly in the given one
  - For instance, <head> and <body> are children of <html> element
- The childNodes collection provides access to all child nodes, including text nodes
- The example below shows the children of document.body:

```
<body>
    <div>Begin</div>
    <ul>
        <li>Information</li>
    </ul>
    <div>End</div>
    <script>
        for (let i = 0; i < document.body.childNodes.length; i++) {
            alert(document.body.childNodes[i]); // Text, DIV, Text, UL, ..., SCRIPT
        }
    </script>
    ...more stuff...
</body>
```

Child Nodes

- Properties **firstChild** and **lastChild** give fast access to the first and last children
- If there exist child nodes, then the following is always true:

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

- There's also a special function **elem.hasChildNodes()** to check whether there are any child nodes

DOM Collections

- childNodes looks like an array, but it is rather a *collection* – a special array-like iterable object
- There are two important consequences:
  - We can use for..of to iterate over it:

```
for (let node of document.body.childNodes) {
    alert(node); // shows all nodes from the collection
}
```

  - Array methods won't work, because it's not an array:

```
alert(document.body.childNodes.filter); // undefined (there's no filter method!)
```

- DOM collections are read-only
  - We can't replace a child by something else, assigning childNodes[i] = ...
- Almost all DOM collections are *live*
  - They reflect the current state of DOM

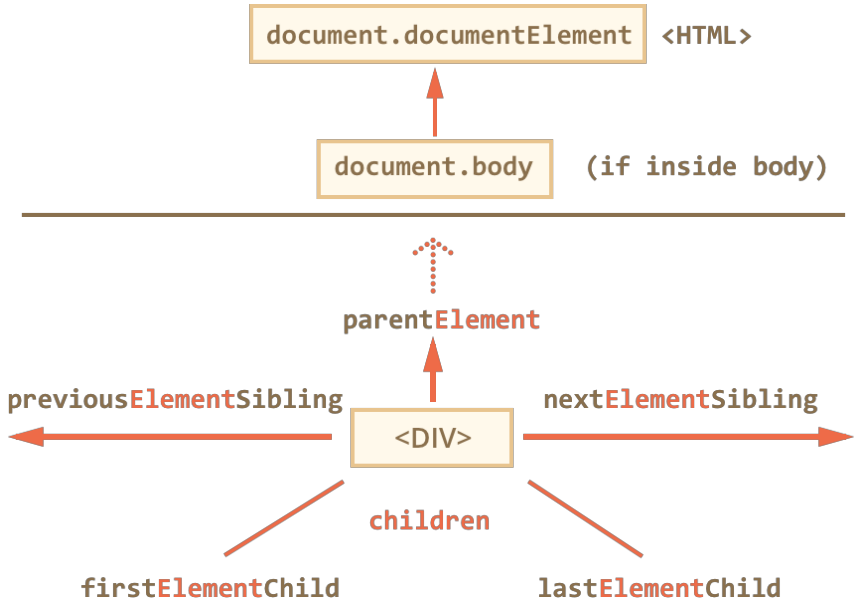
Siblings and the Parent

- **Siblings** are nodes that are children of the same parent
  - The next node of the same parent is available as **nextSibling**
  - The previous node of the same parent is available as **previousSibling**
- The parent is available as **parentNode**

```
<html><head></head><body><script>
    // HTML is "dense" to evade extra "blank" text nodes.

    // parent of <body> is <html>
    alert(document.body.parentNode === document.documentElement); // true

    // after <head> goes <body>
    alert(document.head.nextSibling); // HTMLBodyElement

    // before <body> goes <head>
    alert(document.body.previousSibling); // HTMLHeadElement
</script></body></html>
```

Element-Only Navigation

- Navigation properties listed above refer to *all* nodes
  - For instance, in childNodes we can see text nodes, element nodes, and even comment nodes if there exist
- But for many tasks we want to manipulate only element nodes that represent tags
- The following navigation links take only *element nodes* into account:



Element-Only Navigation

- For example, the following script it shows only the child elements of document.body:

```
<html>
<body>
    <div>Begin</div>
    <ul>
        <li>Information</li>
    </ul>
    <div>End</div>

    <script>
        for (let elem of document.body.children) {
          alert(elem); // DIV, UL, DIV, SCRIPT
        }
    </script>
</body>
</html>
```

Exercise (1)

- For the page:

```
<html>
<body>
    <div>Users:</div>
    <ul>
        <li>John</li>
        <li>Adam</li>
    </ul>
</body>
</html>
```

- How to access:
  - The DOM node?
  - The <ul> DOM node?
  - The second <li> (with Adam)?

Exercise (2)

- Write the code to paint all diagonal table cells in red:
- The result should be:

1:1	2:1	3:1	4:1
1:2	2:2	3:2	4:2
1:3	2:3	3:3	4:3
1:4	2:4	3:4	4:4

- Hint: Use the browser inspector to examine the DOM tree structure

```
<html>
<body>
    <table>
        <tr>
            <td>1:1</td>
            <td>2:1</td>
            <td>3:1</td>
            <td>4:1</td>
        </tr>
        <tr>
            <td>1:2</td>
            <td>2:2</td>
            <td>3:2</td>
            <td>4:2</td>
        </tr>
        <tr>
            <td>1:3</td>
            <td>2:3</td>
            <td>3:3</td>
            <td>4:3</td>
        </tr>
        <tr>
            <td>1:4</td>
            <td>2:4</td>
            <td>3:4</td>
            <td>4:4</td>
        </tr>
    </table>
</body>
</html>
```

Searching Elements

- DOM navigation properties are great when elements are close to each other
- What if they are not? How to get an arbitrary element of the page?
- There are additional searching methods for that, which we'll see on the next slides

getElementById

- If an element has the id attribute, then there's a global variable whose name is identical to that id
- We can use this variable to access the element, like this:

```
<div id="elem">Some element</div>

<script>
    alert(elem); // DOM-element with id="elem"
    alert(window.elem); // accessing global variable like this also works
</script>
```

- However, if we declare another variable with the same name, it shadows the variable created by the browser
- Also, when we look in JS and don't have HTML in view, it's not obvious where the variable comes from

getElementById

- The better alternative is to use a special method document.getElementById(id):

```
<div id="elem">Some element</div>

<script>
    let elem = document.getElementById("elem");
    elem.style.background = "red";
</script>
```

- The id must be unique
  - If there are multiple elements with the same id, the behavior of the corresponding methods is unpredictable
  - The browser may return any of them at random
  - So please stick to the rule and keep id unique

getElementsBy*

- There are also other methods to look for nodes:
  - **elem.getElementsByTagName**(tag) looks for elements with the given tag and returns the collection of them
  - **elem.getElementsByClassName**(className) returns elements that have the given CSS class
  - **document.getElementsByName**(name) returns elements with the given name attribute
    - Exists for historical reasons, very rarely used

```
<div>First</div>
<div>Second</div>
<div>Third</div>

<script>
    // get all divs in the document
    let divs = document.getElementsByTagName("div");
    for (let div of divs) {
        alert(div.innerHTML); // First, Second, Third
    }
</script>
```

QuerySelectorAll

- **elem.querySelectorAll**(css) returns all elements inside elem matching the given CSS selector (elem can also be the document itself)
- That's the most often used and powerful method
- For instance, here we look for all <li> elements that are last children:

```
<ul>
    <li>The</li>
    <li>test</li>
</ul>
<ul>
    <li>has</li>
    <li>passed</li>
</ul>
<script>
    let elements = document.querySelectorAll('ul > li:last-child');

    for (let elem of elements) {
        alert(elem.innerHTML); // "test", "passed"
    }
</script>
```

QuerySelector

- **elem.querySelector**(css) returns the first element for the given CSS selector
  - The result is the same as elem.querySelectorAll(css)[0], but the latter is looking for all elements and picking one, thus elem.querySelector is faster and shorter to write

```
<ul>
    <li>The</li>
    <li>test</li>
</ul>
<ul>
    <li>has</li>
    <li>passed</li>
</ul>
<script>
    let firstList = document.querySelector('ul:first-child');
    firstList.style.color = "red";
</script>
```

- The
- test

- has
- passed

Live Collections

- All methods "getElementsBy*" return a *live* collection
- Such collections always reflect the current state of the document and “auto-update” when it changes.
- In the example below, there are two scripts:
  - The first one creates a reference to the collections of . As of now, its length is 1.
  - The second script runs after the browser meets one more , so its length is 2.

```
<div>First div</div>
<script>
    let divs = document.getElementsByTagName('div');
    alert(divs.length); // 1
</script>

<div>Second div</div>
<script>
    alert(divs.length); // 2
</script>
```

Live Collections

- In contrast, querySelectorAll returns a *static* collection
- It's like a fixed array of elements
- If we use it instead, then both scripts output 1:

```
<script>
    let divs = document.querySelectorAll('div');
    alert(divs.length); // 1
</script>

<div>Second div</div>

<script>
    alert(divs.length); // 1
</script>
```

Summary

- There are 6 main methods to search for nodes in DOM:

Method	Searches by…	Can call on an element?	Live?
getElementById	id	-	-
getElementsByName	name	-	√
getElementsByTagName	tag	√	√
getElementsByClassName	class	√	√
querySelector	CSS-selector	√	-
querySelectorAll	CSS-selector	√	-

  - Note that methods getElementById and getElementsByName can only be called in the context of the document: document.getElementById(...), while other methods can be called on elements too, e.g., elem.querySelectorAll(...) will search inside elem (in the DOM subtree)

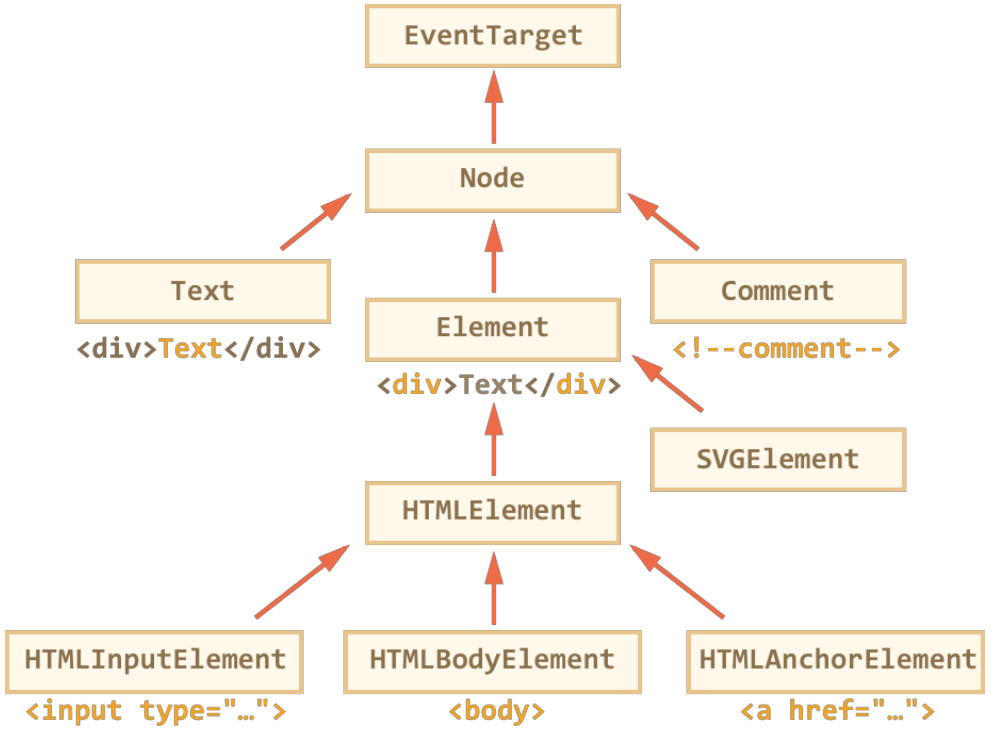
Exercise (3)

- Here's a document with a table and a form:
- How to find?
  - The table with id="age-table"
  - All label elements inside that table (there should be 3 of them)
  - The first td in that table (with the word "Age")
  - The form with the name search
  - The first input in that form
  - The last input in that form

```
<!DOCTYPE HTML>
<html>
<body>
    <form name="search">
        Search the visitors:
        <table id="age-table">
            <tr>
                <td>Age:</td>
                <td id="age-list">
                    <label>
                        <input type="radio" name="age" value="young">less than 18
                    </label>
                    <label>
                        <input type="radio" name="age" value="mature">18-50
                    </label>
                    <label>
                        <input type="radio" name="age" value="senior">more than 50
                    </label>
                </td>
            </tr>
            <tr>
                <td>Additionally:</td>
                <td>
                    <input type="text" name="info[0]">
                    <input type="text" name="info[1]">
                    <input type="text" name="info[2]">
                </td>
            </tr>
        </table>

        <input type="submit" value="Search!">
    </form>
</body>
</html>
```

DOM Node Classes

- Each DOM node belongs to the corresponding built-in class
- The root of the hierarchy is EventTarget, that is inherited by Node, and other DOM nodes inherit from it



DOM Node Classes

- To see the DOM node class name, we can recall that an object has the constructor property, which references to the class constructor:

```
alert(document.body.constructor.name); // HTMLBodyElement
```

- We also can use instanceof to check the inheritance:

```
alert(document.body instanceof HTMLBodyElement); // true
alert(document.body instanceof HTMLElement); // true
alert(document.body instanceof Element); // true
alert(document.body instanceof Node); // true
alert(document.body instanceof EventTarget); // true
```

- To inspect a DOM element in the console, use:
  - **console.log**(elem) shows the element DOM tree
  - **console.dir**(elem) shows the element as a DOM object, good to explore its properties

DOM Node Properties

- DOM nodes have different properties depending on their class
- The full set of properties and methods of a given node comes as the result of the inheritance hierarchy
- For example, let's consider the DOM object for an <input> element
- It belongs to the **HTMLInputElement** class
- It gets properties and methods as a superposition of:
  - HTMLInputElement – this class provides input-specific properties
  - HTMLElement – provides common HTML element methods (and getters/setters)
  - Element – provides generic element methods
  - Node – provides common DOM node properties
  - EventTarget – gives the support for events (to be covered)
  - and finally it inherits from Object, so "pure object" methods like toString are also available

innerHTML

- The **innerHTML** property allows to get the HTML inside the element as a string
- We can also modify it, so it's one of most powerful ways to change the page.
- The example shows the contents of document.body and then replaces it completely:

```
<body>
    <p>A paragraph</p>
    <div>A div</div>

    <script>
        alert(document.body.innerHTML); // read the current contents
        document.body.innerHTML = 'The new BODY!'; // replace it
    </script>
</body>
```

- We can append "more HTML" by using elem.innerHTML+="something"
  - However, we should be very careful about it, because this causes a full overwrite of the element's content

textContent: pure text

- The **textContent** provides access to the *text* inside the element: only text, minus all <tags>
- Compare the two:

```
<body>
    <div id="></div>
    <div></div>

    <script>
        document.body.children[0].innerHTML = "Hello<br/>World";
        document.body.children[1].textContent = "Hello<br/>World";
    </script>
</body>
```

Hello
World
Hello
World

- The first gets the text "as HTML": all tags become tags, so we see the line break
- The second gets the name "as text", so we literally see Hello
World

The “hidden” Property

- The “hidden” attribute and the DOM property specifies whether the element is visible or not
- We can use it in HTML or assign using JavaScript, like this:

```
<body>
    <div>Both divs below are hidden</div>
    <div hidden>With the attribute "hidden"</div>
    <div id="elem">JavaScript assigned the property "hidden"</div>
    <script>
        elem.hidden = true;
    </script>
</body>
```

Both divs below are hidden

- Technically, hidden works the same as style="display:none". But it’s shorter to write.

More Properties

- DOM elements have additional properties, many of them provided by their class:
  - **value** – the value for <input>, <select> and <textarea>
  - **href** – the “href” for <a href="...">
  - **id** – the value of “id” attribute, for all elements
  - and many more…

```
<input type="text" id="elem" value="5"/>
<script>
    alert(elem.type); // text
    alert(elem.id); // elem
    alert(elem.value); // 5
</script>
```

- Most standard HTML attributes have the corresponding DOM property, and we can access them using JavaScript
- You can output the full list of properties of a given element using **console.dir**(elem)

Exercise (4)

- What does this code show?

```
<html>
<body>
    <script>
        let body = document.body;

        body.innerHTML = "<p>My paragraph</p>";

        alert(document.body.lastChild.constructor.name); // what's here?
    </script>
</body>
</html>
```

Exercise (5)

- Create a colored clock like here:

11:08:42

Custom DOM Properties

- DOM nodes are regular JavaScript objects
- We can add our own properties and methods to them
- For instance, let's create a new property in document.body:

```
document.body.myData = {
    name: 'Ceaser',
    title: 'Emperor'
};
alert(document.body.myData.title); // Emperor
```

- We can also modify built-in prototypes like Element.prototype, and add new methods to all elements:

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

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

HTML Attributes

- When the browser loads the page, it “parses” HTML text and generates DOM objects from it
- For element nodes most standard HTML attributes automatically become properties of their corresponding DOM objects
- But the attribute-property mapping is not one-to-one!
  - For example, HTML attribute values are always strings while DOM properties are typed
  - If an HTML attribute is non-standard, there won’t be DOM-property for it
- All HTML attributes are accessible using following methods:
  - **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

HTML Attributes

- Example for working with HTML attributes:

```
<div id="elem" about="Elephant"></div>
<script>
    alert(elem.getAttribute("about")); // 'Elephant', reading
    elem.setAttribute("Test", 123); // writing

    for (let attr of elem.attributes) { // list all attributes
        alert(`${attr.name} = ${attr.value}`);
    }
</script>
```

- The HTML attribute may differ from its corresponding DOM property, for example the style attribute is a string, but the style property is an object:

```
<div id="div" style="color:red;font-size:120%">Hello</div>
<script>
    alert(div.getAttribute('style')); // color:red;font-size:120%
    alert(div.style); // [object CSSStyleDeclaration]
    alert(div.style.color); // red
</script>
```

HTML Custom Attributes

- There is a possible problem with custom attributes
- What if we use a non-standard attribute for our purposes, and later the standard introduces it and makes it do something?
- To avoid conflicts, there exist **data-*** attributes
- All attributes starting with “data-” are reserved for programmers’ use
- They are available in the **dataset** property:

```
<div id="elem" data-about="Elephants">
    <script>
        alert(elem.dataset.about); // Elephants
    </script>
</div>
```

- Multiword attributes like data-order-state become camel-cased: dataset.orderState

Exercise

- Write the code to select the element with data-widget-name attribute from the document and to read the attribute's value

```
<!DOCTYPE html>

<html>
<body>
    <div data-widget-name="menu">Choose the genre</div>

    <script>
        /* your code */
    </script>
</body>
</html>
```

Element Style

- The property elem.style is an object that corresponds to what's written in the "style" attribute
  - Setting elem.style.width="100px" works as if we had in the attribute style="width:100px"
- For multi-word property, camel casing is used:

```
background-color => elem.style.backgroundColor
z-index => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

- For instance, the following script lets the user change the page's background color:

```
<script>
    document.body.style.backgroundColor = prompt('Background color?');
</script>
```

Mind the Units

- CSS units must be provided in style values
- For instance, we should not set elem.style.top to 10, but rather to 10px

```
<div id="elem">
    Hello world
</div>

<script>
    // doesn't work!
    elem.style.margin = 20;
    alert(elem.style.margin); // '' (empty string, the assignment is ignored)

    // now add the CSS unit (px) - and it works
    elem.style.margin = '20px';
    alert(elem.style.margin); // 20px

    alert(elem.style.marginTop); // 20px
    alert(elem.style.marginLeft); // 20px
</script>
```

Styles and Classes

- There are generally two ways to style an element:
  - Create a class in CSS and add it:
  - Write properties directly into style:
- CSS is always the preferred way – not only for HTML, but in JavaScript as well
- We should only manipulate the style property if classes “can’t handle it”
- For instance, style is acceptable if we calculate coordinates of an element dynamically and want to set them from JavaScript, like this:

```
let top = /* complex calculations */;
let left = /* complex calculations */;
elem.style.left = left; // e.g '123px'
elem.style.top = top; // e.g '456px'
```

className

- In the ancient time, there was a limitation in JavaScript: a reserved word like "class" could not be an object property
- So the property “className” was introduced: **elem.className** corresponds to the "class" attribute
- For instance:

```
<h1 id="elem" class="header main"></h1>
<script>
    alert(elem.className); // header main
</script>
```

- If we assign something to elem.className, it replaces the whole strings of classes

classList

- Sometimes we only want to add/remove a single class
- There's another property for that: **elem.classList**
- Methods of classList:
  - **elem.classList.add/remove**("class") – adds/removes the class
  - **elem.classList.toggle**("class") – if the class exists, then removes it, otherwise adds it
  - **elem.classList.contains**("class") – returns true/false, checks for the given class
- For instance:

```
<h1 id="elem" class="header main"></h1>
<script>
    elem.classList.add("article");
    alert(elem.className); // header main article
</script>
```

Computed Styles

- The **style** property operates only on the value of the "style" attribute, without any CSS cascade
- So we can't read anything that comes from CSS classes using elem.style
- For instance, here style doesn't see the margin:

```
<head>
    <style>
        body {
            color: red;
            margin: 5px
        }
    </style>
</head>
<body>
    The red text
    <script>
        alert(document.body.style.color); // empty
        alert(document.body.style.marginTop); // empty
    </script>
</body>
```

Computed Styles

- The method **getComputedStyle**(element) returns an object with style properties, like elem.style, but with respect to all CSS classes:

```
<head>
    <style>
        body {
            color: red;
            margin: 5px
        }
    </style>
</head>
<body>
    The red text
    <script>
        let computedStyle = getComputedStyle(document.body);

        // now we can read the margin and the color from it
        alert(computedStyle.marginTop); // 5px
        alert(computedStyle.color); // rgb(255, 0, 0)
    </script>
</body>
```

Computed and Resolved Values

- There are two concepts in CSS:
  - A *computed* style value is the value after all CSS rules and CSS inheritance is applied, as the result of the CSS cascade. It can look like height:1em or font-size:125%.
  - A *resolved* style value is the one finally applied to the element. Values like 1em or 125% are relative. The browser takes the computed value and makes all units fixed and absolute, for instance: height:20px or font-size:16px
- Originally, getComputedStyle() was created to get computed values, but it turned out that resolved values are much more convenient, and the standard changed
- Nowadays getComputedStyle() returns the resolved value of the property

Exercise (7)

- Move the following div 20px to the right, by increasing its margin-left property
  - Hint: first use getComputedStyle() to get its current marginLeft value

```
<head>
    <style>
        #div1 {
            border: red dotted 1px;
            width: 20%;
            margin: 10px;
            padding: 5px;
        }
    </style>
</head>
<body>
    <div id="div1">
        Hello world
    </div>
</body>
```

Creating Elements

- **document.createElement(tag)** creates a new element with the given tag:

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

- After that, we have a ready DOM element
- To make the element show up, we need to insert it somewhere into document
- There are several methods for inserting a node into a parent element

Method	Description
parentElem.appendChild(*node*)	appends *node* as the last child of *parentElem*
parentElem.insertBefore(*node*, *nextSibling*)	inserts *node* before *nextSibling* into *parentElem*
parentElem.replaceChild(*node*, *oldChild*)	replaces *oldChild* with *node* among children

Creating Elements

- The following example adds a new <li> to the end of <ol>:

```
<ol id="list">
    <li>0</li>
    <li>1</li>
    <li>2</li>
</ol>

<script>
    let newLi = document.createElement("li");
    newLi.innerHTML = "Hello, world!";

    list.appendChild(newLi);
</script>
```

1. 0
2. 1
3. 2
4. Hello, world!

Creating Elements

- The following code inserts a new list item before the second <li>:

```
<ol id="list">
    <li>0</li>
    <li>1</li>
    <li>2</li>
</ol>

<script>
    let newLi = document.createElement("li");
    newLi.innerHTML = "Hello, world!“;

    list.insertBefore(newLi, list.children[1]);
</script>
```

1. 0
2. Hello, world!
3. 1
4. 2

Insertion Methods

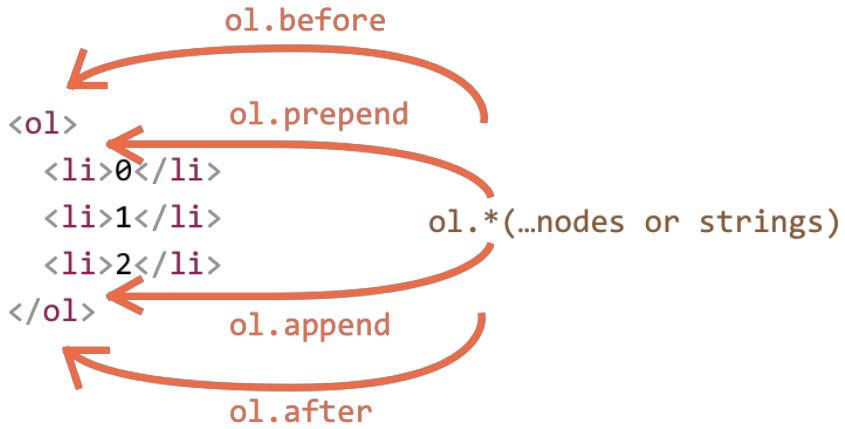
- There is another set of methods that provide more flexible insertions:

Method	Description
node.append(...*nodes* or *strings*)	appends *nodes* or *strings* at the end of node
node.prepend(...*nodes* or *strings*)	inserts *nodes* or *strings* into the beginning of node
node.before(...*nodes* or *strings*)	inserts *nodes* or *strings* before the node
node.after(...*nodes* or *strings*)	inserts *nodes* or *strings* after the node
node.replaceWith(...*nodes* or *strings*)	replaces node with the given *nodes* or *strings*

Insertion Methods

- Here's an example of using these methods to add more items to a list and the text before/after it:

```
<ol id="list">
    <li>0</li>
    <li>1</li>
    <li>2</li>
</ol>

<script>
    list.before("before");
    list.after("after");

    let prepend = document.createElement("li");
    prepend.innerHTML = "prepend";
    list.prepend(prepend);

    let append = document.createElement("li");
    append.innerHTML = "append";
    list.append(append);
</script>
```



before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Cloning Nodes

- Sometimes when we have a big element, it may be faster and simpler to clone it rather than create a new element
- **elem.cloneNode(true)** creates a “deep” clone of the element
  - with all attributes and subelements
- **elem.cloneNode(false)** creates a clone without child elements

Cloning Nodes

- An example of copying a tag showing a message:

```
<head>
    <style>
        .alert {
            padding: 15px;
            border: 1px solid #d6e9c6;
            border-radius: 4px;
            color: #3c763d;
            background-color: #dff0d8;
        }
    </style>
</head>
<body>
    <div class="alert" id="div">
        <strong>Hi there!</strong> You've read an important message.
    </div>
    <script>
        let div2 = div.cloneNode(true); // clone the message
        div2.querySelector('strong').innerHTML = 'Bye there!'; //
change the clone
        div.after(div2); // show the clone after the existing div
    </script>
</body>
```

Hi there! You've read an important message.

Bye there! You've read an important message.

Removal Methods

- To remove nodes, there are the following methods:

Method	Description
parentElem.removeChild(*node*)	removes *elem* from *parentElem* (assuming it's a child)
node.remove()	Removes the node from its place

- The second method is much shorter. The first one exists for historical reasons.
- For example, let's make our message disappear after a second:

```
<div class="alert" id="div">
    <strong>Hi there!</strong> You've read an important message.
</div>

<script>
    setTimeout(() => div.remove(), 1000);
</script>
```

Moving Nodes

- If we want to *move* an element to another place – there's no need to remove it from the old one.
- All insertion methods automatically remove the node from the old place
- For instance, let's swap elements:

```
<div id="first">First</div>
<div id="second">Second</div>

<script>
    // no need to call remove
    second.after(first); // take #second and after it - insert #first
</script>
```

Second
First

Exercise (8)

- Create a function **clear(elem)** that removes everything from inside the element (but keeps the element itself)

```
<ol id="list">
    <li>Hello</li>
    <li>World</li>
</ol>

<script>
    function clear(elem) { /* your code */ }

    clear(list); // clears the list
</script>
```

Exercise (9)

- Write the code to insert the elements 2 and 3 between the two <li> here:

```
<ul id="ul">
    <li id="one">1</li>
    <li id="two">4</li>
</ul>
```

Exercise (10)

- Write a function **createCalendar**(elem, year, month)
- The call should create a calendar for the given year/month and put it inside elem
- The calendar should be a table, where a week is <tr>, and a day is <td>
- The table top should be <th> with weekday names
- For instance, createCalendar(cal, 2018, 6) should generate in element cal the following calendar:

SU	MO	TU	WE	TH	FR	SA
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30