Lesson - 18

DOM
Accessing DOM with JS

SkillUp by Dobrea Vladislav

Lesson Plan

- Browser Environments
- DOM Tree
- Walking the DOM
- Searching
- Node Properties
- Attributes and Properties
- Modifying the document

Browser environment and specs

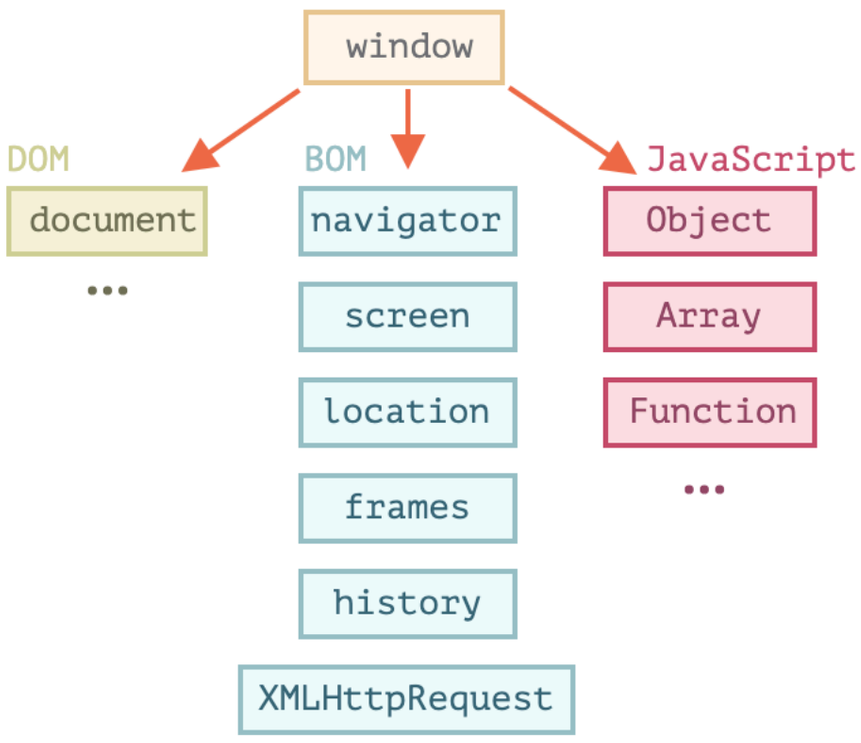
The JavaScript language was initially created for web browsers. Since then it has evolved and become a language with many uses and platforms.

A platform may be a browser, or a web-server or another *host*, even a coffee machine. Each of them provides platform-specific functionality. The JavaScript specification calls that a *host environment*.

A host environment provides own 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 and specs

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:

1. First, it is a global object for JavaScript code, as described in the chapter Global object.
2. Second, it represents the “browser window” and provides methods to control it.

Browser environment and specs

For instance, here we use it as a global object:

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

// global functions are methods of the global object:
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

Document Object Model, or DOM for short, represents all page content as objects that can be modified.

The `document` object is the main "entry point" to the page. 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);
```

Here we used `document.body.style`, but there's much, much more. Properties and methods are described in the specification:

- **DOM Living Standard** at https://dom.spec.whatwg.org

> **DOM is not only for browsers**
>
> The DOM specification explains the structure of a document and provides objects to manipulate it. There are non-browser instruments that use DOM too.
>
> For instance, server-side scripts that download HTML pages and process them can also use DOM. They may support only a part of the specification though.

Browser Object Model

The Browser Object Model (BOM) represents additional objects provided by the browser (host environment) for working with everything except the document.

For instance:

- The navigator object provides background information about the browser and the operating system. There are many properties, but the two most widely known are: `navigator.userAgent` – about the current browser, and `navigator.platform` – about the platform (can help to differ between Windows/Linux/Mac etc).
- The location object allows us to read the current URL and can 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
}
```

Functions `alert/confirm/prompt` are also a part of BOM: they are directly not related to the document, but represent pure browser methods of communicating with the user.

Specifications

BOM is the part of the general HTML specification.

Yes, you heard that right. The HTML spec at https://html.spec.whatwg.org is not only about the "HTML language" (tags, attributes), but also covers a bunch of objects, methods and browser-specific DOM extensions. That's "HTML in broad terms". Also, some parts have additional specs listed at https://spec.whatwg.org.

DOM Tree

The backbone of an HTML document is tags.

According to the Document Object Model (DOM), every HTML tag is an object. Nested tags are “children” of the enclosing one. The text inside a tag is an object as well.

All these objects are accessible using JavaScript, and we can use them to modify the page.

For example, `document.body` is the object representing the `<body>` tag.

Running this code will make the `<body>` red for 3 seconds:

```
1 document.body.style.background = 'red'; // make the background red
2
3 setTimeout(() => document.body.style.background = '', 3000); // return back
```

Here we used `style.background` to change the background color of `document.body`, but there are many other properties, such as:

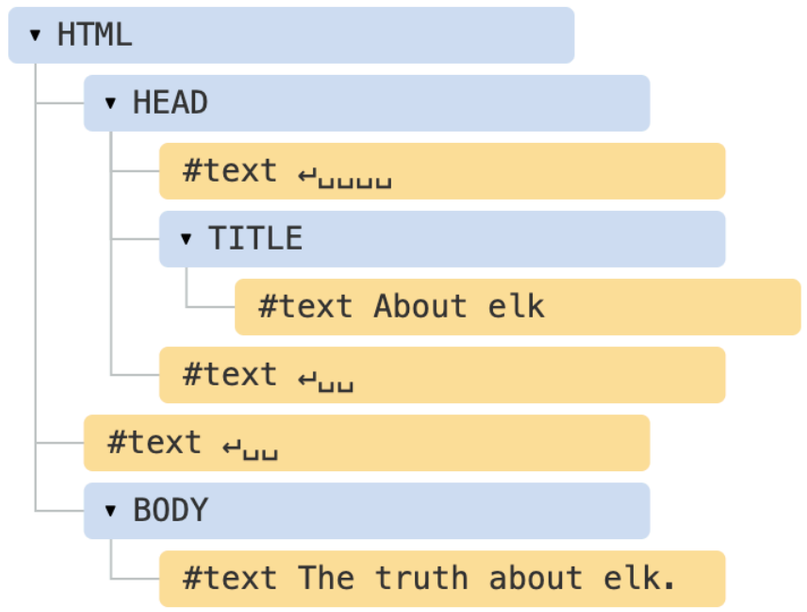
- `innerHTML` – HTML contents of the node.
- `offsetWidth` – the node width (in pixels)
- ...and so on.

Soon we’ll learn more ways to manipulate the DOM, but first we need to know about its structure.

Example of DOM

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

The DOM represents HTML as a tree structure of tags. Here's how it looks:



On the picture above, you can click on element nodes and their children will open/collapse.

Example of DOM

Every tree node is an object.

Tags are *element nodes* (or just elements) and form the tree structure: `<html>` is at the root, then `<head>` and `<body>` are its children, etc.

The text inside elements forms *text nodes*, labelled as `#text`. A text node contains only a string. It may not have children and is always a leaf of the tree.

For instance, the `<title>` tag has the text `"About elk"`.

Please note the special characters in text nodes:

- a newline: ↵ (in JavaScript known as `\n`)
- a space: ␣

Spaces and newlines are totally valid characters, like letters and digits. They form text nodes and become a part of the DOM. So, for instance, in the example above the `<head>` tag contains some spaces before `<title>`, and that text becomes a `#text` node (it contains a newline and some spaces only).

There are only two top-level exclusions:

1. Spaces and newlines before `<head>` are ignored for historical reasons.
2. If we put something after `</body>`, then that is automatically moved inside the `body`, at the end, as the HTML spec requires that all content must be inside `<body>`. So there can't be any spaces after `</body>`.

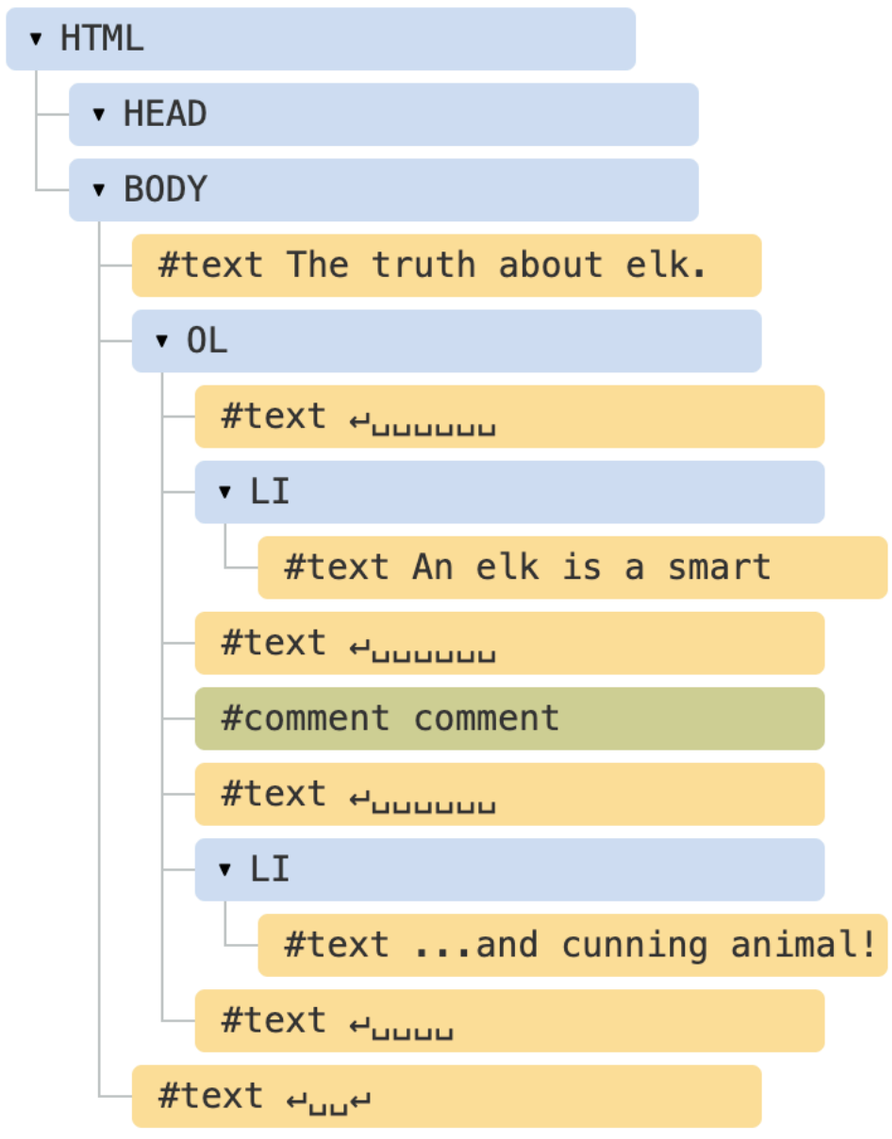
Other node types

There are some other node types besides elements and text nodes.

For example, comments:

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

Other node types



We can see here a new tree node type – *comment node*, labeled as `#comment`, between two text nodes.

We may think – why is a comment added to the DOM? It doesn't affect the visual representation in any way. But there's a rule – if something's in HTML, then it also must be in the DOM tree.

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

Even the `<!DOCTYPE...>` directive at the very beginning of HTML is also a DOM node. It's in the DOM tree right before `<html>`. We are not going to touch that node, we even don't draw it on diagrams for that reason, but it's there.

The `document` object that represents the whole document is, formally, a DOM node as well.

There are 12 node types. In practice we usually work with 4 of them:

1. `document` – the "entry point" into DOM.
2. element nodes – HTML-tags, the tree building blocks.
3. text nodes – contain text.
4. comments – sometimes we can put information there, it won't be shown, but JS can read it from the DOM.

Summary

An HTML/XML document is represented inside the browser as the DOM tree.

- Tags become element nodes and form the structure.
- Text becomes text nodes.
- …etc, everything in HTML has its place in DOM, even comments.

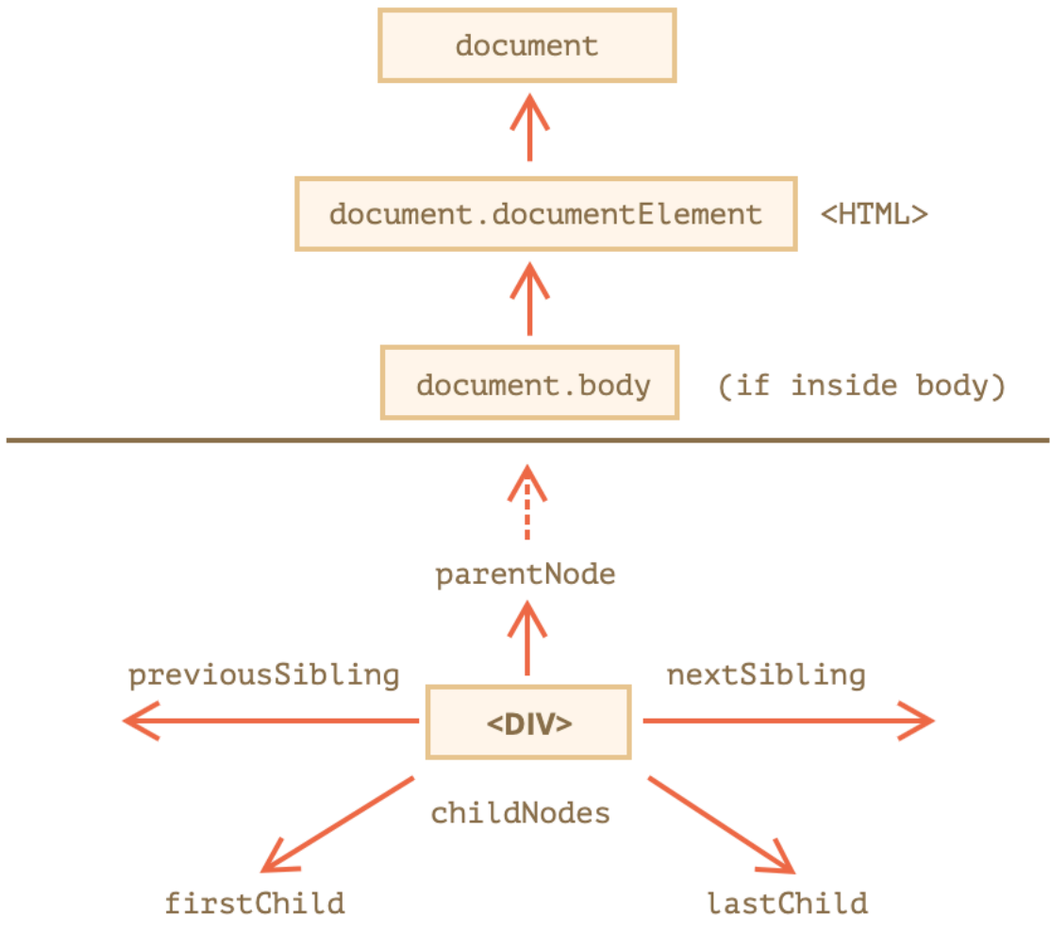
We can use developer tools to inspect DOM and modify it manually.

Walking the DOM

The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object.

All operations on the DOM start with the `document` object. That's the main "entry point" to DOM. From it we can access any node.

Here's a picture of links that allow for travel between DOM nodes:



On top: documentElement and body

The topmost tree nodes are available directly as `document` properties:

`<html>` = `document.documentElement`

The topmost document node is `document.documentElement`. That's the DOM node of the `<html>` tag.

`<body>` = `document.body`

Another widely used DOM node is the `<body>` element – `document.body`.

`<head>` = `document.head`

The `<head>` tag is available as `document.head`.

Children: childNodes, firstChild, lastChild

There are two terms that we'll use from now on:

- **Child nodes (or children)** – elements that are direct children. In other words, they are nested exactly in the given one. For instance, `<head>` and `<body>` are children of `<html>` element.
- **Descendants** – all elements that are nested in the given one, including children, their children and so on.

For instance, here `<body>` has children `<div>` and `<ul>` (and few blank text nodes):

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>
      <b>Information</b>
    </li>
  </ul>
</body>
</html>
```

...And descendants of `<body>` are not only direct children `<div>`, `<ul>` but also more deeply nested elements, such as `<li>` (a child of `<ul>`) and `<b>` (a child of `<li>`) – the entire subtree.

childNodes

The `childNodes` collection lists all child nodes, including text nodes.

The example below shows children of `document.body`:

```
<html>
<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, ..., SCRI
    }
  </script>
  ...more stuff...
</body>
</html>
```

Please note an interesting detail here. If we run the example above, the last element shown is `<script>`. In fact, the document has more stuff below, but at the moment of the script execution the browser did not read it yet, so the script doesn't see it.

Properties `firstChild` and `lastChild` give fast access to the first and last children.

DOM Collections

As we can see, `childNodes` looks like an array. But actually it's not an array, but rather a *collection* – a special array-like iterable object.

There are two important consequences:

1. We can use `for..of` to iterate over it:

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

That's because it's iterable (provides the `Symbol.iterator` property, as required).

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

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

The first thing is nice. The second is tolerable, because we can use `Array.from` to create a "real" array from the collection, if we want array methods:

```
alert( Array.from(document.body.childNodes).filter ); // function
```

Important notes

⚠️ DOM collections are read-only

DOM collections, and even more – *all* navigation properties listed in this chapter are read-only.

We can't replace a child by something else by assigning `childNodes[i] = ...`.

Changing DOM needs other methods. We will see them in the next chapter.

⚠️ DOM collections are live

Almost all DOM collections with minor exceptions are *live*. In other words, they reflect the current state of DOM.

If we keep a reference to `elem.childNodes`, and add/remove nodes into DOM, then they appear in the collection automatically.

⚠️ Don't use `for..in` to loop over collections

Collections are iterable using `for..of`. Sometimes people try to use `for..in` for that.

Please, don't. The `for..in` loop iterates over all enumerable properties. And collections have some "extra" rarely used properties that we usually do not want to get:

```
<body>
<script>
  // shows 0, 1, length, item, values and more.
  for (let prop in document.body.childNodes) alert(prop);
</script>
</body>
```

Siblings and the parent

Siblings are nodes that are children of the same parent.

For instance, here `<head>` and `<body>` are siblings:

```
<html>
  <head>...</head><body>...</body>
</html>
```

- `<body>` is said to be the "next" or "right" sibling of `<head>`,
- `<head>` is said to be the "previous" or "left" sibling of `<body>`.

The next sibling is in `nextSibling` property, and the previous one – in `previousSibling`.

The parent is available as `parentNode`.

For example:

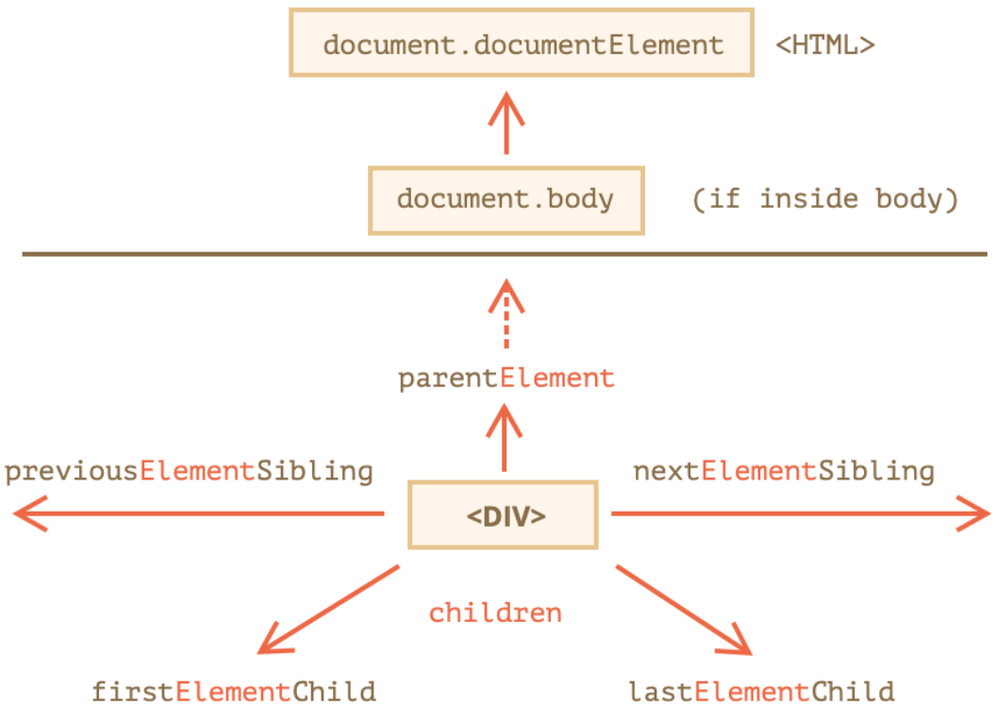
```
// parent of <body> is <html>
alert( document.body.parentNode === document.documentElement ); // true

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

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

Element only navigation

Navigation properties listed above refer to *all* nodes. For instance, in `childNodes` we can see both text nodes, element nodes, and even comment nodes if there exist.

But for many tasks we don't want text or comment nodes. We want to manipulate element nodes that represent tags and form the structure of the page.

So let's see more navigation links that only take *element nodes* into account:



The links are similar to those given above, just with `Element` word inside:

- `children` – only those children that are element nodes.
- `firstElementChild`, `lastElementChild` – first and last element children.
- `previousElementSibling`, `nextElementSibling` – neighbor elements.
- `parentElement` – parent element.

Summary

Given a DOM node, we can go to its immediate neighbors using navigation properties.

There are two main sets of them:

- For all

  nodes: `parentNode`, `childNodes`, `firstChild`, `lastChild`, `previousSibling`, `nextSibling`.
- For element nodes

  only: `parentElement`, `children`, `firstElementChild`, `lastElementChild`, `previousElementSibling`, `nextElementSibling`.

Some types of DOM elements, e.g. tables, provide additional properties and collections to access their content.

Searching: getElement*, querySelector*

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.

document.getElementById or just id

If an element has the `id` attribute, we can get the element using the method `document.getElementById(id)`, no matter where it is.

For instance:

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // get the element
  let elem = document.getElementById('elem');

  // make its background red
  elem.style.background = 'red';
</script>
```

Global id as variable

Also, there's a global variable named by `id` that references the element:

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // elem is a reference to DOM-element with id="elem"
  elem.style.background = 'red';

  // id="elem-content" has a hyphen inside, so it can't be a variable name
  // ...but we can access it using square brackets: window['elem-content']
</script>
```

Things to note!

⚠️ Please don't use id-named global variables to access elements

This behavior is described in the specification, so it's kind of standard. But it is supported mainly for compatibility.

The browser tries to help us by mixing namespaces of JS and DOM. That's fine for simple scripts, inlined into HTML, but generally isn't a good thing. There may be naming conflicts. Also, when one reads JS code and doesn't have HTML in view, it's not obvious where the variable comes from.

Here in the tutorial we use `id` to directly reference an element for brevity, when it's obvious where the element comes from.

In real life `document.getElementById` is the preferred method.

ℹ️ The `id` must be unique

The `id` must be unique. There can be only one element in the document with the given `id`.

If there are multiple elements with the same `id`, then the behavior of methods that use it is unpredictable, e.g. `document.getElementById` may return any of such elements at random. So please stick to the rule and keep `id` unique.

⚠️ Only `document.getElementById`, not `anyElem.getElementById`

The method `getElementById` that can be called only on `document` object. It looks for the given `id` in the whole document.

querySelectorAll

By far, the most versatile method, `elem.querySelectorAll(css)` returns all elements inside `elem` matching the given CSS selector.

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>
```

This method is indeed powerful, because any CSS selector can be used.

Can use pseudo-classes as well

Pseudo-classes in the CSS selector like `:hover` and `:active` are also supported. For instance, `document.querySelectorAll(':hover')` will return the collection with elements that the pointer is over now (in nesting order: from the outermost `<html>` to the most nested one).

querySelector

The call to `elem.querySelector(css)` returns the first element for the given CSS selector.

In other words, the result is the same as `elem.querySelectorAll(css)[0]`, but the latter is looking for *all* elements and picking one, while `elem.querySelector` just looks for one. So it's faster and also shorter to write.

Matches

Previous methods were searching the DOM.

The elem.matches(css) does not look for anything, it merely checks if `elem` matches the given CSS-selector. It returns `true` or `false`.

The method comes in handy when we are iterating over elements (like in an array or something) and trying to filter out those that interest us.

For instance:

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
  // can be any collection instead of document.body.children
  for (let elem of document.body.children) {
    if (elem.matches('a[href$="zip"]')) {
      alert("The archive reference: " + elem.href );
    }
  }
</script>
```

Closest

Ancestors of an element are: parent, the parent of parent, its parent and so on. The ancestors together form the chain of parents from the element to the top.

The method `elem.closest(css)` looks the nearest ancestor that matches the CSS-selector. The `elem` itself is also included in the search.

In other words, the method `closest` goes up from the element and checks each of parents. If it matches the selector, then the search stops, and the ancestor is returned.

For instance:

```
<h1>Contents</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Chapter 1</li>
    <li class="chapter">Chapter 1</li>
  </ul>
</div>

<script>
  let chapter = document.querySelector('.chapter'); // LI

  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

  alert(chapter.closest('h1')); // null (because h1 is not an ancestor)
</script>
```

getElementsBy*

There are also other methods to look for nodes by a tag, class, etc.

Today, they are mostly history, as `querySelector` is more powerful and shorter to write.

So here we cover them mainly for completeness, while you can still find them in the old scripts.

- `elem.getElementsByTagName(tag)` looks for elements with the given tag and returns the collection of them. The `tag` parameter can also be a star `"*"` for "any tags".
- `elem.getElementsByClassName(className)` returns elements that have the given CSS class.
- `document.getElementsByName(name)` returns elements with the given `name` attribute, document-wide. Very rarely used.

For instance:

```
1 // get all divs in the document
2 let divs = document.getElementsByTagName('div');
```

Things to note!

⚠️ Don’t forget the `"s"` letter!

Novice developers sometimes forget the letter `"s"`. That is, they try to call `getElementByTagName` instead of `getElement`**`s`**`ByTagName`.

The `"s"` letter is absent in `getElementById`, because it returns a single element. But `getElementsByTagName` returns a collection of elements, so there’s `"s"` inside.

⚠️ It returns a collection, not an element!

Another widespread novice mistake is to write:

```
// doesn't work
document.getElementsByTagName('input').value = 5;
```

That won’t work, because it takes a *collection* of inputs and assigns the value to it rather than to elements inside it.

We should either iterate over the collection or get an element by its index, and then assign, like this:

```
// should work (if there's an input)
document.getElementsByTagName('input')[0].value = 5;
```

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.

1. The first one creates a reference to the collection of `<div>`. As of now, its length is `1`.
2. The second scripts runs after the browser meets one more `<div>`, 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`:

```
<div>First div</div>

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

<div>Second div</div>

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

Now we can easily see the difference. The static collection did not increase after the appearance of a new `div` in the document.

Summary

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

Method	Searches by...	Can call on an element?	Live?
`querySelector`	CSS-selector	✔	-
`querySelectorAll`	CSS-selector	✔	-
`getElementById`	`id`	-	-
`getElementsByName`	`name`	-	✔
`getElementsByTagName`	tag or `'*'`	✔	✔
`getElementsByClassName`	class	✔	✔

By far the most used are `querySelector` and `querySelectorAll`, but `getElementBy*` can be sporadically helpful or found in the old scripts.

Besides that:

- There is `elem.matches(css)` to check if `elem` matches the given CSS selector.
- There is `elem.closest(css)` to look for the nearest ancestor that matches the given CSS-selector. The `elem` itself is also checked.

And let's mention one more method here to check for the child-parent relationship, as it's sometimes useful:

- `elemA.contains(elemB)` returns true if `elemB` is inside `elemA` (a descendant of `elemA`) or when `elemA==elemB`.

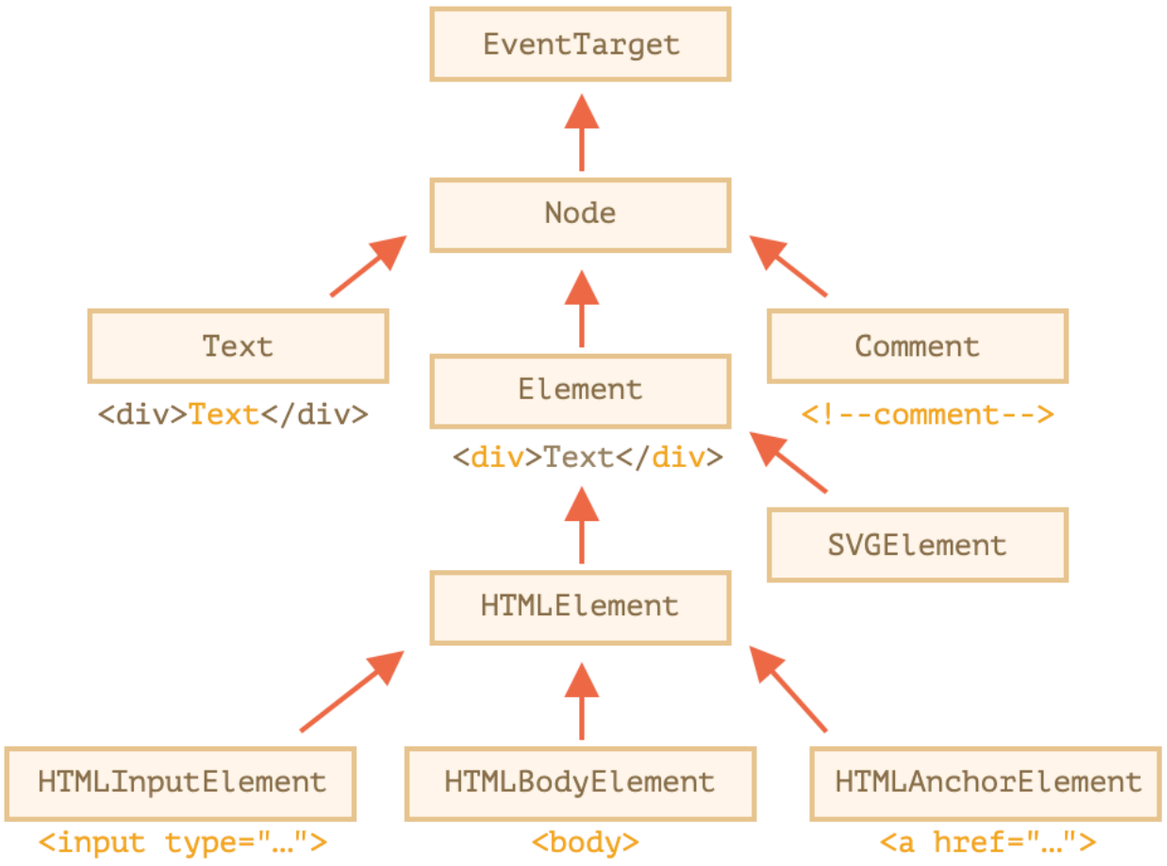
Node properties

Different DOM nodes may have different properties. For instance, an element node corresponding to tag `<a>` has link-related properties, and the one corresponding to `<input>` has input-related properties and so on. Text nodes are not the same as element nodes. But there are also common properties and methods between all of them, because all classes of DOM nodes form a single hierarchy.

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.

Here's the picture, explanations to follow:



Node properties

The classes are:

- EventTarget – is the root "abstract" class. Objects of that class are never created. It serves as a base, so that all DOM nodes support so-called "events", we'll study them later.
- Node – is also an "abstract" class, serving as a base for DOM nodes. It provides the core tree functionality: `parentNode`, `nextSibling`, `childNodes` and so on (they are getters). Objects of `Node` class are never created. But there are concrete node classes that inherit from it, namely: `Text` for text nodes, `Element` for element nodes and more exotic ones like `Comment` for comment nodes.
- Element – is a base class for DOM elements. It provides element-level navigation like `nextElementSibling`, `children` and searching methods like `getElementsByTagName`, `querySelector`. A browser supports not only HTML, but also XML and SVG. The `Element` class serves as a base for more specific classes: `SVGElement`, `XMLElement` and `HTMLElement`.
- HTMLElement – is finally the basic class for all HTML elements. It is inherited by concrete HTML elements:
  - HTMLInputElement – the class for `<input>` elements,
  - HTMLBodyElement – the class for `<body>` elements,
  - HTMLAnchorElement – the class for `<a>` elements,
  - ...and so on, each tag has its own class that may provide specific properties and methods.

So, the full set of properties and methods of a given node comes as the result of the inheritance.

The NodeType prop

The `nodeType` property provides one more, "old-fashioned" way to get the "type" of a DOM node.

It has a numeric value:

- `elem.nodeType == 1` for element nodes,
- `elem.nodeType == 3` for text nodes,
- `elem.nodeType == 9` for the document object,
- there are few other values in the specification.

For instance:

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

  // let's examine what it is?
  alert(elem.nodeType); // 1 => element

  // and the first child is...
  alert(elem.firstChild.nodeType); // 3 => text

  // for the document object, the type is 9
  alert( document.nodeType ); // 9
  </script>
</body>
```

In modern scripts, we can use `instanceof` and other class-based tests to see the node type, but sometimes `nodeType` may be simpler. We can only read `nodeType`, not change it.

Tag: nodeName, tagName

Given a DOM node, we can read its tag name from `nodeName` or `tagName` properties:

For instance:

```
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

Is there any difference between `tagName` and `nodeName`?

Sure, the difference is reflected in their names, but is indeed a bit subtle.

- The `tagName` property exists only for `Element` nodes.
- The `nodeName` is defined for any `Node`:
  - for elements it means the same as `tagName`.
  - for other node types (text, comment, etc.) it has a string with the node type.

In other words, `tagName` is only supported by element nodes (as it originates from `Element` class), while `nodeName` can say something about other node types.

innerHTML: the contents

The innerHTML property allows to get the HTML inside the element as a string.

We can also modify it. So it's one of the 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>
```

innerHTML += does overwrite

We can append HTML to an element by using `elem.innerHTML+="more html"`.

Like this:

```
chatDiv.innerHTML += "<div>Hello<img src='smile.gif'/> !</div>";
chatDiv.innerHTML += "How goes?";
```

But we should be very careful about doing it, because what's going on is *not* an addition, but a full overwrite.

Technically, these two lines do the same:

```
elem.innerHTML += "...";
// is a shorter way to write:
elem.innerHTML = elem.innerHTML + "..."
```

In other words, `innerHTML+=` does this:

1. The old contents is removed.
2. The new `innerHTML` is written instead (a concatenation of the old and the new one).

DOM Properties

We've already seen built-in DOM properties. There are a lot. But technically no one limits us, and if there aren't enough, we can add our own.

DOM nodes are regular JavaScript objects. We can alter them.

For instance, let's create a new property in `document.body`:

```
document.body.myData = {
  name: 'Caesar',
  title: 'Imperator'
};

alert(document.body.myData.title); // Imperator
```

We can add a method as well:

```
document.body.sayTagName = function() {
  alert(this.tagName);
};

document.body.sayTagName(); // BODY (the value of "this" in the method is docu
```

HTML Attributes

In HTML, tags may have attributes. When the browser parses the HTML to create 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.

For instance:

```
<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>
```

Please note that a standard attribute for one element can be unknown for another one. For instance, `"type"` is standard for `<input>` (HTMLInputElement), but not for `<body>` (HTMLBodyElement). Standard attributes are described in the specification for the corresponding element class.

HTML Attributes

So, if an attribute is non-standard, there won't be a DOM-property for it. Is there a way to access such attributes?

Sure. All attributes are accessible by using the 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.

These methods operate exactly with what's written in HTML.

Also one can read all attributes using `elem.attributes`: a collection of objects that belong to a built-in Attr class, with `name` and `value` properties.

Here's a demo of reading a non-standard property:

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

HTML attributes have the following features:

- Their name is case-insensitive (`id` is same as `ID`).
- Their values are always strings.

Non-standard attributes

When writing HTML, we use a lot of standard attributes. But what about non-standard, custom ones? First, let's see whether they are useful or not? What for?

Sometimes non-standard attributes are used to pass custom data from HTML to JavaScript, or to "mark" HTML-elements for JavaScript.

Like this:

```
<!-- mark the div to show "name" here -->
<div show-info="name"></div>
<!-- and age here -->
<div show-info="age"></div>

<script>
  // the code finds an element with the mark and shows what's requested
  let user = {
    name: "Pete",
    age: 25
  };

  for(let div of document.querySelectorAll('[show-info]')) {
    // insert the corresponding info into the field
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // first Pete into "name", then 25 into "age"
  }
</script>
```

Non-standard attributes

Also they can be used to style an element.

For instance, here for the order state the attribute `order-state` is used:

```
<style>
  /* styles rely on the custom attribute "order-state" */
  .order[order-state="new"] {
    color: green;
  }

  .order[order-state="pending"] {
    color: blue;
  }

  .order[order-state="canceled"] {
    color: red;
  }
</style>

<div class="order" order-state="new">
  A new order.
</div>

<div class="order" order-state="pending">
  A pending order.
</div>

<div class="order" order-state="canceled">
  A canceled order.
</div>
```

Non-standard attributes

Why would using an attribute be preferable to having classes like `.order-state-new`, `.order-state-pending`, `order-state-canceled`?

Because an attribute is more convenient to manage. The state can be changed as easy as:

```
// a bit simpler than removing old/adding a new class
div.setAttribute('order-state', 'canceled');
```

But there may be 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? The HTML language is alive, it grows, and more attributes appear to suit the needs of developers. There may be unexpected effects in such case.

To avoid conflicts, there exist data-* attributes.

All attributes starting with "data-" are reserved for programmers' use. They are available in the `dataset` property.

For instance, if an `elem` has an attribute named `"data-about"`, it's available as `elem.dataset.about`.

Like this:

```
<body data-about="Elephants">
<script>
  alert(document.body.dataset.about); // Elephants
</script>
```

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

Summary

- Attributes – is what's written in HTML.
- Properties – is what's in DOM objects.

A small comparison:

	Properties	**Attributes**
Type	Any value, standard properties have types described in the spec	A string
Name	Name is case-sensitive	Name is not case-sensitive

Methods to work with attributes are:

- `elem.hasAttribute(name)` – to check for existence.
- `elem.getAttribute(name)` – to get the value.
- `elem.setAttribute(name, value)` – to set the value.
- `elem.removeAttribute(name)` – to remove the attribute.
- `elem.attributes` is a collection of all attributes.

For most situations using DOM properties is preferable. We should refer to attributes only when DOM properties do not suit us, when we need exactly attributes, for instance:

- We need a non-standard attribute. But if it starts with `data-`, then we should use `dataset`.
- We want to read the value "as written" in HTML. The value of the DOM property may be different, for instance the `href` property is always a full URL, and we may want to get the "original" value.

Modifying Document

- Methods to create new nodes:
  - `document.createElement(tag)` – creates an element with the given tag,
  - `document.createTextNode(value)` – creates a text node (rarely used),
  - `elem.cloneNode(deep)` – clones the element, if `deep==true` then with all descendants.
- Insertion and removal:
  - `node.append(...nodes or strings)` – insert into `node`, at the end,
  - `node.prepend(...nodes or strings)` – insert into `node`, at the beginning,
  - `node.before(...nodes or strings)` –- insert right before `node`,
  - `node.after(...nodes or strings)` –- insert right after `node`,
  - `node.replaceWith(...nodes or strings)` –- replace `node`.
  - `node.remove()` –- remove the `node`.

Text strings are inserted “as text”.

Modifying Document

- There are also “old school” methods:
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

  All these methods return `node`.

- Given some HTML in `html`, `elem.insertAdjacentHTML(where, html)` inserts it depending on the value of `where`:
  - `"beforebegin"` – insert `html` right before `elem`,
  - `"afterbegin"` – insert `html` into `elem`, at the beginning,
  - `"beforeend"` – insert `html` into `elem`, at the end,
  - `"afterend"` – insert `html` right after `elem`.

  Also there are similar methods, `elem.insertAdjacentText` and `elem.insertAdjacentElement`, that insert text strings and elements, but they are rarely used.

- To append HTML to the page before it has finished loading:
  - `document.write(html)`

  After the page is loaded such a call erases the document. Mostly seen in old scripts.

Learning Resources

1. **Browser Env** https://javascript.info/browser-environment
2. **DOM Tree:** https://javascript.info/dom-nodes
3. **Walking DOM:** https://javascript.info/dom-navigation
4. **Searching in DOM:** https://javascript.info/searching-elements-dom
5. **Basic Node Props:** https://javascript.info/basic-dom-node-properties
6. **DOM Attributes and props:** https://javascript.info/dom-attributes-and-properties
7. **Modifying DOM:** https://javascript.info/modifying-document

Home Work

Homework

1. Try to Access document title and change it to: "Today we manipulate with DOM"

2. Ask user question id="question-frontend" (suggestion: use confirm()), and then insert answer in id="answer-frontend" Use if/else to set value as YES or NO.

3. Get a product from user(promt), and then insert it to .shopping-list.

4. Remove item(li) with text "Bread"

5. Promt item, then promt price, then calculate price and set it to, id="your-total-price".

6. Create and array of product objects ex: [{ name: "Carrot", price: 2.50 }, ...] Then Ask user from promt to write item name, and then ask it to provide quantity for this item, and then calculate the total (item quantity * price) and insert it in id="your-total-price". If item is not found, then show message "Sorry the item you are searching is not found".