Document Object Model

October 2022

Agenda

1 INTRO

2 HISTORY – WHY DO WE NEED TO MANIPULATE THE DOM?

3 DOCUMENT OBJECT MODEL

4 ACCESSING THE DOM

5 MODIFYING THE DOM STRUCTURE

Document Object Model (DOM)

The Document Object Model (DOM) is a representation — a model — of a document and its content.

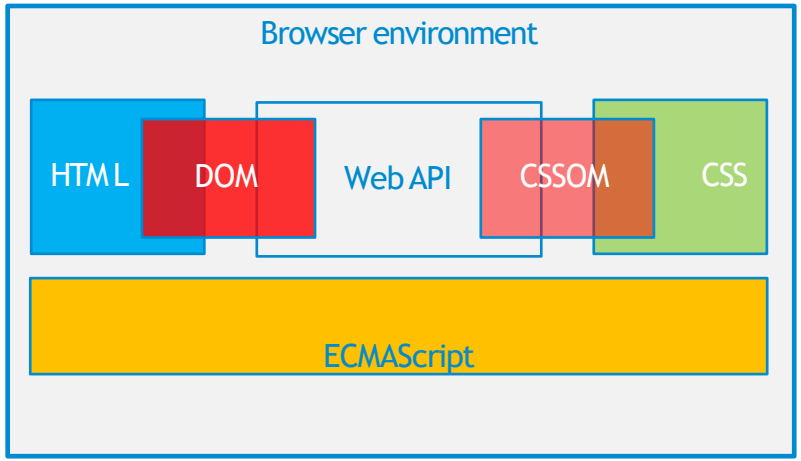
It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects.

DOM is a web standard.

But why do we need to access the HTML markup in the first place? To understand that, we need a bit of a history lesson...

It is 2021 - you probably won't work with the Document Object Model directly, yet it is something you need to know.

Also, we have a CSS Object Model, for manipulating CSS from JavaScript. We deliberately won't touch this, as you should not touch this in production code, too.

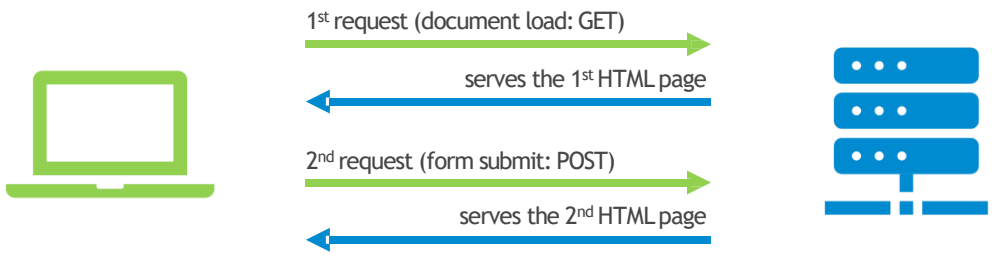


Static HTML pages

Traditionally, webservers served static HTML pages

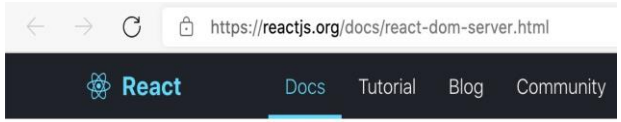
For a page, such as Wikipedia, this makes sense. The main interaction from user side is to read and to navigate between the pages via links.

Every interaction results in a complete, new HTML document.



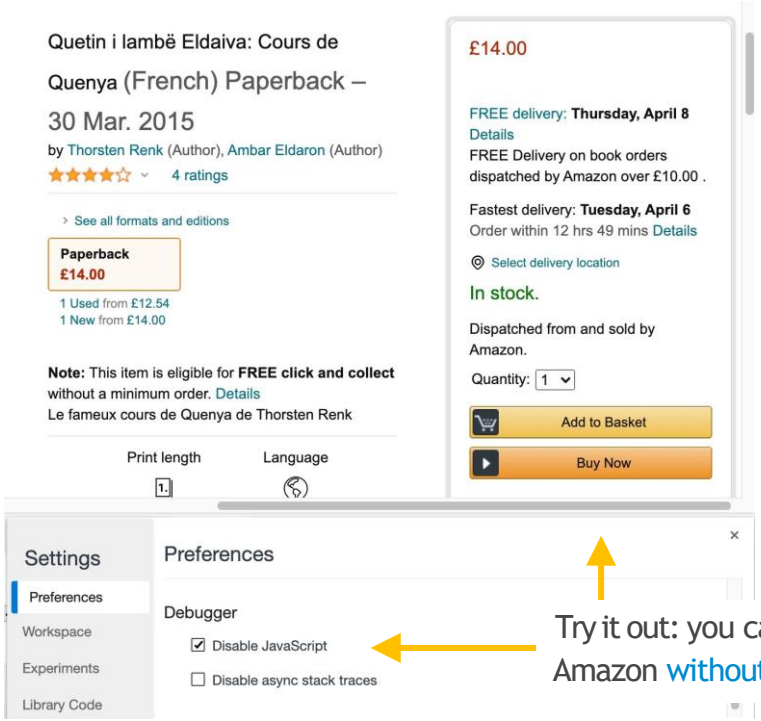
Submitting a form also possible, simply with using the <form> html element: clicking on the submit button will result in a completely new HTML document, too.

These documents are simple HTML text files on the server. Even the folder structure on the server is reflected in the document's URL: docs could be a real folder in the document root.



sure, there could be magic as well...

The life without JavaScript



Try it out: you can order from Amazon without JavaScript.

To this point, there is no need for JavaScript at all.

For many sites, it is still requested that the main functionality must work without JS.

Without JavaScript, the only way to modify *anything** on the page is to reload the page entirely with all its assets.

And it costs a lot: the server bandwidth is expensive, and while you can cache / deliver the static assets (images) with Akamai / Cloudflare, you still need those for that, etc.

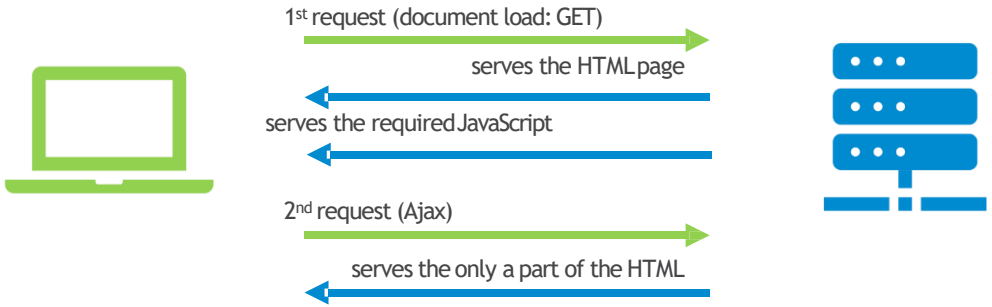
** well, we have iframes as well, if you asked*

Dynamic HTML and Ajax

With JavaScript, we can replace / reload any parts of the document

Changing the HTML markup in JavaScript is done by manipulating on the DOM (Document Object Model).

Basically, the DOM is the HTML representation in JavaScript. And with the DOM we can do anything with the markup. The architecture, however, now changes a bit:



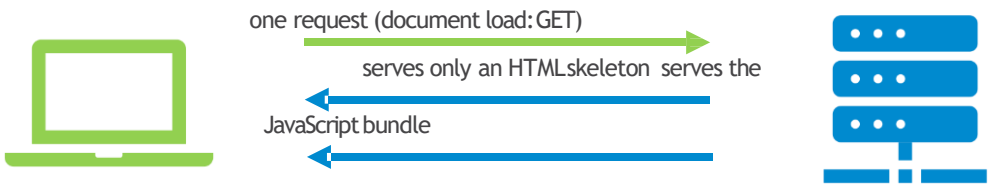
Now we need JavaScript for adding the new / changed content to the page.

Single Page Applications

Modern SPAs (React and Angular, mainly) promise a scalable and simple solution for web development.

With these, however, the architecture has completely changed:

Of course, there are many more requests, however, for the document itself, it is only a single (hence the name) - and even that HTML is merely a skeleton, the real HTML will be built in the browser. Every page.



The HTML markup will be assembled on the client side with JavaScript.

DOM: full access to the document

With the DOM, we can change anything in the entire tree

But is it a problem? - I hear you ask.

Let's say, that the branches of this tree is a component (or a function), and the leaves are variables. Now, just think about, in this system, you can access all of the variables in all of the components from any component.

Everything is globally accessible in this system.

Also, let me show you a real-world example of a DOM tree …

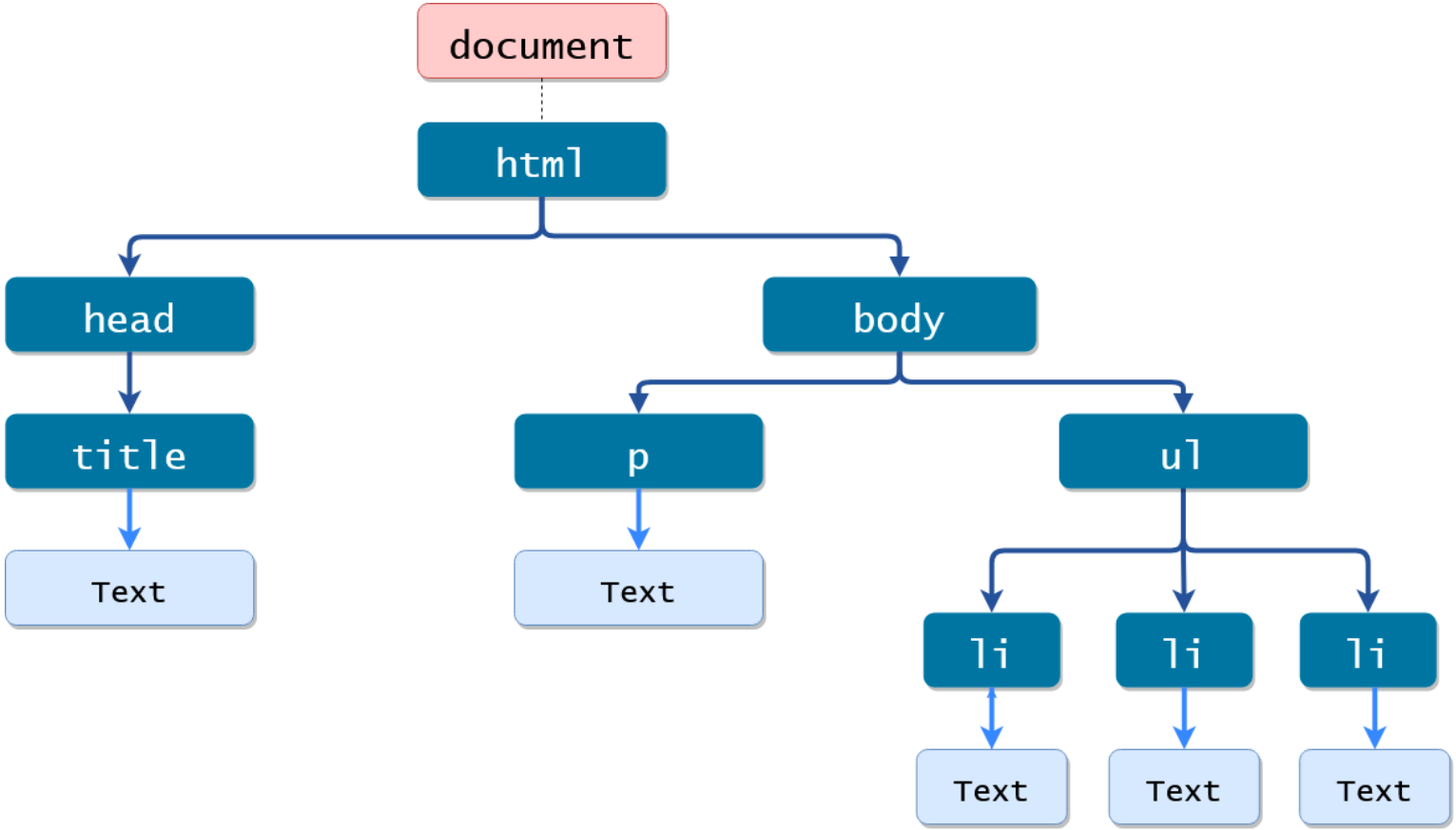
DOM is a tree.

Markup to test (permalink, save, upload, download, hide):

```
<html>
<head>
   <title>With Treebeard and the Ents</title>
</head>
<body>
  <em>Legolas</em>
  <p>Then are we not to see the merry young hobbits again?</p>
  <em>Gandalf</em>
  <p>Who knows? Have patience. Go where you must go, and hope!</p>
</body>
</html>
```

DOM view (hide, refresh):

```
└HTML
  ├HEAD
  │ ├#text:
  │ ├TITLE
  │ │ └#text: With Treebeard and the Ents
  │ └#text:
  ├#text:
  └BODY
    ├#text:
    ├EM
    │ └#text: Legolas
    ├#text:
    ├P
    │ └#text: Then are we not to see the merry young hobbits again?
    ├#text:
    ├EM
    │ └#text: Gandalf
    ├#text:
    ├P
    │ └#text: Who knows? Have patience. Go where you must go, and hope!
    └#text:
```


document
html
head
body
title
p
ul
Text
Text
li
li
li
Text
Text
Text

DOCUMENT OBJECT MODEL

DOM tree

When a web page is loaded, the browser creates the Document Object Model of the page: a tree

In this tree everything is a node, and every node is an object. The DOM can represent HTML or XML documents.

```
…<html> == $0
  ▼<head>
     <title>With Treebeard and the Ents</title>
   </head>
  ▼<body>
     <em class="elf" name="legolas">Legolas</em>
     <p id="question">Then are we not to see the
              merry young hobbits again?</p>
     <em class="maia" name="gandalf">Gandalf</em>
     <p id="answer">Who knows? Have patience.
              Go where you must go, and hope!</p>
   </body>
 </html>
```

```
└HTML
  ├HEAD
  │ ├#text:
  │ ├TITLE
  │ │ └#text: With Treebeard and the Ents
  │ └#text:
  ├#text:
  └BODY
    ├#text:
    ├EM class="elf" name="legolas"
    │ └#text: Legolas
    ├#text:
    ├P id="question"
    │ └#text: Then are we not to see the merry young hobbits again?
    ├#text:
    ├EM class="maia" name="gandalf"
    │ └#text: Gandalf
    ├#text:
    ├P id="answer"
    │ └#text: Who knows? Have patience. Go where you must go, and hope!
    └#text:
```

DOM - elements

Let's break down the DOM!

There are different types of nodes, and the methods return different types of collections.

```
> document
<- ▼#document
      <html>
       ▶<head>…</head>
       ▼<body>
          <em class="elf" name="legolas">Legolas</em>
          <p id="question">Then are we not to see the
                   merry young hobbits again?</p>
          <em class="maia" name="gandalf">Gandalf</em>
          <p id="answer">Who knows? Have patience.
                   Go where you must go, and hope!</p>
        </body>
      </html>
```

document is HTMLDocument

```
> document.toString();
<- "[object HTMLDocument]"
```

```
> let emNodes = document.getElementsByTagName("em");
<- undefined
```

nodeList is HTMLCollection

```
> emNodes.toString();
<- "[object HTMLCollection]"
> emNodes.length;
<- 2
```

node is HTMLElement

```
> emNodes[0].toString();
<- "[object HTMLElement]"
> emNodes[0];
<-   <em class="elf" name="legolas">Legolas</em>
```

DOM - node types

the document node

Document

Element

elements - in HTML: , <p>, ...

the textual content of an element:
Gimli

Text

Comment

comments:
<!– a troll is still stronger, though -->

attributes also nodes:
<gandalf class="maia">Mithrandir</gandalf>

Attr

ShadowRoot

using Web Components, parts of the DOM can be separated*

<![CDATA[In HTML only used foreign content: MathML/SVG]]>

CDATASection

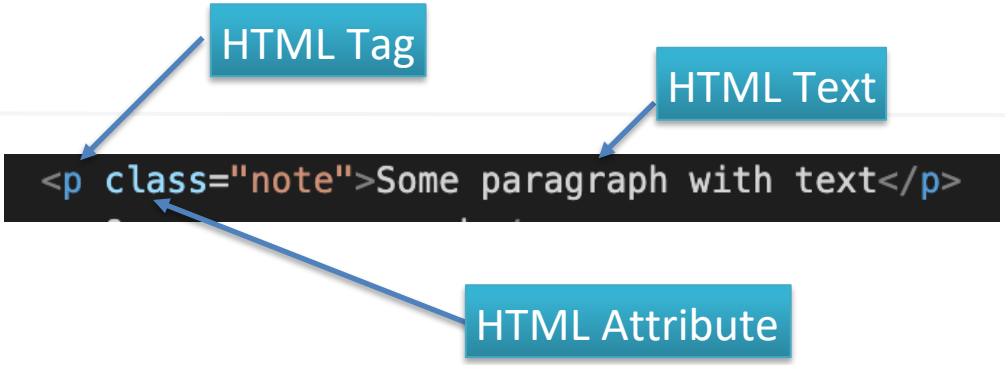
ProcessingInstruction

<?xml-stylesheet type="text/xsl" href="style.xsl"?>

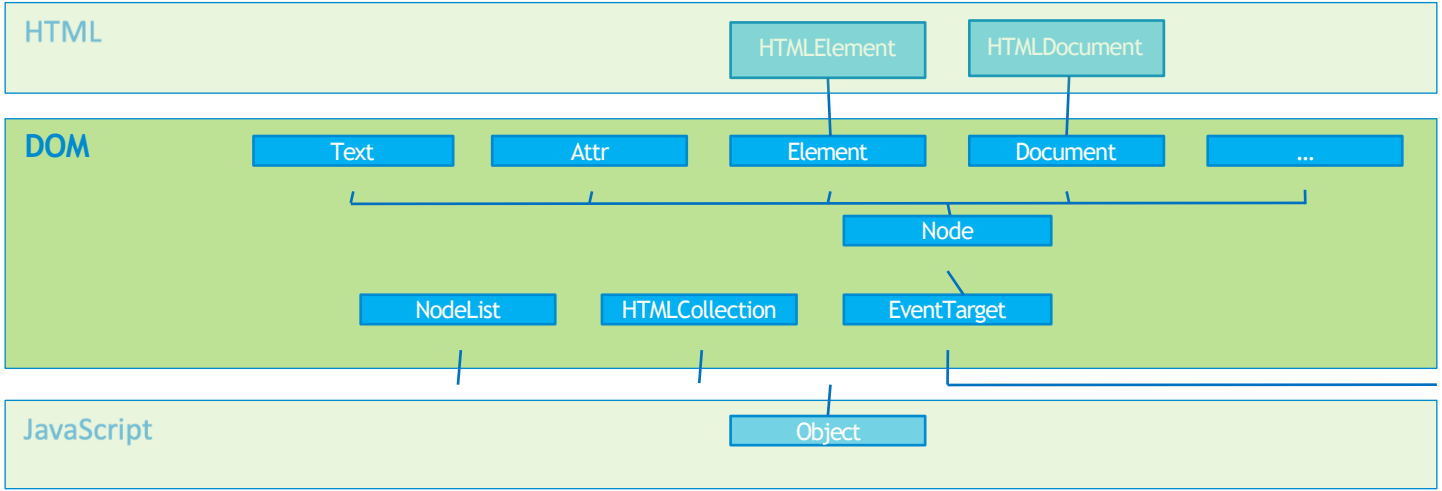
DocumentFragment

* it solves a lot of problems...

DOM - objecthierarchy



Without getting too deep into the topic, the object hierarchy looks like this:



ACCESSING THE DOM

Accessing the DOM

Method	Description	on the document node	on an element node
.getElementById(DOMString elementId)	*Returns the first element within node's descendants whose ID is elementId.*	✓	
.getElementsByTagName(qualifiedName)	*Returns a HTMLCollection of all descendant elements whose qualified name is qualifiedName. Case-insensitive.*	✓ ✅	✓ ✅
.getElementsByClassName(classNames)	*Returns a HTMLCollection of the elements in the object on which the method was invoked (a document or an element) that have all the classes given by classNames.*	✓	✓
.getElementsByName(name)	*Returns a NodeList of elements in the Document that have a name attribute with the value name.*	✓	
.querySelector(selector)	*Returns the first Element within the document that matches the specified selector, or group of selectors. If no matches are found, null is returned.*	✓	✓
.querySelectorAll(selector)	*Returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors.*	✓	✓

Accessors on a sub tree

.getElementsByTagName and .getElementsByClassName can be used on an element node, too.

```
> document
  ▼#document
      <html>
        ▶<head>…</head>
        ▼<body>
          ▶<div>…</div>
          ▼<div>
              <em class="maia" name="gandalf">Gandalf</em>
              <p id="answer">Who knows? Have patience.
                             Go where you must go, and hope!</p>
            </div>
          </body>
      </html>
```

can be used on document

```
> document.getElementsByTagName("p");
  ▼HTMLCollection(2) [p#question, p#answer, question: p#question, answer: p#answer]
    ▶0: p#question
    ▶1: p#answer
     length: 2
    ▶answer: p#answer
    ▶question: p#question
    ▶__proto__: HTMLCollection
```

and on an element node as well

```
> document.getElementsByTagName("div")[0].getElementsByTagName("p");
  ▼HTMLCollection [p#question, question: p#question]
    ▶0: p#question
     length: 1
    ▶question: p#question
    ▶__proto__: HTMLCollection
```

but .getElementById is not

```
> document.getElementsByTagName("div")[0].getElementById("answer");
  ▶Uncaught TypeError: document.getElementsByTagName(...)[0].getElementById is not a function
     at <anonymous>:1:41
```

Accessors by id, tag, class, and name attribute

by id
(returns a single node)

```
> document.getElementById("question");
<-   <p id="question">Then are we not to see the
                merry young hobbits again?</p>
```

by tag name
(returns an HTMLCollection)

```
> document.getElementsByTagName("p");
<- ▼HTMLCollection(2) [p#question, p#answer, question:
    ▶0: p#question
    ▶1: p#answer
     length: 2
    ▶answer: p#answer
    ▶question: p#question
    ▶__proto__: HTMLCollection
```

by class name
(returns an HTMLCollection)

```
> document.getElementsByClassName("elf");
<- ▼HTMLCollection [em.elf, legolas: em.elf] i
    ▶0: em.elf
     length: 1
    ▶legolas: em.elf
    ▶__proto__: HTMLCollection
```

by the name attribute
(returns a NodeList)

```
> document.getElementsByName("legolas");
<- ▼NodeList [em.elf] i
    ▶0: em.elf
     length: 1
    ▶__proto__: NodeList
```

Accessors by CSS selector

With the query selectors, a CSS selector can be used, too.

```
> document
  ▼#document
     <html>
      ▶<head>…</head>
      ▼<body>
        ▼<div>
           <em class="elf" name="legolas">Legolas</em>
           <p id="question">Then are we not to see the
                     merry young hobbits again?</p>
         </div>
        ▶<div>…</div>
        </body>
     </html>
> document.querySelector("body div");
  ▼<div>
     <em class="elf" name="legolas">Legolas</em>
     <p id="question">Then are we not to see the
               merry young hobbits again?</p>
   </div>
> document.querySelectorAll("body div");
  ▶NodeList(2) [div, div]
```

Modifying text content

we can get and set the text content of a node

it also appears on the document, itself

```
> let question = document.getElementById("question");
  undefined
> question;
  <p id="question">Then are we not to see the
                 merry young hobbits again?</p>
> question.innerText = question.innerText + " - said Legolas.";
  "Then are we not to see the merry young hobbits again? - said Legolas."
> question
  <p id="question">Then are we not to see the merry young hobbits again? - said Legolas.</p>
> document
  ▼#document
     <html>
      ▶<head>…</head>
      ▼<body>
        ▼<div>
           <em class="elf" name="legolas">Legolas</em>
           <p id="question">Then are we not to see the merry young hobbits again? - said Legolas.</p>
         </div>
        ▶<div>…</div>
        </body>
     </html>
```

.innerText vs .textContent

We also have textContent, but these are significantly different:

textContent gets the content of all elements, including <script> and <style> elements. In contrast, innerText only shows "human-readable" elements.

textContent returns every element in the node. In contrast, innerText is aware of styling and won't return the text of "hidden" elements.

Moreover, since innerText takes CSS styles into account, reading the value of innerText triggers a reflow to ensure up-to-date computed styles. (Reflows can be computationally expensive, and thus should be avoided when possible.)

```
> question.textContent;
<· "Then are we not to see the merry young hobbits again? - said Legolas."
> document.body.innerText;
<· "Legolas

   Then are we not to see the merry young hobbits again? - said Legolas.

   Gandalf

   Who knows? Have patience. Go where you must go, and hope!"
> document.body.textContent;
<· "

       Legolas
       Then are we not to see the merry young hobbits again? - said Legolas.


       Gandalf
       Who knows? Have patience.
              Go where you must go, and hope!

   "
```

Attributes

We have different methods to get, set, check and for removing attributes.

Legolas

Then are we not to see the merry young hobbits again?

Gandalf

Who knows? Have patience. Go where you must go, and hope!

Welcome | Elements | Memory | Console | Sources | Network

top | Filter | Default levels

```
> let question = document.getElementById("question");
<· undefined
> question.getAttribute("id");
<· "question"
> question.hasAttribute("style");
<· false
> question.setAttribute("style", "color: blue");
<· undefined
> question;
<·   <p id="question" style="color: blue">Then are we not to see the
          merry young hobbits again?</p>
> question.removeAttribute("id");
<· undefined
> question;
<·   <p style="color: blue">Then are we not to see the
          merry young hobbits again?</p>
```

it actually sets an inline style

id is also an attribute and can be removed

Attributes vs properties

Some standard attributes are mirrored as a property on the object

id, class, style, etc. However, the mappings are not trivial: class becomes className, style properties become camelCase.

Setting them as a node property will affect the attributes as well.

```
> document.body
  ▼<body>
     <em class="elf" name="legolas">Legolas</em>
     <p id="question">Then are we not to see the
              merry young hobbits again?</p>
     <em class="maia" name="gandalf">Gandalf</em>
     <p id="answer">Who knows? Have patience.
              Go where you must go, and hope!</p>
   </body>
> let answer = document.body.querySelector("em.maia[name=gandalf] + p#answer");
  undefined
> answer.hasAttribute("style");
  false
> answer.style;
  ▶ CSSStyleDeclaration {alignContent: "", alignItems: "", alignSelf: "", alignme
> answer.style.alignContent = "start";
  "start"
> answer
    <p id="answer" style="align-content: start;">Who knows? Have patience.
              Go where you must go, and hope!</p>
> answer.setAttribute("style", "align-content: center");
  undefined
> answer
    <p id="answer" style="align-content: center">Who knows? Have patience.
              Go where you must go, and hope!</p>
```

there is no style attribute

but there is a style property

setting the property…

… and the attribute, both work

.classList

classList provides a convenient way to modify the classes of an element

Working with classes could be a bit of tricky, because the class attribute is a space separated list
- having dedicated methods could help.

Please check the compatibility, though.

```
> let gandalf = document.querySelector("em.maia");
<· undefined
> gandalf.classList;
<· ▶ DOMTokenList ["maia", value: "maia"]
```

add →

```
> gandalf.classList.add("wizard");
<· undefined
```

remove →

```
> gandalf.classList.remove("maia");
<· undefined
> gandalf.classList;
<· ▶ DOMTokenList ["wizard", value: "wizard"]
```

replace →

```
> gandalf.classList.replace("wizard", "maia");
<· true
> gandalf.classList;
<· ▶ DOMTokenList ["maia", value: "maia"]
```

toggle - on →

```
> gandalf.classList.toggle("hidden");
<· true
> gandalf.classList;
<· ▶ DOMTokenList(2) ["maia", "hidden", value: "maia hidden"]
```

toggle - off →

```
> gandalf.classList.toggle("hidden");
<· false
> gandalf.classList;
<· ▶ DOMTokenList ["maia", value: "maia"]
```

MODIFYING THE DOM STRUCTURE

.innerHTML

With innerHTML, you can insert text as parsed HTML into the DOM...

...but you probably should not do that.

If you use innerHTML with an unknown content (from a CMS, typically), then you just implemented an XSS vulnerability.

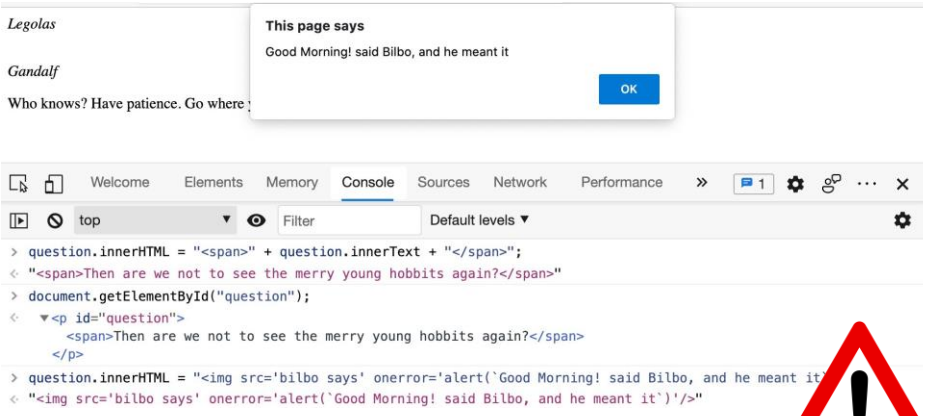
It means that an attacker can run any script on the page, stealing data, starting transactions, anything, really.

You cannot trust a content from a CMS.

Use innerText, simply.

XSS = Cross Site Scripting, when the browser runs injected code.

... in a way you really don't want



it works...

.innerHTML does not run script elements, but…

Don't be fooled by the perceived security

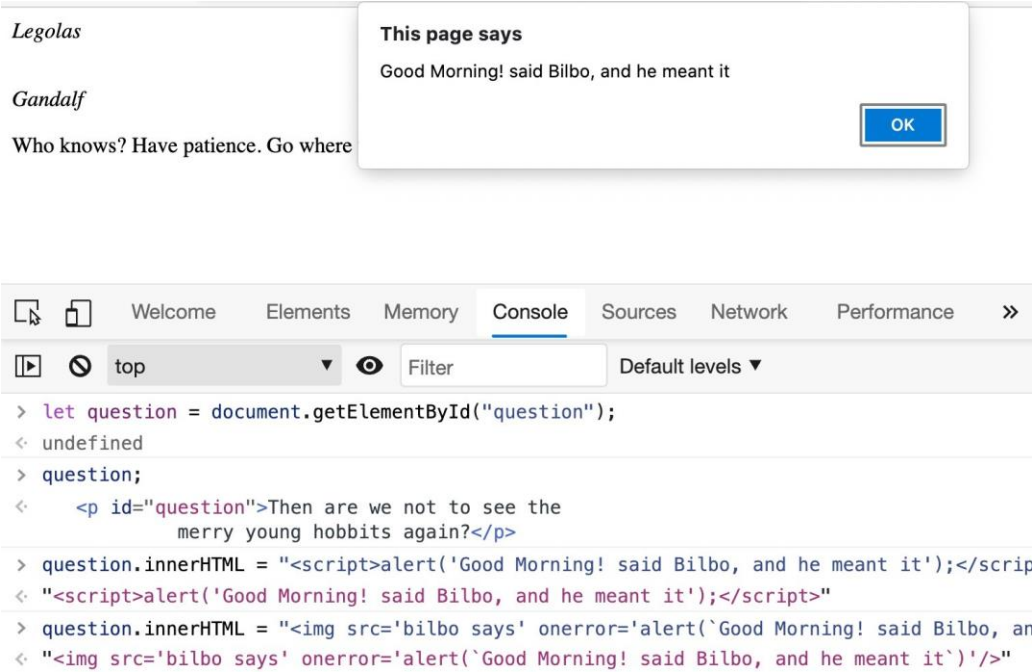
The innerHTML does not run the scripts directly, this works in a different way: when the browser tries to request the image from a wrong URL, it triggers an error.

And then the script will run.

Lesson learned: there will be always a tricky way to run a script from an injected HTML – just because you don't know how, it is still possible!

this does not work

but it does!



.createElement

New DOM nodes can be added

createElement will create a new element, but it won't add to the DOM.

we are not there yet

we have to append it to an element

```
> document.body;
  ▼<body>
      <em class="elf" name="legolas">Legolas</em>
      <p id="question">Then are we not to see the
              merry young hobbits again?</p>
      <em class="maia" name="gandalf">Gandalf</em>
      <p id="answer">Who knows? Have patience.
              Go where you must go, and hope!</p>
    </body>
> let bilbo = document.createElement("em");
  undefined
> document.body.getElementsByTagName("em");
  ▶HTMLCollection(2) [em.elf, em.maia, legolas: em.elf, gandalf: em.maia]
> bilbo.className = "hobbit";
  bilbo.setAttribute("name", "bilbo");
  bilbo.innerText = "Good Morning! said Bilbo, and he meant it.";
  "Good Morning! said Bilbo, and he meant it."
> document.body.appendChild(bilbo);
    <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>
> document.body
  ▼<body>
      <em class="elf" name="legolas">Legolas</em>
      <p id="question">Then are we not to see the
              merry young hobbits again?</p>
      <em class="maia" name="gandalf">Gandalf</em>
      <p id="answer">Who knows? Have patience.
              Go where you must go, and hope!</p>
      <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>
    </body>
```

A bit more complex example

web development is supergreen

```
> let narrator = document.createTextNode("The sun was shining, and the grass was
  very green.");
<· undefined
```

a text node is an object

```
> narrator.toString();
<· "[object Text]"
```

```
> bilbo;
<·   <em class="hobbit" name="bilbo">Good Morning! said Bilbo, and he meant it.</em>
```

inserting a new node with insertBefore

```
> bilbo.insertBefore(narrator, bilbo.firstChild);
<·   "The sun was shining, and the grass was very green."
```

success

```
> bilbo;
<· ▼<em class="hobbit" name="bilbo">
     "The sun was shining, and the grass was very green."
     "Good Morning! said Bilbo, and he meant it."
   </em>
```

wait! wrong order

```
> bilbo.childNodes;
<· ▶NodeList(2) [text, text]
```

1., we get the first child,

2., then we remove that,

3., but as a return value we have that, so we can insert it again

```
> bilbo.appendChild(
    bilbo.removeChild(
      bilbo.firstChild
    )
  );
<·   "The sun was shining, and the grass was very green."
```

```
> bilbo
<· ▼<em class="hobbit" name="bilbo">
     "Good Morning! said Bilbo, and he meant it."
     "The sun was shining, and the grass was very green."
   </em>
```

THANK YOU!