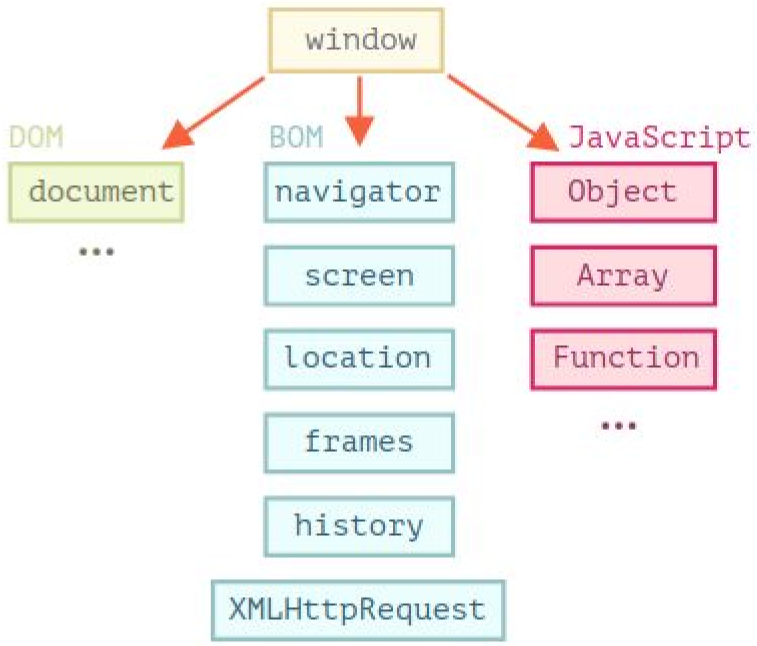
Browser environment, specs

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 its 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.

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
2. Second, it represents the "browser window" and provides methods to control it.

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
```

There are more window-specific methods and properties, we'll cover them later.

DOM (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 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.

BOM (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 URL
}
```

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:

```
document.body.style.background = 'red'; // make the background red
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.

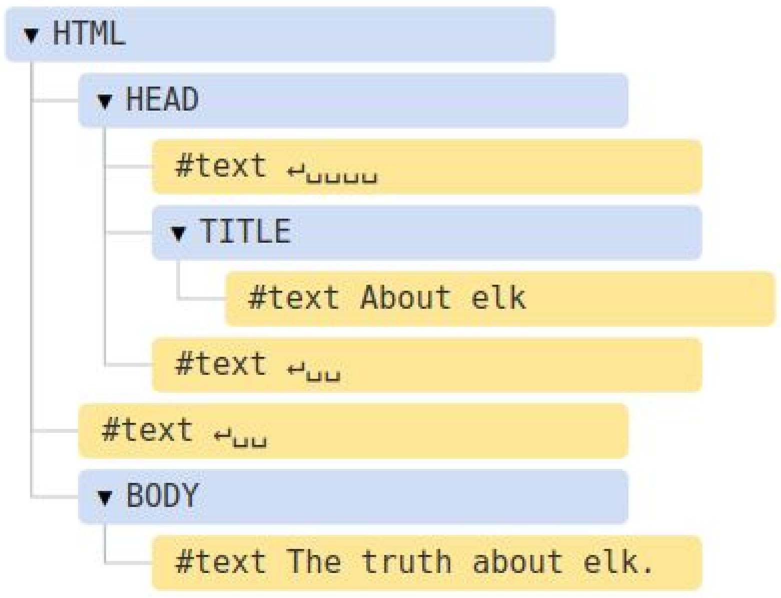
Soon we'll learn more ways to manipulate the DOM, but first we need to know about its structure.

An example of the DOM

Let's start with the following simple document:

```
<!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.

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>`.

In other cases everything's straightforward – if there are spaces (just like any character) in the document, then they become text nodes in the DOM, and if we remove them, then there won't be any.

Here are no space-only text nodes:

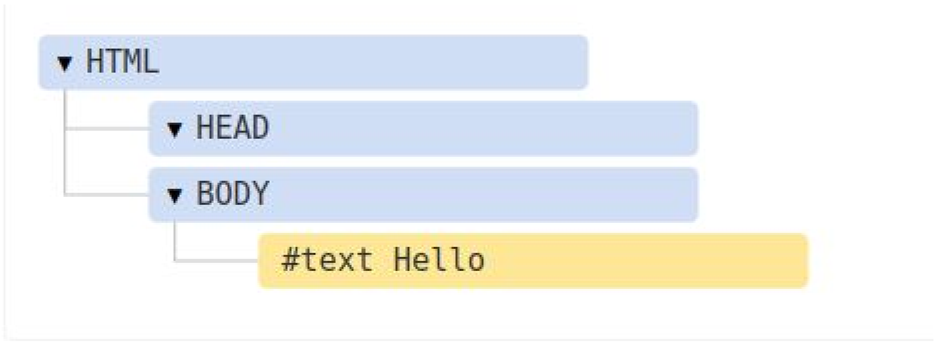
```
<!DOCTYPE HTML>
<html><head><title>About elk</title></head><body>The truth about elk.</body></html>
```

Autocorrection

If the browser encounters malformed HTML, it automatically corrects it when making the DOM.

For instance, the top tag is always `<html>`. Even if it doesn't exist in the document, it will exist in the DOM, because the browser will create it. The same goes for `<body>`.

As an example, if the HTML file is the single word `"Hello"`, the browser will wrap it into `<html>` and `<body>`, and add the required `<head>`, and the DOM will be:



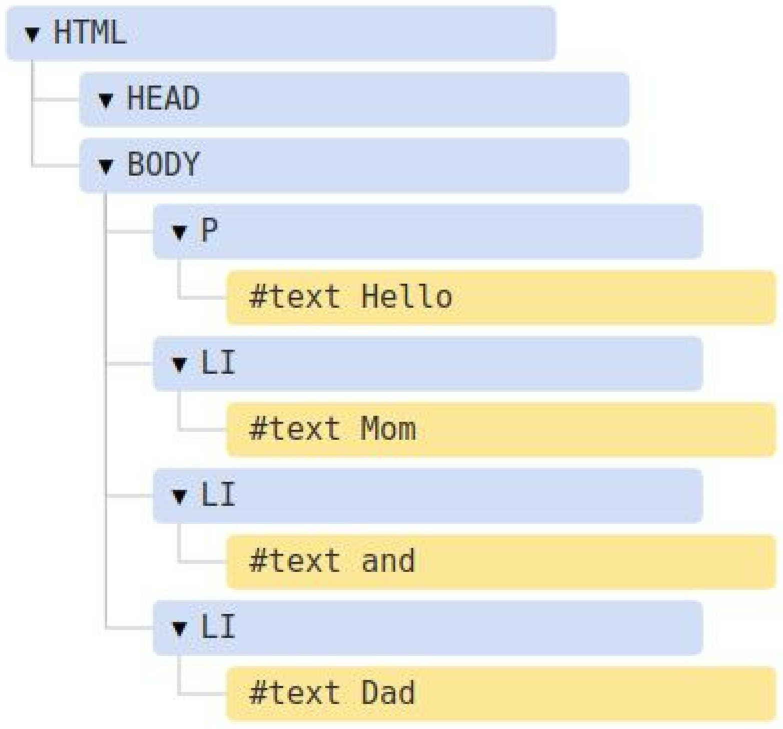
While generating the DOM, browsers automatically process errors in the document, close tags and so on.

A document with unclosed tags:

```
<p>Hello
<li>Mom
```

```
<li>and
<li>Dad
```

…will become a normal DOM as the browser reads tags and restores the missing parts:

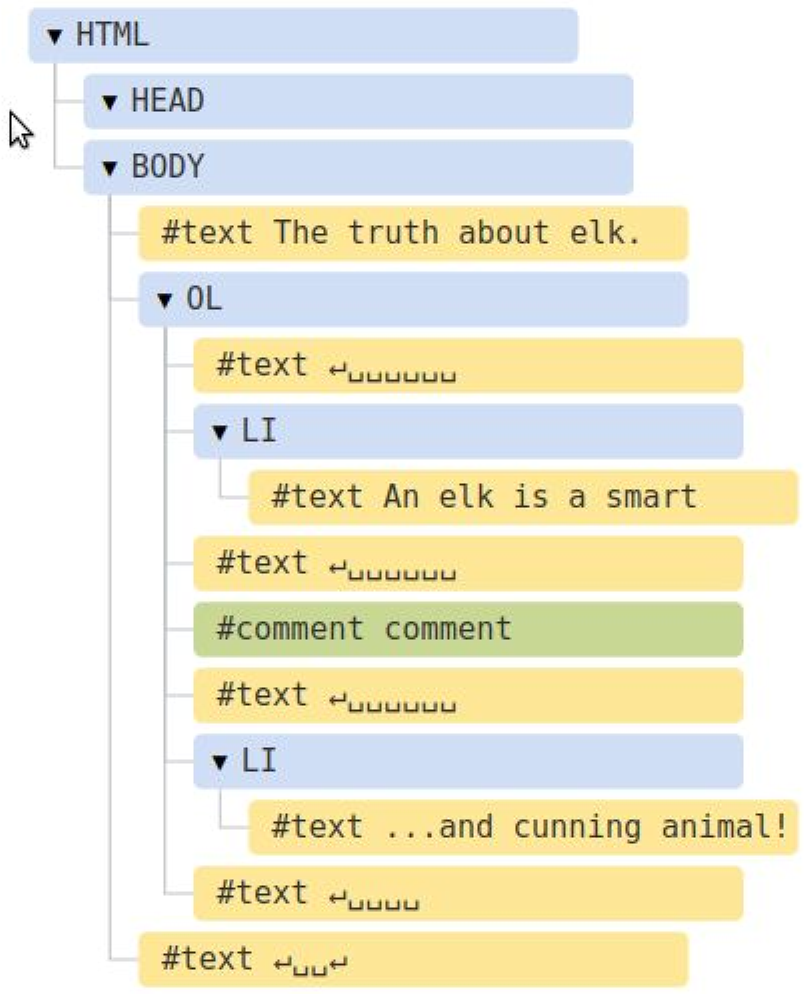


Other node types

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>
```



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.

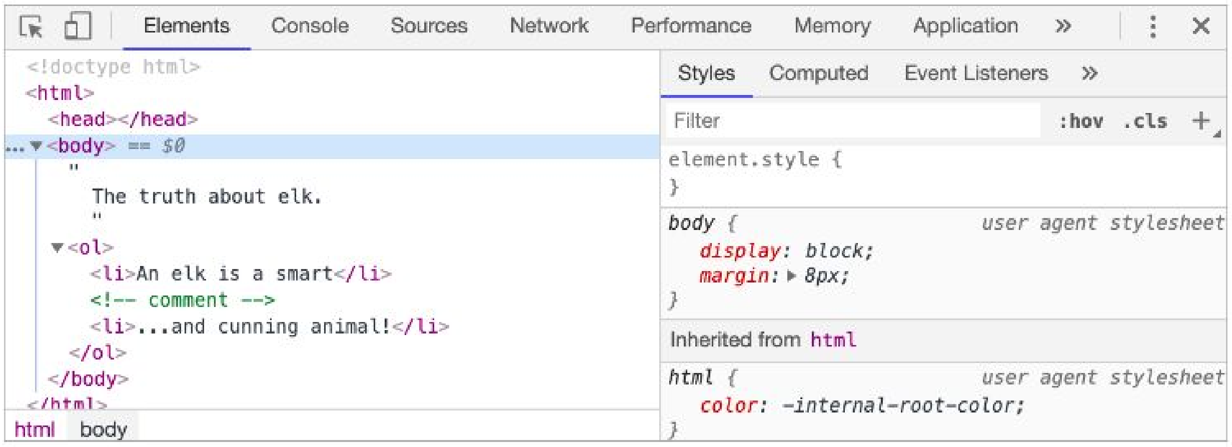
See it for yourself

To see the DOM structure in real-time, try Live DOM Viewer. Just type in the document, and it will show up as a DOM at an instant.

Another way to explore the DOM is to use the browser developer tools. Actually, that's what we use when developing.

To do so, open the web page elk.html, turn on the browser developer tools and switch to the Elements tab.

It should look like this:



```
<!doctype html>
<html>
  <head></head>
...▼<body> == $0
    "
      The truth about elk.
      "
   ▼<ol>
      <li>An elk is a smart</li>
      <!-- comment -->
      <li>...and cunning animal!</li>
    </ol>
  </body>
</html>
```

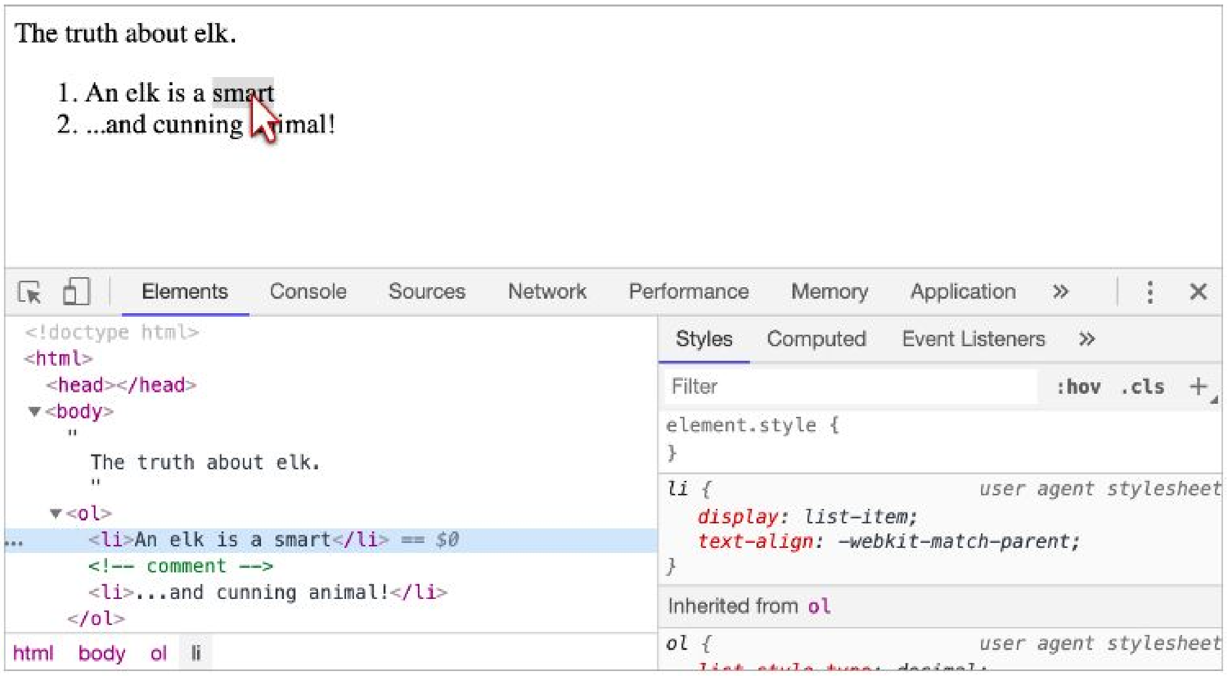
```
element.style {
}
body {                              user agent stylesheet
   display: block;
   margin: ▶ 8px;
}
html {                              user agent stylesheet
   color: -internal-root-color;
}
```

You can see the DOM, click on elements, see their details and so on.

Please note that the DOM structure in developer tools is simplified. Text nodes are shown just as text. And there are no "blank" (space only) text nodes at all. That's fine, because most of the time we are interested in element nodes.

Clicking the button in the left-upper corner allows us to choose a node from the webpage using a mouse (or other pointer devices) and "inspect" it (scroll to it in the Elements tab). This works great when we have a huge HTML page (and corresponding huge DOM) and would like to see the place of a particular element in it.

Another way to do it would be just right-clicking on a webpage and selecting "Inspect" in the context menu.


The truth about elk.

1. An elk is a smart
2. ...and cunning animal!

```
<!doctype html>
<html>
  <head></head>
  ▼<body>
     "
      The truth about elk.
     "
    ▼<ol>
       <li>An elk is a smart</li> == $0
       <!-- comment -->
       <li>...and cunning animal!</li>
     </ol>
```

```
element.style {
}
li {                          user agent stylesheet
   display: list-item;
   text-align: -webkit-match-parent;
}
Inherited from ol
ol {                          user agent stylesheet
```

At the right part of the tools there are the following subtabs:

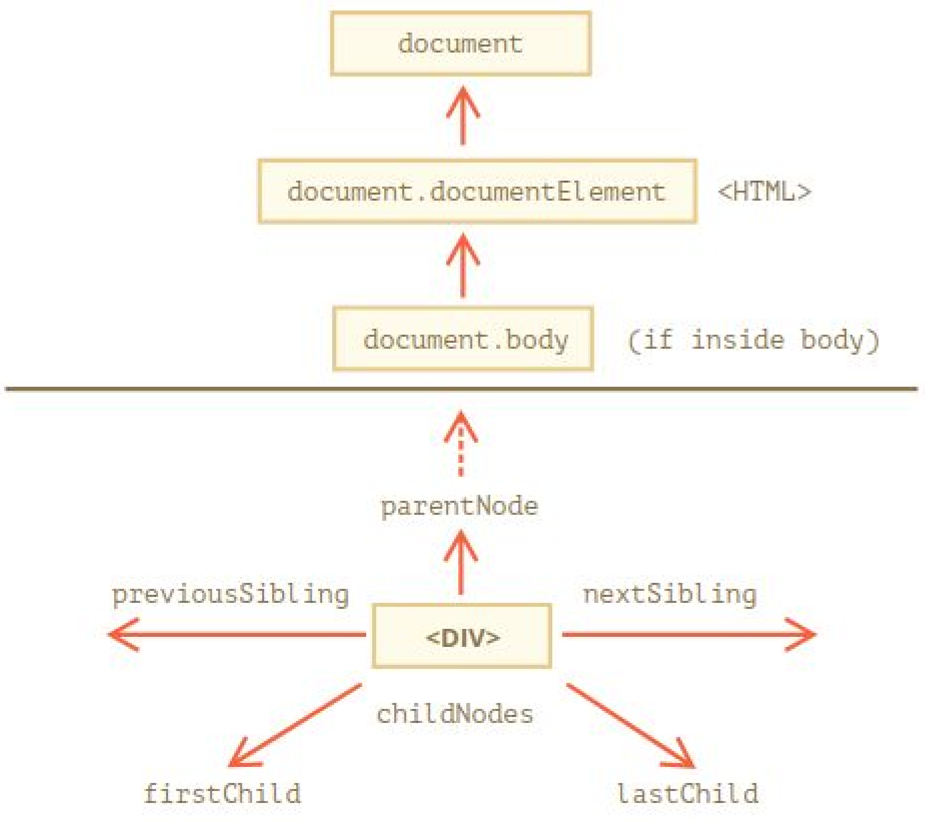
- **Styles** – we can see CSS applied to the current element rule by rule, including built-in rules (gray). Almost everything can be edited in-place, including the dimensions/margins/paddings of the box below.
- **Computed** – to see CSS applied to the element by property: for each property we can see a rule that gives it (including CSS inheritance and such).
- **Event Listeners** – to see event listeners attached to DOM elements (we'll cover them in the next part of the tutorial).
- …and so on.

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.

There’s a catch: document.body can be null

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.
So, in the example below the first alert shows null:

```
<html>
    <head>
        <script>  alert( "From HEAD: " + document.body ); // null, there's no <body> yet </script>
    </head>
    <body>
        <script>
            alert( "From BODY: " + document.body ); // HTMLBodyElement, now it exists
        </script>
    </body>
</html>
```

In the DOM world null means "doesn't exist"

In the DOM, the null value means "doesn't exist" or "no such node".

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.

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, ..., SCRIPT
            }
        </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.

They are just shorthands. 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

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 method!)
```

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
```

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 with 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:

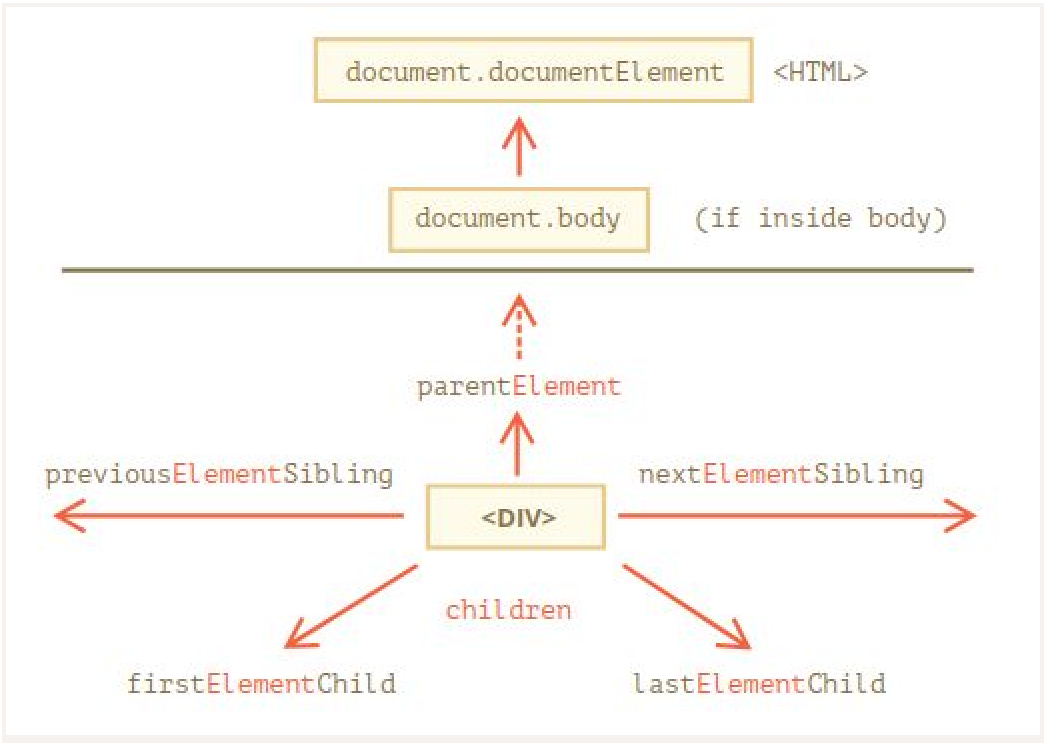
```
// parent of <body> is <html>
alert( document.body.parentNode === document.documentElement ); // true

// 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.

Why parentElement? Can the parent be *not* an element?

The parentElement property returns the “element” parent, while parentNode returns “any node” parent. These properties are usually the same: they both get the parent.

With the one exception of document.documentElement:

```
alert( document.documentElement.parentNode ); // document
alert( document.documentElement.parentElement ); // null
```

The reason is that the root node document.documentElement (<html>) has document as its parent. But document is not an element node, so parentNode returns it and parentElement does not.

This detail may be useful when we want to travel up from an arbitrary element elem to <html>, but not to the document:

```
while(elem = elem.parentElement) { // go up till <html>
    alert( elem );
}
```

Let’s modify one of the examples above: replace childNodes with children. Now it shows only elements:

```
<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>
```

More links: tables

Till now we described the basic navigation properties.

Certain types of DOM elements may provide additional properties, specific to their type, for convenience.

Tables are a great example of that, and represent a particularly important case:

The `<table>` element supports (in addition to the given above) these properties:

- `table.rows` – the collection of `<tr>` elements of the table.
- `table.caption/tHead/tFoot` – references to elements `<caption>`, `<thead>`, `<tfoot>`.
- `table.tBodies` – the collection of `<tbody>` elements (can be many according to the standard, but there will always be at least one – even if it is not in the source HTML, the browser will put it in the DOM).

`<thead>`, `<tfoot>`, `<tbody>` elements provide the `rows` property:

- `tbody.rows` – the collection of `<tr>` inside.

`<tr>`:

- `tr.cells` – the collection of `<td>` and `<th>` cells inside the given `<tr>`.
- `tr.sectionRowIndex` – the position (index) of the given `<tr>` inside the enclosing `<thead>/<tbody>/<tfoot>`.
- `tr.rowIndex` – the number of the `<tr>` in the table as a whole (including all table rows).

`<td>` and `<th>`:

- `td.cellIndex` – the number of the cell inside the enclosing `<tr>`.

An example of usage:

```
<table id="table">
  <tr>
    <td>one</td><td>two</td>
  </tr>
  <tr>
    <td>three</td><td>four</td>
  </tr>
</table>
```

```
<script>
    // get td with "two" (first row, second column)
    let td = table.rows[0].cells[1];
    td.style.backgroundColor = "red"; // highlight it
</script>
```

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>
```

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>
```

…That’s unless we declare a JavaScript variable with the same name, then it takes precedence:

```
<div id="elem"></div>
<script>
    let elem = 5; // now elem is 5, not a reference to <div id="elem">
    alert(elem); // 5
</script>
```

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.

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.

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 elemitself 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:

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

Let’s find all `input` tags inside the table:

```
<table id="table">
  <tr>
    <td>Your age:</td>
    <td>
      <label>
        <input type="radio" name="age" value="young" checked> less than 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> from 18 to 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> more than 60
      </label>
    </td>
  </tr>
</table>

<script>
  let inputs = table.getElementsByTagName('input');
  for (let input of inputs) {
    alert( input.value + ': ' + input.checked );
  }
</script>
```

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;
```

Looking for `.article` elements:

```
<form name="my-form">
  <div class="article">Article</div>
  <div class="long article">Long article</div>
</form>

<script>
  // find by name attribute
  let form = document.getElementsByName('my-form')[0];
  // find by class inside the form
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, found two elements with class "article"
</script>
```

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>
```

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.

Tag: nodeName and 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.

For instance, let's compare `tagName` and `nodeName` for the `document` and a comment node:

```
<body>
    <!-- comment -->
    <script>
        // for comment
        alert( document.body.firstChild.tagName ); // undefined (not an element)
        alert( document.body.firstChild.nodeName ); // #comment
        // for document
        alert( document.tagName ); // undefined (not an element)
        alert( document.nodeName ); // #document
    </script>
</body>
```

If we only deal with elements, then we can use both `tagName` and `nodeName` – there's no difference.

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>
```

We can try to insert invalid HTML, the browser will fix our errors:

```
<body>
  <script>
    document.body.innerHTML = '<b>test'; // forgot to close the tag
    alert( document.body.innerHTML ); // <b>test</b> (fixed)
  </script>
</body>
```

Scripts don’t execute

If innerHTML inserts a <script> tag into the document – it becomes a part of HTML, but doesn’t execute.

Beware: “innerHTML+=” does a full 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).

As the content is "zeroed-out" and rewritten from the scratch, all images and other resources will be reloaded.

In the chatDiv example above the line `chatDiv.innerHTML+="How goes?"` re-creates the HTML content and reloads smile.gif (hope it's cached). If chatDiv has a lot of other text and images, then the reload becomes clearly visible.

There are other side-effects as well. For instance, if the existing text was selected with the mouse, then most browsers will remove the selection upon rewriting `innerHTML`. And if there was an `<input>` with a text entered by the visitor, then the text will be removed. And so on.

Luckily, there are other ways to add HTML besides `innerHTML`, and we'll study them soon.

outerHTML: full HTML of the element

The `outerHTML` property contains the full HTML of the element. That's like `innerHTML` plus the element itself.

Here's an example:

```
<div id="elem">Hello <b>World</b></div>
<script>
    alert(elem.outerHTML); // <div id="elem">Hello <b>World</b></div>
</script>
```

Beware: unlike innerHTML, writing to outerHTML does not change the element. Instead, it replaces it in the DOM.

Yeah, sounds strange, and strange it is, that's why we make a separate note about it here. Take a look.

Consider the example:

```
<div>Hello, world!</div>
<script>
    let div = document.querySelector('div');
    // replace div.outerHTML with <p>...</p>
    div.outerHTML = '<p>A new element</p>'; // (*)
    // Wow! 'div' is still the same!
    alert(div.outerHTML); // <div>Hello, world!</div> (**)
</script>
```

Looks really odd, right?

In the line (*) we replaced div with <p>A new element</p>. In the outer document (the DOM) we can see the new content instead of the <div>. But, as we can see in line (**), the value of the old div variable hasn't changed!

The outerHTML assignment does not modify the DOM element (the object referenced by, in this case, the variable 'div'), but removes it from the DOM and inserts the new HTML in its place.

So what happened in div.outerHTML=... is:

- div was removed from the document.
- Another piece of HTML <p>A new element</p> was inserted in its place.
- div still has its old value. The new HTML wasn't saved to any variable.

It's so easy to make an error here: modify div.outerHTML and then continue to work with div as if it had the new content in it. But it doesn't. Such thing is correct for innerHTML, but not for outerHTML.

We can write to elem.outerHTML, but should keep in mind that it doesn't change the element we're writing to ('elem'). It puts the new HTML in its place instead. We can get references to the new elements by querying the DOM.\

nodeValue/data: text node content

The innerHTML property is only valid for element nodes.

Other node types, such as text nodes, have their counterpart: nodeValue and data properties. These two are almost the same for practical use, there are only minor specification differences. So we'll use data, because it's shorter.

An example of reading the content of a text node and a comment:

```
<body>
    Hello
    <!-- Comment -->
    <script>
        let text = document.body.firstChild;
        alert(text.data); // Hello
        let comment = text.nextSibling;
        alert(comment.data); // Comment
    </script>
</body>
```

textContent: pure text

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

For instance:

```
<div id="news">
    <h1>Headline!</h1>
    <p>Martians attack people!</p>
</div>
<script>
    // Headline! Martians attack people!
    alert(news.textContent);
</script>
```

As we can see, only text is returned, as if all `<tags>` were cut out, but the text in them remained. In practice, reading such text is rarely needed.

Writing to `textContent` is much more useful, because it allows you to write text the "safe way".

Let's say we have an arbitrary string, for instance entered by a user, and want to show it.

- With `innerHTML` we'll have it inserted "as HTML", with all HTML tags.
- With `textContent` we'll have it inserted "as text", all symbols are treated literally.

Compare the two

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
    let name = prompt("What's your name?", "<b>Winnie-the-pooh!</b>");
    elem1.innerHTML = name;
    elem2.textContent = name;
</script>
```

1. The first `<div>` gets the name "as HTML": all tags become tags, so we see the bold name.
2. The second `<div>` gets the name "as text", so we literally see `<b>Winnie-the-pooh!</b>`.

In most cases, we expect the text from a user, and want to treat it as text. We don't want unexpected HTML in our site. An assignment to `textContent` does exactly that.

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:

```
<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>
```

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

Here’s a blinking element:

```
<div id="elem">A blinking element</div>
<script>
    setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

More properties

DOM elements also have additional properties, in particular those that depend on the class:

- value – the value for <input>, <select> and <textarea> (HTMLInputElement, HTMLSelectElement…).
- href – the “href” for <a href="..."> (HTMLAnchorElement).
- id – the value of “id” attribute, for all elements (HTMLElement).
- …and much more…

For instance:

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

Or if we’d like to get them fast or are interested in a concrete browser specification – we can always output the element using `console.dir(elem)` and read the properties. Or explore “DOM properties” in the Elements tab of the browser developer tools.

Attributes and properties

When the browser loads the page, it “reads” (another word: “parses”) the HTML and generates DOM objects from it. For element nodes, most standard HTML attributes automatically become properties of DOM objects.

For instance, if the tag is `<body id="page">`, then the DOM object has `body.id="page"`.

But the attribute-property mapping is not one-to-one! In this chapter we’ll pay attention to separate these two notions, to see how to work with them, when they are the same, and when they are different.

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 document.body)
```

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
```

So, DOM properties and methods behave just like those of regular JavaScript objects:

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

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

Here we can see it:

```
<body id="body" type="...">
    <input id="input" type="text">
    <script>
        alert(input.type); // text
        alert(body.type); // undefined: DOM property not created, because it's non-standard
    </script>
</body>
```

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.

Here's an extended demo of working with attributes:

```
<body>
  <div id="elem" about="Elephant"></div>
  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant', reading
    elem.setAttribute('Test', 123); // (2), writing
    alert( elem.outerHTML ); // (3), see if the attribute is in HTML (yes)
    for (let attr of elem.attributes) { // (4) list all
      alert( `${attr.name} = ${attr.value}` );
    }
  </script>
</body>
```

Please note:

1. getAttribute('About') – the first letter is uppercase here, and in HTML it's all lowercase. But that doesn't matter: attribute names are case-insensitive.
2. We can assign anything to an attribute, but it becomes a string. So here we have "123" as the value.
3. All attributes including ones that we set are visible in outerHTML.
4. The attributes collection is iterable and has all the attributes of the element (standard and non-standard) as objects with name and value properties.

Property-attribute synchronization

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

In the example below id is modified as an attribute, and we can see the property changed too. And then the same backwards:

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

But there are exclusions, for instance `input.value` synchronizes only from attribute → to property, but not back:

```
<input>
<script>
  let input = document.querySelector('input');
  // attribute => property
  input.setAttribute('value', 'text');
  alert(input.value); // text
  // NOT property => attribute
  input.value = 'newValue';
  alert(input.getAttribute('value')); // text (not updated!)
</script>
```

In the example above:

- Changing the attribute `value` updates the property.
- But the property change does not affect the attribute.

That “feature” may actually come in handy, because the user actions may lead to `value` changes, and then after them, if we want to recover the “original” value from HTML, it’s in the attribute.

DOM properties are typed

DOM properties are not always strings. For instance, the input.checked property (for checkboxes) is a boolean:

```
<input id="input" type="checkbox" checked> checkbox
<script>
    alert(input.getAttribute('checked')); // the attribute value is: empty string
    alert(input.checked); // the property value is: true
</script>
```

There are other examples. 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>
    // string
    alert(div.getAttribute('style')); // color:red;font-size:120%
    // object
    alert(div.style); // [object CSSStyleDeclaration]
    alert(div.style.color); // red
</script>
```

Most properties are strings though.

Quite rarely, even if a DOM property type is a string, it may differ from the attribute. For instance, the href DOM property is always a *full* URL, even if the attribute contains a relative URL or just a #hash.

Here’s an example:

```
<a id="a" href="#hello">link</a>
<script>
    // attribute
    alert(a.getAttribute('href')); // #hello
    // property
    alert(a.href ); // full URL in the form http://site.com/page#hello
</script>
```

If we need the value of href or any other attribute exactly as written in the HTML, we can use getAttribute.

Non-standard attributes, dataset

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>
```

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>
```

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`.

Here’s a rewritten “order state” example:

```
<style>
    .order[data-order-state="new"] {
        color: green;
    }
    .order[data-order-state="pending"] {
        color: blue;
    }
    .order[data-order-state="canceled"] {
        color: red;
    }
</style>

<div id="order" class="order" data-order-state="new">
    A new order.
</div>
<script>
    // read
    alert(order.dataset.orderState); // new
```

```
  // modify
  order.dataset.orderState = "pending"; // (*)
</script>
```

Using `data-*` attributes is a valid, safe way to pass custom data.

Please note that we can not only read, but also modify data-attributes. Then CSS updates the view accordingly: in the example above the last line `(*)` changes the color to blue.

Modifying the document

DOM modification is the key to creating "live" pages.
Here we'll see how to create new elements "on the fly" and modify the existing page content.

Example: show a message

Let's demonstrate using an example. We'll add a message on the page that looks nicer than `alert`.

Here's how it will look:

```
<style>
  .alert {
    padding: 15px;
    border: 1px solid #d6e9c6;
    border-radius: 4px;
    color: #3c763d;
    background-color: #dff0d8;
  }
</style>

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

That was an HTML example. Now let’s create the same `div` with JavaScript (assuming that the styles are in the HTML or an external CSS file).

Creating an element

To create DOM nodes, there are two methods:

document.createElement(tag)
Creates a new *element node* with the given tag:

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

document.createTextNode(text)

Creates a new *text node* with the given text:

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

Creating the message
In our case the message is a `div` with `alert` class and the HTML in it:

```
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
```

We created the element, but as of now it’s only in a variable. We can’t see the element on the page, as it’s not yet a part of the document.

Insertion methods

To make the `div` show up, we need to insert it somewhere into the document. For instance, in `document.body`.

There’s a special method `append` for that: `document.body.append(div)`.

Here’s the full code:

```
<style>
    .alert {
        padding: 15px;
        border: 1px solid #d6e9c6;
        border-radius: 4px;
        color: #3c763d;
        background-color: #dff0d8;
    }
</style>

<script>
    let div = document.createElement('div');
    div.className = "alert";
    div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
    document.body.append(div);
</script>
```

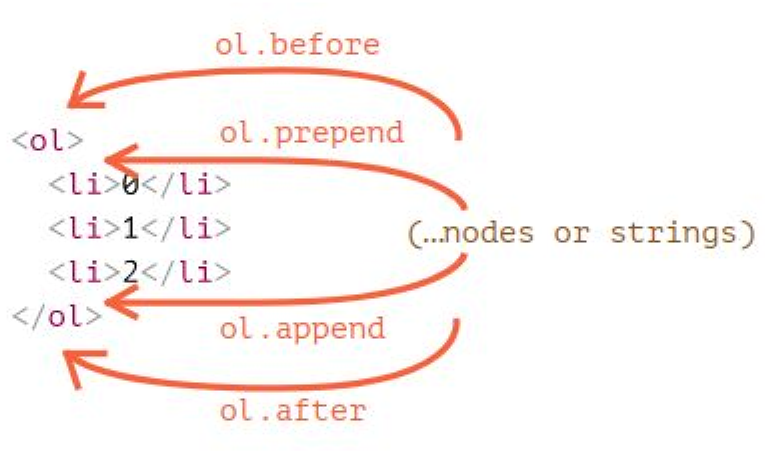
This set of methods provides more ways to insert:

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

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

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

<script>
    ol.before('before'); // insert string "before" before <ol>
    ol.after('after'); // insert string "after" after <ol>
    let liFirst = document.createElement('li');
    liFirst.innerHTML = 'prepend';
    ol.prepend(liFirst); // insert liFirst at the beginning of <ol>

    let liLast = document.createElement('li');
    liLast.innerHTML = 'append';
    ol.append(liLast); // insert liLast at the end of <ol>
</script>
```

Here’s a visual picture what methods do:



So the final list will be:

```
before
<ol id="ol">
    <li>prepend</li>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>append</li>
</ol>
after
```

These methods can insert multiple lists of nodes and text pieces in a single call.

For instance, here a string and an element are inserted:

```
<div id="div"></div>
<script>
    div.before('<p>Hello</p>', document.createElement('hr'));
</script>
```

All text is inserted *as text*.

So the final HTML is:

```
&lt;p&gt;Hello&lt;/p&gt;
<hr>
<div id="div"></div>
```

In other words, strings are inserted in a safe way, like `elem.textContent` does it.

So, these methods can only be used to insert DOM nodes or text pieces.

But what if we want to insert HTML “as html”, with all tags and stuff working, like `elem.innerHTML?`

insertAdjacentHTML/Text/Element

For that we can use another, pretty versatile method: `elem.insertAdjacentHTML(where, html)`.

The first parameter is a code word, specifying where to insert relative to `elem`. Must be one of the following:

- `"beforebegin"` – insert `html` immediately before `elem`,
- `"afterbegin"` – insert `html` into `elem`, at the beginning,
- `"beforeend"` – insert `html` into `elem`, at the end,
- `"afterend"` – insert `html` immediately after `elem`.

The second parameter is an HTML string, that is inserted “as HTML”.
For instance:

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

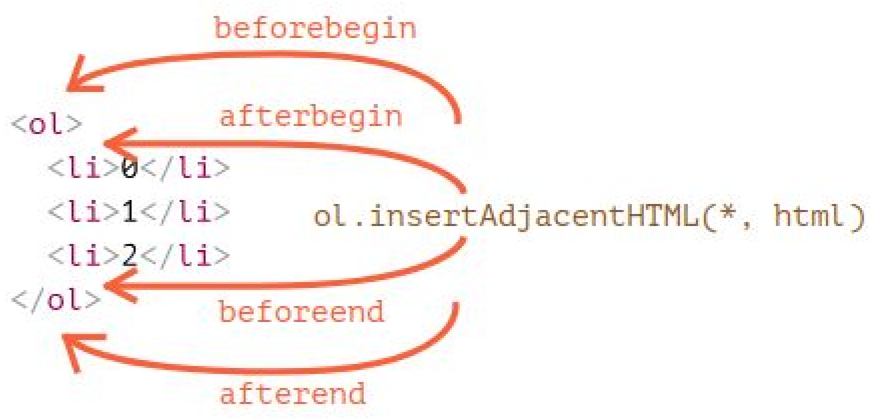
```
<script>
    div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
    div.insertAdjacentHTML('afterend', '<p>Bye</p>');
</script>
```

…Would lead to:

```
<p>Hello</p>
<div id="div"></div>
<p>Bye</p>
```

That's how we can append arbitrary HTML to the page.

Here's the picture of insertion variants:



We can easily notice similarities between this and the previous picture. The insertion points are actually the same, but this method inserts HTML.

The method has two brothers:

- `elem.insertAdjacentText(where, text)` – the same syntax, but a string of `text` is inserted "as text" instead of HTML,
- `elem.insertAdjacentElement(where, elem)` – the same syntax, but inserts an element.

So here's an alternative variant of showing a message:

```
<style>
    .alert {
        padding: 15px;
        border: 1px solid #d6e9c6;
        border-radius: 4px;
        color: #3c763d;
        background-color: #dff0d8;
    }
</style>

<script>
    document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
        <strong>Hi there!</strong> You've read an important message. </div>`);
</script>
```

Node removal

To remove a node, there’s a method `node.remove()`.

Let’s make our message disappear after a second:

```
<style>
    .alert {
        padding: 15px;
        border: 1px solid #d6e9c6;
        border-radius: 4px;
        color: #3c763d;
        background-color: #dff0d8;
    }
</style>

<script>
    let div = document.createElement('div');
    div.className = "alert";
    div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
    document.body.append(div);
    setTimeout(() => div.remove(), 1000);
</script>
```

Please note: 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>

//output
Second
First
```

Cloning nodes: cloneNode

How to insert one more similar message?

We could make a function and put the code there. But the alternative way would be to *clone* the existing `div` and modify the text inside it (if needed).

Sometimes when we have a big element, that may be faster and simpler.

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

An example of copying the message:

```
<style>
    .alert {
        padding: 15px;
        border: 1px solid #d6e9c6;
        border-radius: 4px;
        color: #3c763d;
        background-color: #dff0d8;
    }
</style>

<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>
```

//output

Hi there! You've read an important message.

Bye there! You've read an important message.

DocumentFragment

`DocumentFragment` is a special DOM node that serves as a wrapper to pass around lists of nodes.

We can append other nodes to it, but when we insert it somewhere, then its content is inserted instead.

For example, getListContent below generates a fragment with <li> items, that are later inserted into <ul>:

```
<ul id="ul"> </ul>
<script>
  function getListContent() {
    let fragment = new DocumentFragment();
    for(let i=1; i<=3; i++) {
      let li = document.createElement('li');
      li.append(i);
      fragment.append(li);
    }
    return fragment;
  }
  ul.append(getListContent()); // (*)
</script>
```

Please note, at the last line (*) we append DocumentFragment, but it “blends in”, so the resulting structure will be:

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

DocumentFragment is rarely used explicitly. Why append to a special kind of node, if we can return an array of nodes instead? Rewritten example:

```
<ul id="ul"> </ul>
<script>
  function getListContent() {
    let result = [];
    for(let i=1; i<=3; i++) {
            let li = document.createElement('li');
            li.append(i);
            result.push(li);
        }
    return result;
}
ul.append(...getListContent()); // append + "..." operator = friends!
</script>
```

We mention DocumentFragment mainly because there are some concepts on top of it, like template element, that we'll cover much later.

Old-school insert/remove methods

This information helps to understand old scripts, but not needed for new development.

There are also “old school” DOM manipulation methods, existing for historical reasons. These methods come from really ancient times. Nowadays, there's no reason to use them, as modern methods, such as append, prepend, before, after, remove, replaceWith, are more flexible.

The only reason we list these methods here is that you can find them in many old scripts:

parentElem.appendChild(node)

Appends node as the last child of parentElem. 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>
```

parentElem.insertBefore(node, nextSibling)

Inserts `node` before `nextSibling` into `parentElem`.

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>
```

To insert `newLi` as the first element, we can do it like this:

```
list.insertBefore(newLi, list.firstChild);
```

parentElem.replaceChild(node, oldChild)

Replaces oldChild with node among children of parentElem.

parentElem.removeChild(node)

Removes node from parentElem (assuming node is its child).

The following example removes first `<li>` from `<ol>`:

```
<ol id="list">
    <li>0</li>
    <li>1</li>
    <li>2</li>
</ol>
<script>
    let li = list.firstElementChild;
    list.removeChild(li);
</script>
```

All these methods return the inserted/removed node. In other words, `parentElem.appendChild(node)` returns `node`. But usually the returned value is not used, we just run the method.

A word about "document.write"

There's one more, very ancient method of adding something to a web-page: `document.write`.

The syntax:

```
<p>Somewhere in the page...</p>
<script>
    document.write('<b>Hello from JS</b>');
</script>
<p>The end</p>
```

The call to `document.write(html)` writes the `html` into page "right here and now". The `html` string can be dynamically generated, so it's kind of flexible. We can use JavaScript to create a full-fledged webpage and write it.

The method comes from times when there was no DOM, no standards… Really old times. It still lives, because there are scripts using it.

In modern scripts we can rarely see it, because of the following important limitation:

The call to `document.write` only works while the page is loading.

If we call it afterwards, the existing document content is erased.

For instance:

```
<p>After one second the contents of this page will be replaced...</p>
<script>
    // document.write after 1 second
    // that's after the page loaded, so it erases the existing content
    setTimeout(() => document.write('<b>...By this.</b>'), 1000);
```

```
</script>
```

So it’s kind of unusable at “after loaded” stage, unlike other DOM methods we covered above. That’s the downside.

There’s an upside also. Technically, when `document.write` is called while the browser is reading (“parsing”) incoming HTML, and it writes something, the browser consumes it just as if it were initially there, in the HTML text.

So it works blazingly fast, because there’s *no DOM modification* involved. It writes directly into the page text, while the DOM is not yet built.

So if we need to add a lot of text into HTML dynamically, and we’re at page loading phase, and the speed matters, it may help. But in practice these requirements rarely come together. And usually we can see this method in scripts just because they are old.

Styles and classes

Before we get into JavaScript’s ways of dealing with styles and classes – here’s an important rule. Hopefully it’s obvious enough, but we still have to mention it.

There are generally two ways to style an element:

1. Create a class in CSS and add it: `<div class="...">`
2. Write properties directly into `style`: `<div style="...">`.

JavaScript can modify both classes and `style` properties.

We should always prefer CSS classes to `style`. The latter should only be used if classes “can’t handle it”.

For example, `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', calculated at run-time
elem.style.top = top; // e.g '456px'
```

For other cases, like making the text red, adding a background icon – describe that in CSS and then add the class (JavaScript can do that). That's more flexible and easier to support.

className and classList

Changing a class is one of the most often used actions in scripts.

For instance:

```
<body class="main page">
    <script>
        alert(document.body.className); // main page
  </script>
</body>
```

If we assign something to `elem.className`, it replaces the whole string of classes. Sometimes that's what we need, but often we want to add/remove a single class.

There's another property for that: `elem.classList`.

The `elem.classList` is a special object with methods to `add/remove/toggle` a single class.

For instance:

```
<body class="main page">
    <script>
        // add a class
       document.body.classList.add('article');
       alert(document.body.className); // main page article
    </script>
</body>
```

So we can operate both on the full class string using `className` or on individual classes using `classList`. What we choose depends on our needs.

Methods of classList:

- `elem.classList.add/remove("class")` – adds/removes the class.

- elem.classList.toggle("class") – adds the class if it doesn't exist, otherwise removes it.
- elem.classList.contains("class") – checks for the given class, returns true/false.

Besides, classList is iterable, so we can list all classes with for..of, like this:

```
<body class="main page">
    <script>
        for (let name of document.body.classList) {
            alert(name); // main, and then page
        }
    </script>
</body>
```

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 the same as if we had in the attribute style a string width:100px.

For multi-word property the camelCase is used:

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

For instance:

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

Prefixed properties

Browser-prefixed properties like -moz-border-radius, -webkit-border-radius also follow the same rule: a dash means upper case.

For instance:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Resetting the style property

Sometimes we want to assign a style property, and later remove it.

For instance, to hide an element, we can set elem.style.display = "none".

Then later we may want to remove the style.display as if it were not set. Instead of delete elem.style.display we should assign an empty string to it: elem.style.display = "".

```
// if we run this code, the <body> will blink
document.body.style.display = "none"; // hide
setTimeout(() => document.body.style.display = "", 1000); // back to normal
```

If we set style.display to an empty string, then the browser applies CSS classes and its built-in styles normally, as if there were no such style.display property at all.

Full rewrite with style.cssText

Normally, we use style.* to assign individual style properties. We can't set the full style like div.style="color: red; width: 100px", because div.style is an object, and it's read-only.

To set the full style as a string, there's a special property style.cssText:

```
<div id="div">Button</div>
<script>
    // we can set special style flags like "important" here
    div.style.cssText=`color: red !important;
        background-color: yellow;
        width: 100px;
        text-align: center;
    `;
   alert(div.style.cssText);
</script>
```

This property is rarely used, because such assignment removes all existing styles: it does not add, but replaces them. May occasionally delete something needed. But we can safely use it for new elements, when we know we won't delete an existing style. The same can be accomplished by setting an attribute: `div.setAttribute('style', 'color: red...')`.

Mind the units

Don't forget to add CSS units to values.

For instance, we should not set `elem.style.top to 10`, but rather to `10px`. Otherwise it wouldn't work:

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

        // now add the CSS unit (px) - and it works
        document.body.style.margin = '20px';
        alert(document.body.style.margin); // 20px
  </script>
</body>
```

Computed styles: getComputedStyle

So, modifying a style is easy. But how to *read* it?

For instance, we want to know the size, margins, the color of an element. How to do it?

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>
```

…But what if we need, say, to increase the margin by `20px`? We would want the current value of it.

There's another method for that: `getComputedStyle`.

The syntax is:

```
getComputedStyle(element, [pseudo])
```

element
Element to read the value for.

pseudo
A pseudo-element if required, for instance `::before`. An empty string or no argument means the element itself.

The result is an object with styles, like `elem.style`, but now with respect to all CSS classes.
For instance:

```
<head>
    <style> body { color: red; margin: 5px } </style>
</head>
```

```
<body>
    <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:

1. 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%`.
2. 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`. For geometry properties resolved values may have a floating point, like `width:50.5px`

A long time ago `getComputedStyle` was created to get computed values, but it turned out that resolved values are much more convenient, and the standard changed. So nowadays `getComputedStyle` actually returns the resolved value of the property, usually in `px` for geometry.

getComputedStyle requires the full property name

We should always ask for the exact property that we want, like `paddingLeft` or `marginTop` or `borderTopWidth`. Otherwise the correct result is not guaranteed.

For instance, if there are properties `paddingLeft/paddingTop`, then what should we get for `getComputedStyle(elem).padding`? Nothing, or maybe a “generated” value from known paddings? There’s no standard rule here.

There are other inconsistencies. As an example, some browsers (Chrome) show 10px in the document below, and some of them (Firefox) – do not:

```
<style>
    body {
        margin: 10px;
    }
</style>
<script>
    let style = getComputedStyle(document.body);
    alert(style.margin); // empty string in Firefox
</script>
```

Element size and scrolling

There are many JavaScript properties that allow us to read information about element width, height and other geometry features.

We often need them when moving or positioning elements in JavaScript.

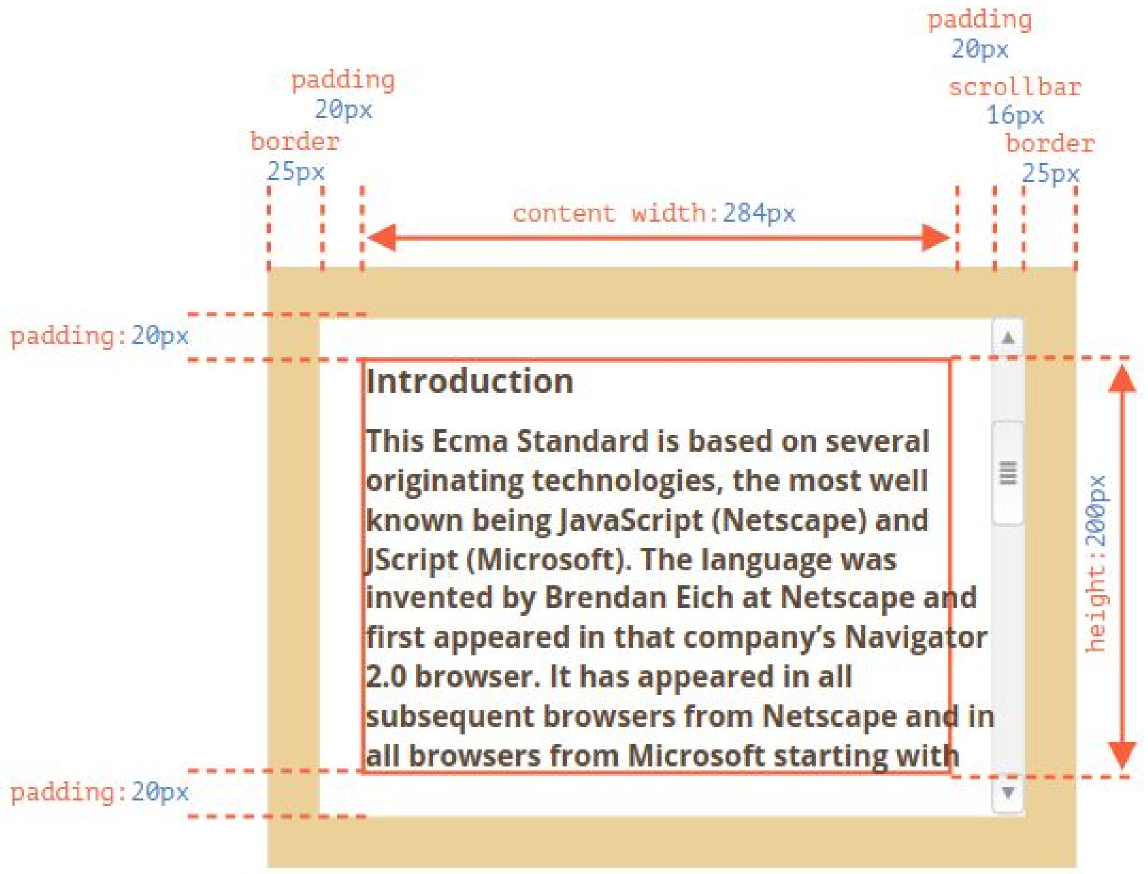
Sample element

As a sample element to demonstrate properties we'll use the one given below:

```
<div id="example">
    ...Text...
</div>
<style>
    #example {
        width: 300px;
        height: 200px;
        border: 25px solid #E8C48F;
        padding: 20px;
        overflow: auto;
    }
```

`</style>`

It has the border, padding and scrolling. The full set of features. There are no margins, as they are not the part of the element itself, and there are no special properties for them.
The element looks like this:


padding 20px
border 25px
content width:284px
padding 20px
scrollbar 16px
border 25px
padding:20px
height:200px
padding:20px
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with

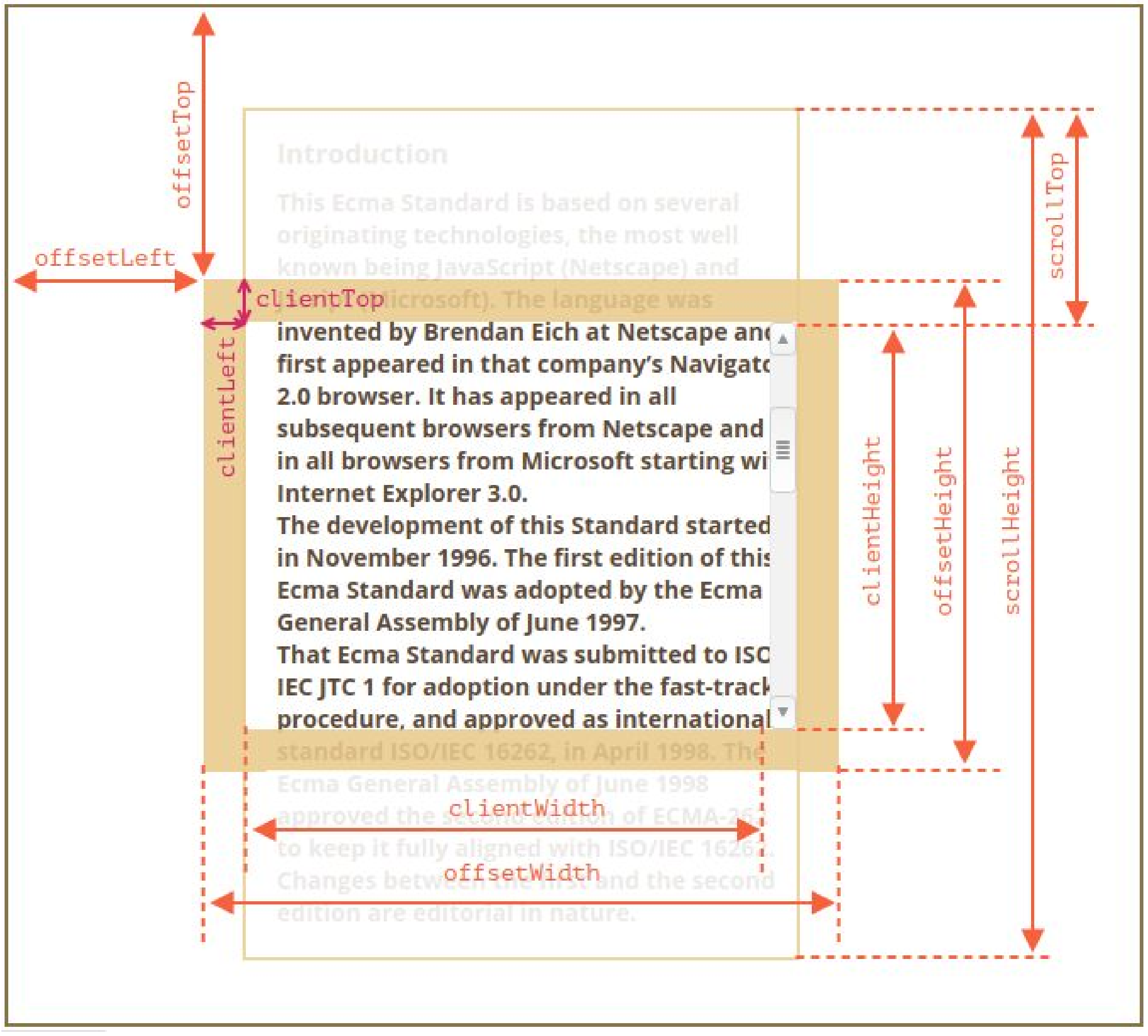
Mind the scrollbar

The picture above demonstrates the most complex case when the element has a scrollbar. Some browsers (not all) reserve the space for it by taking it from the content (labeled as "content width" above).

So, without scrollbar the content width would be `300px`, but if the scrollbar is `16px` wide (the width may vary between devices and browsers) then only `300 - 16 = 284px` remains, and we should take it into account. That’s why examples from this chapter assume that there’s a scrollbar. Without it, some calculations are simpler.

Geometry

Here’s the overall picture with geometry properties:


offsetTop
offsetLeft
clientTop
clientLeft
scrollTop
clientHeight
offsetHeight
scrollHeight
clientWidth
offsetWidth
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company’s Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.
The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997.
That Ecma Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.

Values of these properties are technically numbers, but these numbers are “of pixels”, so these are pixel measurements.

Let’s start exploring the properties starting from the outside of the element.

offsetParent, offsetLeft/Top

These properties are rarely needed, but still they are the “most outer” geometry properties, so we’ll start with them.

The offsetParent is the nearest ancestor that the browser uses for calculating coordinates during rendering.

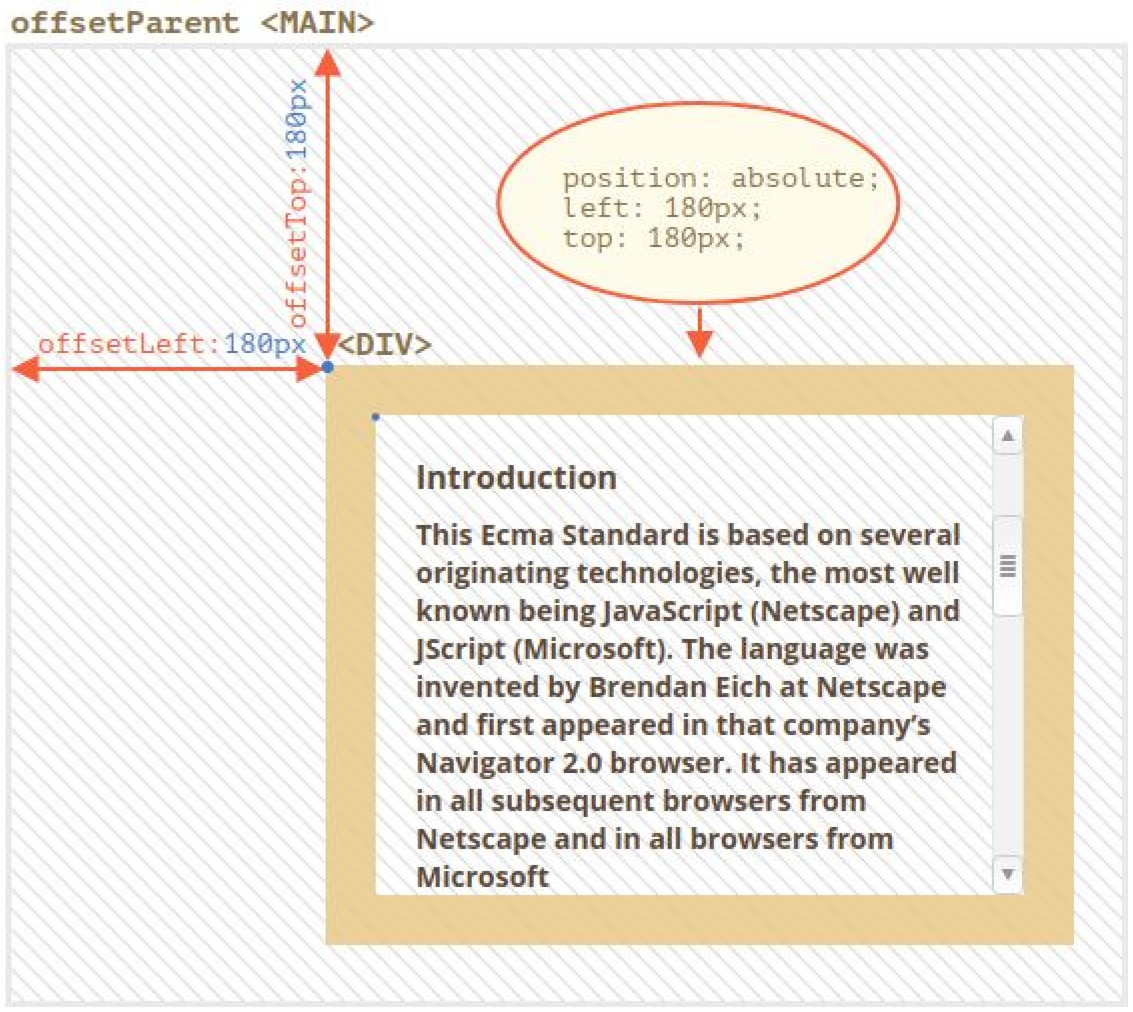
That’s the nearest ancestor that is one of the following:

1. CSS-positioned (position is absolute, relative, fixed or sticky), or
2. <td>, <th>, or <table>, or
3. <body>.

Properties offsetLeft/offsetTop provide x/y coordinates relative to offsetParent upper-left corner.

In the example below the inner <div> has <main> as offsetParent and offsetLeft/offsetTop shifts from its upper-left corner (180):

```
<main style="position: relative" id="main">
    <article>
        <div id="example" style="position: absolute; left: 180px; top: 180px">...</div>
    </article>
</main>
<script>
    alert(example.offsetParent.id); // main
    alert(example.offsetLeft); // 180 (note: a number, not a string "180px")
    alert(example.offsetTop); // 180
</script>
```


offsetParent <MAIN>
offsetTop:180px
offsetLeft:180px
<DIV>
position: absolute;
left: 180px;
top: 180px;
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft

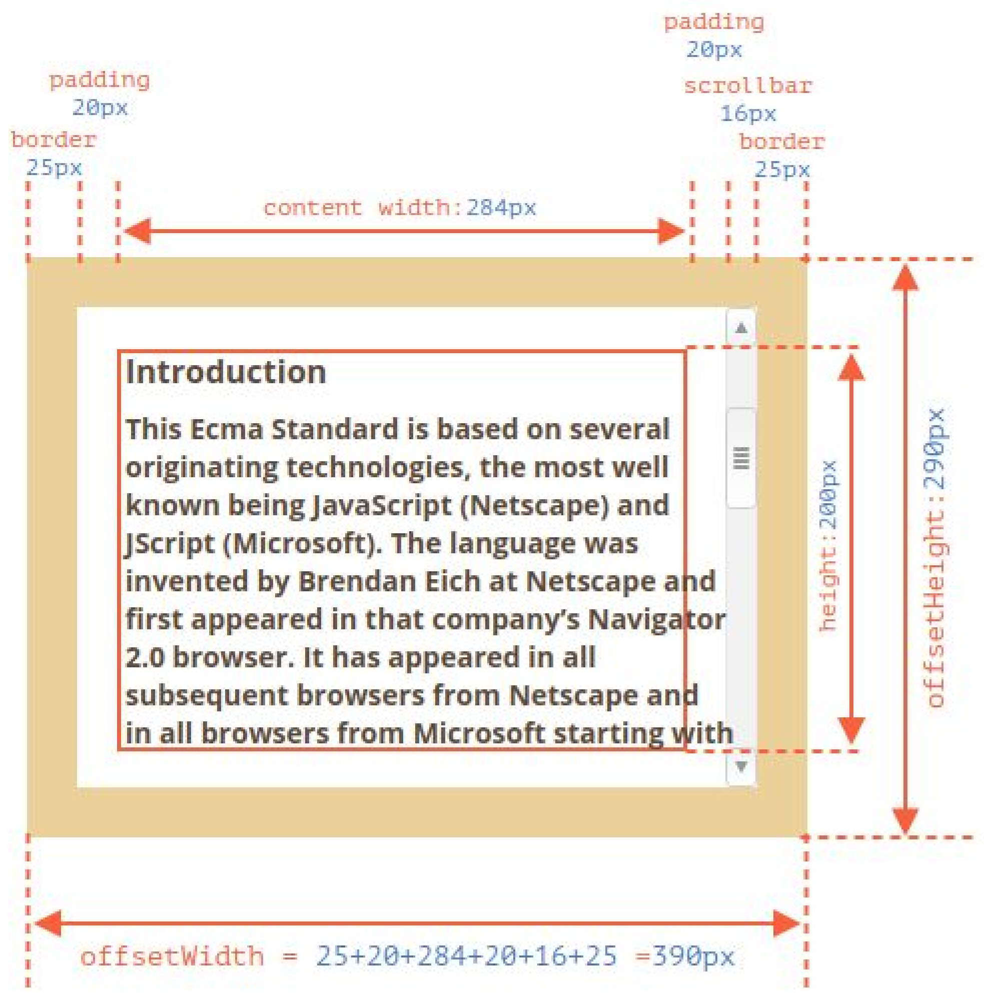
There are several occasions when offsetParent is null:

1. For not shown elements (display:none or not in the document).
2. For <body> and <html>.
3. For elements with position:fixed.

offsetWidth/Height

Now let's move on to the element itself.

These two properties are the simplest ones. They provide the "outer" width/height of the element. Or, in other words, its full size including borders.


padding 20px
border 25px
content width:284px
padding 20px
scrollbar 16px
border 25px
height:200px
offsetHeight:290px
offsetWidth = 25+20+284+20+16+25 =390px
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with

For our sample element:

- offsetWidth = 390 – the outer width, can be calculated as inner CSS-width (300px) plus paddings (2 * 20px) and borders (2 * 25px).
- offsetHeight = 290 – the outer height.

Geometry properties are zero/null for elements that are not displayed

Geometry properties are calculated only for displayed elements.

If an element (or any of its ancestors) has a display:none or is not in the document, then all geometry properties are zero (or null for offsetParent).

For example, offsetParent is null, and offsetWidth, offsetHeight are 0 when we created an element, but haven't inserted it into the document yet, or it (or it's ancestor) has display:none.

We can use this to check if an element is hidden, like this:

```
function isHidden(elem) {
    return !elem.offsetWidth && !elem.offsetHeight;
}
```

Please note that such isHidden returns true for elements that are on-screen, but have zero sizes (like an empty <div>).

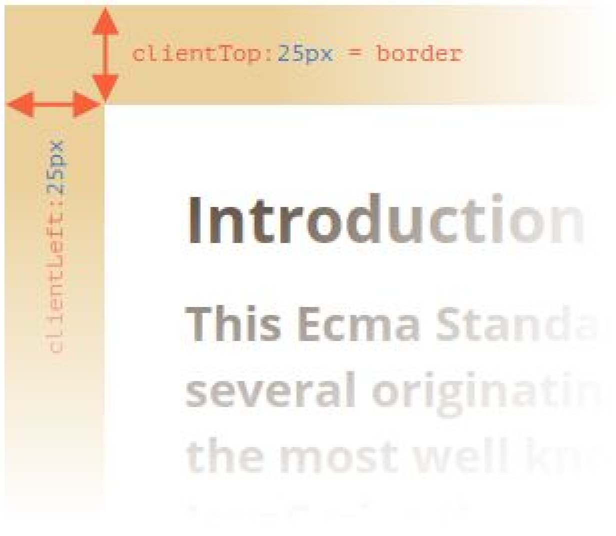
clientTop/Left

Inside the element we have the borders.

To measure them, there are properties clientTop and clientLeft.

In our example:

- clientLeft = 25 – left border width
- clientTop = 25 – top border width



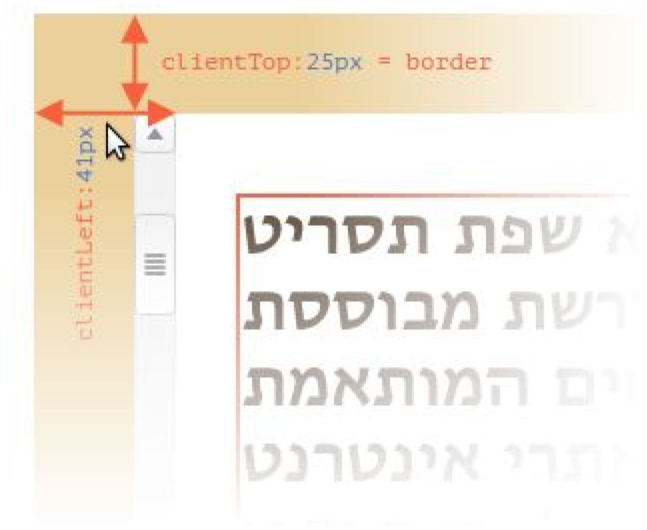
…But to be precise – these properties are not border width/height, but rather relative coordinates of the inner side from the outer side.

What’s the difference?

It becomes visible when the document is right-to-left (the operating system is in Arabic or Hebrew languages). The scrollbar is then not on the right, but on the left, and then clientLeft also includes the scrollbar width.

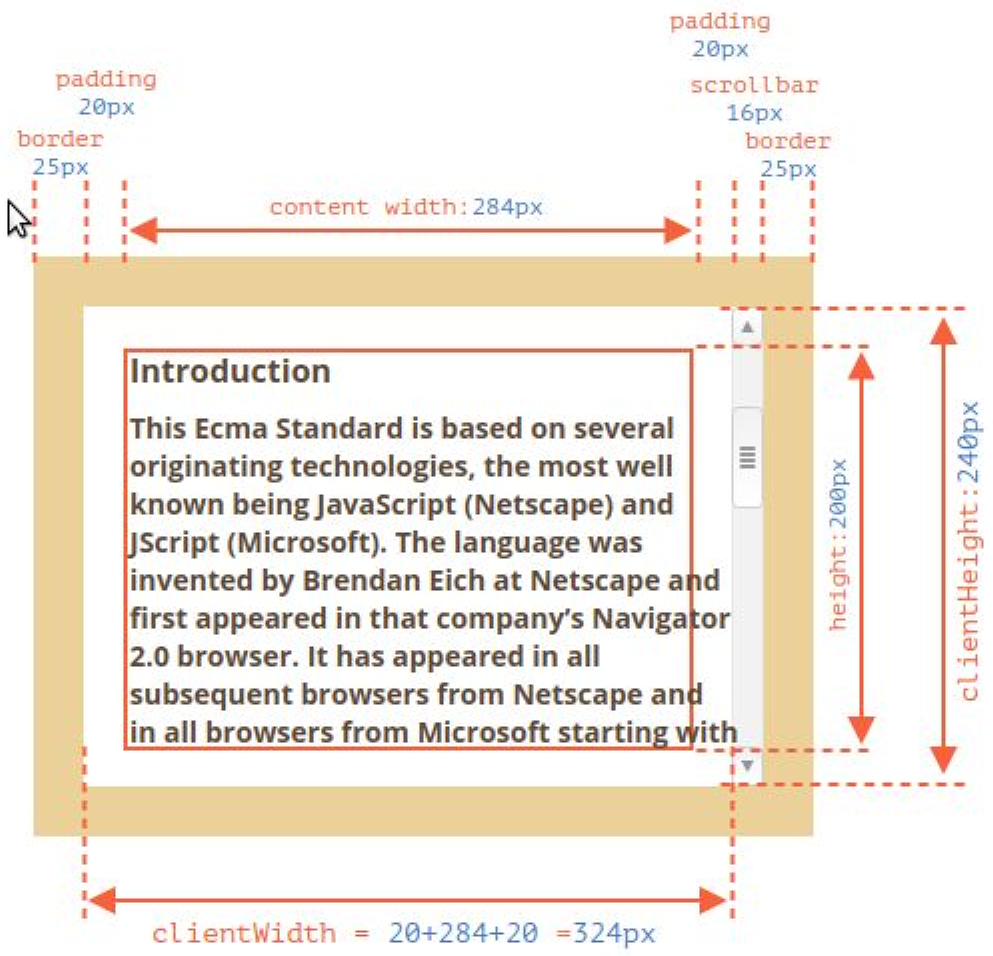
In that case, clientLeft would be not 25, but with the scrollbar width 25 + 16 = 41.

Here’s the example in hebrew:



clientWidth/Height

These properties provide the size of the area inside the element borders.
They include the content width together with paddings, but without the scrollbar:

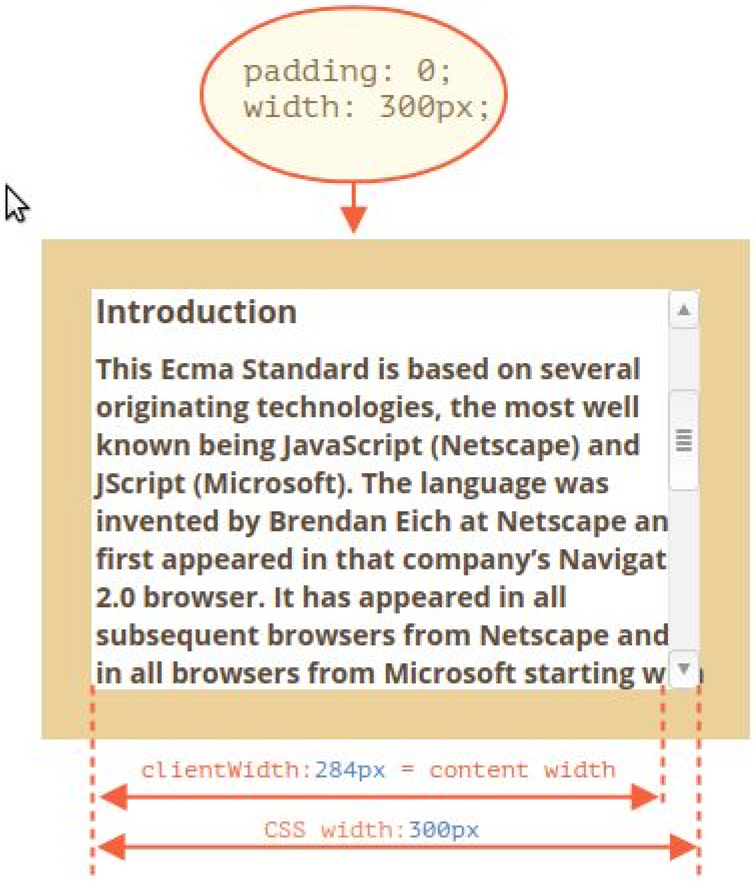

padding
20px
border
25px
padding
20px
scrollbar
16px
border
25px
content width:284px
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with
height:200px
clientHeight:240px
clientWidth = 20+284+20 =324px

On the picture above let's first consider clientHeight.

There's no horizontal scrollbar, so it's exactly the sum of what's inside the borders: CSS-height 200px plus top and bottom paddings (2 * 20px) total 240px.

Now clientWidth – here the content width is not 300px, but 284px, because 16px are occupied by the scrollbar. So the sum is 284px plus left and right paddings, a total 324px.

If there are no paddings, then clientWidth/Height is exactly the content area, inside the borders and the scrollbar (if any).



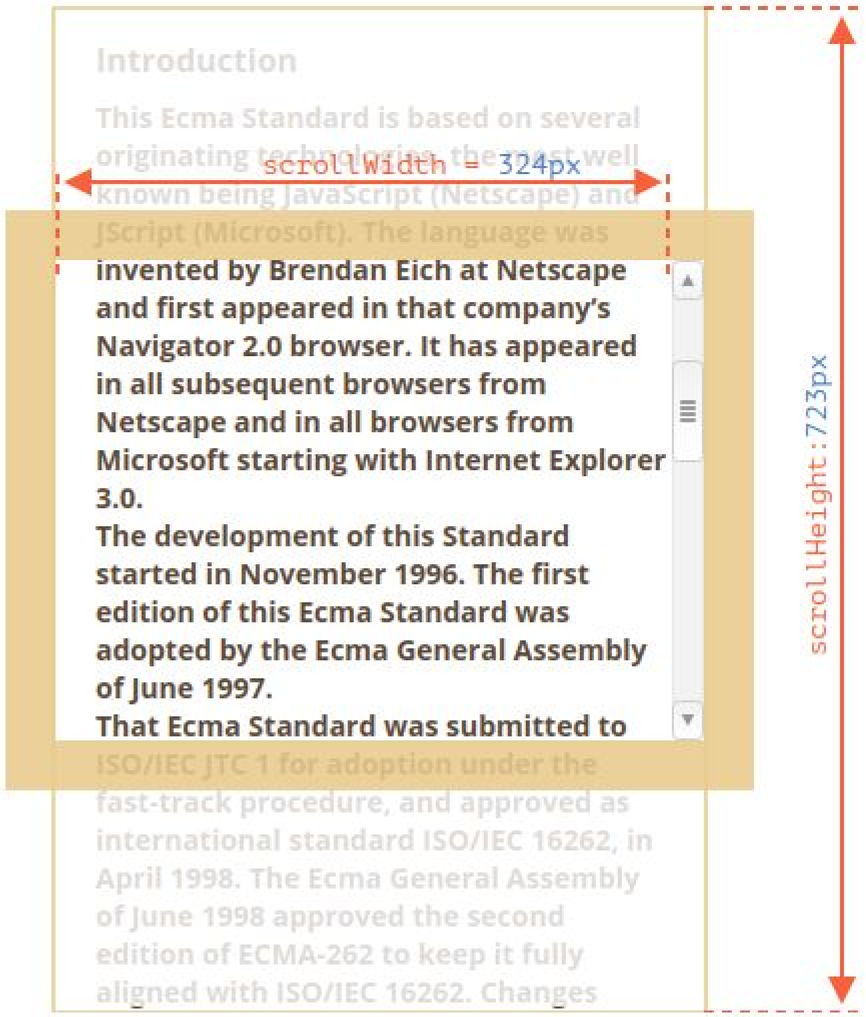
Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape an first appeared in that company's Navigat 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting w

So when there's no padding we can use clientWidth/clientHeight to get the content area size.

scrollWidth/Height

These properties are like clientWidth/clientHeight, but they also include the scrolled out (hidden) parts:


scrollWidth = 324px
scrollHeight: 723px
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.
The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997.
That Ecma Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes

On the picture above:

- scrollHeight = 723 – is the full inner height of the content area including the scrolled out parts.
- scrollWidth = 324 – is the full inner width, here we have no horizontal scroll, so it equals clientWidth.

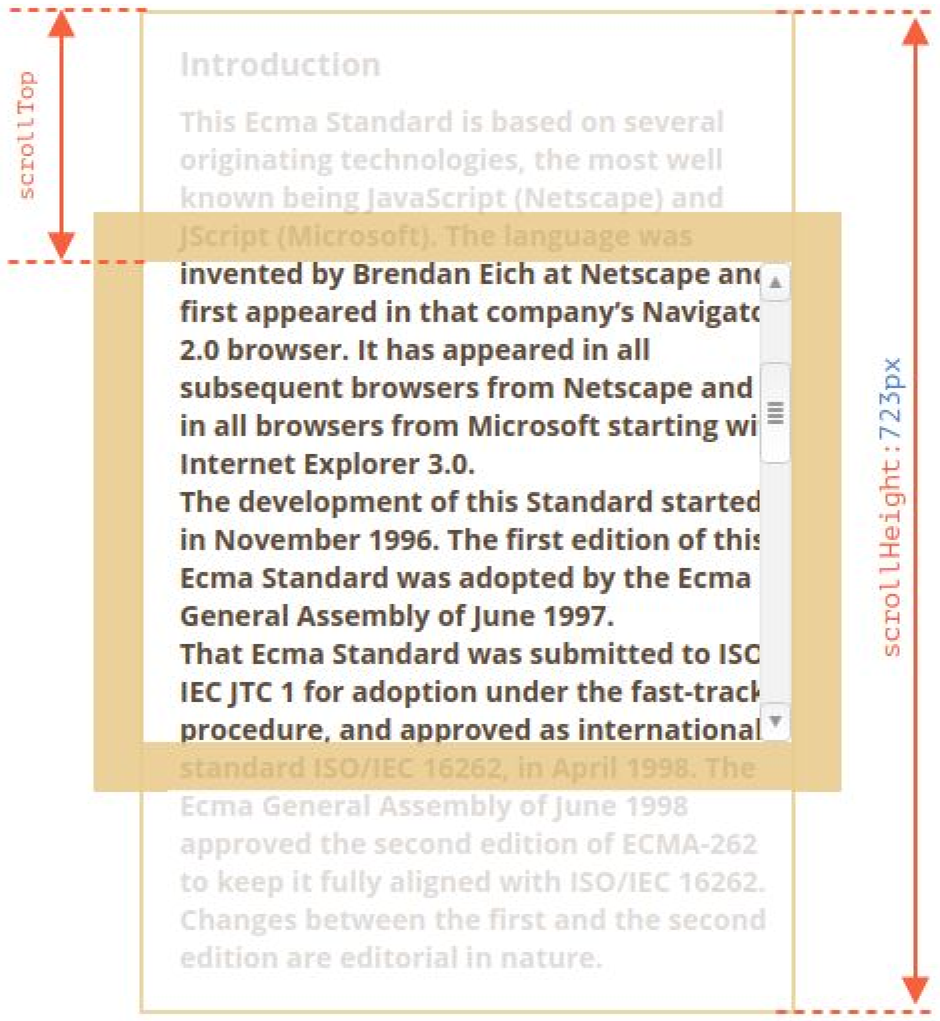
We can use these properties to expand the element wide to its full width/height.
Like this:

```
// expand the element to the full content height
element.style.height = `${element.scrollHeight}px`;
```

scrollLeft/scrollTop

Properties scrollLeft/scrollTop are the width/height of the hidden, scrolled out part of the element.

On the picture below we can see scrollHeight and scrollTop for a block with a vertical scroll.



Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape anc first appeared in that company's Navigatc 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting wi Internet Explorer 3.0.
The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997.
That Ecma Standard was submitted to ISC IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.

In other words, scrollTop is "how much is scrolled up".

scrollLeft/scrollTop can be modified

Most of the geometry properties here are read-only, but scrollLeft/scrollTop can be changed, and the browser will scroll the element.

Setting scrollTop to 0 or Infinity will make the element scroll to the very top/bottom respectively.

Don't take width/height from CSS

We've just covered geometry properties of DOM elements, that can be used to get widths, heights and calculate distances.

But as we know from the chapter Styles and classes, we can read CSS-height and width using getComputedStyle.

So why not to read the width of an element with getComputedStyle, like this?

```
let elem = document.body;
alert( getComputedStyle(elem).width ); // show CSS width for elem
```

Why should we use geometry properties instead? There are two reasons:

1. First, CSS width/height depend on another property: box-sizing that defines "what is" CSS width and height. A change in box-sizing for CSS purposes may break such JavaScript.
2. Second, CSS width/height may be auto, for instance for an inline element:

   ```
   <span id="elem">Hello!</span>
   <script>
     alert( getComputedStyle(elem).width ); // auto
   </script>
   ```

   From the CSS standpoint, width:auto is perfectly normal, but in JavaScript we need an exact size in px that we can use in calculations. So here CSS width is useless.

And there's one more reason: a scrollbar. Sometimes the code that works fine without a scrollbar becomes buggy with it, because a scrollbar takes the space from the content in some browsers. So the real width available for the content is *less* than CSS width. And clientWidth/clientHeight take that into account.

…But with getComputedStyle(elem).width the situation is different. Some browsers (e.g. Chrome) return the real inner width, minus the scrollbar, and some of them (e.g. Firefox) – CSS width (ignore the scrollbar). Such cross-browser differences is the reason not to use getComputedStyle, but rather rely on geometry properties.

On a Desktop Windows OS, Firefox, Chrome, Edge all reserve the space for the scrollbar. But Firefox shows 300px, while Chrome and Edge show less. That's because Firefox returns the CSS width and other browsers return the "real" width.
Please note that the described difference is only about reading getComputedStyle(...).width from JavaScript, visually everything is correct.

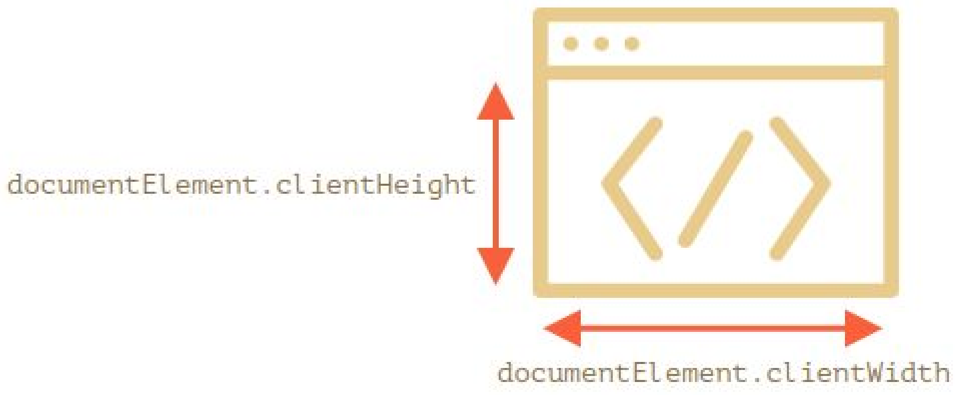
Window sizes and scrolling

How do we find the width and height of the browser window? How do we get the full width and height of the document, including the scrolled out part? How do we scroll the page using JavaScript?

For most such requests, we can use the root document element document.documentElement, that corresponds to the <html> tag. But there are additional methods and peculiarities important enough to consider.

Width/height of the window

To get window width and height we can use clientWidth/clientHeight of document.documentElement:



Not window.innerWidth/Height

Browsers also support properties window.innerWidth/innerHeight. They look like what we want. So why not to use them instead?

If there exists a scrollbar, and it occupies some space, clientWidth/clientHeight provide the width/height without it (subtract it). In other words, they return the width/height of the visible part of the document, available for the content

…And window.innerWidth/innerHeight include the scrollbar.

If there's a scrollbar, and it occupies some space, then these two lines show different values:

```
alert( window.innerWidth ); // full window width
alert( document.documentElement.clientWidth ); // window width minus the scrollbar
```

DOCTYPE is important

Please note: top-level geometry properties may work a little bit differently when there's no <!DOCTYPE HTML> in HTML. Odd things are possible.

In modern HTML we should always write DOCTYPE.

Width/height of the document

Theoretically, as the root document element is document.documentElement, and it encloses all the content, we could measure document full size as document.documentElement.scrollWidth/scrollHeight.

But on that element, for the whole page, these properties do not work as intended. In Chrome/Safari/Opera if there's no scroll, then documentElement.scrollHeight may be even less than documentElement.clientHeight! Sounds like nonsense, weird, right?

To reliably obtain the full document height, we should take the maximum of these properties:

```
let scrollHeight = Math.max(
  document.body.scrollHeight, document.documentElement.scrollHeight,
  document.body.offsetHeight, document.documentElement.offsetHeight,
  document.body.clientHeight, document.documentElement.clientHeight
);
alert('Full document height, with scrolled out part: ' + scrollHeight);
```

Why so? Better don't ask. These inconsistencies come from ancient times, not a "smart" logic.

Get the current scroll

DOM elements have their current scroll state in elem.scrollLeft/scrollTop.

For document scroll document.documentElement.scrollLeft/Top works in most browsers, except older WebKit-based ones, like Safari (bug 5991), where we should use document.body instead of document.documentElement.

Luckily, we don't have to remember these peculiarities at all, because the scroll is available in the special properties window.pageXOffset/pageYOffset:

```
alert('Current scroll from the top: ' + window.pageYOffset);
alert('Current scroll from the left: ' + window.pageXOffset);
```

These properties are read-only.

Scrolling: scrollTo, scrollBy, scrollIntoView

Important:

  To scroll the page from JavaScript, its DOM must be fully built.
  For instance, if we try to scroll the page from the script in <head>, it won't work.

Regular elements can be scrolled by changing scrollTop/scrollLeft.

We can do the same for the page using document.documentElement.scrollTop/Left (except Safari, where document.body.scrollTop/Left should be used instead).

Alternatively, there’s a simpler, universal solution: special methods `window.scrollBy(x,y)` and `window.scrollTo(pageX,pageY)`.

- The method `scrollBy(x,y)` scrolls the page *relative to its current position*. For instance, `scrollBy(0,10)` scrolls the page `10px` down.
- The method `scrollTo(pageX,pageY)` scrolls the page *to absolute coordinates*, so that the top-left corner of the visible part has coordinates `(pageX, pageY)` relative to the document’s top-left corner. It’s like setting `scrollLeft/scrollTop`.

To scroll to the very beginning, we can use `scrollTo(0,0)`.

These methods work for all browsers the same way.

scrollIntoView

For completeness, let’s cover one more method: `elem.scrollIntoView(top)`.

The call to `elem.scrollIntoView(top)` scrolls the page to make elem visible. It has one argument:

- if `top=true` (that’s the default), then the page will be scrolled to make elem appear on the top of the window. The upper edge of the element is aligned with the window top.
- if `top=false`, then the page scrolls to make elem appear at the bottom. The bottom edge of the element is aligned with the window bottom.

Forbid the scrolling

Sometimes we need to make the document “unscrollable”. For instance, when we need to cover it with a large message requiring immediate attention, and we want the visitor to interact with that message, not with the document.

To make the document unscrollable, it’s enough to set `document.body.style.overflow = "hidden"`. The page will freeze on its current scroll.

Try it:

```
document.body.style.overflow = ‘hidden’
document.body.style.overflow = ‘ ’
```

The first (document.body.style.overflow = ‘hidden’) freezes the scroll, the second one (document.body.style.overflow = ‘ ’) resumes it.

We can use the same technique to “freeze” the scroll for other elements, not just for document.body.

Coordinates

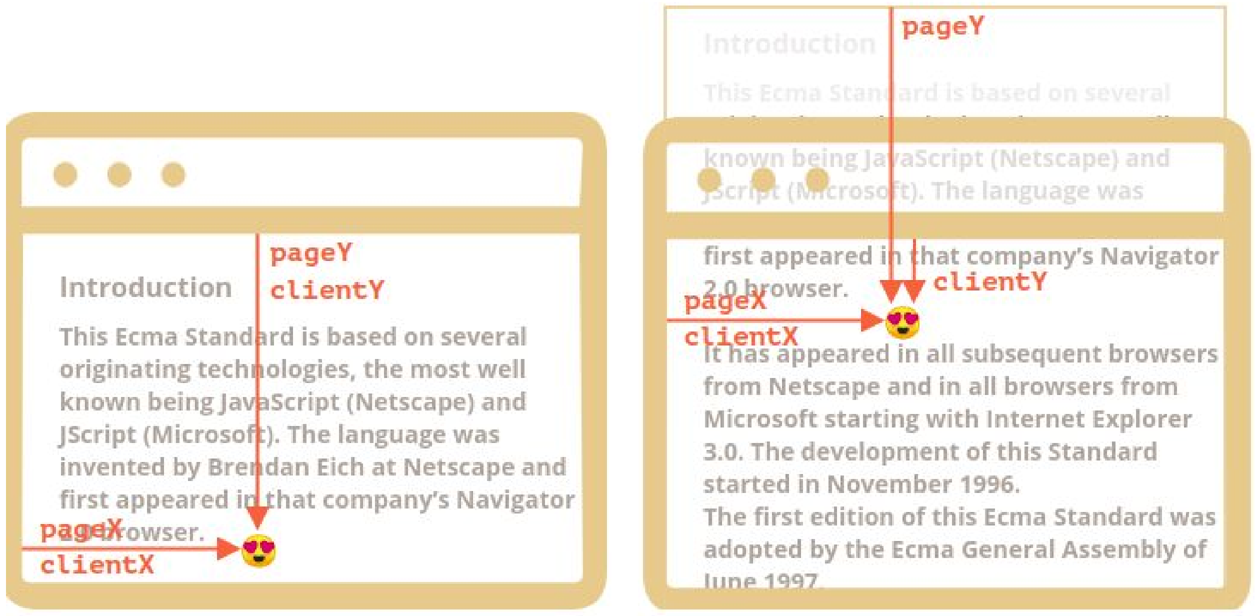
To move elements around we should be familiar with coordinates.

Most JavaScript methods deal with one of two coordinate systems:

1. **Relative to the window** – similar to position:fixed, calculated from the window top/left edge.
   - we’ll denote these coordinates as clientX/clientY, the reasoning for such name will become clear later, when we study event properties.
2. **Relative to the document** – similar to position:absolute in the document root, calculated from the document top/left edge.
   - we’ll denote them pageX/pageY.

When the page is scrolled to the very beginning, so that the top/left corner of the window is exactly the document top/left corner, these coordinates equal each other. But after the document shifts, window-relative coordinates of elements change, as elements move across the window, while document-relative coordinates remain the same.

On this picture we take a point in the document and demonstrate its coordinates before the scroll (left) and after it (right).



When the document scrolled:

- pageY – document-relative coordinate stayed the same, it’s counted from the document top (now scrolled out).
- clientY – window-relative coordinate did change (the arrow became shorter), as the same point became closer to window top.

Element coordinates: getBoundingClientRect

The method elem.getBoundingClientRect() returns window coordinates for a minimal rectangle that encloses elem as an object of built-in DOMRect class.

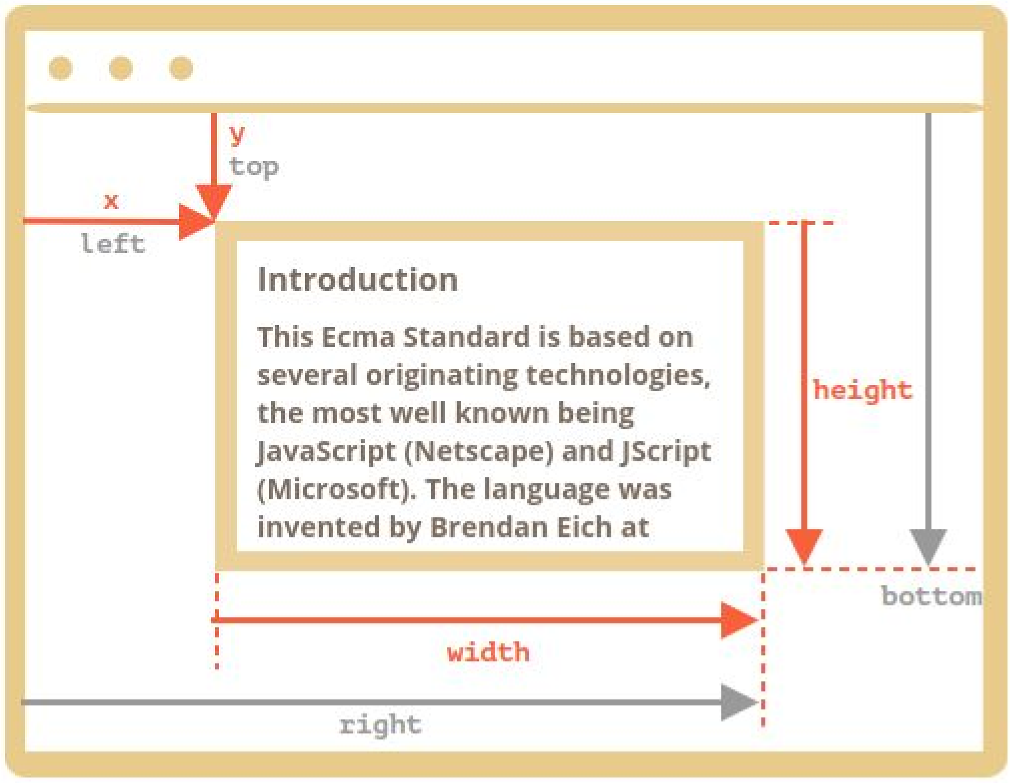
Main DOMRect properties:

- x/y – X/Y-coordinates of the rectangle origin relative to window,
- width/height – width/height of the rectangle (can be negative)

Additionally, there are derived properties:

- top/bottom – Y-coordinate for the top/bottom rectangle edge,
- left/right – X-coordinate for the left/right rectangle edge.

Here’s the picture of elem.getBoundingClientRect() output:


y
top
x
left
Introduction
This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at
height
bottom
width
right

As you can see, x/y and width/height fully describe the rectangle. Derived properties can be easily calculated from them:

- left = x
- top = y
- right = x + width
- bottom = y + height

Please note:

- Coordinates may be decimal fractions, such as 10.5. That’s normal, internally browsers uses fractions in calculations. We don’t have to round them when setting to style.left/top.

- Coordinates may be negative. For instance, if the page is scrolled so that `elem` is now above the window, then `elem.getBoundingClientRect().top` is negative.

Coordinates right/bottom are different from CSS position properties

There are obvious similarities between window-relative coordinates and CSS `position:fixed`.

But in CSS positioning, `right` property means the distance from the right edge, and `bottom` property means the distance from the bottom edge.

If we just look at the picture above, we can see that in JavaScript it is not so. All window coordinates are counted from the top-left corner, including these ones.

elementFromPoint(x, y)

The call to `document.elementFromPoint(x, y)` returns the most nested element at window coordinates `(x, y)`.
The syntax is:

```
let elem = document.elementFromPoint(x, y);
```

For instance, the code below highlights and outputs the tag of the element that is now in the middle of the window:

```
let centerX = document.documentElement.clientWidth / 2;
let centerY = document.documentElement.clientHeight / 2;
let elem = document.elementFromPoint(centerX, centerY);
elem.style.background = "red";
alert(elem.tagName);
```

As it uses window coordinates, the element may be different depending on the current scroll position.

Using for “fixed” positioning

Most of the time we need coordinates in order to position something.

To show something near an element, we can use `getBoundingClientRect` to get its coordinates, and then CSS `position` together with left/top (or right/bottom).

For instance, the function `createMessageUnder(elem, html)` below shows the message under `elem`:

```
let elem = document.getElementById("coords-show-mark");
function createMessageUnder(elem, html) {
    // create message element
    let message = document.createElement('div');
    // better to use a css class for the style here
    message.style.cssText = "position:fixed; color: red";
    // assign coordinates, don't forget "px"!
    let coords = elem.getBoundingClientRect();
    message.style.left = coords.left + "px";
    message.style.top = coords.bottom + "px";
    message.innerHTML = html;
    return message;
}

// Usage:
// add it for 5 seconds in the document
let message = createMessageUnder(elem, 'Hello, world!');
document.body.append(message);
setTimeout(() => message.remove(), 5000);
```

The code can be modified to show the message at the left, right, below, apply CSS animations to “fade it in” and so on. That’s easy, as we have all the coordinates and sizes of the element.

But note the important detail: when the page is scrolled, the message flows away from the button.

The reason is obvious: the message element relies on `position:fixed`, so it remains at the same place of the window while the page scrolls away.

To change that, we need to use document-based coordinates and `position:absolute`

Document coordinates

Document-relative coordinates start from the upper-left corner of the document, not the window.

In CSS, window coordinates correspond to position:fixed, while document coordinates are similar to position:absolute on top.

We can use position:absolute and top/left to put something at a certain place of the document, so that it remains there during a page scroll. But we need the right coordinates first.

There’s no standard method to get the document coordinates of an element. But it’s easy to write it.

The two coordinate systems are connected by the formula:

- pageY = clientY + height of the scrolled-out vertical part of the document.
- pageX = clientX + width of the scrolled-out horizontal part of the document.

The function getCoords(elem) will take window coordinates from elem.getBoundingClientRect() and add the current scroll to them:

```
// get document coordinates of the element
function getCoords(elem) {
    let box = elem.getBoundingClientRect();
    return {
        top: box.top + window.pageYOffset,
        left: box.left + window.pageXOffset
   };
}
```

If in the example above we used it with position:absolute, then the message would stay near the element on scroll.

The modified createMessageUnder function:

```
function createMessageUnder(elem, html) {
    let message = document.createElement('div');
    message.style.cssText = "position:absolute; color: red";
    let coords = getCoords(elem);
    message.style.left = coords.left + "px";
    message.style.top = coords.bottom + "px";
    message.innerHTML = html;
    return message;
}
```

Introduction to browser events

An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

Here's a list of the most useful DOM events, just to take a look at:

Mouse events:

- `click` – when the mouse clicks on an element (touchscreen devices generate it on a tap).
- `contextmenu` – when the mouse right-clicks on an element.
- `mouseover` / `mouseout` – when the mouse cursor comes over / leaves an element.
- `mousedown` / `mouseup` – when the mouse button is pressed / released over an element.
- `mousemove` – when the mouse is moved.

Form element events:

- `submit` – when the visitor submits a `<form>`.
- `focus` – when the visitor focuses on an element, e.g. on an `<input>`.

Keyboard events:

- `keydown` and `keyup` – when the visitor presses and then releases the button.

Document events:

- `DOMContentLoaded` – when the HTML is loaded and processed, DOM is fully built.

CSS events:

- `transitionend` – when a CSS-animation finishes.

There are many other events. We'll get into more details of particular events in next chapters.

Event handlers

To react on events we can assign a *handler* – a function that runs in case of an event.
Handlers are a way to run JavaScript code in case of user actions.
There are several ways to assign a handler. Let's see them, starting from the simplest one.

HTML-attribute

A handler can be set in HTML with an attribute named `on<event>`.

For instance, to assign a `click` handler for an `input`, we can use `onclick`, like here:

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

On mouse click, the code inside `onclick` runs.

Please note that inside `onclick` we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: `onclick="alert("Click!")"`, then it won't work right.

An HTML-attribute is not a convenient place to write a lot of code, so we'd better create a JavaScript function and call it there.

Here a click runs the function `countRabbits()`:

```
<script>
    function countRabbits() {
        for(let i=1; i<=3; i++) {
            alert("Rabbit number " + i);
        }
    }
</script>
<input type="button" onclick="countRabbits()" value="Count rabbits!">
```

As we know, HTML attribute names are not case-sensitive, so `ONCLICK` works as well as `onClick` and `onCLICK…` But usually attributes are lowercase: `onclick`.

DOM property

We can assign a handler using a DOM property `on<event>`.

For instance,

```
<input id="elem" type="button" value="Click me">
<script>
    elem.onclick = function() {
        alert('Thank you');
    };
</script>
```

The handler is always in the DOM property: the HTML-attribute is just one of the ways to initialize it.

These two code pieces work the same:

Only HTML:

```
<input type="button" onclick="alert('Click!')" value="Button">
```

HTML + JS:

```
<input type="button" id="button" value="Button">
<script>
    button.onclick = function() {
        alert('Click!');
    };
</script>
```

As there’s only one `onclick` property, we can’t assign more than one event handler.

In the example below adding a handler with JavaScript overwrites the existing handler:

```
<input type="button" id="elem" onclick="alert('Before')" value="Click me">
<script>
    elem.onclick = function() { // overwrites the existing handler
        alert('After'); // only this will be shown
    };
```

```
</script>
```

By the way, we can assign an existing function as a handler directly:

```
function sayThanks() {
    alert('Thanks!');
}
elem.onclick = sayThanks;
```

To remove a handler – assign `elem.onclick = null`.

Accessing the element: this

The value of `this` inside a handler is the element. The one which has the handler on it.
In the code below `button` shows its contents using `this.innerHTML`:

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

Possible mistakes

If you're starting to work with events – please note some subtleties. The function should be assigned as `sayThanks`, not `sayThanks()`.

```
// right
button.onclick = sayThanks;
```

```
// wrong
button.onclick = sayThanks();
```

If we add parentheses, `sayThanks()` – is a function call. So the last line actually takes the *result* of the function execution, that is `undefined` (as the function returns nothing), and assigns it to `onclick`. That doesn't work.

…On the other hand, in the markup we do need the parentheses:

```
<input type="button" id="button" onclick="sayThanks()">
```

Use functions, not strings.

The assignment `elem.onclick = "alert(1)"` would work too. It works for compatibility reasons, but is strongly not recommended.

Don't use `setAttribute` for handlers.

Such a call won't work:

```
// a click on <body> will generate errors,
// because attributes are always strings, function becomes a string
document.body.setAttribute('onclick', function() { alert(1) });
```

DOM-property case matters.

Assign a handler to `elem.onclick`, not `elem.ONCLICK`, because DOM properties are case-sensitive.

addEventListener

The fundamental problem of the aforementioned ways to assign handlers – we can't assign multiple handlers to one event.

For instance, one part of our code wants to highlight a button on click, and another one wants to show a message.

We'd like to assign two event handlers for that. But a new DOM property will overwrite the existing one:

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); } // replaces the previous handler
```

Web-standard developers understood that long ago and suggested an alternative way of managing handlers using special methods `addEventListener` and `removeEventListener`. They are free of such a problem.

The syntax to add a handler:

```
element.addEventListener(event, handler[, options]);
```

event

Event name, e.g. `"click"`.

handler

The handler function.

options

An additional optional object with properties:

- `once`: if `true`, then the listener is automatically removed after it triggers.
- `capture`: the phase where to handle the event, to be covered later in the chapter Bubbling and capturing. For historical reasons, options can also be `false/true`, that's the same as `{capture: false/true}`.
- `passive`: if `true`, then the handler will not `preventDefault()`, we'll cover that later in Browser default actions.

To remove the handler, use `removeEventListener`:

```
element.removeEventListener(event, handler, [options]);
```

Removal requires the same function

To remove a handler we should pass exactly the same function as was assigned.
That doesn't work:

```
elem.addEventListener( "click" , () => alert('Thanks!'));
// ....
elem.removeEventListener( "click", () => alert('Thanks!'));
```

The handler won't be removed, because removeEventListener gets another function – with the same code, but that doesn't matter.
Here's the right way:

```
function handler() {
    alert( 'Thanks!' );
}
input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

Please note – if we don't store the function in a variable, then we can't remove it. There's no way to "read back" handlers assigned by addEventListener.

Multiple calls to addEventListener allow to add multiple handlers, like this:

```
<input id="elem" type="button" value="Click me"/>
<script>
    function handler1() {
        alert('Thanks!');
    };
    function handler2() {
        alert('Thanks again!');
    }
    elem.onclick = () => alert("Hello");
    elem.addEventListener("click", handler1); // Thanks!
    elem.addEventListener("click", handler2); // Thanks again!
</script>
```

For some events, handlers only work with addEventListener

There exist events that can't be assigned via a DOM-property. Must use addEventListener. For instance, the event DOMContentLoaded, that triggers when the document is loaded and DOM is built.

```
document.onDOMContentLoaded = function() {
    alert("DOM built"); // will never run
};
 document.addEventListener("DOMContentLoaded", function() {
     alert("DOM built"); // this way it works
 });
```

Event object

To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keypress”, but what were the pointer coordinates? Which key was pressed? And so on.

When an event happens, the browser creates an *event object*, puts details into it and passes it as an argument to the handler.

Here’s an example of getting mouse coordinates from the event object:

```
<input type="button" value="Click me" id="elem">
<script>
    elem.onclick = function(event) {
        // show event type, element and coordinates of the click
        alert(event.type + " at " + event.currentTarget);
        alert("Coordinates: " + event.clientX + ":" + event.clientY);
    };
</script>
```

event.type
Event type, here it’s "click".

event.currentTarget
Element that handled the event.

event.clientX / event.clientY
Window-relative coordinates of the cursor, for mouse events.

There are more properties. They depend on the event type, so we’ll study them later when we come to different events in detail.

The event object is also accessible from HTML

If we assign a handler in HTML, we can also use the event object, like this:

```
<input type="button" onclick="alert(event.type)" value="Event type">
```

Object handlers: handleEvent

We can assign not just a function, but an object as an event handler using addEventListener. When an event occurs, its handleEvent method is called.

For instance:

```
<button id="elem">Click me</button>
<script>
  elem.addEventListener('click', {
    handleEvent(event) {
      alert(event.type + " at " + event.currentTarget);
    }
  });
</script>
```

As we can see, when addEventListener receives an object as the handler, it calls object.handleEvent(event) in case of an event.

We could also use a class for that:

```
<button id="elem">Click me</button>
<script>
  class Menu {
    handleEvent(event) {
      switch(event.type) {
        case 'mousedown':
          elem.innerHTML = "Mouse button pressed";
          break;
        case 'mouseup':
```

```
                    elem.innerHTML += "...and released.";
                    break;
            }
        }
    }
    let menu = new Menu();
    elem.addEventListener('mousedown', menu);
    elem.addEventListener('mouseup', menu);
</script>
```

Here the same object handles both events. Please note that we need to explicitly setup the events to listen using addEventListener. The menu object only gets mousedown and mouseup here, not any other types of events.

Bubbling and capturing

Let's start with an example.

This handler is assigned to `<div>`, but also runs if you click any nested tag like `<em>` or `<code>`:

```
<div onclick="alert('The handler!')">
    <em>If you click on <code>EM</code>, the handler on <code>DIV</code> runs.</em>
</div>
```

Isn't it a bit strange? Why does the handler on `<div>` run if the actual click was on `<em>`?

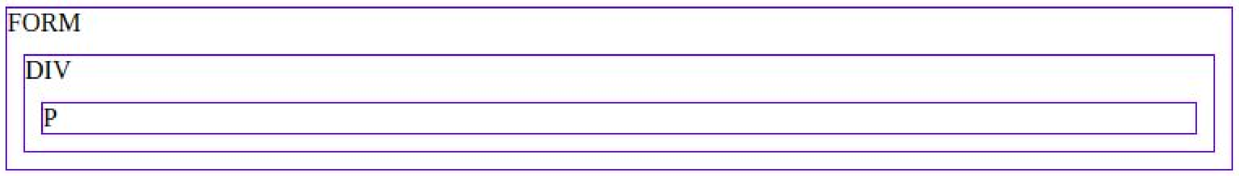
Bubbling

The bubbling principle is simple.

When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

Let's say we have 3 nested elements `FORM > DIV > P` with a handler on each of them:

```
<style>
    body * {
        margin: 10px;
        border: 1px solid blue;
    }
</style>
<form onclick="alert('form')">FORM
    <div onclick="alert('div')">DIV
        <p onclick="alert('p')">P</p>
    </div>
</form>
```



A click on the inner `<p>` first runs `onclick`:

1. On that `<p>`.
2. Then on the outer `<div>`.
3. Then on the outer `<form>`.
4. And so on upwards till the `document` object.

So if we click on `<p>`, then we'll see 3 alerts: `p` → `div` → `form`.
The process is called "bubbling", because events "bubble" from the inner element up through parents like a bubble in the water.

***Almost* all events bubble.**

The key word in this phrase is "almost".

For instance, a `focus` event does not bubble. There are other examples too, we'll meet them. But still it's an exception, rather than a rule, most events do bubble.

event.target

A handler on a parent element can always get the details about where it actually happened.

The most deeply nested element that caused the event is called a *target* element, accessible as event.target**.**

Note the differences from this (=event.currentTarget):

- event.target – is the “target” element that initiated the event, it doesn’t change through the bubbling process.
- this – is the “current” element, the one that has a currently running handler on it.

For instance, if we have a single handler form.onclick, then it can “catch” all clicks inside the form. No matter where the click happened, it bubbles up to <form> and runs the handler.
In form.onclick handler:

- this (=event.currentTarget) is the <form> element, because the handler runs on it.
- event.target is the actual element inside the form that was clicked.

It’s possible that event.target could equal this – it happens when the click is made directly on the <form> element.

Stopping bubbling

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path.

But any handler may decide that the event has been fully processed and stop the bubbling.

The method for it is event.stopPropagation().

For instance, here body.onclick doesn’t work if you click on <button>:

```
<body onclick="alert(`the bubbling doesn't reach here`)">
    <button onclick="event.stopPropagation()">Click me</button>
</body>
```

event.stopImmediatePropagation()

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.
In other words, `event.stopPropagation()` stops the move upwards, but on the current element all other handlers will run.

To stop the bubbling and prevent handlers on the current element from running, there's a method `event.stopImmediatePropagation()`. After it no other handlers execute.

Don't stop bubbling without a need!

Bubbling is convenient. Don't stop it without a real need: obvious and architecturally well thought out.
Sometimes `event.stopPropagation()` creates hidden pitfalls that later may become problems.
For instance:

1. We create a nested menu. Each submenu handles clicks on its elements and calls `stopPropagation` so that the outer menu won't trigger.
2. Later we decide to catch clicks on the whole window, to track users' behavior (where people click). Some analytic systems do that. Usually the code uses `document.addEventListener('click'…)` to catch all clicks.
3. Our analytic won't work over the area where clicks are stopped by `stopPropagation`. Sadly, we've got a "dead zone".

There's usually no real need to prevent the bubbling. A task that seemingly requires that may be solved by other means.

Capturing

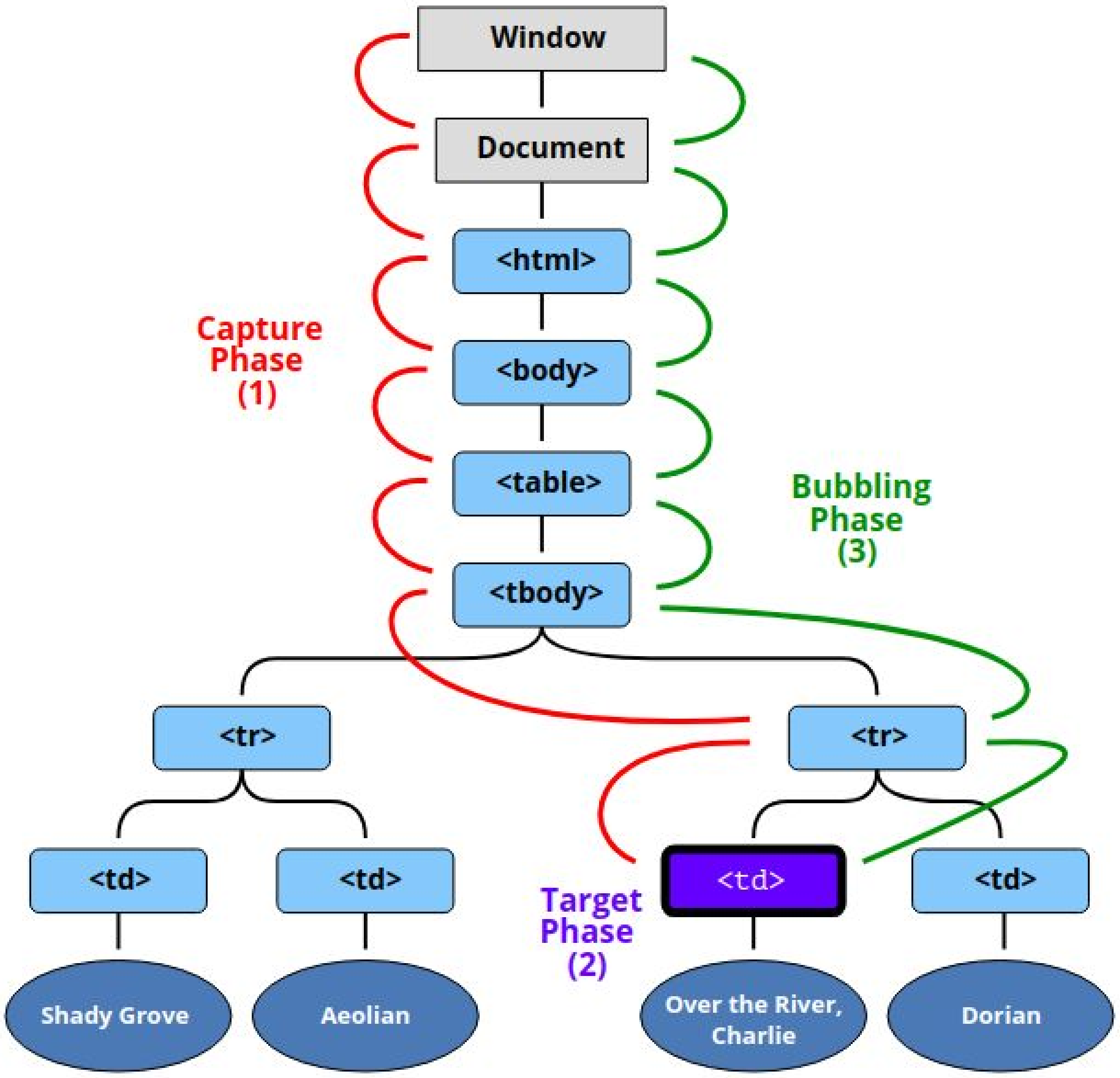
There's another phase of event processing called "capturing". It is rarely used in real code, but sometimes can be useful.

The standard DOM Events describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.

3. Bubbling phase – the event bubbles up from the element.

Here's the picture of a click on `<td>` inside a table, taken from the specification:



That is: for a click on `<td>` the event first goes through the ancestors chain down to the element (capturing phase), then it reaches the target and triggers there (target phase), and then it goes up (bubbling phase), calling handlers on its way.

Before we only talked about bubbling, because the capturing phase is rarely used. Normally it is invisible to us.

Handlers added using `on<event>`-property or using HTML attributes or using two-argument `addEventListener(event, handler)` don’t know anything about capturing, they only run on the 2nd and 3rd phases.

To catch an event on the capturing phase, we need to set the handler `capture` option to `true`:

```
elem.addEventListener(..., {capture: true})
// or, just "true" is an alias to {capture: true}
elem.addEventListener(..., true)
```

There are two possible values of the `capture` option:

- If it’s `false` (default), then the handler is set on the bubbling phase.
- If it’s `true`, then the handler is set on the capturing phase.

Note that while formally there are 3 phases, the 2nd phase (“target phase”: the event reached the element) is not handled separately: handlers on both capturing and bubbling phases trigger at that phase.

Let’s see both capturing and bubbling in action:

```
<style>
  body * {
    margin: 10px;
    border: 1px solid blue;
  }
</style>
<form>FORM
  <div>DIV
```

```
        <p>P</p>
    </div>
</form>
<script>
    for(let elem of document.querySelectorAll('*')) {
        elem.addEventListener("click", e => alert(`Capturing: ${elem.tagName}`), true);
        elem.addEventListener("click", e => alert(`Bubbling: ${elem.tagName}`));
    }
</script>
```

The code sets click handlers on *every* element in the document to see which ones are working.

If you click on `<p>`, then the sequence is:

1. HTML → BODY → FORM → DIV (capturing phase, the first listener):
2. P (target phrase, triggers two times, as we've set two listeners: capturing and bubbling)
3. DIV → FORM → BODY → HTML (bubbling phase, the second listener).

There's a property `event.eventPhase` that tells us the number of the phase on which the event was caught. But it's rarely used, because we usually know it in the handler.

To remove the handler, `removeEventListener` needs the same phase

If we `addEventListener(..., true)`, then we should mention the same phase in `removeEventListener(..., true)` to correctly remove the handler.

Listeners on same element and same phase run in their set order

If we have multiple event handlers on the same phase, assigned to the same element with `addEventListener`, they run in the same order as they are created:

```
elem.addEventListener("click", e => alert(1)); // guaranteed to trigger first
elem.addEventListener("click", e => alert(2));
```

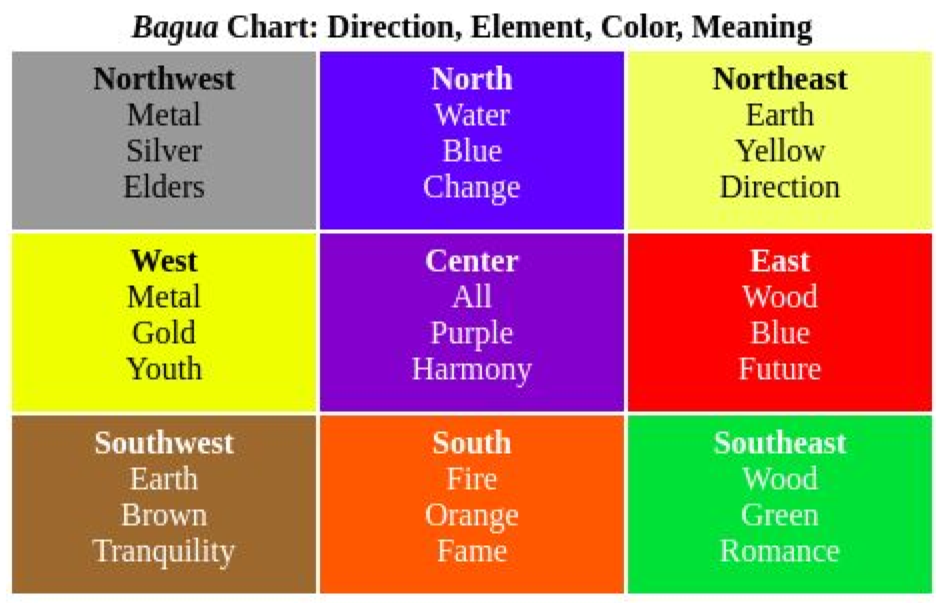
Event delegation

Capturing and bubbling allow us to implement one of most powerful event handling patterns called *event delegation*.

The idea is that if we have a lot of elements handled in a similar way, then instead of assigning a handler to each of them – we put a single handler on their common ancestor.

In the handler we get event.target, see where the event actually happened and handle it.

Let’s see an example – the Ba-Gua diagram reflecting ancient Chinese philosophy.
Here it is:



Bagua Chart: Direction, Element, Color, Meaning		
Northwest Metal<br>Silver<br>Elders	**North** Water Blue Change	**Northeast** Earth Yellow Direction
West Metal Gold Youth	**Center** All Purple Harmony	**East** Wood Blue Future
Southwest Earth Brown Tranquility	**South** Fire Orange Fame	**Southeast** Wood Green Romance

The HTML is like this:

```
<table>
  <tr>
    <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
  </tr>
  <tr>
    <td class="nw"><strong>Northwest</strong><br>Metal<br>Silver<br>Elders</td>
    <td class="n">...</td>
    <td class="ne">...</td>
  </tr>
  <tr>...2 more lines of this kind...</tr>
  <tr>...2 more lines of this kind...</tr>
</table>
```

The table has 9 cells, but there could be 99 or 9999, doesn’t matter.

Our task is to highlight a cell `<td>` on click.

Instead of assigning an `onclick` handler to each `<td>` (can be many) – we’ll setup the “catch-all” handler on `<table>`element.

It will use `event.target` to get the clicked element and highlight it.

The code:

```
let selectedTd;
table.onclick = function(event) {
    let target = event.target; // where was the click?
    if (target.tagName != 'TD') return; // not on TD? Then we're not interested
    highlight(target); // highlight it
};
function highlight(td) {
    if (selectedTd) { // remove the existing highlight if any
        selectedTd.classList.remove('highlight');
    }
    selectedTd = td;
    selectedTd.classList.add('highlight'); // highlight the new td
}
```

Such a code doesn’t care how many cells there are in the table. We can add/remove `<td>` dynamically at any time and the highlighting will still work.

Still, there’s a drawback.

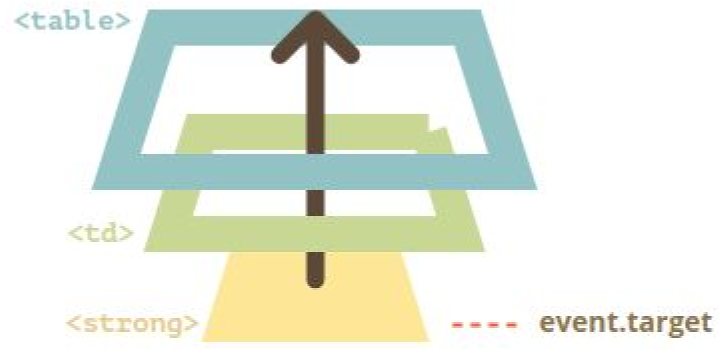
The click may occur not on the `<td>`, but inside it.

In our case if we take a look inside the HTML, we can see nested tags inside `<td>`, like

`<strong>`:

```
<td>
    <strong>Northwest</strong>
     ...
</td>
```

Naturally, if a click happens on that `<strong>` then it becomes the value of `event.target`.



in the handler `table.onclick` we should take such `event.target` and find out whether the click was inside `<td>` or not.

Here’s the improved code:

```
table.onclick = function(event) {
    let td = event.target.closest('td'); // (1)
    if (!td) return; // (2)
    if (!table.contains(td)) return; // (3)
    highlight(td); // (4)
};
```

Explanations:

1. The method elem.closest(selector) returns the nearest ancestor that matches the selector. In our case we look for <td> on the way up from the source element.
2. If event.target is not inside any <td>, then the call returns null, and we don't have to do anything.
3. In case of nested tables, event.target may be a <td> lying outside of the current table. So we check if that's actually *our table's* <td>.
4. And, if it's so, then highlight it.

As a result, we have a fast, efficient highlighting code, that doesn't care about the total number of <td> in the table.

Delegation example: actions in markup

There are other uses for event delegation.

Let's say, we want to make a menu with buttons "Save", "Load", "Search" and so on. And there's an object with methods save, load, search… How to match them?

The first idea may be to assign a separate handler to each button. But there's a more elegant solution. We can add a handler for the whole menu and data-action attributes for buttons that has the method to call:

```
<button data-action="save">Click to Save</button>
```

The handler reads the attribute and executes the method. Take a look at the working example:

```
<div id="menu">
    <button data-action="save">Save</button>
    <button data-action="load">Load</button>
    <button data-action="search">Search</button>
</div>

<script>
    class Menu {
        constructor(elem) {
            this._elem = elem;
            elem.onclick = this.onClick.bind(this); // (*)
        }
        save() {
            alert('saving');
        }
        load() {
            alert('loading');
        }
        search() {
            alert('searching');
        }
       onClick(event) {
           let action = event.target.dataset.action;
           if (action) {
               this[action]();
           }
       };
    }

    new Menu(menu);
</script>
```

Please note that this.onClick is bound to this in (*). That's important, because otherwise this inside it would reference the DOM element (elem), not the Menu object, and this[action] would not be what we need.

So, what advantages does delegation give us here?

- We don't need to write the code to assign a handler to each button. Just make a method and put it in the markup.
- The HTML structure is flexible, we can add/remove buttons at any time.

We could also use classes .action-save, .action-load, but an attribute data-action is better semantically. And we can use it in CSS rules too.

The “behavior” pattern

We can also use event delegation to add “behaviors” to elements *declaratively*, with special attributes and classes.

The pattern has two parts:

1. We add a custom attribute to an element that describes its behavior.
2. A document-wide handler tracks events, and if an event happens on an attributed element – performs the action.

Behavior: Counter

For instance, here the attribute `data-counter` adds a behavior: “increase value on click” to buttons:

```
Counter: <input type="button" value="1" data-counter>
One more counter: <input type="button" value="2" data-counter>
<script>
  document.addEventListener('click', function(event) {
    if (event.target.dataset.counter != undefined) { // if the attribute exists...
      event.target.value++;
    }
  });
</script>
```

If we click a button – its value is increased. Not buttons, but the general approach is important here.

There can be as many attributes with `data-counter` as we want. We can add new ones to HTML at any moment. Using the event delegation we “extended” HTML, added an attribute that describes a new behavior.

Browser default actions

Many events automatically lead to certain actions performed by the browser.

For instance:

- A click on a link – initiates navigation to its URL.
- A click on a form submit button – initiates its submission to the server.
- Pressing a mouse button over a text and moving it – selects the text.

If we handle an event in JavaScript, we may not want the corresponding browser action to happen, and want to implement another behavior instead.

Preventing browser actions

There are two ways to tell the browser we don’t want it to act:

- The main way is to use the `event` object. There’s a method `event.preventDefault()`.
- If the handler is assigned using `on<event>` (not by `addEventListener`), then returning `false` also works the same.

In this HTML a click on a link doesn’t lead to navigation, browser doesn’t do anything:

```
<a href="/" onclick="return false">Click here</a>
or
<a href="/" onclick="event.preventDefault()">here</a>
```

Example: the menu

Consider a site menu, like this:

```
<ul id="menu" class="menu">
    <li><a href="/html">HTML</a></li>
    <li><a href="/javascript">JavaScript</a></li>
    <li><a href="/css">CSS</a></li>
</ul>
```

Menu items are implemented as HTML-links `<a>`, not buttons `<button>`. There are several reasons to do so, for instance:

- Many people like to use “right click” – “open in a new window”. If we use `<button>` or `<span>`, that doesn’t work.
- Search engines follow `<a href="...">` links while indexing.

So we use `<a>` in the markup. But normally we intend to handle clicks in JavaScript. So we should prevent the default browser action.

Like here:

```
menu.onclick = function(event) {
   if (event.target.nodeName != 'A') return;
   let href = event.target.getAttribute('href');
   alert( href ); // ...can be loading from the server, UI generation etc
   return false; // prevent browser action (don't go to the URL)
};
```

If we omit `return false`, then after our code executes the browser will do its “default action” – navigating to the URL in `href`. And we don’t need that here, as we’re handling the click by ourselves.

Follow-up events

Certain events flow one into another. If we prevent the first event, there will be no second.

For instance, `mousedown` on an `<input>` field leads to focusing in it, and the `focus` event. If we prevent the `mousedown` event, there’s no focus.

Try to click on the first `<input>` below – the `focus` event happens. But if you click the second one, there’s no focus.

```
<input value="Focus works" onfocus="this.value=''">
<input onmousedown="return false" onfocus="this.value=''" value="Click me">
```

That’s because the browser action is canceled on mousedown. The focusing is still possible if we use another way to enter the input. For instance, the Tab key to switch from the 1st input into the 2nd. But not with the mouse click any more..

The “passive” handler option

The optional passive: true option of addEventListener signals the browser that the handler is not going to call preventDefault().

Why may that be needed?

There are some events like touchmove on mobile devices (when the user moves their finger across the screen), that cause scrolling by default, but that scrolling can be prevented using preventDefault() in the handler.

So when the browser detects such an event, it has first to process all handlers, and then if preventDefault is not called anywhere, it can proceed with scrolling. That may cause unnecessary delays and “jitters” in the UI.

The passive: true options tells the browser that the handler is not going to cancel scrolling. Then browser scrolls immediately providing a maximally fluent experience, and the event is handled by the way.

For some browsers (Firefox, Chrome), passive is true by default for touchstart and touchmove events.

event.defaultPrevented

The property event.defaultPrevented is true if the default action was prevented, and false otherwise.

There’s an interesting use case for it.

You remember in the chapter Bubbling and capturing we talked about event.stopPropagation() and why stopping bubbling is bad?

Sometimes we can use event.defaultPrevented instead, to signal other event handlers that the event was handled.

Let’s see a practical example.

By default the browser on contextmenu event (right mouse click) shows a context menu with standard options. We can prevent it and show our own, like this:

```
<button>Right-click shows browser context menu</button>
<button oncontextmenu="alert('Draw our menu'); return false">
  Right-click shows our context menu
</button>
```

Now, in addition to that context menu we’d like to implement a document-wide context menu.

Upon right click, the closest context menu should show up.

```
<p>Right-click here for the document context menu</p>
<button id="elem">Right-click here for the button context menu</button>
<script>
  elem.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Button context menu");
  };
  document.oncontextmenu = function(event) {
    event.preventDefault();
    alert("Document context menu");
  };
</script>
```

The problem is that when we click on elem, we get two menus: the button-level and (the event bubbles up) the document-level menu.

How to fix it? One of solutions is to think like: “When we handle right-click in the button handler, let’s stop its bubbling” and use event.stopPropagation():

```
<p>Right-click for the document menu</p>
<button id="elem">Right-click for the button menu (fixed with event.stopPropagation)</button>
```

```
<script>
    elem.oncontextmenu = function(event) {
        event.preventDefault();
        event.stopPropagation();
        alert("Button context menu");
    };
    document.oncontextmenu = function(event) {
        event.preventDefault();
        alert("Document context menu");
    };
</script>
```

Now the button-level menu works as intended. But the price is high. We forever deny access to information about right-clicks for any outer code, including counters that gather statistics and so on. That’s quite unwise.

An alternative solution would be to check in the `document` handler if the default action was prevented? If it is so, then the event was handled, and we don’t need to react on it.

```
<p>Right-click for the document menu (added a check for event.defaultPrevented)</p>
<button id="elem">Right-click for the button menu</button>
<script>
    elem.oncontextmenu = function(event) {
        event.preventDefault();
        alert("Button context menu");
    };
    document.oncontextmenu = function(event) {
        if (event.defaultPrevented) return;
        event.preventDefault();
        alert("Document context menu");
    };
</script>
```

Now everything also works correctly. If we have nested elements, and each of them has a context menu of its own, that would also work. Just make sure to check for `event.defaultPrevented` in each `contextmenu` handler.

event.stopPropagation() and event.preventDefault()

As we can clearly see, `event.stopPropagation()` and `event.preventDefault()` (also known as `return false`) are two different things. They are not related to each other.

Mouse events basics

In this chapter we'll get into more details about mouse events and their properties.
Please note: such events may come not only from "mouse devices", but are also from other devices, such as phones and tablets, where they are emulated for compatibility.

Mouse event types

We can split mouse events into two categories: "simple" and "complex"

Simple events

The most used simple events are:

mousedown/mouseup

Mouse button is clicked/released over an element.

mouseover/mouseout

Mouse pointer comes over/out from an element.

mousemove

Every mouse move over an element triggers that event.

contextmenu

Triggers when opening a context menu is attempted. In the most common case, that happens when the right mouse button is pressed. Although, there are other ways to open a context menu, e.g. using a special keyboard key, so it's not exactly the mouse event.

…There are several other event types too, we'll cover them later.

Complex events

click

Triggers after `mousedown` and then `mouseup` over the same element if the left mouse button was used.

dblclick

Triggers after a double click over an element.

Complex events are made of simple ones, so in theory we could live without them. But they exist, and that's good, because they are convenient.

Events order

An action may trigger multiple events.

For instance, a click first triggers `mousedown`, when the button is pressed, then `mouseup` and `click` when it's released.

In cases when a single action initiates multiple events, their order is fixed. That is, the handlers are called in the order `mousedown` → `mouseup` → `click`.

Getting the button: which

Click-related events always have the `which` property, which allows to get the exact mouse button.

It is not used for `click` and `contextmenu` events, because the former happens only on left-click, and the latter – only on right-click.

But if we track `mousedown` and `mouseup`, then we need it, because these events trigger on any button, so `which` allows to distinguish between "right-mousedown" and "left-mousedown".

There are the three possible values:

- `event.which == 1` – the left button
- `event.which == 2` – the middle button

- `event.which == 3` – the right button

The middle button is somewhat exotic right now and is very rarely used.

Modifiers: shift, alt, ctrl and meta

All mouse events include the information about pressed modifier keys.

Event properties:

- `shiftKey`: Shift
- `altKey`: Alt (or Opt for Mac)
- `ctrlKey`: Ctrl
- `metaKey`: Cmd for Mac

They are `true` if the corresponding key was pressed during the event.

For instance, the button below only works on Alt+Shift+click:

```
<button id="button">Alt+Shift+Click on me!</button>
<script>
    button.onclick = function(event) {
        if (event.altKey && event.shiftKey) {
            alert('Hooray!');
        }
    };
</script>
```

Attention: on Mac it's usually `Cmd` instead of `Ctrl`

On Windows and Linux there are modifier keys Alt, Shift and Ctrl. On Mac there's one more: Cmd, corresponding to the property `metaKey`.

In most applications, when Windows/Linux uses Ctrl, on Mac Cmd is used.

That is: where a Windows user presses Ctrl+Enter or Ctrl+A, a Mac user would press Cmd+Enter or Cmd+A, and so on.

So if we want to support combinations like Ctrl+click, then for Mac it makes sense to use Cmd+click. That’s more comfortable for Mac users.

Even if we’d like to force Mac users to Ctrl+click – that’s kind of difficult. The problem is: a left-click with Ctrl is interpreted as a *right-click* on MacOS, and it generates the `contextmenu` event, not `click` like Windows/Linux.

So if we want users of all operating systems to feel comfortable, then together with `ctrlKey` we should check `metaKey`.

For JS-code it means that we should check `if (event.ctrlKey || event.metaKey)`

Coordinates: clientX/Y, pageX/Y

All mouse events have coordinates in two flavours:

1. Window-relative: `clientX` and `clientY`.
2. Document-relative: `pageX` and `pageY`.

For instance, if we have a window of the size 500x500, and the mouse is in the left-upper corner, then `clientX` and `clientY` are 0. And if the mouse is in the center, then clientX and clientY are 250, no matter what place in the document it is, how far the document was scrolled. They are similar to `position:fixed`.

Move the mouse over the input field to see clientX/clientY (the example is in the iframe, so coordinates are relative to that iframe):

```
<input onmousemove="this.value=event.clientX+':'+event.clientY" value="Mouse over me">
```

Document-relative coordinates `pageX`, `pageY` are counted from the left-upper corner of the document, not the window. You can read more about coordinates in the chapter Coordinates.

Disabling selection

Double mouse click has a side-effect that may be disturbing in some interfaces: it selects the text.

For instance, a double-click on the text below selects it in addition to our handler:

```
<span ondblclick="alert('dblclick')">Double-click me</span>
```

If one presses the left mouse button and, without releasing it, moves the mouse, that also makes the selection, often unwanted.

There are multiple ways to prevent the selection, that you can read in the chapter Selection and Range.

In this particular case the most reasonable way is to prevent the browser action on `mousedown`. It prevents both these selections:

```
Before...
<b ondblclick="alert('Click!')" onmousedown="return false">
    Double-click me
</b>
...After
```

Now the bold element is not selected on double clicks, and pressing the left button on it won't start the selection.

Please note: the text inside it is still selectable. However, the selection should start not on the text itself, but before or after it. Usually that's fine for users.

Preventing copying

If we want to disable selection to protect our page content from copy-pasting, then we can use another event: `oncopy`.

```
<div oncopy="alert('Copying forbidden!');return false">
    Dear user, The copying is forbidden for you. If you know JS or HTML, then you can get everything from the page source though.
</div>
```

If you try to copy a piece of text in the `<div>`, that won't work, because the default action `oncopy` is prevented.

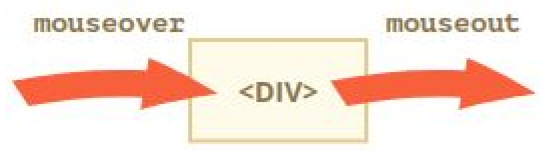
Surely the user has access to HTML-source of the page, and can take the content from there, but not everyone knows how to do it.

Moving the mouse: mouseover/out, mouseenter/leave

Let’s dive into more details about events that happen when the mouse moves between elements.

Events mouseover/mouseout, relatedTarget

The mouseover event occurs when a mouse pointer comes over an element, and mouseout – when it leaves.



These events are special, because they have property relatedTarget. This property complements the target. When a mouse leaves one element for another, one of them becomes a target, and the other one – relatedTarget.

For mouseover:

- event.target – is the element where the mouse came over.
- event.relatedTarget – is the element from which the mouse came (relatedTarget → target).

For mouseout the reverse:

- event.target – is the element that the mouse left.
- event.relatedTarget – is the new under-the-pointer element, that mouse left for (target → relatedTarget).

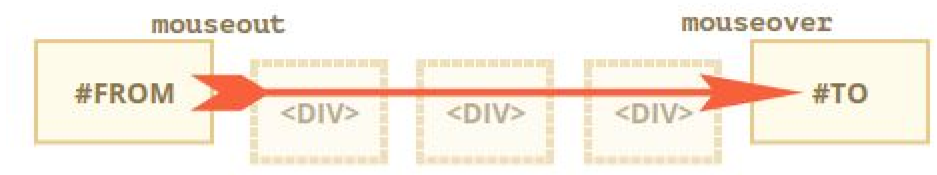
relatedTarget can be null

The relatedTarget property can be null.
That’s normal and just means that the mouse came not from another element, but from out of the window. Or that it left the window.

We should keep that possibility in mind when using event.relatedTarget in our code. If we access event.relatedTarget.tagName, then there will be an error.

Skipping elements

The mousemove event triggers when the mouse moves. But that doesn't mean that every pixel leads to an event.
The browser checks the mouse position from time to time. And if it notices changes then triggers the events.
That means that if the visitor is moving the mouse very fast then some DOM-elements may be skipped:

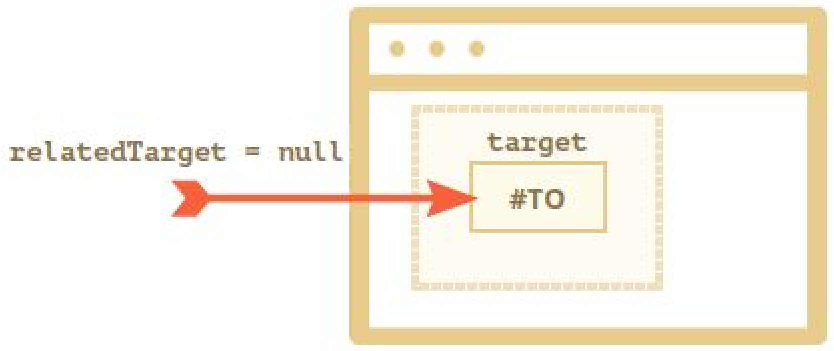


If the mouse moves very fast from #FROM to #TO elements as painted above, then intermediate <div> elements (or some of them) may be skipped. The mouseout event may trigger on #FROM and then immediately mouseover on #TO.

That's good for performance, because there may be many intermediate elements. We don't really want to process in and out of each one.

On the other hand, we should keep in mind that the mouse pointer doesn't "visit" all elements along the way. It can "jump".

In particular, it's possible that the pointer jumps right inside the middle of the page from out of the window. In that case relatedTarget is null, because it came from "nowhere":



You can check it out “live” on a teststand below.

Its HTML has two nested elements: the `<div id="child">` is inside the `<div id="parent">`. If you move the mouse fast over them, then maybe only the child div triggers events, or maybe the parent one, or maybe there will be no events at all.

Also move the pointer into the child div, and then move it out quickly down through the parent one. If the movement is fast enough, then the parent element is ignored. The mouse will cross the parent element without noticing it.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
         #parent {
             background: #99C0C3;
             width: 160px;
             height: 120px;
             position: relative;
        }
        #child {
            background: #FFDE99;
            width: 50%;
            height: 50%;
            position: absolute;
            left: 50%;
            top: 50%;
            transform: translate(-50%, -50%);
        }
        textarea {
            height: 140px;
            width: 300px;
            display: block;
        }
    </style>
</head>
<body>
    <div id="parent">parent
        <div id="child">child</div>
    </div>
    <textarea id="text"></textarea>
    <input onclick="clearText()" value="Clear" type="button">
    <script>
        let parent = document.getElementById('parent');
        parent.onmouseover = parent.onmouseout = parent.onmousemove = handler;
        function handler(event) {
            let type = event.type;
            while (type < 11) type += ' ';
            log(type + " target=" + event.target.id)
            return false;
        }
        function clearText() {
            text.value = "";
            lastMessage = "";
        }
        let lastMessageTime = 0;
        let lastMessage = "";
        let repeatCounter = 1;
        function log(message) {
            if (lastMessageTime == 0) lastMessageTime = new Date();
            let time = new Date();
            if (time - lastMessageTime > 500) {
                message = '------------------------------\n' + message;
            }
            if (message === lastMessage) {
                repeatCounter++;
                if (repeatCounter == 2) {
                    text.value = text.value.trim() + ' x 2\n';
                } else {
                    text.value = text.value.slice(0, text.value.lastIndexOf('x')+1) + repeatCounter +
"\n";
                }
            } else {
                repeatCounter = 1;
                text.value += message + "\n";
            }
            text.scrollTop = text.scrollHeight;
            lastMessageTime = time;
            lastMessage = message;
        }
    </script>
</body>
</html>
```

If mouseover triggered, there must be mouseout

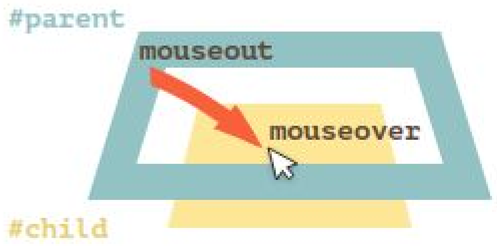
In case of fast mouse movements, intermediate elements may be ignored, but one thing we know for sure: if the pointer “officially” entered an element (mouseover event generated), then upon leaving it we always get mouseout.

Mouseout when leaving for a child

An important feature of mouseout – it triggers, when the pointer moves from an element to its descendant, e.g. from #parent to #child in this HTML:

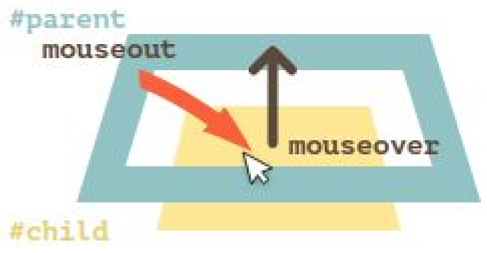
```
<div id="parent">
    <div id="child">...</div>
</div>
```

If we're on #parent and then move the pointer deeper into #child, but we get mouseout on #parent!



That may seem strange, but can be easily explained.

According to the browser logic, the mouse cursor may be only over a *single* element at any time – the most nested one and top by z-index.
So if it goes to another element (even a descendant), then it leaves the previous one.
Please note another important detail of event processing.
The mouseover event on a descendant bubbles up. So, if #parent has mouseover handler, it triggers:



You can see that very well in the example below: <div id="child"> is inside the <div id="parent">. There are mouseover/out handlers on #parent element that output event details.

If you move the mouse from #parent to #child, you see two events on #parent:

1. mouseout [target: parent] (left the parent), then
2. mouseover [target: child] (came to the child, bubbled).

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        #parent {
            background: #99C0C3;
            width: 160px;
            height: 120px;
            position: relative;
        }
        #child {
            background: #FFDE99;
            width: 50%;
            height: 50%;
            position: absolute;
            left: 50%;
            top: 50%;
            transform: translate(-50%, -50%);
        }
        textarea {
            height: 140px;
            width: 300px;
            display: block;
        }
    </style>
</head>
<body>
    <div id="parent" onmouseover="mouselog(event)" onmouseout="mouselog(event)">parent
        <div id="child">child</div>
    </div>
    <textarea id="text"></textarea>
    <input type="button" onclick="text.value=''" value="Clear">
    <script>
        function mouselog(event) {
            let d = new Date();
            text.value += `${d.getHours()}:${d.getMinutes()}:${d.getSeconds()} | ${event.type}
[target: ${event.target.id}]\n`.replace(/(:|^)(\d\D)/, '$10$2');
            text.scrollTop = text.scrollHeight;
        }
    </script>
</body>
</html>
```

As shown, when the pointer moves from #parent element to #child, two handlers trigger on the parent element: mouseout and mouseover:

```
parent.onmouseout = function(event) {
    /* event.target: parent element */
};
parent.onmouseover = function(event) {
    /* event.target: child element (bubbled) */
};
```

If we don't examine event.target inside the handlers, then it may seem that the mouse pointer left #parent element, and then immediately came back over it.

But that's not the case! The pointer is still over the parent, it just moved deeper into the child element.

If there are some actions upon leaving the parent element, e.g. an animation runs in parent.onmouseout, we usually don't want it when the pointer just goes deeper into #parent.

To avoid it, we can check relatedTarget in the handler and, if the mouse is still inside the element, then ignore such event.

Alternatively we can use other events: mouseenter and mouseleave, that we'll be covering now, as they don't have such problems.

Events mouseenter and mouseleave

Events mouseenter/mouseleave are like mouseover/mouseout. They trigger when the mouse pointer enters/leaves the element.

But there are two important differences:

1. Transitions inside the element, to/from descendants, are not counted.
2. Events mouseenter/mouseleave do not bubble.

These events are extremely simple.

When the pointer enters an element – mouseenter triggers. The exact location of the pointer inside the element or its descendants doesn't matter.

When the pointer leaves an element – mouseleave triggers.

This example is similar to the one above, but now the top element has mouseenter/mouseleave instead of mouseover/mouseout.

As you can see, the only generated events are the ones related to moving the pointer in and out of the top element. Nothing happens when the pointer goes to the child and back. Transitions between descendants are ignored

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        #parent {
            background: #99C0C3;
            width: 160px;
            height: 120px;
            position: relative;
        }
        #child {
            background: #FFDE99;
            width: 50%;
            height: 50%;
            position: absolute;
            left: 50%;
            top: 50%;
            transform: translate(-50%, -50%);
        }
        textarea {
            height: 140px;
            width: 300px;
            display: block;
        }
    </style>
</head>
<body>
    <div id="parent" onmouseenter="mouselog(event)"onmouseleave="mouselog(event)">parent
        <div id="child">child</div>
    </div>
    <textarea id="text"></textarea>
        <input type="button" onclick="text.value=''" value="Clear">
        <script>
            function mouselog(event) {
                let d = new Date();
                text.value += `${d.getHours()}:${d.getMinutes()}:${d.getSeconds()} | ${event.type}
[target: ${event.target.id}]\n`.replace(/(:|^)(\d\D)/, '$10$2');
                text.scrollTop = text.scrollHeight;
            }
        </script>
</body>
</html>
```

Event delegation

Events mouseenter/leave are very simple and easy to use. But they do not bubble. So we can't use event delegation with them.

Imagine we want to handle mouse enter/leave for table cells. And there are hundreds of cells.

The natural solution would be – to set the handler on <table> and process events there. But mouseenter/leave don't bubble. So if such event happens on <td>, then only a handler on that <td> is able to catch it.

Handlers for mouseenter/leave on <table> only trigger when the pointer enters/leaves the table as a whole. It's impossible to get any information about transitions inside it.

So, let's use mouseover/mouseout.

Let's start with simple handlers that highlight the element under mouse:

```
// let's highlight an element under the pointer
table.onmouseover = function(event) {
    let target = event.target;
    target.style.background = 'pink';
};
table.onmouseout = function(event) {
    let target = event.target;
    target.style.background = '';
};
```

Here they are in action. As the mouse travels across the elements of this table, the current one is highlighted:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        #text {
            display: block;
            height: 100px;
            width: 456px;
        }
        #table th {
            text-align: center;
            font-weight: bold;
        }
        #table td {
            width: 150px;
            white-space: nowrap;
            text-align: center;
            vertical-align: bottom;
            padding-top: 5px;
            padding-bottom: 12px;
            cursor: pointer;
        }
        #table .nw {
            background: #999;
        }
        #table .n {
            background: #03f;
            color: #fff;
        }
        #table .ne {
            background: #ff6;
        }
        #table .w {
            background: #ff0;
        }
        #table .c {
            background: #60c;
            color: #fff;
        }
        #table .e {
            background: #09f;
            color: #fff;
        }
        #table .sw {
            background: #963;
            color: #fff;
        }
        #table .s {
            background: #f60;
            color: #fff;
        }
        #table .se {
            background: #0c3;
            color: #fff;
        }
        #table .highlight {
            background: red;
        }
    </style>
</head>
<body>
    <table id="table">
        <tr>
            <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
        </tr>
        <tr>
            <td class="nw">
                <strong>Northwest</strong>
                <br>Metal
                <br>Silver
                <br>Elders
            </td>
            <td class="n"><strong>North</strong>
                <br>Water
                <br>Blue
                <br>Change
            </td>
            <td class="ne"><strong>Northeast</strong>
                <br>Earth
                <br>Yellow
                <br>Direction
            </td>
        </tr>
        <tr>
            <td class="w"><strong>West</strong>
                <br>Metal
                <br>Gold
                <br>Youth
            </td>
            <td class="c"><strong>Center</strong>
                <br>All
                <br>Purple
                <br>Harmony
            </td>
            <td class="e"><strong>East</strong>
                <br>Wood
                <br>Blue
                <br>Future
            </td>
        </tr>
        <tr>
            <td class="sw"><strong>Southwest</strong>
```

```
              <br>Earth
              <br>Brown
              <br>Tranquility
          </td>
          <td class="s"><strong>South</strong>
              <br>Fire
              <br>Orange
              <br>Fame
          </td>
          <td class="se"><strong>Southeast</strong>
              <br>Wood
              <br>Green
              <br>Romance
          </td>
      </tr>
  </table>
  <textarea id="text"></textarea>
  <input type="button" onclick="text.value=''" value="Clear">

 <script>
     table.onmouseover = function(event) {
         let target = event.target;
         target.style.background = 'pink';
         text.value += `over -> ${target.tagName}\n`;
         text.scrollTop = text.scrollHeight;
      };
      table.onmouseout = function(event) {
          let target = event.target;
          target.style.background = '';
          text.value += `out <- ${target.tagName}\n`;
          text.scrollTop = text.scrollHeight;
      };
  </script>
```

```
</body>
</html>
```

In our case we’d like to handle transitions between table cells `<td>`: entering a cell and leaving it. Other transitions, such as inside the cell or outside of any cells, don’t interest us. Let’s filter them out.

Here’s what we can do:

- Remember the currently highlighted `<td>` in a variable, let’s call it `currentElem`.
- On `mouseover` – ignore the event if we’re still inside the current `<td>`.
- On `mouseout` – ignore if we didn’t leave the current `<td>`.

Here’s an example of code that accounts for all possible situations:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <meta http-equiv="X-UA-Compatible" content="ie=edge">
   <title>Document</title>
   <style>
       #text {
           display: block;
           height: 100px;
           width: 456px;
       }
       #table th {
           text-align: center;
           font-weight: bold;
       }
       #table td {
           width: 150px;
           white-space: nowrap;
           text-align: center;
```

```
        vertical-align: bottom;
        padding-top: 5px;
        padding-bottom: 12px;
        cursor: pointer;
    }
    #table .nw {
        background: #999;
    }
    #table .n {
        background: #03f;
        color: #fff;
    }
    #table .ne {
        background: #ff6;
    }
    #table .w {
        background: #ff0;
    }
    #table .c {
        background: #60c;
        color: #fff;
    }
    #table .e {
        background: #09f;
        color: #fff;
    }
    #table .sw {
        background: #963;
        color: #fff;
    }
    #table .s {
        background: #f60;
        color: #fff;
        }
        #table .se {
            background: #0c3;
            color: #fff;
        }
        #table .highlight {
            background: red;
        }
    </style>
</head>
<body>
    <table id="table">
        <tr>
            <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
        </tr>
        <tr>
            <td class="nw"><strong>Northwest</strong>
                <br>Metal
                <br>Silver
                <br>Elders
            </td>
            <td class="n"><strong>North</strong>
                <br>Water
                <br>Blue
                <br>Change
             </td>
             <td class="ne"><strong>Northeast</strong>
                 <br>Earth
                 <br>Yellow
                 <br>Direction
             </td>
         </tr>
         <tr>
```

```
            <td class="w"><strong>West</strong>
                <br>Metal
                <br>Gold
                <br>Youth
            </td>
            <td class="c"><strong>Center</strong>
                <br>All
                <br>Purple
                <br>Harmony
            </td>
            <td class="e"><strong>East</strong>
                <br>Wood
                <br>Blue
                <br>Future
            </td>
        </tr>
        <tr>
            <td class="sw"><strong>Southwest</strong>
                <br>Earth
                <br>Brown
                <br>Tranquility
            </td>
            <td class="s"><strong>South</strong>
                <br>Fire
                <br>Orange
                <br>Fame
            </td>
            <td class="se"><strong>Southeast</strong>
                <br>Wood
                <br>Green
                <br>Romance
            </td>
        </tr>
```

```
</table>
<textarea id="text"></textarea>
<input type="button" onclick="text.value=''" value="Clear">
<script>
    // <td> under the mouse right now (if any)
    let currentElem = null;
    table.onmouseover = function(event) {
        // before entering a new element, the mouse always leaves the previous one
        // if currentElem is set, we didn't leave the previous <td>,
        // that's a mouseover inside it, ignore the event
        if (currentElem) return;
        let target = event.target.closest('td');
        // we moved not into a <td> - ignore
        if (!target) return;
        // moved into <td>, but outside of our table (possible in case of nested tables)
        // ignore
        if (!table.contains(target)) return;
        // hooray! we entered a new <td>
        currentElem = target;
        onEnter(currentElem);
    };
    table.onmouseout = function(event) {
        // if we're outside of any <td> now, then ignore the event
        // that's probably a move inside the table, but out of <td>,
        // e.g. from <tr> to another <tr>
        if (!currentElem) return;
        // we're leaving the element – where to? Maybe to a descendant?
        let relatedTarget = event.relatedTarget;
        while (relatedTarget) {
            // go up the parent chain and check – if we're still inside currentElem
            // then that's an internal transition – ignore it
```

```
                if (relatedTarget == currentElem) return;
                relatedTarget = relatedTarget.parentNode;
            }
            // we left the <td>. really.
            onLeave(currentElem);
            currentElem = null;
        };
        function onEnter(elem) {
            elem.style.background = 'pink';
            // show that in textarea
            text.value += `over -> ${currentElem.tagName}.${currentElem.className}\n`;
            text.scrollTop = 1e6;
        }
        function onLeave(elem) {
            elem.style.background = '';
            // show that in textarea
            text.value += `out <- ${elem.tagName}.${elem.className}\n`;
            text.scrollTop = 1e6;
        }
    </script>
</body>
</html>
```

Try to move the cursor in and out of table cells and inside them. Fast or slow – doesn't matter. Only `<td>` as a whole is highlighted, unlike the example before.

Drag'n'Drop with mouse events

Drag'n'Drop is a great interface solution. Taking something and dragging and dropping it is a clear and simple way to do many things, from copying and moving documents (as in file managers) to ordering (dropping items into a cart).

In the modern HTML standard there’s a section about Drag and Drop with special events such as dragstart, dragend, and so on.

Drag’n’Drop algorithm

The basic Drag’n’Drop algorithm looks like this:

1. On mousedown – prepare the element for moving, if needed (maybe create a copy of it).
2. Then on mousemove move it by changing left/top and position:absolute.
3. On mouseup – perform all actions related to a finished Drag’n’Drop.

Here’s the algorithm for drag’n’drop of a ball:

```
ball.onmousedown = function(event) { // (1) start the process
  // (2) prepare to moving: make absolute and on top by z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  document.body.append(ball);
  // ...and put that absolutely positioned ball under the pointer
  moveAt(event.pageX, event.pageY);

  // centers the ball at (pageX, pageY) coordinates
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }
  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }
  // (3) move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);
  // (4) drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
    };
};
```

For the result, visit https://javascript.info/mouse-drag-and-drop

Try to drag'n'drop the mouse and you'll see such behavior.
That's because the browser has its own Drag'n'Drop for images and some other elements that runs automatically and conflicts with ours.

To disable it:

```
ball.ondragstart = function() {
    return false;
};
```

Now everything will be alright.

For the result, visit https://javascript.info/mouse-drag-and-drop

Potential drop targets (droppables)

In previous examples the ball could be dropped just “anywhere” to stay. In real-life we usually take one element and drop it onto another. For instance, a “file” into a “folder” or something else.

Speaking abstractly, we take a “draggable” element and drop it onto a “droppable” element.

We need to know:

- where the element was dropped at the end of Drag'n'Drop – to do the corresponding action,
- and, preferably, know the droppable we're dragging over, to highlight it.

The solution is kind-of interesting and just a little bit tricky, so let's cover it here.

What may be the first idea? Probably to set `mouseover/mouseup` handlers on potential droppables?

But that doesn't work.

The problem is that, while we’re dragging, the draggable element is always above other elements. And mouse events only happen on the top element, not on those below it.

For instance, below are two `<div>` elements, red one on top of the blue one (fully covers). There’s no way to catch an event on the blue one, because the red is on top:

```
<style>
  div {
    width: 50px;
    height: 50px;
    position: absolute;
    top: 0;
  }
</style>
<div style="background:blue" onmouseover="alert('never works')"></div>
<div style="background:red" onmouseover="alert('over red!')"></div>
```

The same with a draggable element. The ball is always on top over other elements, so events happen on it. Whatever handlers we set on lower elements, they won’t work.

That’s why the initial idea to put handlers on potential droppables doesn’t work in practice. They won’t run.

So, what to do?

There’s a method called `document.elementFromPoint(clientX, clientY)`. It returns the most nested element on given window-relative coordinates (or `null` if given coordinates are out of the window).

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <style>
    #gate {
      cursor: pointer;
```

```
            margin-bottom: 100px;
            width: 83px;
            height: 46px;
        }
        #ball {
            cursor: pointer;
            width: 40px;
            height: 40px;
        }
    </style>
</head>
<body>
    <p>Drag the ball.</p>
    <img src="https://en.js.cx/clipart/soccer-gate.svg" id="gate" class="droppable">
    <img src="https://en.js.cx/clipart/ball.svg" id="ball">
    <script>
        let currentDroppable = null;
        ball.onmousedown = function(event) {
            let shiftX = event.clientX - ball.getBoundingClientRect().left;
            let shiftY = event.clientY - ball.getBoundingClientRect().top;
            ball.style.position = 'absolute';
            ball.style.zIndex = 1000;
            document.body.append(ball);
            moveAt(event.pageX, event.pageY);
            function moveAt(pageX, pageY) {
                ball.style.left = pageX - shiftX + 'px';
                ball.style.top = pageY - shiftY + 'px';
            }
            function onMouseMove(event) {
                moveAt(event.pageX, event.pageY);
                ball.hidden = true;
                let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
                ball.hidden = false;
                if (!elemBelow) return;

                let droppableBelow = elemBelow.closest('.droppable');
                if (currentDroppable != droppableBelow) {
                    if (currentDroppable) { // null when we were not over a droppable before this
event
                        leaveDroppable(currentDroppable);
                    }
                    currentDroppable = droppableBelow;
                    if (currentDroppable) { // null if we're not coming over a droppable now
                        // (maybe just left the droppable)
                        enterDroppable(currentDroppable);
                    }
                }
            }
            document.addEventListener('mousemove', onMouseMove);
            ball.onmouseup = function() {
                document.removeEventListener('mousemove', onMouseMove);
                ball.onmouseup = null;
            };
       };
        function enterDroppable(elem) {
            elem.style.background = 'pink';
        }
        function leaveDroppable(elem) {
            elem.style.background = '';
        }
        ball.ondragstart = function() {
            return false;
        };
    </script>
</body>
</html>
```

Now we have the current “drop target”, that we’re flying over, in the variable currentDroppable during the whole process and can use it to highlight or any other stuff.

Keyboard: keydown and keyup

Before we get to the keyboard, please note that on modern devices there are other ways to “input something”. For instance, people use speech recognition (especially on mobile devices) or copy/paste with the mouse.

So if we want to track any input into an <input> field, then keyboard events are not enough. There’s another event named input to track changes of an <input> field, by any means. And it may be a better choice for such a task. We’ll cover it later in the chapter Events: change, input, cut, copy, paste.

Keyboard events should be used when we want to handle keyboard actions (virtual keyboard also counts). For instance, to react on arrow keys Up and Down or hotkeys (including combinations of keys).

Teststand

To better understand keyboard events, you can use the teststand below.

Try different key combinations in the text field.

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <style>
        #kinput {
            font-size: 150%;
            box-sizing: border-box;
            width: 95%;
        }
        #area {
            width: 95%;
            box-sizing: border-box;
            height: 250px;
            border: 1px solid black;
            display: block;
        }
        form label {
            display: inline;
            white-space: nowrap;
        }
    </style>
</head>
<body>
    <form id="form" onsubmit="return false">
        Prevent default for:
        <label>
            <input type="checkbox" name="keydownStop" value="1">  keydown
        </label>
        <label>
            <input type="checkbox" name="keyupStop" value="1"> keyup
        </label>
        <p>
            Ignore:
            <label>
                <input type="checkbox" name="keydownIgnore" value="1">  keydown
            </label>
            <label>
                <input type="checkbox" name="keyupIgnore" value="1"> keyup
            </label>
        </p>
        <p>Focus on the input field and press a key.</p>
        <input type="text" placeholder="Press keys here" id="kinput">
        <textarea id="area"></textarea>
        <input type="button" value="Clear" onclick="area.value = ''" />
    </form>
    <script>
        kinput.onkeydown = kinput.onkeyup = kinput.onkeypress = handle;
        let lastTime = Date.now();
        function handle(e) {
            if (form.elements[e.type + 'Ignore'].checked) return;
            let text = e.type +
                        ' key=' + e.key +
                        ' code=' + e.code +
                        (e.shiftKey ? ' shiftKey' : '') +
                        (e.ctrlKey ? ' ctrlKey' : '') +
                        (e.altKey ? ' altKey' : '') +
                        (e.metaKey ? ' metaKey' : '') +
                        (e.repeat ? ' (repeat)' : '') +
                        "\n";
            if (area.value && Date.now() - lastTime > 250) {
                area.value += new Array(81).join('-') + '\n';
            }
            lastTime = Date.now();
            area.value += text;
            if (form.elements[e.type + 'Stop'].checked) {
                e.preventDefault();
            }
        }
    </script>
</body>
</html>
```

Keydown and keyup

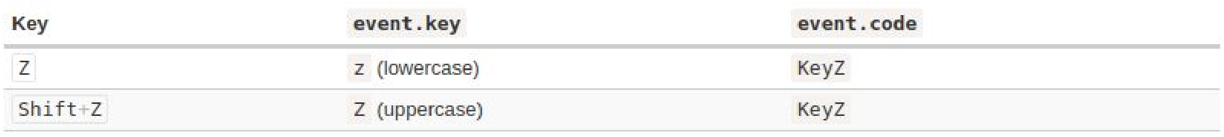
The keydown events happens when a key is pressed down, and then keyup – when it’s released.

event.code and event.key

The key property of the event object allows to get the character, while the code property of the event object allows to get the “physical key code”.

For instance, the same key Z can be pressed with or without Shift. That gives us two different characters: lowercase z and uppercase Z.

The event.key is exactly the character, and it will be different. But event.code is the same:



Key	event.key	event.code
Z	z (lowercase)	KeyZ
Shift+Z	Z (uppercase)	KeyZ

If a user works with different languages, then switching to another language would make a totally different character instead of "Z". That will become the value of event.key, while event.code is always the same: "KeyZ".

“KeyZ” and other key codes

Every key has the code that depends on its location on the keyboard. Key codes described in the UI Events code specification.

For instance:

- Letter keys have codes "Key<letter>": "KeyA", "KeyB" etc.
- Digit keys have codes: "Digit<number>": "Digit0", "Digit1" etc.
- Special keys are coded by their names: "Enter", "Backspace", "Tab" etc.

There are several widespread keyboard layouts, and the specification gives key codes for each of them.

Read the alphanumeric section of the spec for more codes, or just press a key in the teststand above.

What if a key does not give any character? For instance, Shift or F1 or others. For those keys, event.key is approximately the same as event.code:

Key	event.key	event.code
F1	F1	F1
Backspace	Backspace	Backspace
Shift	Shift	ShiftRight or ShiftLeft

Please note that event.code specifies exactly which key is pressed. For instance, most keyboards have two Shift keys: on the left and on the right side. The event.code tells us exactly which one was pressed, and event.key is responsible for the "meaning" of the key: what it is (a "Shift").

Let's say, we want to handle a hotkey: Ctrl+Z (or Cmd+Z for Mac). Most text editors hook the "Undo" action on it. We can set a listener on keydown and check which key is pressed.

There's a dilemma here: in such a listener, should we check the value of event.key or event.code?

On one hand, the value of event.key is a character, it changes depending on the language. If the visitor has several languages in OS and switches between them, the same key gives different characters. So it makes sense to check event.code, it's always the same.

Like this:

```
document.addEventListener('keydown', function(event) {
    if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
        alert('Undo!')
    }
});
```

For the same key, US layout has "Z", while German layout has "Y" (letters are swapped).
Literally, event.code will equal KeyZ for people with German layout when they press Y.
If we check event.code == 'KeyZ' in our code, then for people with German layout such test will pass when they press Y.

That sounds really odd, but so it is. The specification explicitly mentions such behavior.

So, event.code may match a wrong character for unexpected layout. Same letters in different layouts may map to different physical keys, leading to different codes. Luckily, that happens only with several codes, e.g. keyA, keyQ, keyZ (as we've seen), and doesn't happen with special keys such as Shift. You can find the list in the specification.

To reliably track layout-dependent characters, event.key may be a better way.

On the other hand, event.code has the benefit of staying always the same, bound to the physical key location, even if the visitor changes languages. So hotkeys that rely on it work well even in case of a language switch.

Do we want to handle layout-dependant keys? Then event.key is the way to go.

Or we want a hotkey to work even after a language switch? Then event.code may be better.

Auto-repeat

If a key is being pressed for a long enough time, it starts to "auto-repeat": the keydown triggers again and again, and then when it's released we finally get keyup. So it's kind of normal to have many keydown and a single keyup.

For events triggered by auto-repeat, the event object has event.repeat property set to true.

Default actions

Default actions vary, as there are many possible things that may be initiated by the keyboard.

For instance:

- A character appears on the screen (the most obvious outcome).
- A character is deleted (Delete key).
- The page is scrolled (PageDown key).
- The browser opens the "Save Page" dialog (Ctrl+S)
- …and so on.

Preventing the default action on keydown can cancel most of them, with the exception of OS-based special keys. For instance, on Windows Alt+F4 closes the current browser window. And there's no way to stop it by preventing the default action in JavaScript.

For instance, the `<input>` below expects a phone number, so it does not accept keys except digits, `+`, `()` or `-`:

```
<script>
    function checkPhoneKey(key) {
        return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-';
    }
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please"
type="tel">
```

Please note that special keys, such as Backspace, Left, Right, Ctrl+V, do not work in the input. That’s a side-effect of the strict filter `checkPhoneKey`.

Let’s relax it a little bit:

```
<script>
    function checkPhoneKey(key) {
        return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-' ||
    key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' || key == 'Backspace';
    }
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please"
type="tel">
```

Now arrows and deletion works well.

…But we still can enter anything by using a mouse and right-click + Paste. So the filter is not 100% reliable. We can just let it be like that, because most of time it works. Or an alternative approach would be to track the `input` event – it triggers after any modification. There we can check the new value and highlight/modify it when it’s invalid.

Legacy

In the past, there was a `keypress` event, and also `keyCode`, `charCode`, `which` properties of the event object.

There were so many browser incompatibilities while working with them, that developers of the specification had no way, other than deprecating all of them and creating new, modern events (described above in this chapter). The old code still works, as browsers keep supporting them, but there's totally no need to use those any more.

Scrolling

The scroll event allows users to react on a page or element scrolling. There are quite a few good things we can do here.

For instance:

- Show/hide additional controls or information depending on where in the document the user is.
- Load more data when the user scrolls down till the end of the page.

Here's a small function to show the current scroll:

```
window.addEventListener('scroll', function() {
    document.getElementById('showScroll').innerHTML = window.pageYOffset + 'px';
});
```

The `scroll` event works both on the `window` and on scrollable elements.

Prevent scrolling

How do we make something unscrollable?

We can't prevent scrolling by using `event.preventDefault()` in `onscroll` listener, because it triggers *after* the scroll has already happened.

But we can prevent scrolling by `event.preventDefault()` on an event that causes the scroll, for instance `keydown` event for pageUp and pageDown.

If we add an event handler to these events and `event.preventDefault()` in it, then the scroll won't start.

Form properties and methods

Forms and control elements, such as `<input>` have a lot of special properties and events. Working with forms will be much more convenient when we learn them.

Navigation: form and elements

Document forms are members of the special collection `document.forms`.

That’s a so-called “named collection”: it’s both named and ordered. We can use both the name or the number in the document to get the form.

```
document.forms.my - the form with name="my"
document.forms[0] - the first form in the document
```

When we have a form, then any element is available in the named collection `form.elements`.

For instance:

```
<form name="my">
    <input name="one" value="1">
    <input name="two" value="2">
</form>
<script>
    // get the form
    let form = document.forms.my; // <form name="my"> element
    // get the element
    let elem = form.elements.one; // <input name="one"> element
    alert(elem.value); // 1
</script>
```

There may be multiple elements with the same name, that’s often the case with radio buttons. In that case `form.elements[name]` is a collection, for instance:

```
<form>
    <input type="radio" name="age" value="10">
    <input type="radio" name="age" value="20">
```

```
</form>
<script>
    let form = document.forms[0];
    let ageElems = form.elements.age;
    alert(ageElems[0]); // [object HTMLInputElement]
</script>
```

These navigation properties do not depend on the tag structure. All control elements, no matter how deep they are in the form, are available in form.elements.

Fieldsets as “subforms”

A form may have one or many <fieldset> elements inside it. They also have elements property that lists form controls inside them.

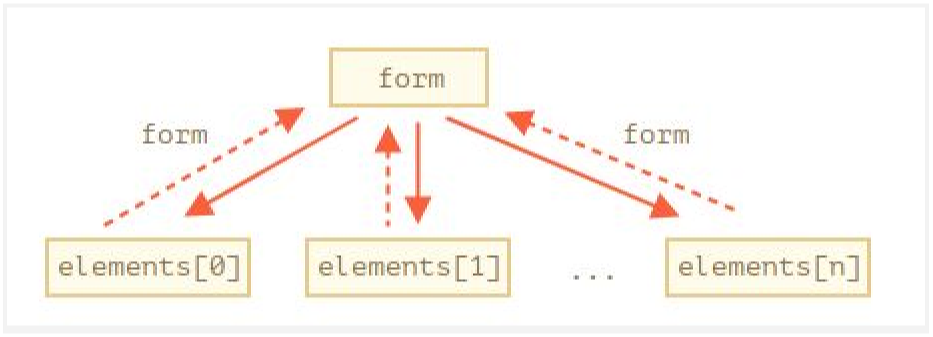
For instance:

```
<body>
    <form id="form">
        <fieldset name="userFields">
            <legend>info</legend>
            <input name="login" type="text">
        </fieldset>
    </form>
    <script>
        alert(form.elements.login); // <input name="login">
        let fieldset = form.elements.userFields;
        alert(fieldset); // HTMLFieldSetElement
        // we can get the input by name both from the form and from the fieldset
        alert(fieldset.elements.login == form.elements.login); // true
    </script>
</body>
```

Backreference: element.form

For any element, the form is available as element.form. So a form references all elements, and elements reference the form.

Here's the picture:



For instance:

```
<form id="form">
    <input type="text" name="login">
</form>
<script>
    // form -> element
    let login = form.login;
    // element -> form
    alert(login.form); // HTMLFormElement
</script>
```

Form elements

Let's talk about form controls.

input and textarea

We can access their value as `input.value` (string) or `input.checked` (boolean) for checkboxes.

Like this:

```
input.value = "New value";
textarea.value = "New text";
input.checked = true; // for a checkbox or radio button
```

select and option

A `<select>` element has 3 important properties:

1. `select.options` – the collection of `<option>` subelements,
2. `select.value` – the value of the currently selected `<option>`,
3. `select.selectedIndex` – the number of the currently selected `<option>`.

They provide three different ways of setting a value for a `<select>`:

1. Find the corresponding `<option>` element and set `option.selected` to `true`.
2. Set `select.value` to the value.
3. Set `select.selectedIndex` to the number of the option.

The first way is the most obvious, but `(2)` and `(3)` are usually more convenient.

Here is an example:

```
<select id="select">
    <option value="apple">Apple</option>
    <option value="pear">Pear</option>
    <option value="banana">Banana</option>
</select>

<script>
    // all three lines do the same thing
    select.options[2].selected = true;
    select.selectedIndex = 2;
    select.value = 'banana';
</script>
```

Unlike most other controls, `<select>` allows to select multiple options at once if it has `multiple` attribute. That’s feature is rarely used. In that case we need to use the first way: add/remove the `selected` property from `<option>` subelements.

We can get their collection as `select.options`, for instance:

```
<select id="select" multiple>
    <option value="blues" selected>Blues</option>
    <option value="rock" selected>Rock</option>
    <option value="classic">Classic</option>
```

```
</select>
<script>
    // get all selected values from multi-select
    let selected = Array.from(select.options)
        .filter(option => option.selected)
        .map(option => option.value);
    alert(selected); // blues,rock
</script>
```

Focusing: focus/blur

An element receives a focus when the user either clicks on it or uses the Tab key on the keyboard. There's also an autofocus HTML attribute that puts the focus into an element by default when a page loads and other means of getting a focus.

Focusing on an element generally means: "prepare to accept the data here", so that's the moment when we can run the code to initialize the required functionality.

The moment of losing the focus ("blur") can be even more important. That's when a user clicks somewhere else or presses Tab to go to the next form field, or there are other means as well.

Losing the focus generally means: "the data has been entered", so we can run the code to check it or even to save it to the server and so on.

Events focus/blur

The focus event is called on focusing, and blur – when the element loses the focus.

Let's use them for validation of an input field.

In the example below:

- The blur handler checks if the field the email is entered, and if not – shows an error.
- The focus handler hides the error message (on blur it will be checked again):

```
<style>
    .invalid { border-color: red; }
```

```
  #error { color: red }
</style>
Your email please: <input type="email" id="input">
<div id="error"></div>
<script>
  input.onblur = function() {
    if (!input.value.includes('@')) { // not email
      input.classList.add('invalid');
      error.innerHTML = 'Please enter a correct email.'
    }
  };
  input.onfocus = function() {
    if (this.classList.contains('invalid')) {
      // remove the "error" indication, because the user wants to re-enter something
      this.classList.remove('invalid');
      error.innerHTML = "";
    }
  };
</script>
```

Modern HTML allows us to do many validations using input attributes: `required`, `pattern` and so on. And sometimes they are just what we need. JavaScript can be used when we want more flexibility. Also we could automatically send the changed value to the server if it’s correct.

Methods focus/blur

Methods `elem.focus()` and `elem.blur()` set/unset the focus on the element.
For instance, let’s make the visitor unable to leave the input if the value is invalid:

```
<style>
  .error {
    background: red;
  }
</style>
```

```
Your email please: <input type="email" id="input">
<input type="text" style="width:220px" placeholder="make email invalid and try to focus here">

<script>
    input.onblur = function() {
        if (!this.value.includes('@')) { // not email
            // show the error
            this.classList.add("error");
            // ...and put the focus back
            input.focus();
        } else {
            this.classList.remove("error");
        }
    };
</script>
```

Please note that we can’t “prevent losing focus” by calling event.preventDefault() in onblur, because onblur works *after* the element loses the focus.

JavaScript-initiated focus loss

A focus loss can occur for many reasons.

One of them is when the visitor clicks somewhere else. But also JavaScript itself may cause it, for instance:

- An alert moves focus to itself, so it causes the focus loss at the element (blur event), and when the alert is dismissed, the focus comes back (focus event).
- If an element is removed from DOM, then it also causes the focus loss. If it is reinserted later, then the focus doesn’t return.

These features sometimes cause focus/blur handlers to misbehave – to trigger when they are not needed.
The best recipe is to be careful when using these events. If we want to track user-initiated focus-loss, then we should avoid causing it ourselves.

Allow focusing on any element: tabindex
By default many elements do not support focusing.
The list varies a bit between browsers, but one thing is always correct: focus/blur support is guaranteed for elements that a visitor can interact with: <button>, <input>, <select>, <a> and so on.

From the other hand, elements that exist to format something, such as <div>, <span>, <table> – are non focusable by default. The method elem.focus() doesn't work on them, and focus/blur events are never triggered.

This can be changed using HTML-attribute tabindex.

Any element becomes focusable if it has tabindex. The value of the attribute is the order number of the element when Tab(or something like that) is used to switch between them.

That is: if we have two elements, the first has tabindex="1", and the second has tabindex="2", then pressing Tabwhile in the first element – moves the focus into the second one.

The switch order is: elements with tabindex from 1 and above go first (in the tabindex order), and then elements without tabindex (e.g. a regular <input>).

Elements with matching tabindex are switched in the document source order (the default order).

There are two special values:

- tabindex="0" puts an element among those without tabindex. That is, when we switch elements, elements with tabindex=0 go after elements with tabindex ≥ 1.
  Usually it's used to make an element focusable, but keep the default switching order. To make an element a part of the form on par with <input>.
- tabindex="-1" allows only programmatic focusing on an element. The Tab key ignores such elements, but method elem.focus() works.

For instance, here’s a list. Click the first item and press Tab:

Click the first item and press Tab. Keep track of the order. Please note that many subsequent Tabs can move the focus out of the iframe with the example.

```
<ul>
    <li tabindex="1">One</li>
    <li tabindex="0">Zero</li>
    <li tabindex="2">Two</li>
    <li tabindex="-1">Minus one</li>
</ul>
<style>
    li { cursor: pointer; }
    :focus { outline: 1px dashed green; }
</style>
```

The order is like this: `1 - 2 - 0`. Normally, `<li>` does not support focusing, but `tabindex` full enables it, along with events and styling with `:focus`.

The property `elem.tabIndex` works too

We can add `tabindex` from JavaScript by using the `elem.tabIndex` property. That has the same effect.

Delegation: focusin/focusout

Events `focus` and `blur` do not bubble.

For instance, we can’t put `onfocus` on the `<form>` to highlight it, like this:

```
<!-- on focusing in the form -- add the class -->
<form onfocus="this.className='focused'">
    <input type="text" name="name" value="Name">
    <input type="text" name="surname" value="Surname">
</form>
```

```
<style> .focused { outline: 1px solid red; } </style>
```

The example above doesn't work, because when user focuses on an `<input>`, the `focus` event triggers on that input only. It doesn't bubble up. So `form.onfocus` never triggers.

There are two solutions.

First, there's a funny historical feature: `focus/blur` do not bubble up, but propagate down on the capturing phase.

This will work:

```
<form id="form">
    <input type="text" name="name" value="Name">
    <input type="text" name="surname" value="Surname">
</form>
<style> .focused { outline: 1px solid red; } </style>
<script>
    // put the handler on capturing phase (last argument true)
    form.addEventListener("focus", () => form.classList.add('focused'), true);
    form.addEventListener("blur", () => form.classList.remove('focused'), true);
</script>
```

Events: change, input, cut, copy, paste

Let's cover various events that accompany data updates.

Event: change

The `change` event triggers when the element has finished changing.

For text inputs that means that the event occurs when it loses focus.

For instance, while we are typing in the text field below – there's no event. But when we move the focus somewhere else, for instance, click on a button – there will be a `change` event:

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```

For other elements: `select`, `input type=checkbox/radio` it triggers right after the selection changes:

```
<select onchange="alert(this.value)">
    <option value="">Select something</option>
    <option value="1">Option 1</option>
    <option value="2">Option 2</option>
    <option value="3">Option 3</option>
</select>
```

Event: input

The `input` event triggers every time after a value is modified by the user.

Unlike keyboard events, it triggers on any value change, even those that do not involve keyboard actions: pasting with a mouse or using speech recognition to dictate the text.

For instance:

```
<input type="text" id="input"> oninput: <span id="result"></span>
<script>
    input.oninput = function() {
        result.innerHTML = input.value;
    };
</script>
```

If we want to handle every modification of an `<input>` then this event is the best choice.

On the other hand, `input` event doesn't trigger on keyboard input and other actions that do not involve value change, e.g. pressing arrow keys ⇦ ⇨ while in the input.

Can't prevent anything in `oninput`

The `input` event occurs after the value is modified.

So we can't use `event.preventDefault()` there – it's just too late, there would be no effect.

Events: cut, copy, paste

These events occur on cutting/copying/pasting a value.
They belong to ClipboardEvent class and provide access to the data that is copied/pasted.
We also can use `event.preventDefault()` to abort the action, then nothing gets copied/pasted.

For instance, the code below prevents all such events and shows what we are trying to cut/copy/paste:

```
<input type="text" id="input">
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));
    return false;
  };
</script>
```

Forms: event and method submit

The `submit` event triggers when the form is submitted, it is usually used to validate the form before sending it to the server or to abort the submission and process it in JavaScript.

The method `form.submit()` allows you to initiate form sending from JavaScript. We can use it to dynamically create and send our own forms to the server.

Let’s see more details of them.

Event: submit

There are two main ways to submit a form:

1. The first – to click `<input type="submit">` or `<input type="image">`.
2. The second – press Enter on an input field.

Both actions lead to a `submit` event on the form. The handler can check the data, and if there are errors, show them and call `event.preventDefault()`, then the form won’t be sent to the server.

In the form below:

1. Go into the text field and press Enter.

2. Click <input type="submit">.

Both actions show alert and the form is not sent anywhere due to return false:

```
<form onsubmit="alert('submit!');return false">
    First: Enter in the input field <input type="text" value="text"><br>
    Second: Click "submit": <input type="submit" value="Submit">
</form>
```

Relation between submit and click

When a form is sent using Enter on an input field, a click event triggers on the <input type="submit">.
That's rather funny, because there was no click at all.
Here's the demo:

```
<form onsubmit="return false">
    <input type="text" size="30" value="Focus here and press enter">
    <input type="submit" value="Submit" onclick="alert('click')">
</form>
```

Method: submit

To submit a form to the server manually, we can call form.submit().

Sometimes that's used to manually create and send a form, like this:

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';
form.innerHTML = '<input name="q" value="test">';
// the form must be in the document to submit it
document.body.append(form);
form.submit();
```

Page: DOMContentLoaded, load, beforeunload, unload

The lifecycle of an HTML page has three important events:

- `DOMContentLoaded` – the browser fully loaded HTML, and the DOM tree is built, but external resources like pictures `<img>` and stylesheets may be not yet loaded.
- `load` – not only HTML is loaded, but also all the external resources: images, styles etc.
- `beforeunload/unload` – the user is leaving the page.

Each event may be useful:

- `DOMContentLoaded` event – DOM is ready, so the handler can lookup DOM nodes, initialize the interface.
- `load` event – external resources are loaded, so styles are applied, image sizes are known etc.
- `beforeunload` event – the user is leaving: we can check if the user saved the changes and ask them whether they really want to leave.
- `unload` – the user almost left, but we still can initiate some operations, such as sending out statistics.

Let’s explore the details of these events.

DOMContentLoaded

The `DOMContentLoaded` event happens on the `document` object.
We must use `addEventListener` to catch it:

```
document.addEventListener("DOMContentLoaded", ready);
// not "document.onDOMContentLoaded = ..."
```

For instance:

```
<script>
    function ready() {
        alert('DOM is ready');
        // image is not yet loaded (unless was cached), so the size is 0x0
        alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
```

```
  }
  document.addEventListener("DOMContentLoaded", ready);
</script>
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

In the example the DOMContentLoaded handler runs when the document is loaded, so it can see all the elements, including <img> below. But it doesn’t wait for the image to load. So alert shows zero sizes.
At first sight, the DOMContentLoaded event is very simple. The DOM tree is ready – here’s the event. There are few peculiarities though.

DOMContentLoaded and scripts
When the browser processes an HTML-document and comes across a <script> tag, it needs to execute before continuing building the DOM. That’s a precaution, as scripts may want to modify DOM, and even document.write into it, so DOMContentLoaded has to wait.

So DOMContentLoaded definitely happens after such scripts:

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    alert("DOM ready!");
  });
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>
<script>
  alert("Library loaded, inline script executed");
</script>
```
In the example above, we first see “Library loaded…”, and then “DOM ready!” (all scripts are executed).

Scripts that don’t block DOMContentLoaded

There are two exceptions from this rule:

1. Scripts with the async attribute, that we’ll cover a bit later, don’t block DOMContentLoaded.

2. Scripts that are generated dynamically with document.createElement('script') and then added to the webpage also don't block this event.

DOMContentLoaded and styles

External style sheets don't affect DOM, so DOMContentLoaded does not wait for them.
But there's a pitfall. If we have a script after the style, then that script must wait until the stylesheet loads:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
    // the script doesn't not execute until the stylesheet is loaded
    alert(getComputedStyle(document.body).marginTop);
</script>
```

The reason for this is that the script may want to get coordinates and other style-dependent properties of elements, like in the example above. Naturally, it has to wait for styles to load.

As DOMContentLoaded waits for scripts, it now waits for styles before them as well.

Built-in browser autofill

Firefox, Chrome and Opera autofill forms on DOMContentLoaded.

For instance, if the page has a form with login and password, and the browser remembers the values, then on DOMContentLoaded it may try to autofill them (if approved by the user).

So if DOMContentLoaded is postponed by long-loading scripts, then autofill also awaits. You probably saw that on some sites (if you use browser autofill) – the login/password fields don't get autofilled immediately, but there's a delay till the page fully loads. That's actually the delay until the DOMContentLoaded event.

window.onload

The load event on the window object triggers when the whole page is loaded including styles, images and other resources. This event is available via the onload property.

The example below correctly shows image sizes, because window.onload waits for all images:

```
<script>
  window.onload = function() { // same as window.addEventListener('load', (event) => {
    alert('Page loaded');
    // image is loaded at this time
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  };
</script>
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

window.onunload

When a visitor leaves the page, the `unload` event triggers on the window. We can do something there that doesn't involve a delay, like closing related popup windows.

The notable exception is sending analytics.

Let's say we gather data about how the page is used: mouse clicks, scrolls, viewed page areas, and so on.

Naturally, `unload` event is when the user leaves us, and we'd like to save the data on our server.

There exists a special `navigator.sendBeacon(url, data)` method for such needs, described in the specification https://w3c.github.io/beacon/.

It sends the data in the background. The transition to another page is not delayed: the browser leaves the page, but still performs `sendBeacon`.

Here's how to use it:

```
let analyticsData = { /* object with gathered data */ };
window.addEventListener("unload", function() {
  navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));
};
```

- The request is sent as POST.
- We can send not only a string, but also forms and other formats, as described in the chapter Fetch, but usually it's a stringified object.
- The data is limited by 64kb.

When the sendBeacon request is finished, the browser probably has already left the document, so there’s no way to get server response (which is usually empty for analytics).

There’s also a keepalive flag for doing such “after-page-left” requests in fetch method for generic network requests. You can find more information in the chapter Fetch API.

window.onbeforeunload

If a visitor initiates navigation away from the page or tries to close the window, the onbeforeunload handler asks for additional confirmation.

If we cancel the event, the browser may ask the visitor if they are sure.

You can try it by running this code and then reloading the page:

```
window.onbeforeunload = function() {
    return false;
};
```

For historical reasons, returning a non-empty string also counts as canceling the event. Some time ago browsers used to show it as a message, but as the modern specification says, they shouldn’t.

Here’s an example:

```
window.onbeforeunload = function() {
    return "There are unsaved changes. Leave now?";
};
```

The behavior was changed, because some webmasters abused this event handler by showing misleading and annoying messages. So right now old browsers still may show it as a message, but aside from that – there’s no way to customize the message shown to the user.

readyState

What happens if we set the DOMContentLoaded handler after the document is loaded?

Naturally, it never runs.
There are cases when we are not sure whether the document is ready or not. We’d like our function to execute when the DOM is loaded, be it now or later.

The document.readyState property tells us about the current loading state.

There are 3 possible values:

- "loading" – the document is loading.
- "interactive" – the document was fully read.
- "complete" – the document was fully read and all resources (like images) are loaded too.

So we can check document.readyState and setup a handler or execute the code immediately if it’s ready.

Like this:

```
function work() { /*...*/ }
if (document.readyState == 'loading') {
    // loading yet, wait for the event
    document.addEventListener('DOMContentLoaded', work);
} else {
    // DOM is ready!
    work();
}
```

There’s also the readystatechange event that triggers when the state changes, so we can print all these states like this:

```
// current state
console.log(document.readyState);
// print state changes
document.addEventListener('readystatechange', () => console.log(document.readyState));
```

The readystatechange event is an alternative mechanics of tracking the document loading state, Let’s see the full events flow for the completeness.

Here’s a document with <iframe>, <img> and handlers that log events:

```
<script>
    log('initial readyState:' + document.readyState);
    document.addEventListener('readystatechange', () => log('readyState:' +
document.readyState));
    document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));
    window.onload = () => log('window onload');
</script>
<img src="http://en.js.cx/clipart/train.gif" id="img">
<script>
    img.onload = () => log('img onload');
</script>
```

The working example is in the sandbox.

The typical output:

1. [1] initial readyState:loading
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

The numbers in square brackets denote the approximate time of when it happens. Events labeled with the same digit happen approximately at the same time (± a few ms).

- document.readyState becomes interactive right before DOMContentLoaded. These two things actually mean the same.

document.readyState becomes complete when all resources (iframe and img) are loaded. Here we can see that it happens in about the same time as img.onload (img is the last resource) and window.onload. Switching to complete state means the same as window.onload. The difference is that window.onload always works after all other load handlers.it appeared long ago. Nowadays, it is rarely used.

Scripts: async, defer

In modern websites, scripts are often “heavier” than HTML: their download size is larger, and processing time is also longer.

When the browser loads HTML and comes across a `<script>...</script>` tag, it can’t continue building the DOM. It must execute the script right now. The same happens for external scripts `<script src="..."></script>`: the browser must wait until the script downloads, executes it, and only after process the rest of the page.

That leads to two important issues:

1. Scripts can’t see DOM elements below them, so they can’t add handlers etc.
2. If there’s a bulky script at the top of the page, it “blocks the page”. Users can’t see the page content till it downloads and runs:

```
<p>...content before script...</p>
<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<!-- This isn't visible until the script loads -->
<p>...content after script...</p>
```

There are some workarounds to that. For instance, we can put a script at the bottom of the page. Then it can see elements above it, and it doesn’t block the page content from showing:

```
<body>
   ...all content is above the script...
   <script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
</body>
```

But this solution is far from perfect. For example, the browser notices the script (and can start downloading it) only after it downloaded the full HTML document. For long HTML documents, that may be a noticeable delay.

Such things are invisible for people using very fast connections, but many people in the world still have slow internet speeds and use a far-from-perfect mobile internet connection.

Luckily, there are two `<script>` attributes that solve the problem for us: `defer` and `async`.

defer

The `defer` attribute tells the browser that it should go on working with the page, and load the script “in background”, then run the script when it loads.

Here’s the same example as above, but with `defer`:

```
<p>...content before script...</p>
<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<!-- visible immediately -->
<p>...content after script...</p>
```

- Scripts with `defer` never block the page.
- Scripts with `defer` always execute when the DOM is ready, but before the DOMContentLoaded event.

The following example demonstrates that:

```
<p>...content before scripts...</p>
<script>
    document.addEventListener('DOMContentLoaded', () => alert("DOM ready after defer!")); // (2)
</script>
<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>
<p>...content after scripts...</p>
```

1. The page content shows up immediately.
2. DOMContentLoaded waits for the deferred script. It only triggers when the script `(2)` is downloaded and executed.

Deferred scripts keep their relative order, just like regular scripts.

So, if we have a long script first, and then a smaller one, then the latter one waits.

```
<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>
<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>
```

The small script downloads first, runs second

Browsers scan the page for scripts and download them in parallel, to improve performance. So in the example above both scripts download in parallel. The `small.js` probably makes it first.

But the specification requires scripts to execute in the document order, so it waits for `long.js` to execute.

The defer attribute is only for external scripts

The defer attribute is ignored if the `<script>` tag has no `src`.

async

The async attribute means that a script is completely independent:

- The page doesn't wait for async scripts, the contents are processed and displayed.
- DOMContentLoaded and async scripts don't wait for each other:
    - DOMContentLoaded may happen both before an async script (if an async script finishes loading after the page is complete)
    - …or after an async script (if an async script is short or was in HTTP-cache)
- Other scripts don't wait for async scripts, and async scripts don't wait for them.

So, if we have several async scripts, they may execute in any order. Whatever loads first – runs first:

```
<p>...content before scripts...</p>
<script>
    document.addEventListener('DOMContentLoaded', () => alert("DOM ready!"));
</script>
<script async src="https://javascript.info/article/script-async-defer/long.js"></script>
<script async src="https://javascript.info/article/script-async-defer/small.js"></script>
<p>...content after scripts...</p>
```

1. The page content shows up immediately: `async` doesn't block it.
2. DOMContentLoaded may happen both before and after async, no guarantees here.

3. Async scripts don’t wait for each other. A smaller script `small.js` goes second, but probably loads before `long.js`, so runs first. That’s called a “load-first” order.

Async scripts are great when we integrate an independent third-party script into the page: counters, ads and so on, as they don’t depend on our scripts, and our scripts shouldn’t wait for them:

```
<!-- Google Analytics is usually added like this -->
<script async src="https://google-analytics.com/analytics.js"></script>
```

Dynamic scripts

We can also add a script dynamically using JavaScript:

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
document.body.append(script); // (*)
```

The script starts loading as soon as it’s appended to the document `(*)`.

Dynamic scripts behave as “async” by default.

That is:

- They don’t wait for anything, nothing waits for them.
- The script that loads first – runs first (“load-first” order).

```
let script = document.createElement('script');
script.src = "/article/script-async-defer/long.js";
script.async = false;
document.body.append(script);
```

For example, here we add two scripts. Without `script.async=false` they would execute in load-first order (the `small.js` probably first). But with that flag the order is “as in the document”:

```
function loadScript(src) {
    let script = document.createElement('script');
    script.src = src;
    script.async = false;
    document.body.append(script);
}
// long.js runs first because of async=false
loadScript("/article/script-async-defer/long.js");
loadScript("/article/script-async-defer/small.js");
```

Resource loading: onload and onerror

The browser allows us to track the loading of external resources – scripts, iframes, pictures and so on.

There are two events for it:

- onload – successful load,
- onerror – an error occurred.

Loading a script

Let’s say we need to load a third-party script and call a function that resides there.

We can load it dynamically, like this:

```
let script = document.createElement('script');
script.src = "my.js";
document.head.append(script);
```
…But how to run the function that is declared inside that script? We need to wait until the script loads, and only then we can call it.

script.onload

The main helper is the load event. It triggers after the script is loaded and executed. For instance:

```
let script = document.createElement('script');
// can load any script, from any domain
script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"
document.head.append(script);
script.onload = function() {
    // the script creates a helper function "_"
    alert(_); // the function is available
};
```

So in onload we can use script variables, run functions etc.

…And what if the loading failed? For instance, there’s no such script (error 404) or the server is down (unavailable).

script.onerror

Errors that occur during the loading of the script can be tracked in an error event. For instance, let’s request a script that doesn’t exist:

```
let script = document.createElement('script');
script.src = "https://example.com/404.js"; // no such script
document.head.append(script);
script.onerror = function() {
    alert("Error loading " + this.src); // Error loading https://example.com/404.js
};
```

Please note that we can’t get HTTP error details here. We don’t know if it was an error 404 or 500 or something else. Just that the loading failed.

Events onload/onerror track only the loading itself.

Errors that may occur during script processing and execution are out of scope for these events. That is: if a script loaded successfully, then onload triggers, even if it has programming errors in it. To track script errors, one can use window.onerror global handler.

Other resources

The load and error events also work for other resources, basically for any resource that has an external src. For example:

```
let img = document.createElement('img');
img.src = "https://js.cx/clipart/train.gif"; // (*)
img.onload = function() {
    alert(`Image loaded, size ${img.width}x${img.height}`);
};
```

```
img.onerror = function() {
    alert("Error occurred while loading image");
};
```

There are some notes though:

- Most resources start loading when they are added to the document. But `<img>` is an exception. It starts loading when it gets a src `(*)`.
- For `<iframe>`, the `iframe.onload` event triggers when the iframe loading finished, both for successful load and in case of an error.

Crossorigin policy

There’s a rule: scripts from one site can’t access contents of the other site. So, e.g. a script at `https://facebook.com` can’t read the user’s mailbox at `https://gmail.com`.

Or, to be more precise, one origin (domain/port/protocol triplet) can’t access the content from another one. So even if we have a subdomain, or just another port, these are different origins with no access to each other.

This rule also affects resources from other domains. If we’re using a script from another domain, and there’s an error in it, we can’t get error details.

For example, let’s take a script `error.js` that consists of a single (bad) function call:

```
// 📁 error.js
noSuchFunction();
```

Now load it from the same site where it’s located:

```
<script>
window.onerror = function(message, url, line, col, errorObj) {
    alert(`${message}\n${url}, ${line}:${col}`);
};
</script>
<script src="/article/onload-onerror/crossorigin/error.js"></script>
```

We can see a good error report, like this:

```
Uncaught ReferenceError: noSuchFunction is not defined
https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1
```

Now let’s load the same script from another domain:

```
<script>
  window.onerror = function(message, url, line, col, errorObj) {
    alert(`${message}\n${url}, ${line}:${col}`);
  };
</script>
<script src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>
```

The report is different, like this:

```
Script error.
, 0:0
```

Details may vary depending on the browser, but the idea is the same: any information about the internals of a script, including error stack traces, is hidden. Exactly because it’s from another domain.

Why do we need error details?

There are many services (and we can build our own) that listen for global errors using `window.onerror`, save errors and provide an interface to access and analyze them. That’s great, as we can see real errors, triggered by our users. But if a script comes from another origin, then there’s not much information about errors in it, as we’ve just seen.

Similar cross-origin policy (CORS) is enforced for other types of resources as well.

To allow cross-origin access, the `<script>` tag needs to have the `crossorigin` attribute, plus the remote server must provide special headers.

There are three levels of cross-origin access:

1. **No crossorigin attribute** – access prohibited.
2. **crossorigin="anonymous"** – access allowed if the server responds with the header Access-Control-Allow-Origin with * or our origin. Browser does not send authorization information and cookies to the remote server.
3. **crossorigin="use-credentials"** – access allowed if the server sends back the header Access-Control-Allow-Origin with our origin and Access-Control-Allow-Credentials: true. Browser sends authorization information and cookies to the remote server.

In our case, we didn't have any crossorigin attribute. So cross-origin access was prohibited. Let's add it.
We can choose between "anonymous" (no cookies sent, one server-side header needed) and "use-credentials"(sends cookies too, two server-side headers needed).

If we don't care about cookies, then "anonymous" is the way to go:

```
<script>
    window.onerror = function(message, url, line, col, errorObj) {
        alert(`${message}\n${url}, ${line}:${col}`);
    };
</script>
<script crossorigin="anonymous"
src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>
```

Now, assuming that the server provides an Access-Control-Allow-Origin header, everything's fine. We have the full error report.

Selection and Range

Range

The basic concept of selection is Range: basically, a pair of "boundary points": range start and range end.

Each point is represented as a parent DOM node with the relative offset from its start. If the parent node is an element node, then the offset is a child number, for a text node it's the position in the text. Examples to follow.

Let's select something.

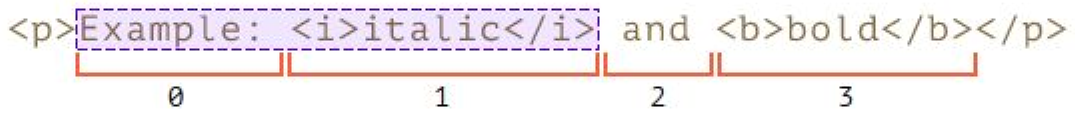
First, we can create a range (the constructor has no parameters):
`let range = new Range();`

Then we can set the selection boundaries using `range.setStart(node, offset)` and `range.setEnd(node, offset)`.

For example, consider this fragment of HTML:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

Let's select `"Example: <i>italic</i>"`. That's two first children of `<p>` (counting text nodes):



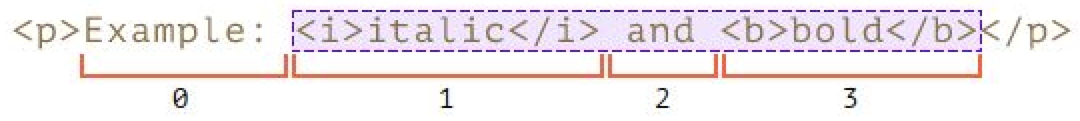
```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
<script>
  let range = new Range();
  range.setStart(p, 0);
  range.setEnd(p, 2);
  // toString of a range returns its content as text (without tags)
  alert(range); // Example: italic
  // apply this range for document selection (explained later)
  document.getSelection().addRange(range);
</script>
```

- `range.setStart(p, 0)` – sets the start at the 0th child of `<p>` (that's the text node `"Example: "`).
- `range.setEnd(p, 2)` – spans the range up to (but not including) 2nd child of `<p>` (that's the text node `" and "`, but as the end is not included, so the last selected node is `<i>`

Here’s a more flexible test stand where you try more variants:

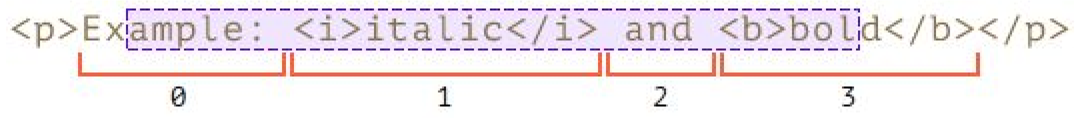
```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
From <input id="start" type="number" value=1> – To <input id="end" type="number" value=4>
<button id="button">Click to select</button>
<script>
  button.onclick = () => {
    let range = new Range();
    range.setStart(p, start.value);
    range.setEnd(p, end.value);
    // apply the selection, explained later
    document.getSelection().removeAllRanges();
    document.getSelection().addRange(range);
  };
</script>
```

E.g. selecting from 1 to 4 gives range `<i>italic</i> and <b>bold</b>`.



Selecting parts of text nodes

Let’s select the text partially, like this:



That’s also possible, we just need to set the start and the end as a relative offset in text nodes.

We need to create a range, that:

- starts from position 2 in `<p>` first child (taking all but two first letters of "Ex**ample:** ")
- ends at the position 3 in `<b>` first child (taking first three letters of “**bol**d”, but no more):

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
<script>
    let range = new Range();
    range.setStart(p.firstChild, 2);
    range.setEnd(p.querySelector('b').firstChild, 3);
    alert(range); // ample: italic and bol
    // use this range for selection (explained later)
    window.getSelection().addRange(range);
</script>
```

Range methods

There are many convenience methods to manipulate ranges.

Set range start:

- setStart(node, offset) set start at: position offset in node
- setStartBefore(node) set start at: right before node
- setStartAfter(node) set start at: right after node

Set range end (similar methods):

- setEnd(node, offset) set end at: position offset in node
- setEndBefore(node) set end at: right before node
- setEndAfter(node) set end at: right after node

As it was demonstrated, node can be both a text or element node: for text nodes offset skips that many of characters, while for element nodes that many child nodes.

Others:

- selectNode(node) set range to select the whole node
- selectNodeContents(node) set range to select the whole node contents
- collapse(toStart) if toStart=true set end=start, otherwise set start=end, thus collapsing the range

- cloneRange() creates a new range with the same start/end

To manipulate the content within the range:

- deleteContents() – remove range content from the document
- extractContents() – remove range content from the document and return as DocumentFragment
- cloneContents() – clone range content and return as DocumentFragment
- insertNode(node) – insert node into the document at the beginning of the range
- surroundContents(node) – wrap node around range content. For this to work, the range must contain both opening and closing tags for all elements inside it: no partial ranges like <i>abc.

With these methods we can do basically anything with selected nodes.

Here’s the test stand to see them in action:

```
Click buttons to run methods on the selection, "resetExample" to reset it.
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
<p id="result"></p>
<script>
  let range = new Range();
  // Each demonstrated method is represented here:
  let methods = {
    deleteContents() {
      range.deleteContents()
    },
    extractContents() {
      let content = range.extractContents();
      result.innerHTML = "";
      result.append("extracted: ", content);
    },
    cloneContents() {
      let content = range.cloneContents();
      result.innerHTML = "";
```

```
            result.append("cloned: ", content);
        },
        insertNode() {
            let newNode = document.createElement('u');
            newNode.innerHTML = "NEW NODE";
            range.insertNode(newNode);
        },
        surroundContents() {
            let newNode = document.createElement('u');
            try {
                range.surroundContents(newNode);
            } catch(e) {
                alert(e)
            }
        },
        resetExample() {
            p.innerHTML = `Example: <i>italic</i> and <b>bold</b>`;
            result.innerHTML = "";
            range.setStart(p.firstChild, 2);
            range.setEnd(p.querySelector('b').firstChild, 3);
            window.getSelection().removeAllRanges();
            window.getSelection().addRange(range);
        }
    };
    for(let method in methods) {
        document.write(`<div><button
onclick="methods.${method}()">${method}</button></div>`);
    }
    methods.resetExample();
</script>
```

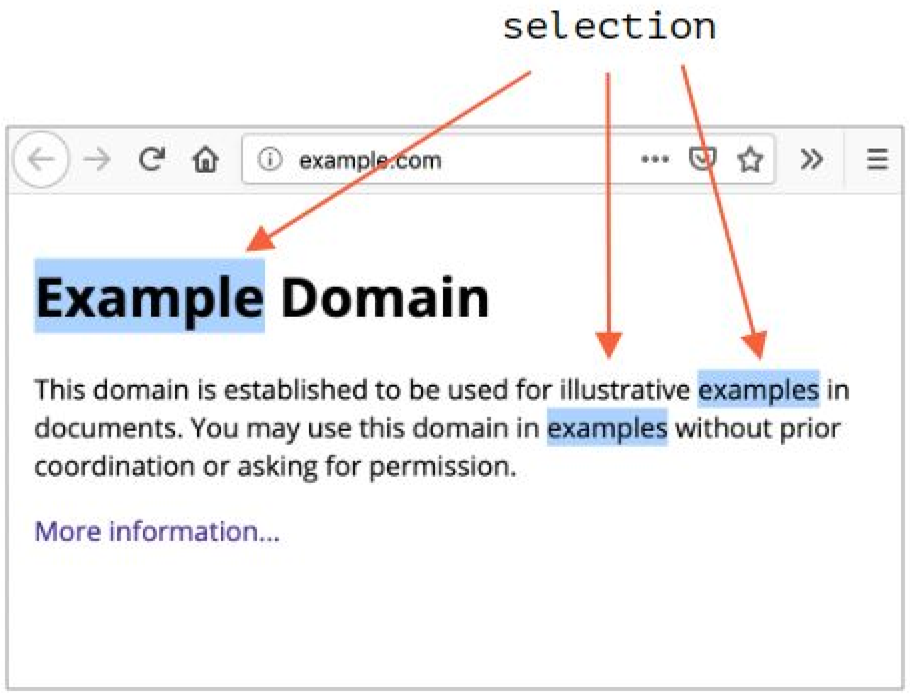
Selection

Range is a generic object for managing selection ranges. We may create such objects, pass them around – they do not visually select anything on their own.

The document selection is represented by a Selection object, that can be obtained as window.getSelection() or document.getSelection().

A selection may include zero or more ranges. At least, the Selection API specification says so. In practice though, only Firefox allows to select multiple ranges in the document by using Ctrl+click (Cmd+click for Mac).

Here's a screenshot of a selection with 3 ranges, made in Firefox:



Example Domain

This domain is established to be used for illustrative examples in documents. You may use this domain in examples without prior coordination or asking for permission.

More information...

Other browsers support at maximum 1 range. As we'll see, some of Selection methods imply that there may be many ranges, but again, in all browsers except Firefox, there's at maximum 1.

Selection properties

Similar to a range, a selection has a start, called "anchor", and the end, called "focus".

The main selection properties are:

- anchorNode – the node where the selection starts,

- anchorOffset – the offset in anchorNode where the selection starts,
- focusNode – the node where the selection ends,
- focusOffset – the offset in focusNode where the selection ends,
- isCollapsed – true if selection selects nothing (empty range), or doesn't exist.
- rangeCount – count of ranges in the selection, maximum 1 in all browsers except Firefox.

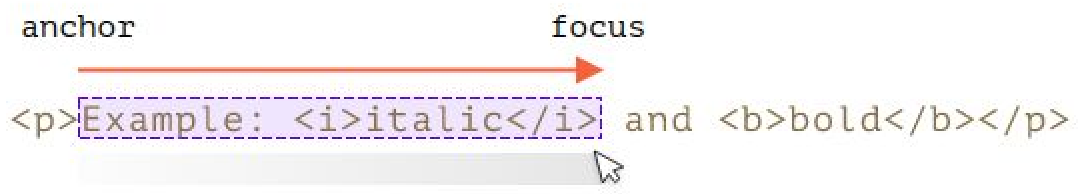
Selection end may be in the document before start

There are many ways to select the content, depending on the user agent: mouse, hotkeys, taps on a mobile etc.

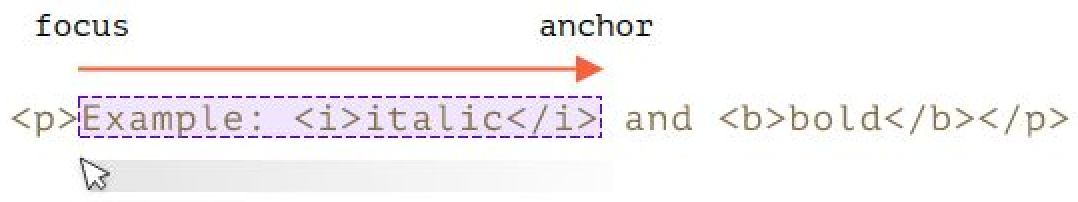
Some of them, such as a mouse, allow the same selection to be created in two directions: "left-to-right" and "right-to-left".

If the start (anchor) of the selection goes in the document before the end (focus), this selection is said to have a "forward" direction.

E.g. if the user starts selecting with mouse and goes from "Example" to "italic":



Otherwise, if they go from the end of "italic" to "Example", the selection is directed "backward", its focus will be before the anchor



That's different from Range objects that are always directed forward: the range start can't be after its end.

Selection events

There are events on to keep track of selection:

- `elem.onselectstart` – when a selection starts on `elem`, e.g. the user starts moving the mouse with pressed button.
    - Preventing the default action makes the selection not start.
- `document.onselectionchange` – whenever a selection changes.
    - Please note: this handler can be set only on `document`.

Selection tracking demo

Here’s a small demo that shows selection boundaries dynamically as it changes:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>
From <input id="from" disabled> – To <input id="to" disabled>
<script>
  document.onselectionchange = function() {
    let {anchorNode, anchorOffset, focusNode, focusOffset} = document.getSelection();
    from.value = `${anchorNode && anchorNode.data}:${anchorOffset}`;
    to.value = `${focusNode && focusNode.data}:${focusOffset}`;
  };
</script>
```

Selection getting demo

To get the whole selection:

- As text: just call `document.getSelection().toString()`.
- As DOM nodes: get the underlying ranges and call their `cloneContents()` method (only first range if we don’t support Firefox multiselection).

And here’s the demo of getting the selection both as text and as DOM nodes:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>
Cloned: <span id="cloned"></span>
<br>
As text: <span id="astext"></span>
<script>
```

```
  document.onselectionchange = function() {
    let selection = document.getSelection();
    cloned.innerHTML = astext.innerHTML = "";
    // Clone DOM nodes from ranges (we support multiselect here)
    for (let i = 0; i < selection.rangeCount; i++) {
      cloned.append(selection.getRangeAt(i).cloneContents());
    }
    // Get as text
    astext.innerHTML += selection;
  };
</script>
```

Selection methods

Selection methods to add/remove ranges:

- `getRangeAt(i)` – get i-th range, starting from `0`. In all browsers except firefox, only `0` is used.
- `addRange(range)` – add `range` to selection. All browsers except Firefox ignore the call, if the selection already has an associated range.
- `removeRange(range)` – remove `range` from the selection.
- `removeAllRanges()` – remove all ranges.
- `empty()` – alias to `removeAllRanges`.

Also, there are convenience methods to manipulate the selection range directly, without `Range`:

- `collapse(node, offset)` – replace selected range with a new one that starts and ends at the given `node`, at position `offset`.
- `setPosition(node, offset)` – alias to `collapse`.
- `collapseToStart()` – collapse (replace with an empty range) to selection start,
- `collapseToEnd()` – collapse to selection end,
- `extend(node, offset)` – move focus of the selection to the given `node`, position `offset`,
- `setBaseAndExtent(anchorNode, anchorOffset, focusNode, focusOffset)` – replace selection range with the given start `anchorNode/anchorOffset` and end `focusNode/focusOffset`. All content in-between them is selected.

- selectAllChildren(node) – select all children of the node.
- deleteFromDocument() – remove selected content from the document.
- containsNode(node, allowPartialContainment = false) – checks whether the selection contains node(partially if the second argument is true)

So, for many tasks we can call Selection methods, no need to access the underlying Range object.

Selection in form controls

Form elements, such as input and textarea provide a special API for selection, without Selection or Range objects. As an input value is a pure text, not HTML, there’s no need for such objects, everything’s much simpler.

Properties:

- input.selectionStart – position of selection start (writeable),
- input.selectionEnd – position of selection end (writeable),
- input.selectionDirection – selection direction, one of: “forward”, “backward” or “none” (if e.g. selected with a double mouse click),

Events:

- input.onselect – triggers when something is selected.

Methods:

- input.select() – selects everything in the text control (can be textarea instead of input),
- input.setSelectionRange(start, end, [direction]) – change the selection to span from position start till end, in the given direction (optional).
- input.setRangeText(replacement, [start], [end], [selectionMode]) – replace a range of text with the new text.
  Optional arguments start and end, if provided, set the range start and end, otherwise user selection is used.
  The last argument, selectionMode, determines how the selection will be set after the text has been replaced. The possible values are:
    - "select" – the newly inserted text will be selected.

    ○ "start" – the selection range collapses just before the inserted text (the cursor will be immediately before it).
    ○ "end" – the selection range collapses just after the inserted text (the cursor will be right after it).
    ○ "preserve" – attempts to preserve the selection. This is the default.

Now let’s see these methods in action.

Example: tracking selection

For example, this code uses onselect event to track selection:

```
<textarea id="area" style="width:80%;height:60px">
Selecting in this text updates values below.
</textarea>
<br>
From <input id="from" disabled> – To <input id="to" disabled>
<script>
  area.onselect = function() {
    from.value = area.selectionStart;
    to.value = area.selectionEnd;
  };
</script>
```

Please note:

- onselect triggers when something is selected, but not when the selection is removed.

Example: moving cursor

We can change selectionStart and selectionEnd, that sets the selection.
An important edge case is when selectionStart and selectionEnd equal each other. Then it’s exactly the cursor position. Or, to rephrase, when nothing is selected, the selection is collapsed at the cursor position.

So, by setting selectionStart and selectionEnd to the same value, we move the cursor.

For example:

```
<textarea id="area" style="width:80%;height:60px">
Focus on me, the cursor will be at position 10.
</textarea>
<script>
  area.onfocus = () => {
    // zero delay setTimeout to run after browser "focus" action finishes
    setTimeout(() => {
      // we can set any selection
      // if start=end, the cursor it exactly at that place
      area.selectionStart = area.selectionEnd = 10;
    });
  };
</script>
```

Example: modifying selection

To modify the content of the selection, we can use `input.setRangeText()` method. Of course, we can read `selectionStart/End` and, with the knowledge of the selection, change the corresponding substring of `value`, but `setRangeText` is more powerful and often more convenient.

That's a somewhat complex method. In its simplest one-argument form it replaces the user selected range and removes the selection.

For example, here the user selection will be wrapped by `*...*`:

```
<input id="input" style="width:200px" value="Select here and click the button">
<button id="button">Wrap selection in stars *...*</button>
<script>
  button.onclick = () => {
    if (input.selectionStart == input.selectionEnd) {
      return; // nothing is selected
    }
    let selected = input.value.slice(input.selectionStart, input.selectionEnd);
    input.setRangeText(`*${selected}*`);
  };
```

```
</script>
```

With more arguments, we can set range start and end.

In this example we find "THIS" in the input text, replace it and keep the replacement selected:

```
<input id="input" style="width:200px" value="Replace THIS in text">
<button id="button">Replace THIS</button>
<script>
  button.onclick = () => {
    let pos = input.value.indexOf("THIS");
    if (pos >= 0) {
      input.setRangeText("*THIS*", pos, pos + 4, "select");
      input.focus(); // focus to make selection visible
    }
  };
</script>
```

Example: insert at cursor

If nothing is selected, or we use equal start and end in setRangeText, then the new text is just inserted, nothing is removed.

We can also insert something “at the cursor” using setRangeText.

Here’s a button that inserts "HELLO" at the cursor position and puts the cursor immediately after it. If the selection is not empty, then it gets replaced (we can detect it by comparing selectionStart!=selectionEnd and do something else instead):

```
<input id="input" style="width:200px" value="Text Text Text Text Text">
<button id="button">Insert "HELLO" at cursor</button>
<script>
  button.onclick = () => {
    input.setRangeText("HELLO", input.selectionStart, input.selectionEnd, "end");
    input.focus();
  };
</script>
```

Making unselectable

To make something unselectable, there are three ways:

1. Use CSS property `user-select: none`.

   ```
   <style>
     #elem {
       user-select: none;
     }
   </style>
   ```

2. `<div>Selectable <div id="elem">Unselectable</div> Selectable</div>`

   Prevent default action in `onselectstart` or `mousedown` events.

   ```
   <div>Selectable <div id="elem">Unselectable</div> Selectable</div>
   <script>
    elem.onselectstart = () => false;
   </script
   ```

3. We can also clear the selection post-factum after it happens with `document.getSelection().empty()`. That's rarely used, as this causes unwanted blinking as the selection appears-disappears.

Event loop: microtasks and macrotasks

Browser JavaScript execution flow, as well as in Node.js, is based on an *event loop*.

Understanding how event loops works is important for optimizations, and sometimes for the right architecture.

In this chapter we first cover theoretical details about how things work, and then see practical applications of that knowledge.

Event Loop

The concept of *event loop* is very simple. There's an endless loop, when the JavaScript engine waits for tasks, executes them and then sleeps waiting for more tasks.

The general algorithm of the engine:

1. While there are tasks:

    - execute them, starting with the oldest task.
2. Sleep until a task appears, then go to 1.

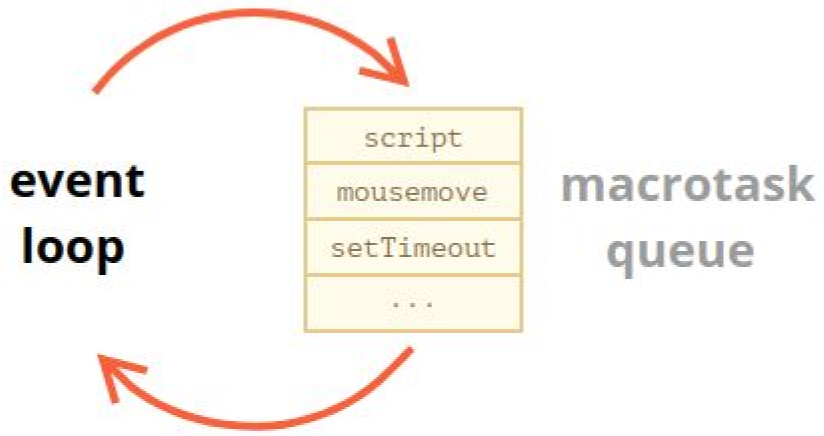
That's a formalization for what we see when browsing a page. JavaScript engine does nothing most of the time, only runs if a script/handler/event activates.

Examples of tasks:

- When an external script `<script src="...">` loads, the task is to execute it.
- When a user moves their mouse, the task is to dispatch `mousemove` event and execute handlers.
- When the time is due for a scheduled `setTimeout`, the task is to run its callback.
- …and so on.

Tasks are set – the engine handles them – then waits for more tasks (while sleeping and consuming close to zero CPU).

It may happen that a task comes while the engine is busy, then it's enqueued.
The tasks form a queue, so-called "macrotask queue" (v8 term):



For instance, while the engine is busy executing a `script`, a user may move their mouse causing `mousemove`, and `setTimeout` may be due and so on, these tasks form a queue, as illustrated on the picture above.

Tasks from the queue are processed on "first come – first served" basis. When the engine browser is done with the `script`, it handles `mousemove` event, then `setTimeout` handler, and so on.

So far, quite simple, right?
Two more details:

1. Rendering never happens while the engine executes a task. Doesn't matter if the task takes a long time. Changes to DOM are painted only after the task is complete.
2. If a task takes too long, the browser can't do other tasks, process user events, so after a time it raises an alert like "Page Unresponsive" suggesting to kill the task with the whole page. That happens when there are a lot of complex calculations or a programming error leading to infinite loop.

That was a theory. Now let's see how we can apply that knowledge.

Use-case 1: splitting CPU-hungry tasks

Let's say we have a CPU-hungry task.

For example, syntax-highlighting (used to colorize code examples on this page) is quite CPU-heavy. To highlight the code, it performs the analysis, creates many colored elements, adds them to the document – for a large amount of text that takes a lot of time.

While the engine is busy with syntax highlighting, it can't do other DOM-related stuff, process user events, etc. It may even cause the browser to "hiccup" or even "hang" for a bit, which is unacceptable.

We can avoid problems by splitting the big task into pieces. Highlight first 100 lines, then schedule `setTimeout (with zero-delay)` for the next 100 lines, and so on.

To demonstrate this approach, for the sake of simplicity, instead of text-highlighting, let's take a function that counts from `1` to `1000000000`.

If you run the code below, the engine will "hang" for some time. For server-side JS that's clearly noticeable, and if you are running it in-browser, then try to click other buttons on the page – you'll see that no other events get handled until the counting finishes.

```
let i = 0;
let start = Date.now();
function count() {
    // do a heavy job
    for (let j = 0; j < 1e9; j++) {
        i++;
    }
    alert("Done in " + (Date.now() - start) + 'ms');
}
count();
```

The browser may even show a “the script takes too long” warning.

Let’s split the job using nested `setTimeout` calls:

```
let i = 0;
let start = Date.now();
function count() {

    // do a piece of the heavy job (*)
    do {
        i++;
    } while (i % 1e6 != 0);
    if (i == 1e9) {
        alert("Done in " + (Date.now() - start) + 'ms');
    } else {
        setTimeout(count); // schedule the new call (**)
    }
}
count();
```

Now the browser interface is fully functional during the “counting” process.

A single run of `count` does a part of the job `(*)`, and then re-schedules itself `(**)` if needed:

1. First run counts: `i=1...1000000`.
2. Second run counts: `i=1000001..2000000`.
3. …and so on.

Now, if a new side task (e.g. onclick event) appears while the engine is busy executing part 1, it gets queued and then executes when part 1 finished, before the next part. Periodic returns to the event loop between count executions provide just enough “air” for the JavaScript engine to do something else, to react to other user actions.

The notable thing is that both variants – with and without splitting the job by setTimeout – are comparable in speed. There’s not much difference in the overall counting time.

Use case 2: progress indication

Another benefit of splitting heavy tasks for browser scripts is that we can show progress indication.

Usually the browser renders after the currently running code is complete. Doesn’t matter if the task takes a long time. Changes to DOM are painted only after the task is finished.

On one hand, that’s great, because our function may create many elements, add them one-by-one to the document and change their styles – the visitor won’t see any “intermediate”, unfinished state. An important thing, right?

Here’s the demo, the changes to i won’t show up until the function finishes, so we’ll see only the last value:

```
<div id="progress"></div>
<script>
    function count() {
        for (let i = 0; i < 1e6; i++) {
            i++;
            progress.innerHTML = i;
        }
    }
    count();
</script>
```

…But we also may want to show something during the task, e.g. a progress bar.

If we split the heavy task into pieces using setTimeout, then changes are painted out in-between them.

This looks prettier:

```
<div id="progress"></div>
<script>
  let i = 0;
  function count() {
    // do a piece of the heavy job (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);
    if (i < 1e7) {
      setTimeout(count);
    }
  }
  count();
```

Now the <div> shows increasing values of i, a kind of a progress bar.

Macrotasks and Microtasks

Along with *macrotasks*, described in this chapter, there exist *microtasks*, mentioned in the chapter Microtasks.

Microtasks come solely from our code. They are usually created by promises: an execution of .then/catch/finallyhandler becomes a microtask. Microtasks are used "under the cover" of await as well, as it's another form of promise handling.

There's also a special function queueMicrotask(func) that queues func for execution in the microtask queue.

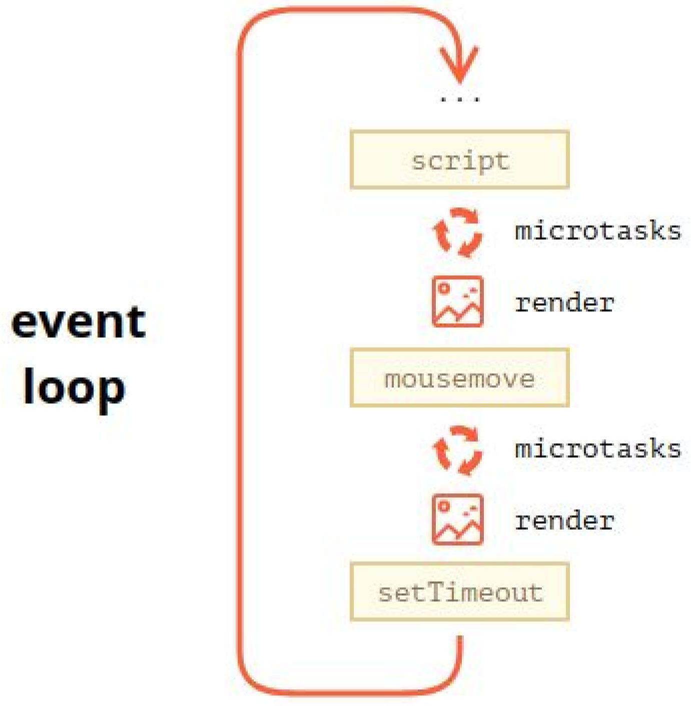
Immediately after every *macrotask*, the engine executes all tasks from *microtask* queue, prior to running any other macrotasks or rendering or anything else.

For instance, take a look:

```
setTimeout(() => alert("timeout"));
Promise.resolve()
  .then(() => alert("promise"));
alert("code");
```

What's going to be the order here?

1. code shows first, because it's a regular synchronous call.
2. promise shows second, because .then passes through the microtask queue, and runs after the current code.
3. timeout shows last, because it's a macrotask.



All microtasks are completed before any other event handling or rendering or any other macrotask takes place.

That's important, as it guarantees that the application environment is basically the same (no mouse coordinate changes, no new network data, etc) between microtasks.

If we'd like to execute a function asynchronously (after the current code), but before changes are rendered or new events handled, we can schedule it with queueMicrotask.

Here’s an example with “counting progress bar”, similar to the one shown previously, but queueMicrotask is used instead of setTimeout. You can see that it renders at the very end. Just like the synchronous code:

```
<div id="progress"></div>
<script>
  let i = 0;
  function count() {
    // do a piece of the heavy job (*)
    do {
      i++;
      progress.innerHTML = i;
    } while (i % 1e3 != 0);
    if (i < 1e6) {
      queueMicrotask(count);
    }
  }
  count();
</script>
```
