TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}June 19, 2022Browser environment, specsThe JavaScript language was initially created for web browsers. Since then, it has evolved into a language with many uses and platforms.

A platform may be a browser, or a web-server or another host, or even a “smart” coffee machine if it can run JavaScript. Each of these provides platform-specific functionality. The JavaScript specification calls that a host environment.

A host environment provides its own objects and functions in addition 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:

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

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

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

// global functions are methods of the global object:
window.sayHi();
```

And we can use it as a browser window, to show the window height:

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

There are more window-specific methods and properties, which we'll cover later.
DOM (Document Object Model)The Document Object Model, or DOM for short, represents all page content as objects that can be modified.
The document object is the main "entry point" to the page. We can change or create anything on the page using it.
For instance:

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

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

Here, we used document.body.style, but there's much, much more. Properties and

methods are described in the specification: DOM Living Standard.

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 the DOM. They may support only a part of the specification though.

CSSOM for styling

There's also a separate specification, CSS Object Model (CSSOM) for CSS rules and stylesheets, that explains how they are represented as objects, and how to read and write them.
The CSSOM is used together with the DOM when we modify style rules for the document. In practice though, the CSSOM is rarely required, because we rarely need to modify CSS rules from JavaScript (usually we just add/remove CSS classes, not modify their CSS rules), but that's also possible.

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 differentiate 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
```

}

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

Specifications

The BOM is a part of the general HTML specification.

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

SummaryTalking about standards, we have:

DOM specification
Describes the document structure, manipulations, and events, see https://dom.spec.whatwg.org.
CSSOM specification
Describes stylesheets and style rules, manipulations with them, and their binding to documents, see https://www.w3.org/TR/cssom-1/.
HTML specification
Describes the HTML language (e.g. tags) and also the BOM (browser object model) – various browser functions: setTimeout, alert, location and so on, see https://html.spec.whatwg.org. It takes the DOM specification and extends it with many additional properties and methods.

Additionally, some classes are described separately at https://spec.whatwg.org/.
Please note these links, as there’s so much to learn that it’s impossible to cover everything and remember it all.
When you’d like to read about a property or a method, the Mozilla manual at https://developer.mozilla.org/en-US/ is also a nice resource, but the corresponding spec may be better: it’s more complex and longer to read, but will make your fundamental knowledge sound and complete.
To find something, it’s often convenient to use an internet search “WHATWG [term]” or “MDN [term]”, e.g https://google.com?q=whatwg+localstorage, https://google.com?q=mdn+localstorage.
Now, we’ll get down to learning the DOM, because the document plays the central role in the UI.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/browser-environment","identifier":"\/browser-environment"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}October 14, 2022DOM treeThe 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 DOMLet'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:

%¾ HTML%¾ HEAD#text !µ$#$#%¾ TITLE#text About elk#text !µ#text !µ%¾ BODY#text !µ$#$#The


```
let node1 = {"name":"HTML","nodeType":1,"children":
[{"name":"HEAD","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"\n  "},
{"name":"TITLE","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"About elk"}]},
{"name":"#text","nodeType":3,"content":"\n"}]},
{"name":"#text","nodeType":3,"content":"\n"},{"name":"BODY","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"\n  The truth about elk.\n"}]}]}
```


```
drawHtmlTree(node1, 'div.domtree', 690, 320);
```

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: !µ (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:

Spaces and newlines before <head> are ignored for historical reasons.
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>
```

%¾ HTML%¾ HEAD%¾ TITLE#text About elk%¾ BODY#text The truth about elk.

```
let node2 = {"name":"HTML","nodeType":1,"children":
[{"name":"HEAD","nodeType":1,"children":[{"name":"TITLE","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"About elk"}]}]},
{"name":"BODY","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"The
truth about elk."}]}]}

drawHtmlTree(node2, 'div.domtree', 690, 210);
```

Spaces at string start/end and space-only text nodes are usually hidden in tools
Browser tools (to be covered soon) that work with DOM usually do not show spaces at the start/end of the text and empty text nodes (line-breaks) between tags. Developer tools save screen space this way.
On further DOM pictures we’ll sometimes omit them when they are irrelevant. Such spaces usually do not affect how the document is displayed.

AutocorrectionIf 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:
%¾ HTML%¾ HEAD%¾ BODY#text Hello

let node3 = {"name":"HTML","nodeType":1,"children":
[{"name":"HEAD","nodeType":1,"children":[]},{"name":"BODY","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"Hello"}]}]}

drawHtmlTree(node3, 'div.domtree', 690, 150);

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:
%¾ HTML%¾ HEAD%¾ BODY%¾ P#text Hello%¾ LI#text Mom%¾ LI#text and%¾ LI#text Dad

let node4 = {"name":"HTML","nodeType":1,"children":
[{"name":"HEAD","nodeType":1,"children":[]},{"name":"BODY","nodeType":1,"children":
[{"name":"P","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"Hello"}]},
{"name":"LI","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"Mom"}]},
{"name":"LI","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"and"}]},
{"name":"LI","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"Dad"}]}]}]}

drawHtmlTree(node4, 'div.domtree', 690, 360);

Tables always have <tbody>
An interesting "special case" is tables. By DOM specification they must have <tbody> tag, but HTML text may omit it. Then the browser creates <tbody> in the DOM automatically.
For the HTML:

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

DOM-structure will be:
%¾ TABLE%¾ TBODY%¾ TR%¾ TD#text 1

```
let node5 = {"name":"TABLE","nodeType":1,"children":
[{"name":"TBODY","nodeType":1,"children":[{"name":"TR","nodeType":1,"children":
[{"name":"TD","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"1"}]}]}]}]};

drawHtmlTree(node5,  'div.domtree', 600, 200);
```

You see? The <tbody> appeared out of nowhere. We should keep this in mind while working with tables to avoid surprises.

Other node typesThere 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>
```

%¾ HTML%¾ HEAD%¾ BODY#text !µ$#$#The truth about elk.!µ$#$#%¾ OL#text !µ$#$#$#$#%
smart#text !µ$#$#$#$##comment comment#text !µ$#$#$#$#%¾ LI#text ...and cunning animal!#text !µ

```
let node6 = {"name":"HTML","nodeType":1,"children":
[{"name":"HEAD","nodeType":1,"children":[]},{"name":"BODY","nodeType":1,"children":
```

[{"name":"#text","nodeType":3,"content":"\n The truth about elk.\n "},
{"name":"OL","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"\n   "},
{"name":"LI","nodeType":1,"children":[{"name":"#text","nodeType":3,"content":"An elk is a smart"}]},{"name":"#text","nodeType":3,"content":"\n   "},
{"name":"#comment","nodeType":8,"content":"comment"},
{"name":"#text","nodeType":3,"content":"\n   "},{"name":"LI","nodeType":1,"children":
[{"name":"#text","nodeType":3,"content":"...and cunning animal!"}]},
{"name":"#text","nodeType":3,"content":"\n "}]},
{"name":"#text","nodeType":3,"content":"\n\n\n"}]}];

drawHtmlTree(node6, 'div.domtree', 690, 500);

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>. Few people know about that. We are not going to touch that node, we even don't draw it on diagrams, but it's there.
The document object that represents the whole document is, formally, a DOM node as well.
There are 12 node types. In practice we usually work with 4 of them:

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

See it for yourselfTo 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:

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.

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.

The best way to study them is to click around. Most values are editable in-place.
Interaction with consoleAs we work the DOM, we also may want to apply JavaScript to it. Like: get a node and run some code to modify it, to see the result. Here are few tips to travel between the Elements tab and the console.
For the start:

Select the first <li> in the Elements tab.
Press Esc – it will open console right below the Elements tab.

Now the last selected element is available as $0, the previously selected is $1 etc.
We can run commands on them. For instance, $0.style.background = 'red' makes the selected list item red, like this:

That’s how to get a node from Elements in Console.
There’s also a road back. If there’s a variable referencing a DOM node, then we can use the command inspect(node) in Console to see it in the Elements pane.
Or we can just output the DOM node in the console and explore “in-place”, like document.body below:

That’s for debugging purposes of course. From the next chapter on we’ll access and modify DOM using JavaScript.
The browser developer tools are a great help in development: we can explore the DOM, try things and see what goes wrong.
SummaryAn HTML/XML document is represented inside the browser as the DOM tree.

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

We can use developer tools to inspect DOM and modify it manually.
Here we covered the basics, the most used and important actions to start with. There’s an extensive documentation about Chrome Developer Tools at https://developers.google.com/web/tools/chrome-devtools. The best way to learn the tools is to click here and there, read menus: most options are obvious. Later, when you know them in general, read the docs and pick up the rest.
DOM nodes have properties and methods that allow us to travel between them, modify them, move around the page, and more. We’ll get down to them in the next chapters.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/dom-nodes","identifier":"\/dom-nodes"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}December 12, 2021Walking the DOMThe 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:

Let’s discuss them in more detail.

On top: documentElement and bodyThe 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, lastChildThere 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 and <ul> (and few blank text nodes):

<html>
<body>
  Begin

      Information

</body>
</html>

…And descendants of <body> are not only direct children , <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 collectionsAs 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:

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).

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 by something else by assigning childNodes[i] = ...

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

DOM collections are live

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

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

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

Collections are iterable using for..of. Sometimes people try to use for..in for that. Please, don't. The for..in loop iterates over all enumerable properties. And collections have some "extra" rarely used properties that we usually do not want to get:

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

Siblings and the parentSiblings 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 navigationNavigation properties listed above refer to all nodes. For instance, in childNodes we can see both text nodes, element nodes, and even comment nodes if they 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: tablesTill 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>
```

The specification: tabular data.

There are also additional navigation properties for HTML forms. We'll look at them later when we start working with forms.

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

There are two main sets of them:

For all nodes: parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling.

For element nodes only: parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling.

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

TasksDOM childrenimportance: 5Look at this page:

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

For each of the following, give at least one way of how to access them:

The DOM node?
The <ul> DOM node?
The second <li> (with Pete)?

solutionThere are many ways, for instance:
The DOM node:

```
document.body.firstElementChild
// or
document.body.children[0]
// or (the first node is space, so we take 2nd)
document.body.childNodes[1]
```

The <ul> DOM node:

```
document.body.lastElementChild
// or
document.body.children[1]
```

The second <li> (with Pete):

```
// get <ul>, and then get its last element child
document.body.lastElementChild.lastElementChild
```

The sibling questionimportance: 5If elem – is an arbitrary DOM element node…

Is it true that elem.lastChild.nextSibling is always null?

Is it true that elem.children[0].previousSibling is always null ?

solution
Yes, true. The element elem.lastChild is always the last one, it has no nextSibling.
No, wrong, because elem.children[0] is the first child among elements. But there may exist non-element nodes before it. So previousSibling may be a text node.

Please note: for both cases if there are no children, then there will be an error.
If there are no children, elem.lastChild is null, so we can’t access elem.lastChild.nextSibling. And the collection elem.children is empty (like an empty array []).
Select all diagonal cellsimportance: 5Write the code to paint all diagonal table cells in red.
You’ll need to get all diagonal <td> from the <table> and paint them using the code:

```
// td should be the reference to the table cell
td.style.backgroundColor = 'red';
```

The result should be:

Open a sandbox for the task.solutionWe’ll be using rows and cells properties to access diagonal table cells.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/dom-navigation","identifier":"\/dom-navigation"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}October 14, 2022Searching: 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 idIf 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, 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.

Only document.getElementById, not anyElem.getElementById

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

querySelectorAllBy 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).

querySelectorThe 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.
matchesPrevious 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>
```

closestAncestors 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 for the nearest ancestor that matches the CSS-selector. The elem itself is also included in the search.
In other words, the method closest goes up from the element and checks each of parents. If it matches the selector, then the search stops, and the ancestor is returned.
For instance:

```
        <h1>Contents</h1>

<div class="contents">
  <ul class="book">
    <li class="chapter">Chapter 1</li>
    <li class="chapter">Chapter 2</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>
```

Don’t forget the "s" letter!

Novice developers sometimes forget the letter "s". That is, they try to call getElementByTagName instead of getElementsByTagName.
The "s" letter is absent in getElementById, because it returns a single element. But getElementsByTagName returns a collection of elements, so there’s "s" inside.

It returns a collection, not an element!

Another widespread novice mistake is to write:

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

That won’t work, because it takes a collection of inputs and assigns the value to it rather than to elements inside it.
We should either iterate over the collection or get an element by its index, and then assign, like this:

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

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 collectionsAll methods "getElementsBy*" return a live collection. Such collections always reflect the current state of the document and “auto-update” when it changes.
In the example below, there are two scripts.

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

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

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

<div>Second div</div>

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

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.

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

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

'
'

By far the most used are querySelector and querySelectorAll, but getElement(s)By* can be sporadically helpful or found in the old scripts.
Besides that:

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

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

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

TasksSearch for elementsimportance: 4Here’s the document with the table and form.
How to find?…

The table with id="age-table".
All label elements inside that table (there should be 3 of them).
The first td in that table (with the word “Age”).
The form with name="search".
The first input in that form.
The last input in that form.

Open the page table.html in a separate window and make use of browser tools for that.
solutionThere are many ways to do it.
Here are some of them:

```
        // 1. The table with `id="age-table"`.
let table = document.getElementById('age-table')

// 2. All label elements inside that table
table.getElementsByTagName('label')
// or
document.querySelectorAll('#age-table label')

// 3. The first td in that table (with the word "Age")
table.rows[0].cells[0]
```

```
// or
table.getElementsByTagName('td')[0]
// or
table.querySelector('td')

// 4. The form with the name "search"
// assuming there's only one element with name="search" in the document
let form = document.getElementsByName('search')[0]
// or, form specifically
document.querySelector('form[name="search"]')

// 5. The first input in that form.
form.getElementsByTagName('input')[0]
// or
form.querySelector('input')

// 6. The last input in that form
let inputs = form.querySelectorAll('input') // find all inputs
inputs[inputs.length-1] // take the last one
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/searching-elements-dom","identifier":"\/searching-elements-dom"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}August 5, 2022Node properties: type, tag and contentsLet's get a more in-depth look at DOM nodes.
In this chapter we'll see more into what they are and learn their most used properties.
DOM node classesDifferent DOM nodes may have different properties. For instance, an element node corresponding to tag <a> has link-related properties, and the one corresponding to <input> has input-related properties and so on. Text nodes are not the same as element nodes. But there are also common properties and methods between all of them, because all classes of DOM nodes form a single hierarchy.
Each DOM node belongs to the corresponding built-in class.

The root of the hierarchy is EventTarget, that is inherited by Node, and other DOM nodes inherit from it.
Here's the picture, explanations to follow:

The classes are:

EventTarget – is the root "abstract" class for everything.
Objects of that class are never created. It serves as a base, so that all DOM nodes support so-called "events", we'll study them later.

Node – is also an "abstract" class, serving as a base for DOM nodes.
It provides the core tree functionality: parentNode, nextSibling, childNodes and so on (they are getters). Objects of Node class are never created. But there are other classes that inherit from it (and so inherit the Node functionality).

Document, for historical reasons often inherited by HTMLDocument (though the latest spec doesn't dictate it) – is a document as a whole.
The document global object belongs exactly to this class. It serves as an entry point to the DOM.

CharacterData – an "abstract" class, inherited by:

Text – the class corresponding to a text inside elements, e.g. Hello in Hello .
Comment – the class for comments. They are not shown, but each comment becomes a member of DOM.

Element – is the base class for DOM elements.
It provides element-level navigation like nextElementSibling, children and searching methods like getElementsByTagName, querySelector.
A browser supports not only HTML, but also XML and SVG. So the Element class serves as a base for more specific classes: SVGElement, XMLElement (we don't need them here) and HTMLElement.

Finally, HTMLElement is the basic class for all HTML elements. We'll work with it most of the time.

It is inherited by concrete HTML elements:

HTMLInputElement – the class for <input> elements,
HTMLBodyElement – the class for <body> elements,
HTMLAnchorElement – the class for <a> elements,
…and so on.

There are many other tags with their own classes that may have specific properties and methods, while some elements, such as , <section>, <article> do not have any specific properties, so they are instances of HTMLElement class.
So, the full set of properties and methods of a given node comes as the result of the chain of inheritance.
For example, let's consider the DOM object for an <input> element. It belongs to HTMLInputElement class.
It gets properties and methods as a superposition of (listed in inheritance order):

HTMLInputElement – this class provides input-specific properties,
HTMLElement – it provides common HTML element methods (and getters/setters),
Element – provides generic element methods,
Node – provides common DOM node properties,
EventTarget – gives the support for events (to be covered),
…and finally it inherits from Object, so "plain object" methods like hasOwnProperty are also available.

To see the DOM node class name, we can recall that an object usually has the constructor property. It references the class constructor, and constructor.name is its name:

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

…Or we can just toString it:

```
alert( document.body ); // [object HTMLBodyElement]
```

We also can use instanceof to check the inheritance:

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

As we can see, DOM nodes are regular JavaScript objects. They use prototype-based classes for inheritance.
That's also easy to see by outputting an element with console.dir(elem) in a browser. There in the console you can see HTMLElement.prototype, Element.prototype and so on.

console.dir(elem) versus console.log(elem)

Most browsers support two commands in their developer tools: console.log and console.dir. They output their arguments to the console. For JavaScript objects these

commands usually do the same.
But for DOM elements they are different:

console.log(elem) shows the element DOM tree.
console.dir(elem) shows the element as a DOM object, good to explore its properties.

Try it on document.body.

IDL in the spec

In the specification, DOM classes aren’t described by using JavaScript, but a special Interface description language (IDL), that is usually easy to understand.
In IDL all properties are prepended with their types. For instance, DOMString, boolean and so on.
Here’s an excerpt from it, with comments:

```
// Define HTMLInputElement
// The colon ":" means that HTMLInputElement inherits from HTMLElement
interface HTMLInputElement: HTMLElement {
  // here go properties and methods of <input> elements

  // "DOMString" means that the value of a property is a string
  attribute DOMString accept;
  attribute DOMString alt;
  attribute DOMString autocomplete;
  attribute DOMString value;

  // boolean value property (true/false)
  attribute boolean autofocus;
  ...
  // now the method: "void" means that the method returns no value
  void select();
  ...
}
```

The “nodeType” propertyThe nodeType property provides one more, “old-fashioned” way to get the “type” of a DOM node.
It has a numeric value:

elem.nodeType == 1 for element nodes,

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

For instance:

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

  // let's examine: what type of node is in elem?
  alert(elem.nodeType); // 1 => element

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

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

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

The tag name is always uppercase except in XML mode

The browser has two modes of processing documents: HTML and XML. Usually the HTML-mode is used for webpages. XML-mode is enabled when the browser receives an XML-document with the header: Content-Type: application/xml+xhtml.
In HTML mode tagName/nodeName is always uppercased: it's BODY either for <body> or <BoDy>.
In XML mode the case is kept "as is". Nowadays XML mode is rarely used.

innerHTML: the contentsThe 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 overwriteWe 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:

The old contents is removed.
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 elementThe 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 A new element . In the outer document (the DOM) we can see the new content instead of the . 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 A new element 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 contentThe 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>
```

For text nodes we can imagine a reason to read or modify them, but why comments?
Sometimes developers embed information or template instructions into HTML in them, like this:

```
        <!-- if isAdmin -->
  <div>Welcome, Admin!</div>
<!-- /if -->
```

…Then JavaScript can read it from data property and process embedded instructions.
textContent: pure textThe 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 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>
```

The first gets the name “as HTML”: all tags become tags, so we see the bold name.
The second gets the name “as text”, so we literally see Winnie-the-Pooh!.

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” propertyThe “hidden” attribute and the DOM property specifies whether the element is visible or not.
We can use it in HTML or assign it 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 propertiesDOM 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>
```

Most standard HTML attributes have the corresponding DOM property, and we can access it like that.
If we want to know the full list of supported properties for a given class, we can find them in the specification. For instance, HTMLInputElement is documented at https://html.spec.whatwg.org/#htmlinputelement.
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.
SummaryEach DOM node belongs to a certain class. The classes form a hierarchy. The full set of properties and methods come as the result of inheritance.
Main DOM node properties are:

nodeType
We can use it to see if a node is a text or an element node. It has a numeric value: 1 for elements,3 for text nodes, and a few others for other node types. Read-only.
nodeName/tagName
For elements, tag name (uppercased unless XML-mode). For non-element nodes nodeName describes what it is. Read-only.
innerHTML
The HTML content of the element. Can be modified.
outerHTML
The full HTML of the element. A write operation into elem.outerHTML does not touch elem itself. Instead it gets replaced with the new HTML in the outer context.
nodeValue/data
The content of a non-element node (text, comment). These two are almost the same, usually we use data. Can be modified.

textContent
The text inside the element: HTML minus all <tags>. Writing into it puts the text inside the element, with all special characters and tags treated exactly as text. Can safely insert user-generated text and protect from unwanted HTML insertions.
hidden
When set to true, does the same as CSS display:none.

DOM nodes also have other properties depending on their class. For instance, <input> elements (HTMLInputElement) support value, type, while <a> elements (HTMLAnchorElement) support href etc. Most standard HTML attributes have a corresponding DOM property.
However, HTML attributes and DOM properties are not always the same, as we'll see in the next chapter.
TasksCount descendantsimportance: 5There's a tree structured as nested ul/li.
Write the code that for each <li> shows:

What's the text inside it (without the subtree)
The number of nested <li> – all descendants, including the deeply nested ones.

Demo in new windowOpen a sandbox for the task.solutionLet's make a loop over <li>:

```
for (let li of document.querySelectorAll('li')) {
  ...
}
```

In the loop we need to get the text inside every li.
We can read the text from the first child node of li, that is the text node:

```
for (let li of document.querySelectorAll('li')) {
  let title = li.firstChild.data;

  // title is the text in <li> before any other nodes
}
```

Then we can get the number of descendants as li.getElementsByTagName('li').length.

Open the solution in a sandbox.What's in the nodeType?importance: 5What does the script show?

```
<html>

<body>
  <script>
    alert(document.body.lastChild.nodeType);
  </script>
</body>

</html>
```

solutionThere’s a catch here.
At the time of <script> execution the last DOM node is exactly <script>, because the browser did not process the rest of the page yet.
So the result is 1 (element node).

```
<html>

<body>
  <script>
    alert(document.body.lastChild.nodeType);
  </script>
</body>

</html>
```

Tag in commentimportance: 3What does this code show?

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

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // what's here?
</script>
```

solutionThe answer: BODY.

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

  body.innerHTML = "<!--" + body.tagName + "-->";

  alert( body.firstChild.data ); // BODY
</script>
```

What’s going on step by step:

The content of <body> is replaced with the comment. The comment is <!--BODY-->, because body.tagName == "BODY". As we remember, tagName is always uppercase in HTML.
The comment is now the only child node, so we get it in body.firstChild.
The data property of the comment is its contents (inside <!--...-->): "BODY".

Where's the "document" in the hierarchy?importance: 4Which class does the document belong to?
What’s its place in the DOM hierarchy?
Does it inherit from Node or Element, or maybe HTMLElement?
solutionWe can see which class it belongs by outputting it, like:

```
alert(document); // [object HTMLDocument]
```

Or:

```
alert(document.constructor.name); // HTMLDocument
```

So, document is an instance of HTMLDocument class.
What’s its place in the hierarchy?
Yeah, we could browse the specification, but it would be faster to figure out manually.
Let’s traverse the prototype chain via __proto__.
As we know, methods of a class are in the prototype of the constructor. For instance, HTMLDocument.prototype has methods for documents.
Also, there’s a reference to the constructor function inside the prototype:

```
alert(HTMLDocument.prototype.constructor === HTMLDocument); // true
```

To get a name of the class as a string, we can use constructor.name. Let’s do it for the whole document prototype chain, till class Node:

```
alert(HTMLDocument.prototype.constructor.name); // HTMLDocument
alert(HTMLDocument.prototype.__proto__.constructor.name); // Document
alert(HTMLDocument.prototype.__proto__.__proto__.constructor.name); // Node
```

That’s the hierarchy.
We also could examine the object using console.dir(document) and see these names by opening __proto__. The console takes them from constructor internally.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/basic-dom-node-properties","identifier":"\/basic-dom-node-properties"}); };var disqus_shortname =

"javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}August 21, 2022Attributes and propertiesWhen 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 propertiesWe’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 attributesIn 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:

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.
We can assign anything to an attribute, but it becomes a string. So here we have "123" as the value.
All attributes including ones that we set are visible in outerHTML.
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 synchronizationWhen 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 !’ 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 typedDOM 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, datasetWhen 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.

Summary

Attributes – is what's written in HTML.

Properties – is what's in DOM objects.

A small comparison:

Properties
Attributes

Type
Any value, standard properties have types described in the spec
A string

Name
Name is case-sensitive
Name is not case-sensitive

Methods to work with attributes are:

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

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

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

TasksGet the attributeimportance: 5Write the code to select the element with data-widget-name attribute from the document and to read its value.

```
        <!DOCTYPE html>
<html>
<body>

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

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

solution

```
        <!DOCTYPE html>
<html>
<body>

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

  <script>
    // getting it
    let elem = document.querySelector('[data-widget-name]');

    // reading the value
    alert(elem.dataset.widgetName);
    // or
    alert(elem.getAttribute('data-widget-name'));
  </script>
</body>
</html>
```

Make external links orangeimportance: 3Make all external links orange by altering their style property.
A link is external if:

Its href has :// in it
But doesn’t start with http://internal.com.

Example:

```
        <a name="list">the list</a>
<ul>
  <li><a href="http://google.com">http://google.com</a></li>
  <li><a href="/tutorial">/tutorial.html</a></li>
  <li><a href="local/path">local/path</a></li>
  <li><a href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
  <li><a href="http://nodejs.org">http://nodejs.org</a></li>
  <li><a href="http://internal.com/test">http://internal.com/test</a></li>
</ul>

<script>
  // setting style for a single link
  let link = document.querySelector('a');
  link.style.color = 'orange';
</script>
```

The result should be:

Open a sandbox for the task.solutionFirst, we need to find all external references. There are two ways.

The first is to find all links using document.querySelectorAll('a') and then filter out what we need:

```
let links = document.querySelectorAll('a');

for (let link of links) {
  let href = link.getAttribute('href');
  if (!href) continue; // no attribute

  if (!href.includes('://')) continue; // no protocol

  if (href.startsWith('http://internal.com')) continue; // internal

  link.style.color = 'orange';
}
```

Please note: we use link.getAttribute('href'). Not link.href, because we need the value from HTML.
…Another, simpler way would be to add the checks to CSS selector:

```
// look for all links that have :// in href
// but href doesn't start with http://internal.com
let selector = 'a[href*="://"]:not([href^="http://internal.com"])';
let links = document.querySelectorAll(selector);

links.forEach(link => link.style.color = 'orange');
```

Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/dom-attributes-and-properties","identifier":"\/dom-attributes-and-properties"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}April 30, 2022Modifying the documentDOM 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 messageLet’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 the HTML example. Now let’s create the same div with JavaScript (assuming that the styles are in the HTML/CSS already).
Creating an elementTo 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');
```

Most of the time we need to create element nodes, such as the div for the message.
Creating the messageCreating the message div takes 3 steps:

```
// 1. Create <div> element
let div = document.createElement('div');

// 2. Set its class to "alert"
div.className = "alert";

// 3. Fill it with the content
div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";
```

We've created the element. But as of now it's only in a variable named div, not in the page yet. So we can't see it.
Insertion methodsTo make the div show up, we need to insert it somewhere into document. For instance, into <body> element, referenced by 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>
```

Here we called append on document.body, but we can call append method on any other element, to put another element into it. For instance, we can append something to by calling div.append(anotherElement).
Here are more insertion methods, they specify different places where 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.

Arguments of these methods are an arbitrary list of DOM nodes to insert, or text strings (that become text nodes automatically).
Let’s see them in action.
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 of what the 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
```

As said, these methods can insert multiple 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>
```

Please note: the text is inserted “as text”, not “as HTML”, with proper escaping of characters such as <, >.
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’d like to insert an HTML string “as html”, with all tags and stuff working,

in the same manner as elem.innerHTML does it?
insertAdjacentHTML/Text/ElementFor 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.

They exist mainly to make the syntax “uniform”. In practice, only insertAdjacentHTML is used most of the time. Because for elements and text, we have methods append/ prepend/before/after – they are shorter to write and can insert nodes/text pieces.
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 removalTo 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>
```

Cloning nodes: cloneNodeHow 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>
```

DocumentFragmentDocumentFragment 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

Old school

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.
Summary

Methods to create new nodes:

document.createElement(tag) – creates an element with the given tag,
document.createTextNode(value) – creates a text node (rarely used),
elem.cloneNode(deep) – clones the element, if deep==true then with all descendants.

Insertion and removal:

node.append(...nodes or strings) – insert into node, at the end,
node.prepend(...nodes or strings) – insert into node, at the beginning,
node.before(...nodes or strings) –- insert right before node,
node.after(...nodes or strings) –- insert right after node,
node.replaceWith(...nodes or strings) –- replace node.
node.remove() –- remove the node.

Text strings are inserted “as text”.

There are also “old school” methods:

parent.appendChild(node)

parent.insertBefore(node, nextSibling)
parent.removeChild(node)
parent.replaceChild(newElem, node)

All these methods return node.

Given some HTML in html, elem.insertAdjacentHTML(where, html) inserts it depending on the value of where:

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

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

To append HTML to the page before it has finished loading:

document.write(html)

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

TaskscreateTextNode vs innerHTML vs textContentimportance: 5We have an empty DOM element elem and a string text.
Which of these 3 commands will do exactly the same?

elem.append(document.createTextNode(text))
elem.innerHTML = text
elem.textContent = text

solutionAnswer: 1 and 3.
Both commands result in adding the text “as text” into the elem.
Here’s an example:

```
        <div id="elem1"></div>
<div id="elem2"></div>
<div id="elem3"></div>
<script>
  let text = '<b>text</b>';

  elem1.append(document.createTextNode(text));
  elem2.innerHTML = text;
  elem3.textContent = text;
</script>
```

Clear the elementimportance: 5Create a function clear(elem) that removes everything from the element.

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

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

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

solutionFirst, let’s see how not to do it:

```
        function clear(elem) {
  for (let i=0; i < elem.childNodes.length; i++) {
     elem.childNodes[i].remove();
  }
}
```

That won’t work, because the call to remove() shifts the collection elem.childNodes, so elements start from the index 0 every time. But i increases, and some elements will be skipped.
The for..of loop also does the same.
The right variant could be:

```
        function clear(elem) {
  while (elem.firstChild) {
    elem.firstChild.remove();
  }
}
```

And also there’s a simpler way to do the same:

```
        function clear(elem) {
  elem.innerHTML = '';
}
```

Why does "aaa" remain?importance: 1In the example below, the call table.remove() removes the table from the document.
But if you run it, you can see that the text "aaa" is still visible.
Why does that happen?

```
        <table id="table">
  aaa
  <tr>
    <td>Test</td>
  </tr>
</table>

<script>
  alert(table); // the table, as it should be

  table.remove();
  // why there's still "aaa" in the document?
</script>
```

solutionThe HTML in the task is incorrect. That’s the reason of the odd thing.
The browser has to fix it automatically. But there may be no text inside the <table>: according to the spec only table-specific tags are allowed. So the browser shows "aaa" before the <table>.
Now it’s obvious that when we remove the table, it remains.
The question can be easily answered by exploring the DOM using the browser tools. You’ll see "aaa" before the <table>.
The HTML standard specifies in detail how to process bad HTML, and such behavior of the browser is correct.
Create a listimportance: 4Write an interface to create a list from user input.
For every list item:

Ask a user about its content using prompt.
Create the <li> with it and add it to <ul>.
Continue until the user cancels the input (by pressing Esc or via an empty entry).

All elements should be created dynamically.
If a user types HTML-tags, they should be treated like a text.
Demo in new windowsolutionPlease note the usage of textContent to assign the <li> content.
Open the solution in a sandbox.Create a tree from the objectimportance: 5Write a

function createTree that creates a nested ul/li list from the nested object.
For instance:

```
        let data = {
  "Fish": {
    "trout": {},
    "salmon": {}
  },

  "Tree": {
    "Huge": {
      "sequoia": {},
      "oak": {}
    },
    "Flowering": {
      "apple tree": {},
      "magnolia": {}
    }
  }
};
```

The syntax:

```
        let container = document.getElementById('container');
createTree(container, data); // creates the tree in the container
```

The result (tree) should look like this:

Choose one of two ways of solving this task:

Create the HTML for the tree and then assign to container.innerHTML.
Create tree nodes and append with DOM methods.

Would be great if you could do both.

P.S. The tree should not have “extra” elements like empty for the leaves.
Open a sandbox for the task.solutionThe easiest way to walk the object is to use recursion.

The solution with innerHTML.
The solution with DOM.

Show descendants in a treeimportance: 5There’s a tree organized as nested ul/li. Write the code that adds to each <li> the number of its descendants. Skip leaves (nodes without children).
The result:

Open a sandbox for the task.solutionTo append text to each <li> we can alter the text node data.
Open the solution in a sandbox.Create a calendarimportance: 4Write a function createCalendar(elem, year, month).
The call should create a calendar for the given year/month and put it inside elem.
The calendar should be a table, where a week is <tr>, and a day is <td>. The table top should be <th> with weekday names: the first day should be Monday, and so on till Sunday.
For instance, createCalendar(cal, 2012, 9) should generate in element cal the following calendar:

P.S. For this task it’s enough to generate the calendar, should not yet be clickable.
Open a sandbox for the task.solutionWe’ll create the table as a string: "<table>...</table>", and then assign it to innerHTML.
The algorithm:

Create the table header with <th> and weekday names.
Create the date object d = new Date(year, month-1). That’s the first day of month (taking into account that months in JavaScript start from 0, not 1).
First few cells till the first day of the month d.getDay() may be empty. Let’s fill them in with <td></td>.
Increase the day in d: d.setDate(d.getDate()+1). If d.getMonth() is not yet the next month, then add the new cell <td> to the calendar. If that’s a Sunday, then add a newline “</tr><tr>”.
If the month has finished, but the table row is not yet full, add empty <td> into it, to make it square.

Open the solution in a sandbox.Colored clock with setIntervalimportance: 4Create a colored clock like here:

Use HTML/CSS for the styling, JavaScript only updates time in elements.
Open a sandbox for the task.solutionFirst, let's make HTML/CSS.
Each component of the time would look great in its own :

```
<div id="clock">
  <span class="hour">hh</span>:<span class="min">mm</span>:<span
class="sec">ss</span>
</div>
```

Also we'll need CSS to color them.
The update function will refresh the clock, to be called by setInterval every second:

```
function update() {
  let clock = document.getElementById('clock');
  let date = new Date(); // (*)
  let hours = date.getHours();
  if (hours < 10) hours = '0' + hours;
  clock.children[0].innerHTML = hours;

  let minutes = date.getMinutes();
  if (minutes < 10) minutes = '0' + minutes;
  clock.children[1].innerHTML = minutes;

  let seconds = date.getSeconds();
  if (seconds < 10) seconds = '0' + seconds;
  clock.children[2].innerHTML = seconds;
}
```

In the line (*) we every time check the current date. The calls to setInterval are not reliable: they may happen with delays.
The clock-managing functions:

```
        let timerId;

function clockStart() { // run the clock
  if (!timerId) { // only set a new interval if the clock is not running
    timerId = setInterval(update, 1000);
  }
  update(); // (*)
}

function clockStop() {
  clearInterval(timerId);
  timerId = null; // (**)
}
```

Please note that the call to update() is not only scheduled in clockStart(), but immediately run in the line (*). Otherwise the visitor would have to wait till the first execution of setInterval. And the clock would be empty till then.
Also it is important to set a new interval in clockStart() only when the clock is not running. Otherways clicking the start button several times would set multiple concurrent intervals. Even worse – we would only keep the timerID of the last interval, losing references to all others. Then we wouldn't be able to stop the clock ever again! Note that we need to clear the timerID when the clock is stopped in the line (**), so that it can be started again by running clockStart().
Open the solution in a sandbox.Insert the HTML in the listimportance: 5Write the code to insert 2
3 between two <li> here:

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

solutionWhen we need to insert a piece of HTML somewhere, insertAdjacentHTML is the best fit.
The solution:

```
one.insertAdjacentHTML('afterend', '<li>2</li><li>3</li>');
```

Sort the tableimportance: 5There’s a table:

```
<table>
<thead>
  <tr>
    <th>Name</th><th>Surname</th><th>Age</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>John</td><td>Smith</td><td>10</td>
  </tr>
  <tr>
    <td>Pete</td><td>Brown</td><td>15</td>
  </tr>
  <tr>
    <td>Ann</td><td>Lee</td><td>5</td>
  </tr>
  <tr>
    <td>...</td><td>...</td><td>...</td>
  </tr>
</tbody>
</table>
```

There may be more rows in it.
Write the code to sort it by the "name" column.
Open a sandbox for the task.solutionThe solution is short, yet may look a bit tricky, so here I provide it with extensive comments:

```
        let sortedRows = Array.from(table.tBodies[0].rows) // 1
  .sort((rowA, rowB) =>
rowA.cells[0].innerHTML.localeCompare(rowB.cells[0].innerHTML));

table.tBodies[0].append(...sortedRows); // (3)
```

The step-by-step algorthm:

Get all <tr>, from <tbody>.
Then sort them comparing by the content of the first <td> (the name field).
Now insert nodes in the right order by .append(...sortedRows).

We don’t have to remove row elements, just “re-insert”, they leave the old place automatically.
P.S. In our case, there’s an explicit <tbody> in the table, but even if HTML table doesn’t have <tbody>, the DOM structure always has it.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/modifying-document","identifier":"\/modifying-document"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}September 29, 2022Styles and classesBefore 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:

Create a class in CSS and add it:
Write properties directly into 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 classListChanging a class is one of the most often used actions in scripts.
In the ancient time, there was a limitation in JavaScript: a reserved word like "class" could not be an object property. That limitation does not exist now, but at that time it was impossible to have a "class" property, like elem.class.
So for classes the similar-looking property "className" was introduced: the elem.className corresponds to the "class" attribute.
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 styleThe 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 propertySometimes 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.
Also there is a special method for that, elem.style.removeProperty('style property'). So, We can remove a property like this:

```
        document.body.style.background = 'red'; //set background to red

setTimeout(() => document.body.style.removeProperty('background'), 1000); // remove background after 1 second
```

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 unitsDon’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

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

Please note: the browser “unpacks” the property style.margin in the last lines and infers style.marginLeft and style.marginTop from it.
Computed styles: getComputedStyleSo, 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:

A computed style value is the value after all CSS rules and CSS inheritance is applied, as the result of the CSS cascade. It can look like height:1em or font-size:125%.
A resolved style value is the one finally applied to the element. Values like 1em or 125% are relative. The browser takes the computed value and makes all units fixed and absolute, for instance: height:20px or font-size:16px. 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.

Styles applied to :visited links are hidden!

Visited links may be colored using :visited CSS pseudoclass.
But getComputedStyle does not give access to that color, because otherwise an arbitrary page could find out whether the user visited a link by creating it on the page and checking the styles.
JavaScript may not see the styles applied by :visited. And also, there’s a limitation in CSS that forbids applying geometry-changing styles in :visited. That’s to guarantee that there’s no side way for an evil page to test if a link was visited and hence to break the privacy.

SummaryTo manage classes, there are two DOM properties:

className – the string value, good to manage the whole set of classes.
classList – the object with methods add/remove/toggle/contains, good for individual classes.

To change the styles:

The style property is an object with camelCased styles. Reading and writing to it has the same meaning as modifying individual properties in the "style" attribute. To see how to apply important and other rare stuff – there’s a list of methods at MDN.

The style.cssText property corresponds to the whole "style" attribute, the full string of styles.

To read the resolved styles (with respect to all classes, after all CSS is applied and final values are calculated):

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

TasksCreate a notificationimportance: 5Write a function showNotification(options) that creates a notification: with the given content. The notification should automatically disappear after 1.5 seconds.
The options are:

```
// shows an element with the text "Hello" near the right-top of the window
showNotification({
  top: 10, // 10px from the top of the window (by default 0px)
  right: 10, // 10px from the right edge of the window (by default 0px)
  html: "Hello!", // the HTML of notification
  className: "welcome" // an additional class for the div (optional)
});
```

Demo in new windowUse CSS positioning to show the element at given top/right coordinates. The source document has the necessary styles.
Open a sandbox for the task.solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/styles-and-classes","identifier":"\/styles-and-classes"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}June 26, 2022Element size and scrollingThere 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 elementAs 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:

You can open the document in the sandbox.

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.

The padding-bottom area may be filled with text

Usually paddings are shown empty on our illustrations, but if there’s a lot of text in the element and it overflows, then browsers show the “overflowing” text at padding-bottom, that’s normal.

GeometryHere’s the overall picture with geometry properties:

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/TopThese 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:

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

Properties offsetLeft/offsetTop provide x/y coordinates relative to offsetParent upper-left corner.
In the example below the inner 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>
```

There are several occasions when offsetParent is null:

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

offsetWidth/HeightNow 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.

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 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 its 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.

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

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, total 324px.
If there are no paddings, then clientWidth/Height is exactly the content area, inside the borders and the scrollbar (if any).

So when there's no padding we can use clientWidth/clientHeight to get the content area size.
scrollWidth/HeightThese properties are like clientWidth/clientHeight, but they also include the scrolled out (hidden) parts:

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`;
```

Click the button to expand the element:
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
text text text text text text text text text text text text text text text text text text text text
element.style.height = ${element.scrollHeight}px
scrollLeft/scrollTopProperties 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.

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.

If you click the element below, the code elem.scrollTop += 10 executes. That makes the element content scroll 10px down.
ClickMe123456789
Setting scrollTop to 0 or a big value, such as 1e9 will make the element scroll to the very top/bottom respectively.

Don’t take width/height from CSSWe’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:

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.

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.
If your browser reserves the space for a scrollbar (most browsers for Windows do), then you can test it below.

The element with text has CSS width:300px.
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.
SummaryElements have the following geometry properties:

offsetParent – is the nearest positioned ancestor or td, th, table, body.
offsetLeft/offsetTop – coordinates relative to the upper-left edge of offsetParent.
offsetWidth/offsetHeight – "outer" width/height of an element including borders.
clientLeft/clientTop – the distances from the upper-left outer corner to the upper-left inner (content + padding) corner. For left-to-right OS they are always the widths of left/top borders. For right-to-left OS the vertical scrollbar is on the left so clientLeft includes

its width too.
clientWidth/clientHeight – the width/height of the content including paddings, but without the scrollbar.
scrollWidth/scrollHeight – the width/height of the content, just like clientWidth/clientHeight, but also include scrolled-out, invisible part of the element.
scrollLeft/scrollTop – width/height of the scrolled out upper part of the element, starting from its upper-left corner.

All properties are read-only except scrollLeft/scrollTop that make the browser scroll the element if changed.
TasksWhat's the scroll from the bottom?importance: 5The elem.scrollTop property is the size of the scrolled out part from the top. How to get the size of the bottom scroll (let’s call it scrollBottom)?
Write the code that works for an arbitrary elem.
P.S. Please check your code: if there’s no scroll or the element is fully scrolled down, then it should return 0.
solutionThe solution is:

```
let scrollBottom = elem.scrollHeight - elem.scrollTop - elem.clientHeight;
```

In other words: (full height) minus (scrolled out top part) minus (visible part) – that’s exactly the scrolled out bottom part.
What is the scrollbar width?importance: 3Write the code that returns the width of a standard scrollbar.
For Windows it usually varies between 12px and 20px. If the browser doesn’t reserve any space for it (the scrollbar is half-translucent over the text, also happens), then it may be 0px.
P.S. The code should work for any HTML document, do not depend on its content.
solutionTo get the scrollbar width, we can create an element with the scroll, but without borders and paddings.
Then the difference between its full width offsetWidth and the inner content area width clientWidth will be exactly the scrollbar:

```
        // create a div with the scroll
let div = document.createElement('div');

div.style.overflowY = 'scroll';
div.style.width = '50px';
div.style.height = '50px';

// must put it in the document, otherwise sizes will be 0
document.body.append(div);
let scrollWidth = div.offsetWidth - div.clientWidth;

div.remove();

alert(scrollWidth);
```

Place the ball in the field centerimportance: 5Here’s how the source document looks:

What are coordinates of the field center?
Calculate them and use to place the ball into the center of the green field:

The element should be moved by JavaScript, not CSS.
The code should work with any ball size (10, 20, 30 pixels) and any field size, not be bound to the given values.

P.S. Sure, centering could be done with CSS, but here we want exactly JavaScript. Further we’ll meet other topics and more complex situations when JavaScript must be used. Here we do a “warm-up”.
Open a sandbox for the task.solutionThe ball has position:absolute. It means that its left/top coordinates are measured from the nearest positioned element, that is #field

(because it has position:relative).
The coordinates start from the inner left-upper corner of the field:

The inner field width/height is clientWidth/clientHeight. So the field center has coordinates (clientWidth/2, clientHeight/2).
…But if we set ball.style.left/top to such values, then not the ball as a whole, but the left-upper edge of the ball would be in the center:

```
ball.style.left = Math.round(field.clientWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2) + 'px';
```

Here's how it looks:

To align the ball center with the center of the field, we should move the ball to the half of its width to the left and to the half of its height to the top:

```
ball.style.left = Math.round(field.clientWidth / 2 - ball.offsetWidth / 2) + 'px';
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight / 2) + 'px';
```

Now the ball is finally centered.

Attention: the pitfall!
The code won't work reliably while <img> has no width/height:

```
<img src="ball.png" id="ball">
```

When the browser does not know the width/height of an image (from tag attributes or CSS), then it assumes them to equal 0 until the image finishes loading.
So the value of ball.offsetWidth will be 0 until the image loads. That leads to wrong coordinates in the code above.
After the first load, the browser usually caches the image, and on reloads it will have the size immediately. But on the first load the value of ball.offsetWidth is 0.
We should fix that by adding width/height to <img>:

```
<img src="ball.png" width="40" height="40" id="ball">
```

…Or provide the size in CSS:

```
#ball {
  width: 40px;
  height: 40px;
}
```

Open the solution in a sandbox.The difference: CSS width versus clientWidthimportance: 5What's the difference between getComputedStyle(elem).width and elem.clientWidth?
Give at least 3 differences. The more the better.
solutionDifferences:

clientWidth is numeric, while getComputedStyle(elem).width returns a string with px at the end.
getComputedStyle may return non-numeric width like "auto" for an inline element.
clientWidth is the inner content area of the element plus paddings, while CSS width (with standard box-sizing) is the inner content area without paddings.
If there's a scrollbar and the browser reserves the space for it, some browser substract that space from CSS width (cause it's not available for content any more), and some do not. The clientWidth property is always the same: scrollbar size is substracted if reserved.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before

commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/size-and-scroll","identifier":"\/size-and-scroll"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}June 26, 2022Window sizes and scrollingHow 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 this type of information, we can use the root document element document.documentElement, that corresponds to the <html> tag. But there are additional methods and peculiarities to consider.
Width/height of the windowTo get window width and height, we can use the clientWidth/clientHeight of document.documentElement:

For instance, this button shows the height of your window:

```
alert(document.documentElement.clientHeight)
```

Not window.innerWidth/innerHeight

Browsers also support properties like 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.
window.innerWidth/innerHeight includes 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
```

In most cases, we need the available window width in order to draw or position something within scrollbars (if there are any), so we should use documentElement.clientHeight/clientWidth.

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 documentTheoretically, as the root document element is document.documentElement, and it encloses all the content, we could measure the document's 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! 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 scrollDOM elements have their current scroll state in their scrollLeft/scrollTop properties.

For document scroll, document.documentElement.scrollLeft/scrollTop 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.

Also available as window properties scrollX and scrollY

For historical reasons, both properties exist, but they are the same:

window.pageXOffset is an alias of window.scrollX.

window.pageYOffset is an alias of window.scrollY.

Scrolling: scrollTo, scrollBy, scrollIntoView

Important:

To scroll the page with JavaScript, its DOM must be fully built.

For instance, if we try to scroll the page with a 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/scrollLeft (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 button below demonstrates this:
window.scrollBy(0,10)

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).
window.scrollTo(0,0)

These methods work for all browsers the same way.
scrollIntoViewFor 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 will be 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 will be aligned with the window bottom.

The button below scrolls the page to position itself at the window top:
this.scrollIntoView()
And this button scrolls the page to position itself at the bottom:
this.scrollIntoView(false)
Forbid the scrollingSometimes we need to make the document “unscrollable”. For instance, when we need to cover the page 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” at its current scroll position.
Try it:
document.body.style.overflow = ‘hidden’
document.body.style.overflow = ‘’
The first button freezes the scroll, while the second one releases it.
We can use the same technique to freeze the scroll for other elements, not just for document.body.
The drawback of the method is that the scrollbar disappears. If it occupied some space, then that space is now free and the content “jumps” to fill it.
That looks a bit odd, but can be worked around if we compare clientWidth before and

after the freeze. If it increased (the scrollbar disappeared), then add padding to document.body in place of the scrollbar to keep the content width the same.
SummaryGeometry:

Width/height of the visible part of the document (content area width/height):
document.documentElement.clientWidth/clientHeight

Width/height of the whole document, with the scrolled out part:

```
let scrollHeight = Math.max(
  document.body.scrollHeight, document.documentElement.scrollHeight,
  document.body.offsetHeight, document.documentElement.offsetHeight,
  document.body.clientHeight, document.documentElement.clientHeight
);
```

Scrolling:

Read the current scroll: window.pageYOffset/pageXOffset.

Change the current scroll:

window.scrollTo(pageX,pageY) – absolute coordinates,
window.scrollBy(x,y) – scroll relative the current place,
elem.scrollIntoView(top) – scroll to make elem visible (align with the top/bottom of the window).

Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/size-and-scroll-

window","identifier":"\/size-and-scroll-window"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document","item":"https://javascript.info/document"}]}January 25, 2023CoordinatesTo move elements around we should be familiar with coordinates.
Most JavaScript methods deal with one of two coordinate systems:

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.

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: getBoundingClientRectThe 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.

For instance click this button to see its window coordinates:

```
function showRect(elem) {
  let r = elem.getBoundingClientRect();
  alert(`x:${r.x}
y:${r.y}
width:${r.width}
height:${r.height}
top:${r.top}
bottom:${r.bottom}
left:${r.left}
right:${r.right}
`);
}
```

If you scroll the page and repeat, you'll notice that as window-relative button position changes, its window coordinates (y/top/bottom if you scroll vertically) change as well.
Here's the picture of elem.getBoundingClientRect() output:

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 browser 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.

Why derived properties are needed? Why does top/left exist if there's x/y?

Mathematically, a rectangle is uniquely defined with its starting point (x,y) and the direction vector (width,height). So the additional derived properties are for convenience.
Technically it's possible for width/height to be negative, that allows for "directed" rectangle, e.g. to represent mouse selection with properly marked start and end.
Negative width/height values mean that the rectangle starts at its bottom-right corner and then "grows" left-upwards.
Here's a rectangle with negative width and height (e.g. width=-200, height=-100):

As you can see, left/top do not equal x/y in such case.
In practice though, elem.getBoundingClientRect() always returns positive width/height, here we mention negative width/height only for you to understand why these seemingly duplicate properties are not actually duplicates.

Internet Explorer: no support for x/y

Internet Explorer doesn't support x/y properties for historical reasons.
So we can either make a polyfill (add getters in DomRect.prototype) or just use top/left, as they are always the same as x/y for positive width/height, in particular in the result of elem.getBoundingClientRect().

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.

For out-of-window coordinates the elementFromPoint returns null

The method document.elementFromPoint(x,y) only works if (x,y) are inside the visible area.

If any of the coordinates is negative or exceeds the window width/height, then it returns null.

Here’s a typical error that may occur if we don’t check for it:

```
let elem = document.elementFromPoint(x, y);
// if the coordinates happen to be out of the window, then elem = null
elem.style.background = ''; // Error!
```

Using for “fixed” positioningMost of 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);
```

Click the button to run it:
Button with id=“coords-show-mark”, the message will appear under it
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 coordinatesDocument-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,
    right: box.right + window.pageXOffset,
    bottom: box.bottom + 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;
}
```

SummaryAny point on the page has coordinates:

Relative to the window – elem.getBoundingClientRect().
Relative to the document – elem.getBoundingClientRect() plus the current page scroll.

Window coordinates are great to use with position:fixed, and document coordinates do well with position:absolute.
Both coordinate systems have their pros and cons; there are times we need one or the other one, just like CSS position absolute and fixed.
TasksFind window coordinates of the fieldimportance: 5In the iframe below you can see a document with the green “field”.
Use JavaScript to find window coordinates of corners pointed by with arrows.
There’s a small feature implemented in the document for convenience. A click at any place shows coordinates there.

Your code should use DOM to get window coordinates of:

Upper-left, outer corner (that’s simple).
Bottom-right, outer corner (simple too).
Upper-left, inner corner (a bit harder).
Bottom-right, inner corner (there are several ways, choose one).

The coordinates that you calculate should be the same as those returned by the mouse click.
P.S. The code should also work if the element has another size or border, not bound to any fixed values.
Open a sandbox for the task.solutionOuter cornersOuter cornersOuter corners are basically what we get from elem.getBoundingClientRect().
Coordinates of the upper-left corner answer1 and the bottom-right corner answer2:

```
let coords = elem.getBoundingClientRect();

let answer1 = [coords.left, coords.top];
let answer2 = [coords.right, coords.bottom];
```

Left-upper inner cornerLeft-upper inner cornerThat differs from the outer corner by the border width. A reliable way to get the distance is clientLeft/clientTop:

```
let answer3 = [coords.left + field.clientLeft, coords.top + field.clientTop];
```

Right-bottom inner cornerRight-bottom inner cornerIn our case we need to substract the border size from the outer coordinates.
We could use CSS way:

```
let answer4 = [
  coords.right - parseInt(getComputedStyle(field).borderRightWidth),
  coords.bottom - parseInt(getComputedStyle(field).borderBottomWidth)
];
```

An alternative way would be to add clientWidth/clientHeight to coordinates of the left-upper corner. That’s probably even better:

```
        let answer4 = [
  coords.left + elem.clientLeft + elem.clientWidth,
  coords.top + elem.clientTop + elem.clientHeight
];
```

Open the solution in a sandbox.Show a note near the elementimportance: 5Create a function positionAt(anchor, position, elem) that positions elem, depending on position near anchor element.
The position must be a string with any one of 3 values:

"top" – position elem right above anchor
"right" – position elem immediately at the right of anchor
"bottom" – position elem right below anchor

It’s used inside function showNote(anchor, position, html), provided in the task source code, that creates a “note” element with given html and shows it at the given position near the anchor.
Here’s the demo of notes:

Open a sandbox for the task.solutionIn this task we only need to accurately calculate the coordinates. See the code for details.
Please note: the elements must be in the document to read offsetHeight and other properties.
A hidden (display:none) or out of the document element has no size.
Open the solution in a sandbox.Show a note near the element (absolute)importance: 5Modify the solution of the previous task so that the note uses position:absolute instead of position:fixed.
That will prevent its “runaway” from the element when the page scrolls.
Take the solution of that task as a starting point. To test the scroll, add the style <body style="height: 2000px">.
solutionThe solution is actually pretty simple:

Use position:absolute in CSS instead of position:fixed for .note.
Use the function getCoords() from the chapter Coordinates to get document-relative coordinates.

Open the solution in a sandbox.Position the note inside (absolute)importance: 5Extend

the previous task Show a note near the element (absolute): teach the function positionAt(anchor, position, elem) to insert elem inside the anchor.
New values for position:

top-out, right-out, bottom-out – work the same as before, they insert the elem over/right/under anchor.
top-in, right-in, bottom-in – insert elem inside the anchor: stick it to the upper/right/bottom edge.

For instance:

```
// shows the note above blockquote
positionAt(blockquote, "top-out", note);

// shows the note inside blockquote, at the top
positionAt(blockquote, "top-in", note);
```

The result:

As the source code, take the solution of the task Show a note near the element (absolute).
solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/coordinates","identifier":"\/coordinates"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, Interfaces{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"}]}Introduction to EventsAn introduction to browser events, event properties and handling patterns.

Introduction to browser eventsBubbling and capturingEvent delegationBrowser default actionsDispatching custom eventsCtrl + !•Previous lessonCtrl + !’Next lessonShareTutorial map
TutorialBrowser: Document, Events, InterfacesIntroduction to Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Introduction to Events","item":"https://javascript.info/events"}]}September 21, 2022Introduction to browser eventsAn 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.

Keyboard events:

keydown and keyup – when a keyboard key is pressed and released.

Form element events:

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

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 upcoming chapters.
Event handlersTo 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-attributeA 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 lowercased: onclick.
DOM propertyWe can assign a handler using a DOM property on<event>.

For instance, elem.onclick:

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

If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property. So this way is actually the same as the previous one.
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>
```

In the first example, the HTML attribute is used to initialize the button.onclick, while in the second example – the script, that's all the difference.
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>
```

To remove a handler – assign elem.onclick = null.
Accessing the element: thisThe 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 mistakesIf you’re starting to work with events – please note some subtleties.
We can set an existing function as a handler:

```
function sayThanks() {
  alert('Thanks!');
}

elem.onclick = sayThanks;
```

But be careful: the function should be assigned as sayThanks, not sayThanks().

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

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

If we add parentheses, then sayThanks() becomes 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()">
```

The difference is easy to explain. When the browser reads the attribute, it creates a handler function with body from the attribute content.
So the markup generates this property:

```
        button.onclick = function() {
  sayThanks(); // <-- the attribute content goes here
};
```

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.
addEventListenerThe fundamental problem of the aforementioned ways to assign handlers is that we can’t assign multiple handlers to one event.
Let’s say, one part of our code wants to highlight a button on click, and another one wants to show a message on the same click.
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
```

Developers of web standards understood that long ago and suggested an alternative way of managing handlers using the special methods addEventListener and removeEventListener which aren’t bound by such constraint.
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 call preventDefault(), we’ll explain 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.
This 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, as it's a different function object. 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 it 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>
```

As we can see in the example above, we can set handlers both using a DOM-property and addEventListener. But generally we use only one of these ways.

For some events, handlers only work with addEventListener
There exist events that can't be assigned via a DOM-property. Only with addEventListener.
For instance, the DOMContentLoaded event, that triggers when the document is loaded and the DOM has been built.

```
// will never run
document.onDOMContentLoaded = function() {
  alert("DOM built");
};
```

```
// this way it works
document.addEventListener("DOMContentLoaded", function() {
  alert("DOM built");
});
```

So addEventListener is more universal. Although, such events are an exception rather than the rule.

Event objectTo properly handle an event we'd want to know more about what's happened. Not just a "click" or a "keydown", 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 pointer 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>
```

Some properties of event object:

event.type
Event type, here it's "click".
event.currentTarget
Element that handled the event. That's exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.
event.clientX / event.clientY
Window-relative coordinates of the cursor, for pointer events.

There are more properties. Many of them depend on the event type: keyboard events have one set of properties, pointer events – another one, we'll study them later when as we move on to the details of different events.

The event object is also available in HTML handlers
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">
```

That's possible because when the browser reads the attribute, it creates a handler like this: function(event) { alert(event.type) }. That is: its first argument is called "event", and the body is taken from the attribute.

Object handlers: handleEventWe 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>
  let obj = {
    handleEvent(event) {
      alert(event.type + " at " + event.currentTarget);
    }
  };

  elem.addEventListener('click', obj);
</script>
```

As we can see, when addEventListener receives an object as the handler, it calls obj.handleEvent(event) in case of an event.
We could also use objects of a custom class, like this:

```
        <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.
The method handleEvent does not have to do all the job by itself. It can call other event-specific methods instead, like this:

```
        <button id="elem">Click me</button>

<script>
  class Menu {
    handleEvent(event) {
      // mousedown -> onMousedown
      let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);
      this[method](event);
    }

    onMousedown() {
      elem.innerHTML = "Mouse button pressed";
    }

    onMouseup() {
      elem.innerHTML += "...and released.";
    }
  }

  let menu = new Menu();
  elem.addEventListener('mousedown', menu);
  elem.addEventListener('mouseup', menu);
</script>
```

Now event handlers are clearly separated, that may be easier to support.
SummaryThere are 3 ways to assign event handlers:

HTML attribute: onclick="...".
DOM property: elem.onclick = function.
Methods: elem.addEventListener(event, handler[, phase]) to add, removeEventListener to remove.

HTML attributes are used sparingly, because JavaScript in the middle of an HTML tag looks a little bit odd and alien. Also can’t write lots of code in there.

DOM properties are ok to use, but we can’t assign more than one handler of the particular event. In many cases that limitation is not pressing.
The last way is the most flexible, but it is also the longest to write. There are few events that only work with it, for instance transitionend and DOMContentLoaded (to be covered). Also addEventListener supports objects as event handlers. In that case the method handleEvent is called in case of the event.
No matter how you assign the handler – it gets an event object as the first argument. That object contains the details about what’s happened.
We’ll learn more about events in general and about different types of events in the next chapters.
TasksHide on clickimportance: 5Add JavaScript to the button to make disappear when we click it.
The demo:

Open a sandbox for the task.solutionOpen the solution in a sandbox.Hide selfimportance: 5Create a button that hides itself on click.
Like this:

solutionCan use this in the handler to reference “the element itself” here:

```
<input type="button" onclick="this.hidden=true" value="Click to hide">
```

Which handlers run?importance: 5There’s a button in the variable. There are no handlers on it.
Which handlers run on click after the following code? Which alerts show up?

```
button.addEventListener("click", () => alert("1"));
```

```
button.removeEventListener("click", () => alert("1"));

button.onclick = () => alert(2);
```

solutionThe answer: 1 and 2.
The first handler triggers, because it’s not removed by removeEventListener. To remove the handler we need to pass exactly the function that was assigned. And in the code a new function is passed, that looks the same, but is still another function.
To remove a function object, we need to store a reference to it, like this:

```
function handler() {
  alert(1);
}

button.addEventListener("click", handler);
button.removeEventListener("click", handler);
```

The handler button.onclick works independently and in addition to addEventListener.
Move the ball across the fieldimportance: 5Move the ball across the field to a click. Like this:

Requirements:

The ball center should come exactly under the pointer on click (if possible without crossing the field edge).
CSS-animation is welcome.
The ball must not cross field boundaries.
When the page is scrolled, nothing should break.

Notes:

The code should also work with different ball and field sizes, not be bound to any fixed values.
Use properties event.clientX/event.clientY for click coordinates.

Open a sandbox for the task.solutionFirst we need to choose a method of positioning the ball.
We can’t use position:fixed for it, because scrolling the page would move the ball from the field.
So we should use position:absolute and, to make the positioning really solid, make field itself positioned.
Then the ball will be positioned relatively to the field:

```
        #field {
  width: 200px;
  height: 150px;
  position: relative;
}

#ball {
  position: absolute;
  left: 0; /* relative to the closest positioned ancestor (field) */
  top: 0;
  transition: 1s all; /* CSS animation for left/top makes the ball fly */
}
```

Next we need to assign the correct ball.style.left/top. They contain field-relative coordinates now.
Here’s the picture:

We have event.clientX/clientY – window-relative coordinates of the click.
To get field-relative left coordinate of the click, we can substract the field left edge and the border width:

```
        let left = event.clientX - fieldCoords.left - field.clientLeft;
```

Normally, ball.style.left means the “left edge of the element” (the ball). So if we assign that left, then the ball edge, not center, would be under the mouse cursor.
We need to move the ball half-width left and half-height up to make it center.
So the final left would be:

```
let left = event.clientX - fieldCoords.left - field.clientLeft - ball.offsetWidth/2;
```

The vertical coordinate is calculated using the same logic.
Please note that the ball width/height must be known at the time we access ball.offsetWidth. Should be specified in HTML or CSS.
Open the solution in a sandbox.Create a sliding menuimportance: 5Create a menu that opens/collapses on click:

P.S. HTML/CSS of the source document is to be modified.
Open a sandbox for the task.solutionHTML/CSSHTML/CSSFirst let’s create HTML/CSS.
A menu is a standalone graphical component on the page, so it’s better to put it into a single DOM element.
A list of menu items can be laid out as a list ul/li.
Here’s the example structure:

```
<div class="menu">
  <span class="title">Sweeties (click me)!</span>
  <ul>
    <li>Cake</li>
    <li>Donut</li>
    <li>Honey</li>
  </ul>
</div>
```

We use for the title, because has an implicit display:block on it, and it will occupy 100% of the horizontal width.
Like this:

```
<div style="border: solid red 1px" onclick="alert(1)">Sweeties (click me)!</div>
```

So if we set onclick on it, then it will catch clicks to the right of the text.
As has an implicit display: inline, it occupies exactly enough place to fit all the text:

```
<span style="border: solid red 1px" onclick="alert(1)">Sweeties (click me)!</span>
```

Toggling the menuToggling the menuToggling the menu should change the arrow and show/hide the menu list.
All these changes are perfectly handled by CSS. In JavaScript we should label the current state of the menu by adding/removing the class .open.
Without it, the menu will be closed:

```
.menu ul {
  margin: 0;
  list-style: none;
  padding-left: 20px;
  display: none;
}

.menu .title::before {
  content: '%¶ ';
  font-size: 80%;
  color: green;
}
```

…And with .open the arrow changes and the list shows up:

```
.menu.open .title::before {
  content: '%¼ ';
}

.menu.open ul {
  display: block;
}
```

Open the solution in a sandbox.Add a closing buttonimportance: 5There’s a list of messages.
Use JavaScript to add a closing button to the right-upper corner of each message.
The result should look like this:

Open a sandbox for the task.solutionTo add the button we can use either position:absolute (and make the pane position:relative) or float:right. The float:right has the benefit that the button never overlaps the text, but position:absolute gives more freedom. So the choice is yours.
Then for each pane the code can be like:

```
pane.insertAdjacentHTML("afterbegin", '<button class="remove-button">[x]</button>');
```

Then the <button> becomes pane.firstChild, so we can add a handler to it like this:

```
pane.firstChild.onclick = () => pane.remove();
```

Open the solution in a sandbox.Carouselimportance: 4Create a “carousel” – a

ribbon of images that can be scrolled by clicking on arrows.

Later we can add more features to it: infinite scrolling, dynamic loading etc.
P.S. For this task HTML/CSS structure is actually 90% of the solution.
Open a sandbox for the task.solutionThe images ribbon can be represented as ul/li list of images <img>.
Normally, such a ribbon is wide, but we put a fixed-size around to “cut” it, so that only a part of the ribbon is visible:

To make the list show horizontally we need to apply correct CSS properties for <li>, like display: inline-block.
For <img> we should also adjust display, because by default it’s inline. There’s extra space reserved under inline elements for “letter tails”, so we can use display:block to remove it.
To do the scrolling, we can shift <ul>. There are many ways to do it, for instance by changing margin-left or (better performance) use transform: translateX():

The outer has a fixed width, so “extra” images are cut.
The whole carousel is a self-contained “graphical component” on the page, so we’d better wrap it into a single and style things inside it.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/introduction-browser-events","identifier":"\/introduction-browser-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesIntroduction to Events<script type="application/ld+json">{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Introduction to Events","item":"https://javascript.info/events"}]}</script>October 14, 2022Bubbling and capturingLet’s start with an example.

This handler is assigned to , but also runs if you click any nested tag like 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 run if the actual click was on ?
BubblingThe 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:

On that <p>.
Then on the outer .
Then on the outer <form>.
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.targetA 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.

Check it out:
Resultscript.jsexample.cssindex.htmlform.onclick = function(event) {
  event.target.style.backgroundColor = 'yellow';

```
  // chrome needs some time to paint yellow
  setTimeout(() => {
    alert("target = " + event.target.tagName + ", this=" + this.tagName);
    event.target.style.backgroundColor = ''
  }, 0);
};form {
  background-color: green;
  position: relative;
  width: 150px;
  height: 150px;
  text-align: center;
  cursor: pointer;
}

div {
  background-color: blue;
  position: absolute;
  top: 25px;
  left: 25px;
  width: 100px;
  height: 100px;
}

p {
  background-color: red;
  position: absolute;
  top: 25px;
  left: 25px;
  width: 50px;
  height: 50px;
  line-height: 50px;
  margin: 0;
}

body {
  line-height: 25px;
  font-size: 16px;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="example.css">
</head>
```

```
<body>
  A click shows both <code>event.target</code> and <code>this</code> to compare:

  <form id="form">FORM
    <div>DIV
      <p>P</p>
    </div>
  </form>

  <script src="script.js"></script>
</body>
```
</html>It’s possible that event.target could equal this – it happens when the click is made directly on the <form> element.

Stopping bubblingA 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:

We create a nested menu. Each submenu handles clicks on its elements and calls stopPropagation so that the outer menu won't trigger.

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.

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. One of them is to use custom events, we'll cover them later. Also we can write our data into the event object in one handler and read it in another one, so we can pass to handlers on parents information about the processing below.

CapturingThere'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:

Capturing phase – the event goes down to the element.

Target phase – the event reached the target element.

Bubbling phase – the event bubbles up from the element.

Here's the picture, taken from the specification, of the capturing (1), target (2) and bubbling (3) phases for a click event on a <td> inside a table:

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.

Until now, we only talked about bubbling, because the capturing phase is rarely used. In fact, the capturing phase was invisible for us, because 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:

HTML !’ BODY !’ FORM !’ DIV -> P (capturing phase, the first listener):
P !’ DIV !’ FORM !’ BODY !’ HTML (bubbling phase, the second listener).

Please note, the P shows up twice, because we’ve set two listeners: capturing and bubbling. The target triggers at the end of the first and at the beginning of the second phase.
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 the 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));
```

The event.stopPropagation() during the capturing also prevents the bubbling
The event.stopPropagation() method and its sibling event.stopImmediatePropagation() can also be called on the capturing phase. Then not only the futher capturing is stopped, but the bubbling as well.
In other words, normally the event goes first down (“capturing”) and then up (“bubbling”). But if event.stopPropagation() is called during the capturing phase, then the event travel stops, no bubbling will occur.

SummaryWhen an event happens – the most nested element where it happens gets labeled as the “target element” (event.target).

Then the event moves down from the document root to event.target, calling handlers assigned with addEventListener(..., true) on the way (true is a shorthand for {capture: true}).
Then handlers are called on the target element itself.
Then the event bubbles up from event.target to the root, calling handlers assigned using on<event>, HTML attributes and addEventListener without the 3rd argument or with the 3rd argument false/{capture:false}.

Each handler can access event object properties:

event.target – the deepest element that originated the event.
event.currentTarget (=this) – the current element that handles the event (the one that has the handler on it)
event.eventPhase – the current phase (capturing=1, target=2, bubbling=3).

Any event handler can stop the event by calling event.stopPropagation(), but that’s not recommended, because we can’t really be sure we won’t need it above, maybe for completely different things.
The capturing phase is used very rarely, usually we handle events on bubbling. And there’s a logical explanation for that.
In real world, when an accident happens, local authorities react first. They know best the area where it happened. Then higher-level authorities if needed.
The same for event handlers. The code that set the handler on a particular element knows maximum details about the element and what it does. A handler on a particular <td> may be suited for that exactly <td>, it knows everything about it, so it should get the chance first. Then its immediate parent also knows about the context, but a little bit less, and so on till the very top element that handles general concepts and runs the last one.
Bubbling and capturing lay the foundation for “event delegation” – an extremely powerful event handling pattern that we study in the next chapter.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,

codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/bubbling-and-capturing","identifier":"\/bubbling-and-capturing"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesIntroduction to Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Introduction to Events","item":"https://javascript.info/events"}]}February 3, 2022Event delegationCapturing and bubbling allow us to implement one of the 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 to see where the event actually happened and handle it.

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

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 assign 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 :

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

</td>

Naturally, if a click happens on that 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:

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.
If event.target is not inside any <td>, then the call returns immediately, as there's nothing to do.
In case of nested tables, event.target may be a <td>, but lying outside of the current table. So we check if that's actually our table's <td>.
And, if it's so, then highlight it.

As the 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 markupThere 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” patternWe can also use event delegation to add “behaviors” to elements declaratively, with special attributes and classes.
The pattern has two parts:

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

Behavior: CounterFor 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.

For document-level handlers – always addEventListener

When we assign an event handler to the document object, we should always use addEventListener, not document.on<event>, because the latter will cause conflicts: new handlers overwrite old ones.
For real projects it’s normal that there are many handlers on document set by different parts of the code.

Behavior: TogglerOne more example of behavior. A click on an element with the attribute data-toggle-id will show/hide the element with the given id:

```
        <button data-toggle-id="subscribe-mail">
  Show the subscription form
</button>

<form id="subscribe-mail" hidden>
  Your mail: <input type="email">
</form>

<script>
  document.addEventListener('click', function(event) {
    let id = event.target.dataset.toggleId;
    if (!id) return;

    let elem = document.getElementById(id);

    elem.hidden = !elem.hidden;
  });
</script>
```

Let’s note once again what we did. Now, to add toggling functionality to an element – there’s no need to know JavaScript, just use the attribute data-toggle-id.
That may become really convenient – no need to write JavaScript for every such element. Just use the behavior. The document-level handler makes it work for any element of the page.
We can combine multiple behaviors on a single element as well.
The “behavior” pattern can be an alternative to mini-fragments of JavaScript.
SummaryEvent delegation is really cool! It’s one of the most helpful patterns for DOM events.
It’s often used to add the same handling for many similar elements, but not only for that.
The algorithm:

Put a single handler on the container.
In the handler – check the source element event.target.
If the event happened inside an element that interests us, then handle the event.

Benefits:
Simplifies initialization and saves memory: no need to add many handlers.
Less code: when adding or removing elements, no need to add/remove handlers.
DOM modifications: we can mass add/remove elements with innerHTML and the like.
The delegation has its limitations of course:
First, the event must be bubbling. Some events do not bubble. Also, low-level handlers

should not use event.stopPropagation().
Second, the delegation may add CPU load, because the container-level handler reacts on events in any place of the container, no matter whether they interest us or not. But usually the load is negligible, so we don’t take it into account.
TasksHide messages with delegationimportance: 5There’s a list of messages with removal buttons [x]. Make the buttons work.
Like this:

P.S. Should be only one event listener on the container, use event delegation.
Open a sandbox for the task.solutionOpen the solution in a sandbox.Tree menuimportance: 5Create a tree that shows/hides node children on click:

Requirements:

Only one event handler (use delegation)
A click outside the node title (on an empty space) should not do anything.

Open a sandbox for the task.solutionThe solution has two parts.

Wrap every tree node title into . Then we can CSS-style them on :hover and handle clicks exactly on text, because width is exactly the text width (unlike without it).
Set a handler to the tree root node and handle clicks on that titles.

Open the solution in a sandbox.Sortable tableimportance: 4Make the table sortable: clicks on <th> elements should sort it by corresponding column.
Each <th> has the type in the attribute, like this:

```
      <table id="grid">
  <thead>
    <tr>
      <th data-type="number">Age</th>
      <th data-type="string">Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>5</td>
      <td>John</td>
```

```
    </tr>
    <tr>
      <td>10</td>
      <td>Ann</td>
    </tr>
    ...
  </tbody>
</table>
```

In the example above the first column has numbers, and the second one – strings.
The sorting function should handle sort according to the type.
Only "string" and "number" types should be supported.
The working example:

P.S. The table can be big, with any number of rows and columns.
Open a sandbox for the task.solutionOpen the solution in a sandbox.Tooltip behaviorimportance: 5Create JS-code for the tooltip behavior.
When a mouse comes over an element with data-tooltip, the tooltip should appear over it, and when it’s gone then hide.
An example of annotated HTML:

```
<button data-tooltip="the tooltip is longer than the element">Short button</button>
<button data-tooltip="HTML<br>tooltip">One more button</button>
```

Should work like this:

In this task we assume that all elements with data-tooltip have only text inside. No nested tags (yet).
Details:

The distance between the element and the tooltip should be 5px.
The tooltip should be centered relative to the element, if possible.
The tooltip should not cross window edges. Normally it should be above the element, but if the element is at the page top and there’s no space for the tooltip, then below it.

The tooltip content is given in the data-tooltip attribute. It can be arbitrary HTML.

You'll need two events here:

mouseover triggers when a pointer comes over an element.
mouseout triggers when a pointer leaves an element.

Please use event delegation: set up two handlers on document to track all "overs" and "outs" from elements with data-tooltip and manage tooltips from there.
After the behavior is implemented, even people unfamiliar with JavaScript can add annotated elements.
P.S. Only one tooltip may show up at a time.
Open a sandbox for the task.solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/event-delegation","identifier":"\/event-delegation"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesIntroduction to Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Introduction to Events","item":"https://javascript.info/events"}]}January 21, 2022Browser default actionsMany 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 actionsThere 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; the browser doesn't do anything:

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

In the next example we'll use this technique to create a JavaScript-powered menu.

Returning false from a handler is an exception

The value returned by an event handler is usually ignored.

The only exception is return false from a handler assigned using on<event>.
In all other cases, return value is ignored. In particular, there's no sense in returning true.

Example: the menuConsider 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>
```

Here's how it looks with some CSS:

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

, 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.
By the way, using event delegation here makes our menu very flexible. We can add nested lists and style them using CSS to “slide down”.

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

Nested context menus architecture

There are also alternative ways to implement nested context menus. One of them is to have a single global object with a handler for document.oncontextmenu, and also methods that allow us to store other handlers in it.

The object will catch any right-click, look through stored handlers and run the appropriate one.

But then each piece of code that wants a context menu should know about that object and use its help instead of the own contextmenu handler.

SummaryThere are many default browser actions:

mousedown – starts the selection (move the mouse to select).
click on <input type="checkbox"> – checks/unchecks the input.
submit – clicking an <input type="submit"> or hitting Enter inside a form field causes this event to happen, and the browser submits the form after it.
keydown – pressing a key may lead to adding a character into a field, or other actions.
contextmenu – the event happens on a right-click, the action is to show the browser context menu.
…there are more…

All the default actions can be prevented if we want to handle the event exclusively by JavaScript.
To prevent a default action – use either event.preventDefault() or return false. The second method works only for handlers assigned with on<event>.
The passive: true option of addEventListener tells the browser that the action is not going to be prevented. That's useful for some mobile events, like touchstart and touchmove, to tell the browser that it should not wait for all handlers to finish before scrolling.
If the default action was prevented, the value of event.defaultPrevented becomes true, otherwise it's false.

Stay semantic, don't abuse

Technically, by preventing default actions and adding JavaScript we can customize the behavior of any elements. For instance, we can make a link <a> work like a button, and a button <button> behave as a link (redirect to another URL or so).
But we should generally keep the semantic meaning of HTML elements. For instance, <a> should perform navigation, not a button.
Besides being "just a good thing", that makes your HTML better in terms of accessibility.
Also if we consider the example with <a>, then please note: a browser allows us to open such links in a new window (by right-clicking them and other means). And people like that. But if we make a button behave as a link using JavaScript and even look like a link using CSS, then <a>-specific browser features still won't work for it.

TasksWhy "return false" doesn't work?importance: 3Why in the code below return false doesn't work at all?

```
        <script>
  function handler() {
    alert( "..." );
    return false;
  }
</script>

<a href="https://w3.org" onclick="handler()">the browser will go to w3.org</a>
```

The browser follows the URL on click, but we don’t want it.
How to fix?
solutionWhen the browser reads the on* attribute like onclick, it creates the handler from its content.
For onclick="handler()" the function will be:

```
        function(event) {
  handler() // the content of onclick
}
```

Now we can see that the value returned by handler() is not used and does not affect the result.
The fix is simple:

```
        <script>
  function handler() {
    alert("...");
    return false;
  }
</script>

<a href="https://w3.org" onclick="return handler()">w3.org</a>
```

Also we can use event.preventDefault(), like this:

```
        <script>
  function handler(event) {
    alert("...");
    event.preventDefault();
  }
</script>

<a href="https://w3.org" onclick="handler(event)">w3.org</a>
```

Catch links in the elementimportance: 5Make all links inside the element with id="contents" ask the user if they really want to leave. And if they don't then don't follow. Like this:

Details:

HTML inside the element may be loaded or regenerated dynamically at any time, so we can't find all links and put handlers on them. Use event delegation.
The content may have nested tags. Inside links too, like ....

Open a sandbox for the task.solutionThat’s a great use of the event delegation pattern.
In real life instead of asking we can send a “logging” request to the server that saves the information about where the visitor left. Or we can load the content and show it right in the page (if allowable).
All we need is to catch the contents.onclick and use confirm to ask the user. A good idea would be to use link.getAttribute('href') instead of link.href for the URL. See the solution for details.
Open the solution in a sandbox.Image galleryimportance: 5Create an image gallery where the main image changes by the click on a thumbnail.
Like this:

P.S. Use event delegation.
Open a sandbox for the task.solutionThe solution is to assign the handler to the container and track clicks. If a click is on the <a> link, then change src of #largeImg to the href of the thumbnail.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/default-browser-action","identifier":"\/default-browser-action"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesIntroduction to Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Introduction to Events","item":"https://javascript.info/events"}]}October 14, 2022Dispatching custom eventsWe can not only assign handlers, but also generate events from JavaScript.
Custom events can be used to create “graphical components”. For instance, a root element of our own JS-based menu may trigger events telling what happens with the menu: open (menu open), select (an item is selected) and so on. Another code may listen for the events and observe what’s happening with the menu.
We can generate not only completely new events, that we invent for our own purposes, but also built-in ones, such as click, mousedown etc. That may be helpful for automated testing.
Event constructorBuilt-in event classes form a hierarchy, similar to DOM element classes. The root is the built-in Event class.
We can create Event objects like this:

```
let event = new Event(type[, options]);
```

Arguments:

type – event type, a string like "click" or our own like "my-event".

options – the object with two optional properties:

bubbles: true/false – if true, then the event bubbles.
cancelable: true/false – if true, then the “default action” may be prevented. Later we’ll see what it means for custom events.

By default both are false: {bubbles: false, cancelable: false}.

dispatchEventAfter an event object is created, we should “run” it on an element using the call elem.dispatchEvent(event).
Then handlers react on it as if it were a regular browser event. If the event was created with the bubbles flag, then it bubbles.
In the example below the click event is initiated in JavaScript. The handler works same way as if the button was clicked:

```
<button id="elem" onclick="alert('Click!');">Autoclick</button>

<script>
  let event = new Event("click");
  elem.dispatchEvent(event);
</script>
```

event.isTrusted

There is a way to tell a “real” user event from a script-generated one.

The property event.isTrusted is true for events that come from real user actions and false for script-generated events.

Bubbling exampleWe can create a bubbling event with the name "hello" and catch it on document.

All we need is to set bubbles to true:

```
<h1 id="elem">Hello from the script!</h1>

<script>
  // catch on document...
  document.addEventListener("hello", function(event) { // (1)
    alert("Hello from " + event.target.tagName); // Hello from H1
  });

  // ...dispatch on elem!
  let event = new Event("hello", {bubbles: true}); // (2)
  elem.dispatchEvent(event);

  // the handler on document will activate and display the message.

</script>
```

Notes:

We should use addEventListener for our custom events, because on<event> only exists for built-in events, document.onhello doesn’t work.

Must set bubbles:true, otherwise the event won’t bubble up.

The bubbling mechanics is the same for built-in (click) and custom (hello) events. There are also capturing and bubbling stages.
MouseEvent, KeyboardEvent and othersHere’s a short list of classes for UI Events from the UI Event specification:

UIEvent
FocusEvent
MouseEvent
WheelEvent
KeyboardEvent
…

We should use them instead of new Event if we want to create such events. For instance, new MouseEvent("click").
The right constructor allows to specify standard properties for that type of event.
Like clientX/clientY for a mouse event:

```
        let event = new MouseEvent("click", {
  bubbles: true,
  cancelable: true,
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // 100
```

Please note: the generic Event constructor does not allow that.
Let’s try:

```
        let event = new Event("click", {
  bubbles: true, // only bubbles and cancelable
  cancelable: true, // work in the Event constructor
  clientX: 100,
  clientY: 100
});

alert(event.clientX); // undefined, the unknown property is ignored!
```

Technically, we can work around that by assigning directly event.clientX=100 after creation. So that's a matter of convenience and following the rules. Browser-generated events always have the right type.
The full list of properties for different UI events is in the specification, for instance, MouseEvent.
Custom eventsFor our own, completely new events types like "hello" we should use new CustomEvent. Technically CustomEvent is the same as Event, with one exception.
In the second argument (object) we can add an additional property detail for any custom information that we want to pass with the event.
For instance:

```
        <h1 id="elem">Hello for John!</h1>

<script>
  // additional details come with the event to the handler
```

```
  elem.addEventListener("hello", function(event) {
    alert(event.detail.name);
  });

  elem.dispatchEvent(new CustomEvent("hello", {
    detail: { name: "John" }
  }));
</script>
```

The detail property can have any data. Technically we could live without, because we can assign any properties into a regular new Event object after its creation. But CustomEvent provides the special detail field for it to evade conflicts with other event properties.

Besides, the event class describes “what kind of event” it is, and if the event is custom, then we should use CustomEvent just to be clear about what it is.

event.preventDefault()Many browser events have a “default action”, such as navigating to a link, starting a selection, and so on.

For new, custom events, there are definitely no default browser actions, but a code that dispatches such event may have its own plans what to do after triggering the event.

By calling event.preventDefault(), an event handler may send a signal that those actions should be canceled.

In that case the call to elem.dispatchEvent(event) returns false. And the code that dispatched it knows that it shouldn’t continue.

Let’s see a practical example – a hiding rabbit (could be a closing menu or something else).

Below you can see a #rabbit and hide() function that dispatches "hide" event on it, to let all interested parties know that the rabbit is going to hide.

Any handler can listen for that event with rabbit.addEventListener('hide',...) and, if needed, cancel the action using event.preventDefault(). Then the rabbit won’t disappear:

```
        <pre id="rabbit">
  |\   /|
   \|_|/
```

```
   /. .\
  =\_Y_/=
   {>o<}
</pre>
<button onclick="hide()">Hide()</button>

<script>
  function hide() {
    let event = new CustomEvent("hide", {
      cancelable: true // without that flag preventDefault doesn't work
    });
    if (!rabbit.dispatchEvent(event)) {
      alert('The action was prevented by a handler');
    } else {
      rabbit.hidden = true;
    }
  }

  rabbit.addEventListener('hide', function(event) {
    if (confirm("Call preventDefault?")) {
      event.preventDefault();
    }
  });
</script>
```

Please note: the event must have the flag cancelable: true, otherwise the call event.preventDefault() is ignored.

Events-in-events are synchronousUsually events are processed in a queue. That is: if the browser is processing onclick and a new event occurs, e.g. mouse moved, then its handling is queued up, corresponding mousemove handlers will be called after onclick processing is finished.

The notable exception is when one event is initiated from within another one, e.g. using dispatchEvent. Such events are processed immediately: the new event handlers are called, and then the current event handling is resumed.

For instance, in the code below the menu-open event is triggered during the onclick. It’s processed immediately, without waiting for onclick handler to end:

```
        <button id="menu">Menu (click me)</button>

<script>
  menu.onclick = function() {
    alert(1);

    menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    }));

    alert(2);
  };

  // triggers between 1 and 2
  document.addEventListener('menu-open', () => alert('nested'));
</script>
```

The output order is: 1 !’ nested !’ 2.

Please note that the nested event menu-open is caught on the document. The propagation and handling of the nested event is finished before the processing gets back to the outer code (onclick).

That’s not only about dispatchEvent, there are other cases. If an event handler calls methods that trigger other events – they are processed synchronously too, in a nested fashion.

Let’s say we don’t like it. We’d want onclick to be fully processed first, independently from menu-open or any other nested events.

Then we can either put the dispatchEvent (or another event-triggering call) at the end of onclick or, maybe better, wrap it in the zero-delay setTimeout:

```
        <button id="menu">Menu (click me)</button>

<script>
  menu.onclick = function() {
    alert(1);

    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })));

    alert(2);
  };

  document.addEventListener('menu-open', () => alert('nested'));
</script>
```

Now dispatchEvent runs asynchronously after the current code execution is finished, including menu.onclick, so event handlers are totally separate.
The output order becomes: 1 !’ 2 !’ nested.
SummaryTo generate an event from code, we first need to create an event object.
The generic Event(name, options) constructor accepts an arbitrary event name and the options object with two properties:

bubbles: true if the event should bubble.
cancelable: true if the event.preventDefault() should work.

Other constructors of native events like MouseEvent, KeyboardEvent and so on accept properties specific to that event type. For instance, clientX for mouse events.
For custom events we should use CustomEvent constructor. It has an additional option named detail, we should assign the event-specific data to it. Then all handlers can access it as event.detail.
Despite the technical possibility of generating browser events like click or keydown, we should use them with great care.
We shouldn’t generate browser events as it’s a hacky way to run handlers. That’s bad architecture most of the time.
Native events might be generated:

As a dirty hack to make 3rd-party libraries work the needed way, if they don’t provide other means of interaction.
For automated testing, to “click the button” in the script and see if the interface reacts correctly.

Custom events with our own names are often generated for architectural purposes, to signal what happens inside our menus, sliders, carousels etc.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/dispatch-events","identifier":"\/dispatch-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, Interfaces{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"}]}UI EventsHere we cover most important user interface events and how to work with them.
Mouse eventsMoving the mouse: mouseover/out, mouseenter/leaveDrag'n'Drop with mouse eventsPointer eventsKeyboard: keydown and keyupScrollingCtrl + !•Previous lessonCtrl + !’Next lessonShareTutorial map
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI Events","item":"https://javascript.info/event-details"}]}June 19, 2022Mouse eventsIn 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 typesWe’ve already seen some of these events:

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.
click
Triggers after mousedown and then mouseup over the same element if the left mouse button was used.
dblclick
Triggers after two clicks on the same element within a short timeframe. Rarely used nowadays.
contextmenu
Triggers when the right mouse button is pressed. There are other ways to open a context menu, e.g. using a special keyboard key, it triggers in that case also, so it’s not

exactly the mouse event.

…There are several other events too, we'll cover them later.
Events orderAs you can see from the list above, a user action may trigger multiple events.
For instance, a left-button 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.
Click the button below and you'll see the events. Try double-click too.
On the teststand below, all mouse events are logged, and if there is more than a 1 second delay between them, they are separated by a horizontal rule.
Also, we can see the button property that allows us to detect the mouse button; it's explained below.

Mouse buttonClick-related events always have the button property, which allows to get the exact mouse button.
We usually don't use it for click and contextmenu events, because the former happens only on left-click, and the latter – only on right-click.
On the other hand, mousedown and mouseup handlers may need event.button, because these events trigger on any button, so button allows to distinguish between "right-mousedown" and "left-mousedown".
The possible values of event.button are:

Button state	event.button
Left button (primary)	0
Middle button (auxiliary)	1
Right button (secondary)	2
X1 button (back)	3

X2 button (forward)
4

Most mouse devices only have the left and right buttons, so possible values are 0 or 2. Touch devices also generate similar events when one taps on them.
Also there’s event.buttons property that has all currently pressed buttons as an integer, one bit per button. In practice this property is very rarely used, you can find details at MDN if you ever need it.

The outdated event.which

Old code may use event.which property that’s an old non-standard way of getting a button, with possible values:

event.which == 1 – left button,
event.which == 2 – middle button,
event.which == 3 – right button.

As of now, event.which is deprecated, we shouldn’t use it.

Modifiers: shift, alt, ctrl and metaAll 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).

There are also mobile devices

Keyboard combinations are good as an addition to the workflow. So that if the visitor uses a keyboard – they work.
But if their device doesn’t have it – then there should be a way to live without modifier keys.

Coordinates: clientX/Y, pageX/YAll mouse events provide coordinates in two flavours:

Window-relative: clientX and clientY.
Document-relative: pageX and pageY.

We already covered the difference between them in the chapter Coordinates.
In short, document-relative coordinates pageX/Y are counted from the left-upper corner of the document, and do not change when the page is scrolled, while clientX/Y are counted from the current window left-upper corner. When the page is scrolled, they change.
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, no matter how the page is scrolled.
And if the mouse is in the center, then clientX and clientY are 250, no matter what place in the document it is. They are similar to position:fixed in that aspect.
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">

Preventing selection on mousedownDouble mouse click has a side effect that may be disturbing in some interfaces: it selects text.
For instance, double-clicking on the text below selects it in addition to our handler:

Double-click me

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...

  Double-click me

...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 , 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.

SummaryMouse events have the following properties:

Button: button.

Modifier keys (true if pressed): altKey, ctrlKey, shiftKey and metaKey (Mac).

If you want to handle Ctrl, then don’t forget Mac users, they usually use Cmd, so it’s better to check if (e.metaKey || e.ctrlKey).

Window-relative coordinates: clientX/clientY.

Document-relative coordinates: pageX/pageY.

The default browser action of mousedown is text selection, if it’s not good for the interface, then it should be prevented.
In the next chapter we’ll see more details about events that follow pointer movement

and how to track element changes under it.
TasksSelectable listimportance: 5Create a list where elements are selectable, like in file-managers.

A click on a list element selects only that element (adds the class .selected), deselects all others.
If a click is made with Ctrl (Cmd for Mac), then the selection is toggled on the element, but other elements are not modified.

The demo:

P.S. For this task we can assume that list items are text-only. No nested tags.
P.P.S. Prevent the native browser selection of the text on clicks.
Open a sandbox for the task.solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/mouse-events-basics","identifier":"\/mouse-events-basics"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI Events","item":"https://javascript.info/event-details"}]}April 17, 2022Moving the mouse: mouseover/out, mouseenter/leaveLet's dive into more details about events that happen when the mouse moves between elements.
Events mouseover/mouseout, relatedTargetThe 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 target. When a mouse leaves one element for another, one of them becomes 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).

In the example below each face and its features are separate elements. When you move the mouse, you can see mouse events in the text area.
Each event has the information about both target and relatedTarget:
Resultscript.jsstyle.cssindex.htmlcontainer.onmouseover = container.onmouseout = handler;

```
function handler(event) {

  function str(el) {
    if (!el) return "null"
    return el.classNam e || el.tagName;
  }

  log.value += event.type + ':  ' +
    'target=' + str(event.target) +
    ',  relatedTarget=' + str(event.relatedTarget) + "\n";
  log.scrollTop = log.scrollHeight;

  if (event.type == 'mouseover') {
    event.target.style.background = 'pink'
  }
  if (event.type == 'mouseout') {
    event.target.style.background = ''
  }
}body,
html {
  margin: 0;
  padding: 0;
}

#container {
  border: 1px solid brown;
  padding: 10px;
  width: 330px;
  margin-bottom: 5px;
  box-sizing: border-box;
}

#log {
```

```
  height: 120px;
  width: 350px;
  display: block;
  box-sizing: border-box;
}

[class^="smiley-"] {
  display: inline-block;
  width: 70px;
  height: 70px;
  border-radius: 50%;
  margin-right: 20px;
}

.smiley-green {
  background: #a9db7a;
  border: 5px solid #92c563;
  position: relative;
}

.smiley-green .left-eye {
  width: 18%;
  height: 18%;
  background: #84b458;
  position: relative;
  top: 29%;
  left: 22%;
  border-radius: 50%;
  float: left;
}

.smiley-green .right-eye {
  width: 18%;
  height: 18%;
  border-radius: 50%;
  position: relative;
  background: #84b458;
  top: 29%;
  right: 22%;
  float: right;
}

.smiley-green .smile {
  position: absolute;
  top: 67%;
  left: 16.5%;
```

```
  width: 70%;
  height: 20%;
  overflow: hidden;
}

.smiley-green .smile:after,
.smiley-green .smile:before {
  content: "";
  position: absolute;
  top: -50%;
  left: 0%;
  border-radius: 50%;
  background: #84b458;
  height: 100%;
  width: 97%;
}

.smiley-green .smile:after {
  background: #84b458;
  height: 80%;
  top: -40%;
  left: 0%;
}

.smiley-yellow {
  background: #eed16a;
  border: 5px solid #dbae51;
  position: relative;
}

.smiley-yellow .left-eye {
  width: 18%;
  height: 18%;
  background: #dba652;
  position: relative;
  top: 29%;
  left: 22%;
  border-radius: 50%;
  float: left;
}

.smiley-yellow .right-eye {
  width: 18%;
  height: 18%;
  border-radius: 50%;
  position: relative;
```

```
  background: #dba652;
  top: 29%;
  right: 22%;
  float: right;
}

.smiley-yellow .smile {
  position: absolute;
  top: 67%;
  left: 19%;
  width: 65%;
  height: 14%;
  background: #dba652;
  overflow: hidden;
  border-radius: 8px;
}

.smiley-red {
  background: #ee9295;
  border: 5px solid #e27378;
  position: relative;
}

.smiley-red .left-eye {
  width: 18%;
  height: 18%;
  background: #d96065;
  position: relative;
  top: 29%;
  left: 22%;
  border-radius: 50%;
  float: left;
}

.smiley-red .right-eye {
  width: 18%;
  height: 18%;
  border-radius: 50%;
  position: relative;
  background: #d96065;
  top: 29%;
  right: 22%;
  float: right;
}

.smiley-red .smile {
```

```
  position: absolute;
  top: 57%;
  left: 16.5%;
  width: 70%;
  height: 20%;
  overflow: hidden;
}

.smiley-red .smile:after,
.smiley-red .smile:before {
  content: "";
  position: absolute;
  top: 50%;
  left: 0%;
  border-radius: 50%;
  background: #d96065;
  height: 100%;
  width: 97%;
}

.smiley-red .smile:after {
  background: #d96065;
  height: 80%;
  top: 60%;
  left: 0%;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <div id="container">
    <div class="smiley-green">
      <div class="left-eye"></div>
      <div class="right-eye"></div>
      <div class="smile"></div>
    </div>

    <div class="smiley-yellow">
      <div class="left-eye"></div>
      <div class="right-eye"></div>
      <div class="smile"></div>
```

```
    </div>

    <div class="smiley-red">
      <div class="left-eye"></div>
      <div class="right-eye"></div>
      <div class="smile"></div>
    </div>
  </div>

  <textarea id="log">Events will show up here!
</textarea>

  <script src="script.js"></script>

</body>
</html>
```

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 elementsThe 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 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 is inside the .
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.
Resultscript.jsstyle.cssindex.htmllet parent = document.getElementById('parent');
parent.onmouseover = parent.onmouseout = parent.onmousemove = handler;

```
function handler(event) {
  let type = event.type;
  while (type.length < 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;
}#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;
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <div id="parent">parent
    <div id="child">child</div>
  </div>
  <textarea id="text"></textarea>
  <input onclick="clearText()" value="Clear" type="button">
```

```
  <script src="script.js"></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 childAn 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, 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: is inside the . 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:

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

```
Resultscript.jsstyle.cssindex.htmlfunction 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;
}#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;
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</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 src="script.js"></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 mouseleaveEvents mouseenter/mouseleave are like mouseover/mouseout. They trigger when the mouse pointer enters/leaves the element.
But there are two important differences:

Transitions inside the element, to/from descendants, are not counted.
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
Resultscript.jsstyle.cssindex.htmlfunction 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;
}#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;
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <div id="parent" onmouseenter="mouselog(event)"
onmouseleave="mouselog(event)">parent
    child

  <textarea id="text"></textarea>
  <input type="button" onclick="text.value='" value="Clear">

```
  <script src="script.js"></script>

</body>

</html>
```
Event delegationEvents 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:
Resultscript.jsstyle.cssindex.htmltable.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;
};#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;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</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 src="script.js"></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:

```
        // <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;
};

// any functions to handle entering/leaving an element
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;
}
```

Once again, the important features are:

It uses event delegation to handle entering/leaving of any <td> inside the table. So it relies on mouseover/out instead of mouseenter/leave that don’t bubble and hence allow no delegation.
Extra events, such as moving between descendants of <td> are filtered out, so that onEnter/Leave runs only if the pointer leaves or enters <td> as a whole.

Here’s the full example with all details:
Resultscript.jsstyle.cssindex.html// <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;
```

```
};

// any functions to handle entering/leaving an element
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;
}#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;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</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 src="script.js"></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.

SummaryWe covered events mouseover, mouseout, mousemove, mouseenter and mouseleave.

These things are good to note:

A fast mouse move may skip intermediate elements.

Events mouseover/out and mouseenter/leave have an additional property: relatedTarget. That's the element that we are coming from/to, complementary to target.

Events mouseover/out trigger even when we go from the parent element to a child element. The browser assumes that the mouse can be only over one element at one time – the deepest one.

Events mouseenter/leave are different in that aspect: they only trigger when the mouse comes in and out the element as a whole. Also they do not bubble.

TasksImproved tooltip behaviorimportance: 5Write JavaScript that shows a tooltip over an element with the attribute data-tooltip. The value of this attribute should become the tooltip text.

That's like the task Tooltip behavior, but here the annotated elements can be nested. The most deeply nested tooltip is shown.

Only one tooltip may show up at the same time.

For instance:

```
        <div data-tooltip="Here – is the house interior" id="house">
  <div data-tooltip="Here – is the roof" id="roof"></div>
  ...
  <a href="https://en.wikipedia.org/wiki/The_Three_Little_Pigs" data-tooltip="Read
on…">Hover over me</a>
</div>
```

The result in iframe:

Open a sandbox for the task.solutionOpen the solution in a sandbox."Smart" tooltipimportance: 5Write a function that shows a tooltip over an element only if the visitor moves the mouse to it, but not through it.

In other words, if the visitor moves the mouse to the element and stops there – show the tooltip. And if they just moved the mouse through, then no need, who wants extra blinking?

Technically, we can measure the mouse speed over the element, and if it's slow then we assume that it comes "over the element" and show the tooltip, if it's fast – then we ignore it.

Make a universal object new HoverIntent(options) for it.

Its options:

elem – element to track.

over – a function to call if the mouse came to the element: that is, it moves slowly or stopped over it.

out – a function to call when the mouse leaves the element (if over was called).

An example of using such object for the tooltip:

```
        // a sample tooltip
let tooltip = document.createElement('div');
tooltip.className = "tooltip";
tooltip.innerHTML = "Tooltip";

// the object will track mouse and call over/out
new HoverIntent({
  elem,
  over() {
    tooltip.style.left = elem.getBoundingClientRect().left + 'px';
    tooltip.style.top = elem.getBoundingClientRect().bottom + 5 + 'px';
```

```
    document.body.append(tooltip);
  },
  out() {
    tooltip.remove();
  }
});
```

The demo:

If you move the mouse over the “clock” fast then nothing happens, and if you do it slow or stop on them, then there will be a tooltip.
Please note: the tooltip doesn’t “blink” when the cursor moves between the clock subelements.
Open a sandbox with tests.solutionThe algorithm looks simple:

Put onmouseover/out handlers on the element. Also can use onmouseenter/leave here, but they are less universal, won’t work if we introduce delegation.
When a mouse cursor entered the element, start measuring the speed on mousemove.
If the speed is slow, then run over.
When we’re going out of the element, and over was executed, run out.

But how to measure the speed?
The first idea can be: run a function every 100ms and measure the distance between previous and new coordinates. If it’s small, then the speed is small.
Unfortunately, there’s no way to get “current mouse coordinates” in JavaScript. There’s no function like getCurrentMouseCoordinates().
The only way to get coordinates is to listen for mouse events, like mousemove, and take coordinates from the event object.
So let’s set a handler on mousemove to track coordinates and remember them. And then compare them, once per 100ms.
P.S. Please note: the solution tests use dispatchEvent to see if the tooltip works right.
Open the solution with tests in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/mousemove-mouseover-mouseout-mouseenter-mouseleave","identifier":"\/mousemove-mouseover-mouseout-mouseenter-mouseleave"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://

schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Browser: Document, Events,
Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI
Events","item":"https://javascript.info/event-details"}]}July 27, 2022Drag'n'Drop with mouse eventsDrag’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.
These events allow us to support special kinds of drag’n’drop, such as handling dragging a file from OS file-manager and dropping it into the browser window. Then JavaScript can access the contents of such files.
But native Drag Events also have limitations. For instance, we can’t prevent dragging from a certain area. Also we can’t make the dragging “horizontal” or “vertical” only. And there are many other drag’n’drop tasks that can’t be done using them. Also, mobile device support for such events is very weak.
So here we’ll see how to implement Drag’n’Drop using mouse events.
Drag’n’Drop algorithmThe basic Drag’n’Drop algorithm looks like this:

On mousedown – prepare the element for moving, if needed (maybe create a clone of it, add a class to it or whatever).
Then on mousemove move it by changing left/top with position:absolute.
On mouseup – perform all actions related to finishing the drag’n’drop.

These are the basics. Later we’ll see how to add other features, such as highlighting current underlying elements while we drag over them.
Here’s the implementation of dragging a ball:

```
        ball.onmousedown = function(event) {
  // (1) prepare to moving: make absolute and on top by z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;

  // move it out of any current parents directly into body
  // to make it positioned relative to the body
  document.body.append(ball);

  // 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';
  }
```

```
  // move our absolutely positioned ball under the pointer
  moveAt(event.pageX, event.pageY);

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // (2) move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);

  // (3) drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};
```

If we run the code, we can notice something strange. On the beginning of the drag’n’drop, the ball “forks”: we start dragging its “clone”.
Here’s an example in action:

Try to drag’n’drop with the mouse and you’ll see such behavior.
That’s because the browser has its own drag’n’drop support for images and some other elements. It runs automatically and conflicts with ours.
To disable it:

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

Now everything will be all right.
In action:

Another important aspect – we track mousemove on document, not on ball. From the first sight it may seem that the mouse is always over the ball, and we can put mousemove on it.
But as we remember, mousemove triggers often, but not for every pixel. So after swift move the pointer can jump from the ball somewhere in the middle of document (or even outside of the window).
So we should listen on document to catch it.
Correct positioningIn the examples above the ball is always moved so that its center is under the pointer:

```
ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
```

Not bad, but there's a side effect. To initiate the drag'n'drop, we can mousedown anywhere on the ball. But if "take" it from its edge, then the ball suddenly "jumps" to become centered under the mouse pointer.
It would be better if we keep the initial shift of the element relative to the pointer.
For instance, if we start dragging by the edge of the ball, then the pointer should remain over the edge while dragging.

Let's update our algorithm:

When a visitor presses the button (mousedown) – remember the distance from the pointer to the left-upper corner of the ball in variables shiftX/shiftY. We'll keep that distance while dragging.
To get these shifts we can substract the coordinates:

```
// onmousedown
let shiftX = event.clientX - ball.getBoundingClientRect().left;
let shiftY = event.clientY - ball.getBoundingClientRect().top;
```

Then while dragging we position the ball on the same shift relative to the pointer, like this:

```
        // onmousemove
// ball has position:absolute
ball.style.left = event.pageX - shiftX + 'px';
ball.style.top = event.pageY - shiftY + 'px';
```

The final code with better positioning:

```
        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);

  // moves the ball at (pageX, pageY) coordinates
  // taking initial shifts into account
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
  }

  function onMouseMove(event) {
    moveAt(event.pageX, event.pageY);
  }

  // move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);
```

```
  // drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};

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

In action (inside <iframe>):

The difference is especially noticeable if we drag the ball by its right-bottom corner. In the previous example the ball “jumps” under the pointer. Now it fluently follows the pointer from the current position.

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 abstract, we take a “draggable” element and drop it onto “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 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). If there are multiple overlapping elements on the same coordinates, then the topmost one is returned.

We can use it in any of our mouse event handlers to detect the potential droppable under the pointer, like this:

```
        // in a mouse event handler
ball.hidden = true; // (*) hide the element that we drag

let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
// elemBelow is the element below the ball, may be droppable

ball.hidden = false;
```

Please note: we need to hide the ball before the call (*). Otherwise we'll usually have a ball on these coordinates, as it's the top element under the pointer: elemBelow=ball. So we hide it and immediately show again.

We can use that code to check what element we're "flying over" at any time. And handle the drop when it happens.

An extended code of onMouseMove to find "droppable" elements:

```
// potential droppable that we're flying over right now
let currentDroppable = null;

function onMouseMove(event) {
  moveAt(event.pageX, event.pageY);

  ball.hidden = true;
  let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
  ball.hidden = false;

  // mousemove events may trigger out of the window (when the ball is dragged off-screen)
  // if clientX/clientY are out of the window, then elementFromPoint returns null
  if (!elemBelow) return;

  // potential droppables are labeled with the class "droppable" (can be other logic)
  let droppableBelow = elemBelow.closest('.droppable');

  if (currentDroppable != droppableBelow) {
    // we're flying in or out...
    // note: both values can be null
    //   currentDroppable=null if we were not over a droppable before this event (e.g over an empty space)
    //   droppableBelow=null if we're not over a droppable now, during this event

    if (currentDroppable) {
      // the logic to process "flying out" of the droppable (remove highlight)
      leaveDroppable(currentDroppable);
    }
    currentDroppable = droppableBelow;
    if (currentDroppable) {
      // the logic to process "flying in" of the droppable
      enterDroppable(currentDroppable);
    }
  }
}
```

In the example below when the ball is dragged over the soccer goal, the goal is highlighted.

Resultstyle.cssindex.html#gate {
  cursor: pointer;
  margin-bottom: 100px;
  width: 83px;
  height: 46px;
}

#ball {
  cursor: pointer;
  width: 40px;
  height: 40px;
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  Drag the ball.

  <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.
SummaryWe considered a basic Drag’n’Drop algorithm.
The key components:

Events flow: ball.mousedown !’ document.mousemove !’ ball.mouseup (don’t forget to cancel native ondragstart).
At the drag start – remember the initial shift of the pointer relative to the element: shiftX/ shiftY and keep it during the dragging.
Detect droppable elements under the pointer using document.elementFromPoint.

We can lay a lot on this foundation.

On mouseup we can intellectually finalize the drop: change data, move elements around.
We can highlight the elements we’re flying over.
We can limit dragging by a certain area or direction.
We can use event delegation for mousedown/up. A large-area event handler that checks event.target can manage Drag’n’Drop for hundreds of elements.
And so on.

There are frameworks that build architecture over it: DragZone, Droppable, Draggable and other classes. Most of them do the similar stuff to what’s described above, so it should be easy to understand them now. Or roll your own, as you can see that that’s easy enough to do, sometimes easier than adapting a third-party solution.
TasksSliderimportance: 5Create a slider:

Drag the blue thumb with the mouse and move it.
Important details:

When the mouse button is pressed, during the dragging the mouse may go over or below the slider. The slider will still work (convenient for the user).
If the mouse moves very fast to the left or to the right, the thumb should stop exactly at the edge.

Open a sandbox for the task.solutionAs we can see from HTML/CSS, the slider is a with a colored background, that contains a runner – another with position:relative.
To position the runner we use position:relative, to provide the coordinates relative to its

parent, here it’s more convenient here than position:absolute.
Then we implement horizontal-only Drag’n’Drop with limitation by width.
Open the solution in a sandbox.Drag superheroes around the fieldimportance: 5This task can help you to check understanding of several aspects of Drag’n’Drop and DOM.
Make all elements with class draggable – draggable. Like a ball in the chapter.
Requirements:

Use event delegation to track drag start: a single event handler on document for mousedown.
If elements are dragged to top/bottom window edges – the page scrolls up/down to allow further dragging.
There is no horizontal scroll (this makes the task a bit simpler, adding it is easy).
Draggable elements or their parts should never leave the window, even after swift mouse moves.

The demo is too big to fit it here, so here’s the link.
Demo in new windowOpen a sandbox for the task.solutionTo drag the element we can use position:fixed, it makes coordinates easier to manage. At the end we should switch it back to position:absolute to lay the element into the document.
When coordinates are at window top/bottom, we use window.scrollTo to scroll it.
More details in the code, in comments.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/mouse-drag-and-drop","identifier":"\/mouse-drag-and-drop"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI Events","item":"https://javascript.info/event-details"}]}January 26, 2023Pointer eventsPointer events are a modern way to handle input from a variety of pointing devices, such as a mouse, a pen/stylus, a touchscreen, and so on.
The brief historyLet’s make a small overview, so that you understand the general picture and the place of Pointer Events among other event types.

Long ago, in the past, there were only mouse events.
Then touch devices became widespread, phones and tablets in particular. For the existing scripts to work, they generated (and still generate) mouse events. For instance, tapping a touchscreen generates mousedown. So touch devices worked well with web pages.

But touch devices have more capabilities than a mouse. For example, it’s possible to touch multiple points at once (“multi-touch”). Although, mouse events don’t have necessary properties to handle such multi-touches.

So touch events were introduced, such as touchstart, touchend, touchmove, that have touch-specific properties (we don’t cover them in detail here, because pointer events are even better).
Still, it wasn’t enough, as there are many other devices, such as pens, that have their own features. Also, writing code that listens for both touch and mouse events was cumbersome.

To solve these issues, the new standard Pointer Events was introduced. It provides a single set of events for all kinds of pointing devices.

As of now, Pointer Events Level 2 specification is supported in all major browsers, while the newer Pointer Events Level 3 is in the works and is mostly compatible with Pointer Events level 2.
Unless you develop for old browsers, such as Internet Explorer 10, or for Safari 12 or below, there’s no point in using mouse or touch events any more – we can switch to pointer events.
Then your code will work well with both touch and mouse devices.
That said, there are some important peculiarities that one should know in order to use Pointer Events correctly and avoid surprises. We’ll make note of them in this article.
Pointer event typesPointer events are named similarly to mouse events:

Pointer event	Similar mouse event
pointerdown	mousedown
pointerup	mouseup
pointermove	mousemove

pointerover
mouseover

pointerout
mouseout

pointerenter
mouseenter

pointerleave
mouseleave

pointercancel
-

gotpointercapture
-

lostpointercapture
-

As we can see, for every mouse<event>, there’s a pointer<event> that plays a similar role. Also there are 3 additional pointer events that don’t have a corresponding mouse... counterpart, we’ll explain them soon.

Replacing mouse<event> with pointer<event> in our code

We can replace mouse<event> events with pointer<event> in our code and expect things to continue working fine with mouse.
The support for touch devices will also “magically” improve. Although, we may need to add touch-action: none in some places in CSS. We’ll cover it below in the section about pointercancel.

Pointer event propertiesPointer events have the same properties as mouse events, such as clientX/Y, target, etc., plus some others:

pointerId – the unique identifier of the pointer causing the event.
Browser-generated. Allows us to handle multiple pointers, such as a touchscreen with stylus and multi-touch (examples will follow).

pointerType – the pointing device type. Must be a string, one of: “mouse”, “pen” or “touch”.
We can use this property to react differently on various pointer types.

isPrimary – is true for the primary pointer (the first finger in multi-touch).

Some pointer devices measure contact area and pressure, e.g. for a finger on the touchscreen, there are additional properties for that:

width – the width of the area where the pointer (e.g. a finger) touches the device. Where unsupported, e.g. for a mouse, it’s always 1.
height – the height of the area where the pointer touches the device. Where unsupported, it’s always 1.
pressure – the pressure of the pointer tip, in range from 0 to 1. For devices that don’t support pressure must be either 0.5 (pressed) or 0.
tangentialPressure – the normalized tangential pressure.
tiltX, tiltY, twist – pen-specific properties that describe how the pen is positioned relative to the surface.

These properties aren’t supported by most devices, so they are rarely used. You can find the details about them in the specification if needed.
Multi-touchOne of the things that mouse events totally don’t support is multi-touch: a user can touch in several places at once on their phone or tablet, or perform special gestures.
Pointer Events allow handling multi-touch with the help of the pointerId and isPrimary properties.
Here’s what happens when a user touches a touchscreen in one place, then puts another finger somewhere else on it:

At the first finger touch:

pointerdown with isPrimary=true and some pointerId.

For the second finger and more fingers (assuming the first one is still touching):

pointerdown with isPrimary=false and a different pointerId for every finger.

Please note: the pointerId is assigned not to the whole device, but for each touching finger. If we use 5 fingers to simultaneously touch the screen, we have 5 pointerdown events, each with their respective coordinates and a different pointerId.
The events associated with the first finger always have isPrimary=true.
We can track multiple touching fingers using their pointerId. When the user moves and then removes a finger, we get pointermove and pointerup events with the same pointerId as we had in pointerdown.
Here's the demo that logs pointerdown and pointerup events:

Please note: you must be using a touchscreen device, such as a phone or a tablet, to actually see the difference in pointerId/isPrimary. For single-touch devices, such as a mouse, there'll be always same pointerId with isPrimary=true, for all pointer events.
Event: pointercancelThe pointercancel event fires when there's an ongoing pointer interaction, and then something happens that causes it to be aborted, so that no more pointer events are generated.
Such causes are:

The pointer device hardware was physically disabled.
The device orientation changed (tablet rotated).
The browser decided to handle the interaction on its own, considering it a mouse gesture or zoom-and-pan action or something else.

We'll demonstrate pointercancel on a practical example to see how it affects us.
Let's say we're implementing drag'n'drop for a ball, just as in the beginning of the article Drag'n'Drop with mouse events.
Here is the flow of user actions and the corresponding events:

The user presses on an image, to start dragging

pointerdown event fires

Then they start moving the pointer (thus dragging the image)

pointermove fires, maybe several times

And then the surprise happens! The browser has native drag'n'drop support for images, that kicks in and takes over the drag'n'drop process, thus generating pointercancel event.

The browser now handles drag’n’drop of the image on its own. The user may even drag the ball image out of the browser, into their Mail program or a File Manager.
No more pointermove events for us.

So the issue is that the browser “hijacks” the interaction: pointercancel fires in the beginning of the “drag-and-drop” process, and no more pointermove events are generated.
Here’s the drag’n’drop demo with loggin of pointer events (only up/down, move and cancel) in the textarea:

We’d like to implement the drag’n’drop on our own, so let’s tell the browser not to take it over.
Prevent the default browser action to avoid pointercancel.
We need to do two things:

Prevent native drag’n’drop from happening:

We can do this by setting ball.ondragstart = () => false, just as described in the article Drag'n'Drop with mouse events.
That works well for mouse events.

For touch devices, there are other touch-related browser actions (besides drag’n’drop). To avoid problems with them too:

Prevent them by setting #ball { touch-action: none } in CSS.
Then our code will start working on touch devices.

After we do that, the events will work as intended, the browser won’t hijack the process and doesn’t emit pointercancel.
This demo adds these lines:

As you can see, there’s no pointercancel any more.
Now we can add the code to actually move the ball, and our drag’n’drop will work for mouse devices and touch devices.
Pointer capturingPointer capturing is a special feature of pointer events.
The idea is very simple, but may seem quite odd at first, as nothing like that exists for any other event type.
The main method is:

elem.setPointerCapture(pointerId) – binds events with the given pointerId to elem. After the call all pointer events with the same pointerId will have elem as the target (as if happened on elem), no matter where in document they really happened.

In other words, elem.setPointerCapture(pointerId) retargets all subsequent events with the given pointerId to elem.
The binding is removed:

automatically when pointerup or pointercancel events occur,
automatically when elem is removed from the document,
when elem.releasePointerCapture(pointerId) is called.

Now what is it good for? It’s time to see a real-life example.
Pointer capturing can be used to simplify drag’n’drop kind of interactions.
Let’s recall how one can implement a custom slider, described in the Drag'n'Drop with mouse events.
We can make a slider element to represent the strip and the “runner” (thumb) inside it:

```
        <div class="slider">
  <div class="thumb"></div>
</div>
```

With styles, it looks like this:

And here’s the working logic, as it was described, after replacing mouse events with similar pointer events:

The user presses on the slider thumb – pointerdown triggers.
Then they move the pointer – pointermove triggers, and our code moves the thumb element along.

…As the pointer moves, it may leave the slider thumb element, go above or below it. The thumb should move strictly horizontally, remaining aligned with the pointer.

In the mouse event based solution, to track all pointer movements, including when it goes above/below the thumb, we had to assign mousemove event handler on the whole document.
That's not a cleanest solution, though. One of the problems is that when a user moves the pointer around the document, it may trigger event handlers (such as mouseover) on some other elements, invoke totally unrelated UI functionality, and we don't want that. This is the place where setPointerCapture comes into play.

We can call thumb.setPointerCapture(event.pointerId) in pointerdown handler,
Then future pointer events until pointerup/cancel will be retargeted to thumb.
When pointerup happens (dragging complete), the binding is removed automatically, we don't need to care about it.

So, even if the user moves the pointer around the whole document, events handlers will be called on thumb. Nevertheless, coordinate properties of the event objects, such as clientX/clientY will still be correct – the capturing only affects target/currentTarget.
Here's the essential code:

```
        thumb.onpointerdown = function(event) {
  // retarget all pointer events (until pointerup) to thumb
  thumb.setPointerCapture(event.pointerId);

  // start tracking pointer moves
  thumb.onpointermove = function(event) {
    // moving the slider: listen on the thumb, as all pointer events are retargeted to it
    let newLeft = event.clientX - slider.getBoundingClientRect().left;
    thumb.style.left = newLeft + 'px';
  };

  // on pointer up finish tracking pointer moves
  thumb.onpointerup = function(event) {
    thumb.onpointermove = null;
    thumb.onpointerup = null;
```

```
    // ...also process the "drag end" if needed
  };
};

// note: no need to call thumb.releasePointerCapture,
// it happens on pointerup automatically
```

The full demo:

In the demo, there’s also an additional element with onmouseover handler showing the current date.
Please note: while you’re dragging the thumb, you may hover over this element, and its handler does not trigger.
So the dragging is now free of side effects, thanks to setPointerCapture.
At the end, pointer capturing gives us two benefits:

The code becomes cleaner as we don’t need to add/remove handlers on the whole document any more. The binding is released automatically.
If there are other pointer event handlers in the document, they won’t be accidentally triggered by the pointer while the user is dragging the slider.

Pointer capturing eventsThere’s one more thing to mention here, for the sake of completeness.
There are two events associated with pointer capturing:

gotpointercapture fires when an element uses setPointerCapture to enable capturing.
lostpointercapture fires when the capture is released: either explicitly with releasePointerCapture call, or automatically on pointerup/pointercancel.

SummaryPointer events allow handling mouse, touch and pen events simultaneously, with a single piece of code.
Pointer events extend mouse events. We can replace mouse with pointer in event names and expect our code to continue working for mouse, with better support for other device types.
For drag’n’drops and complex touch interactions that the browser may decide to hijack and handle on its own – remember to cancel the default action on events and set touch-action: none in CSS for elements that we engage.
Additional abilities of pointer events are:

Multi-touch support using pointerId and isPrimary.
Device-specific properties, such as pressure, width/height, and others.
Pointer capturing: we can retarget all pointer events to a specific element until pointerup/pointercancel.

As of now, pointer events are supported in all major browsers, so we can safely switch to them, especially if IE10- and Safari 12- are not needed. And even with those browsers, there are polyfills that enable the support of pointer events.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/pointer-events","identifier":"\/pointer-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI Events","item":"https://javascript.info/event-details"}]}April 25, 2022Keyboard: keydown and keyupBefore we get to 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 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).
TeststandTo better understand keyboard events, you can use the teststand below.
Try different key combinations in the text field.
Resultscript.jsstyle.cssindex.htmlkinput.onkeydown = kinput.onkeyup = kinput.onkeypress = handle;

```
let lastTime = Date.now();

function handle(e) {
  if (form.elements[e.type + 'Ignore'].checked) return;

  area.scrollTop = 1e6;

  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();
  }
}#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;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</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" readonly></textarea>
    <input type="button" value="Clear" onclick="area.value = "" />
  </form>
  <script src="script.js"></script>


</body>
</html>Keydown and keyupThe keydown events happens when a key is pressed down, and then keyup – when it’s released.
event.code and event.keyThe 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.

Case matters: "KeyZ", not "keyZ"

Seems obvious, but people still make mistakes.

Please evade mistypes: it’s KeyZ, not keyZ. The check like event.code=="keyZ" won’t work: the first letter of "Key" must be uppercase.

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!')
  }
});
```

On the other hand, there's a problem with event.code. For different keyboard layouts, the same key may have different characters.
For example, here are US layout ("QWERTY") and German layout ("QWERTZ") under it (from Wikipedia):

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. 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-repeatIf 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 actionsDefault 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') || ['+','(',')','-'].includes(key);
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please"
type="tel">
```

The onkeydown handler here uses checkPhoneKey to check for the key pressed. If it’s valid (from 0..9 or one of +-()), then it returns true, otherwise false.
As we know, the false value returned from the event handler, assigned using a DOM property or an attribute, such as above, prevents the default action, so nothing appears in the <input> for keys that don’t pass the test. (The true value returned doesn’t affect anything, only returning false matters)
Please note that special keys, such as Backspace, Left, Right, do not work in the input. That’s a side effect of the strict filter checkPhoneKey. These keys make it return false.
Let’s relax the filter a little bit by allowing arrow keys Left, Right and Delete, Backspace:

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

Now arrows and deletion works well.

Even though we have the key filter, one still can enter anything using a mouse and right-click + Paste. Mobile devices provide other means to enter values. So the filter is not 100% reliable.

The alternative approach would be to track the oninput event – it triggers after any modification. There we can check the new input.value and modify it/highlight the <input> when it’s invalid. Or we can use both event handlers together.

LegacyIn 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.

Mobile KeyboardsWhen using virtual/mobile keyboards, formally known as IME (Input-Method Editor), the W3C standard states that a KeyboardEvent’s e.keyCode should be 229 and e.key should be "Unidentified".

While some of these keyboards might still use the right values for e.key, e.code, e.keyCode… when pressing certain keys such as arrows or backspace, there’s no guarantee, so your keyboard logic might not always work on mobile devices.

SummaryPressing a key always generates a keyboard event, be it symbol keys or special keys like Shift or Ctrl and so on. The only exception is Fn key that sometimes presents on a laptop keyboard. There’s no keyboard event for it, because it’s often

implemented on lower level than OS.
Keyboard events:

keydown – on pressing the key (auto-repeats if the key is pressed for long),
keyup – on releasing the key.

Main keyboard event properties:

code – the “key code” ("KeyA", "ArrowLeft" and so on), specific to the physical location of the key on keyboard.
key – the character ("A", "a" and so on), for non-character keys, such as Esc, usually has the same value as code.

In the past, keyboard events were sometimes used to track user input in form fields. That’s not reliable, because the input can come from various sources. We have input and change events to handle any input (covered later in the chapter Events: change, input, cut, copy, paste). They trigger after any kind of input, including copy-pasting or speech recognition.
We should use keyboard events when we really want keyboard. For example, to react on hotkeys or special keys.
TasksExtended hotkeysimportance: 5Create a function runOnKeys(func, code1, code2, ... code_n) that runs func on simultaneous pressing of keys with codes code1, code2, …, code_n.
For instance, the code below shows alert when "Q" and "W" are pressed together (in any language, with or without CapsLock)

```
runOnKeys(
  () => alert("Hello!"),
  "KeyQ",
  "KeyW"
);
```

Demo in new windowsolutionWe should use two handlers: document.onkeydown and document.onkeyup.
Let’s create a set pressed = new Set() to keep currently pressed keys.
The first handler adds to it, while the second one removes from it. Every time on keydown we check if we have enough keys pressed, and run the function if it is so.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use

the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/keyboard-events","identifier":"\/keyboard-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesUI Events{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"UI Events","item":"https://javascript.info/event-details"}]}August 27, 2020ScrollingThe scroll event allows reacting to 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';
});
```

In action:
Current scroll = scroll the window
The scroll event works both on the window and on scrollable elements.
Prevent scrollingHow 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.
There are many ways to initiate a scroll, so it's more reliable to use CSS, overflow property.
Here are few tasks that you can solve or look through to see applications of onscroll.
TasksEndless pageimportance: 5Create an endless page. When a visitor scrolls it to the end, it auto-appends current date-time to the text (so that a visitor can scroll more).
Like this:

Please note two important features of the scroll:

The scroll is “elastic”. We can scroll a little beyond the document start or end in some browsers/devices (empty space below is shown, and then the document will automatically “bounces back” to normal).
The scroll is imprecise. When we scroll to page end, then we may be in fact like 0-50px away from the real document bottom.

So, “scrolling to the end” should mean that the visitor is no more than 100px away from the document end.
P.S. In real life we may want to show “more messages” or “more goods”.
Open a sandbox for the task.solutionThe core of the solution is a function that adds more dates to the page (or loads more stuff in real-life) while we’re at the page end.
We can call it immediately and add as a window.onscroll handler.
The most important question is: “How do we detect that the page is scrolled to bottom?”
Let’s use window-relative coordinates.
The document is represented (and contained) within <html> tag, that is document.documentElement.
We can get window-relative coordinates of the whole document as document.documentElement.getBoundingClientRect(), the bottom property will be window-relative coordinate of the document bottom.
For instance, if the height of the whole HTML document is 2000px, then:

```
// when we're on the top of the page
// window-relative top = 0
document.documentElement.getBoundingClientRect().top = 0

// window-relative bottom = 2000
// the document is long, so that is probably far beyond the window bottom
document.documentElement.getBoundingClientRect().bottom = 2000
```

If we scroll 500px below, then:

```
// document top is above the window 500px
document.documentElement.getBoundingClientRect().top = -500
```

```
// document bottom is 500px closer
document.documentElement.getBoundingClientRect().bottom = 1500
```

When we scroll till the end, assuming that the window height is 600px:

```
// document top is above the window 1400px
document.documentElement.getBoundingClientRect().top = -1400
// document bottom is below the window 600px
document.documentElement.getBoundingClientRect().bottom = 600
```

Please note that the bottom can’t be 0, because it never reaches the window top. The lowest limit of the bottom coordinate is the window height (we assumed it to be 600), we can’t scroll it any more up.
We can obtain the window height as document.documentElement.clientHeight.
For our task, we need to know when the document bottom is not no more than 100px away from it (that is: 600-700px, if the height is 600).
So here’s the function:

```
function populate() {
  while(true) {
    // document bottom
    let windowRelativeBottom =
document.documentElement.getBoundingClientRect().bottom;

    // if the user hasn't scrolled far enough (>100px to the end)
    if (windowRelativeBottom > document.documentElement.clientHeight + 100) break;

    // let's add more data
    document.body.insertAdjacentHTML("beforeend", `<p>Date: ${new Date()}</p>`);
  }
}
```

Open the solution in a sandbox.Up/down buttonimportance: 5Create a “to the top” button to help with page scrolling.

It should work like this:

While the page is not scrolled down at least for the window height – it's invisible.
When the page is scrolled down more than the window height – there appears an "upwards" arrow in the left-top corner. If the page is scrolled back, it disappears.
When the arrow is clicked, the page scrolls to the top.

Like this (top-left corner, scroll to see):

Open a sandbox for the task.solutionOpen the solution in a sandbox.Load visible imagesimportance: 4Let's say we have a slow-speed client and want to save their mobile traffic.
For that purpose we decide not to show images immediately, but rather replace them with placeholders, like this:

```
<img src="placeholder.svg" width="128" height="128" data-src="real.jpg">
```

So, initially all images are placeholder.svg. When the page scrolls to the position where the user can see the image – we change src to the one in data-src, and so the image loads.
Here's an example in iframe:

Scroll it to see images load "on-demand".
Requirements:

When the page loads, those images that are on-screen should load immediately, prior to any scrolling.
Some images may be regular, without data-src. The code should not touch them.
Once an image is loaded, it should not reload any more when scrolled in/out.

P.S. If you can, make a more advanced solution that would "preload" images that are one page below/after the current position.
P.P.S. Only vertical scroll is to be handled, no horizontal scrolling.
Open a sandbox for the task.solutionThe onscroll handler should check which images

are visible and show them.
We also want to run it when the page loads, to detect immediately visible images and load them.
The code should execute when the document is loaded, so that it has access to its content.
Or put it at the <body> bottom:

```
    // ...the page content is above...

function isVisible(elem) {

  let coords = elem.getBoundingClientRect();

  let windowHeight = document.documentElement.clientHeight;

  // top elem edge is visible?
  let topVisible = coords.top > 0 && coords.top < windowHeight;

  // bottom elem edge is visible?
  let bottomVisible = coords.bottom < windowHeight && coords.bottom > 0;

  return topVisible || bottomVisible;
}
```

The showVisible() function uses the visibility check, implemented by isVisible(), to load visible images:

```
    function showVisible() {
  for (let img of document.querySelectorAll('img')) {
    let realSrc = img.dataset.src;
    if (!realSrc) continue;

    if (isVisible(img)) {
      img.src = realSrc;
      img.dataset.src = '';
    }
  }
}
```

```
showVisible();
window.onscroll = showVisible;
```

P.S. The solution also has a variant of isVisible that “preloads” images that are within 1 page above/below the current document scroll.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/onscroll","identifier":"\/onscroll"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, Interfaces{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"}]}Forms, controlsSpecial properties and events for forms <form> and controls: <input>, <select> and other.
Form properties and methodsFocusing: focus/blurEvents: change, input, cut, copy, pasteForms: event and method submitCtrl + !•Previous lessonCtrl + !’Next lessonShareTutorial map
TutorialBrowser: Document, Events, InterfacesForms, controls{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Forms, controls","item":"https://javascript.info/forms-controls"}]}February 9, 2022Form properties and methodsForms 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 elementsDocument 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. This is typical with radio buttons and checkboxes.
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>
```

Shorter notation: form.name

There’s a shorter notation: we can access the element as form[index/name]. In other words, instead of form.elements.login we can write form.login.

That also works, but there’s a minor issue: if we access an element, and then change its name, then it is still available under the old name (as well as under the new one).

That’s easy to see in an example:

```
        <form id="form">
  <input name="login">
</form>

<script>
  alert(form.elements.login == form.login); // true, the same <input>

  form.login.name = "username"; // change the name of the input

  // form.elements updated the name:
  alert(form.elements.login); // undefined
  alert(form.elements.username); // input

  // form allows both names: the new one and the old one
```

```
  alert(form.username == form.login); // true
</script>
```

That’s usually not a problem, however, because we rarely change names of form elements.

Backreference: element.formFor 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 elementsLet’s talk about form controls.
input and textareaWe can access their value as input.value (string) or input.checked (boolean) for checkboxes and radio buttons.

Like this:

```
input.value = "New value";
textarea.value = "New text";

input.checked = true; // for a checkbox or radio button
```

Use textarea.value, not textarea.innerHTML
Please note that even though <textarea>...</textarea> holds its value as nested HTML, we should never use textarea.innerHTML to access it.
It stores only the HTML that was initially on the page, not the current value.

select and optionA <select> element has 3 important properties:

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

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

Find the corresponding <option> element (e.g. among select.options) and set its option.selected to true.
If we know a new value: set select.value to the new value.
If we know the new option number: set select.selectedIndex to that number.

Here is an example of all three methods:

```
<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';
  // please note: options start from zero, so index 2 means the 3rd option.
</script>
```

Unlike most other controls, <select> allows to select multiple options at once if it has multiple attribute. This attribute is rarely used, though.
For multiple selected values, use the first way of setting values: add/remove the selected property from <option> subelements.
Here's an example of how to get selected values from a multi-select:

```
        <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>
```

The full specification of the <select> element is available in the specification https://html.spec.whatwg.org/multipage/forms.html#the-select-element.
new OptionIn the specification there's a nice short syntax to create an <option> element:

```
option = new Option(text, value, defaultSelected, selected);
```

This syntax is optional. We can use document.createElement('option') and set attributes manually. Still, it may be shorter, so here are the parameters:

text – the text inside the option,
value – the option value,
defaultSelected – if true, then selected HTML-attribute is created,
selected – if true, then the option is selected.

The difference between defaultSelected and selected is that defaultSelected sets the HTML-attribute (that we can get using option.getAttribute('selected'), while selected sets whether the option is selected or not.
In practice, one should usually set both values to true or false. (Or, simply omit them; both default to false.)
For instance, here's a new "unselected" option:

```
let option = new Option("Text", "value");
// creates <option value="value">Text</option>
```

The same option, but selected:

```
let option = new Option("Text", "value", true, true);
```

Option elements have properties:

option.selected
Is the option selected.
option.index
The number of the option among the others in its <select>.
option.text
Text content of the option (seen by the visitor).

References
Specification: https://html.spec.whatwg.org/multipage/forms.html.

SummaryForm navigation:

document.forms
A form is available as document.forms[name/index].
form.elements
Form elements are available as form.elements[name/index], or can use just form[name/index]. The elements property also works for <fieldset>.
element.form
Elements reference their form in the form property.

Value is available as input.value, textarea.value, select.value, etc. (For checkboxes and radio buttons, use input.checked to determine whether a value is selected.)
For <select>, one can also get the value by the index select.selectedIndex or through the options collection select.options.
These are the basics to start working with forms. We'll meet many examples further in the tutorial.
In the next chapter we'll cover focus and blur events that may occur on any element, but are mostly handled on forms.
TasksAdd an option to selectimportance: 5There's a <select>:

```
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>
```

Use JavaScript to:

Show the value and the text of the selected option.
Add an option: <option value="classic">Classic</option>.

Make it selected.

Note, if you've done everything right, your alert should show blues.
solutionThe solution, step by step:

```
        <select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>

<script>
  // 1)
  let selectedOption = genres.options[genres.selectedIndex];
  alert( selectedOption.value );

  // 2)
  let newOption = new Option("Classic", "classic");
  genres.append(newOption);

  // 3)
  newOption.selected = true;
</script>
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/form-elements","identifier":"\/form-elements"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialBrowser: Document, Events, InterfacesForms, controls{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":

[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},
{"@type":"ListItem","position":3,"name":"Forms, controls","item":"https://javascript.info/forms-controls"}]}June 19, 2022Focusing: focus/blurAn element receives the 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 onto an element by default when a page loads and other means of getting the 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.

There are important peculiarities when working with focus events. We'll do the best to cover them further on.

Events focus/blurThe 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 has an email 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/blurMethods 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>
```

It works in all browsers except Firefox (bug).

If we enter something into the input and then try to use Tab or click away from the <input>, then onblur returns the focus back.

Please note that we can't "prevent losing focus" by calling event.preventDefault() in onblur, because onblur works after the element lost the focus.

In practice though, one should think well, before implementing something like this, because we generally should show errors to the user, but should not prevent their progress in filling our form. They may want to fill other fields first.

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: tabindexBy 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.
On the other hand, elements that exist to format something, such as , , <table> – are unfocusable 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 Tab while 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 without 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 "e 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 in 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/focusoutEvents 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>
```

Second, there are focusin and focusout events – exactly the same as focus/blur, but they bubble.
Note that they must be assigned using elem.addEventListener, not on<event>.
So here’s another working variant:

```
        <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>
  form.addEventListener("focusin", () => form.classList.add('focused'));
  form.addEventListener("focusout", () => form.classList.remove('focused'));
</script>
```

SummaryEvents focus and blur trigger on an element focusing/losing focus.
Their specials are:

They do not bubble. Can use capturing state instead or focusin/focusout.
Most elements do not support focus by default. Use tabindex to make anything focusable.

The current focused element is available as document.activeElement.

TasksEditable divimportance: 5Create a that turns into <textarea> when clicked.
The textarea allows to edit the HTML in the .
When the user presses Enter or it loses focus, the <textarea> turns back into , and its content becomes HTML in .
Demo in new windowOpen a sandbox for the task.solutionOpen the solution in a sandbox.Edit TD on clickimportance: 5Make table cells editable on click.

On click – the cell should become “editable” (textarea appears inside), we can change HTML. There should be no resize, all geometry should remain the same.
Buttons OK and CANCEL appear below the cell to finish/cancel the editing.
Only one cell may be editable at a moment. While a <td> is in “edit mode”, clicks on other cells are ignored.
The table may have many cells. Use event delegation.

The demo:

Open a sandbox for the task.solution
On click – replace innerHTML of the cell by <textarea> with same sizes and no border.
Can use JavaScript or CSS to set the right size.
Set textarea.value to td.innerHTML.
Focus on the textarea.
Show buttons OK/CANCEL under the cell, handle clicks on them.

Open the solution in a sandbox.Keyboard-driven mouseimportance: 4Focus on the mouse. Then use arrow keys to move it:
Demo in new windowP.S. Don’t put event handlers anywhere except the #mouse element.
P.P.S. Don’t modify HTML/CSS, the approach should be generic and work with any element.
Open a sandbox for the task.solutionWe can use mouse.onclick to handle the click and make the mouse “moveable” with position:fixed, then mouse.onkeydown to handle arrow keys.
The only pitfall is that keydown only triggers on elements with focus. So we need to add tabindex to the element. As we’re forbidden to change HTML, we can use mouse.tabIndex property for that.
P.S. We also can replace mouse.onclick with mouse.onfocus.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/focus-blur","identifier":"\/focus-blur"}); };var disqus_shortname = "javascriptinfo";var

disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesForms, controls{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Forms, controls","item":"https://javascript.info/forms-controls"}]}June 10, 2022Events: change, input, cut, copy, pasteLet’s cover various events that accompany data updates.

Event: changeThe 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: inputThe 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 does 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 !æ !è 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, pasteThese events occur on cutting/copying/pasting a value.
They belong to ClipboardEvent class and provide access to the data that is cut/copied/pasted.
We also can use event.preventDefault() to abort the action, then nothing gets copied/pasted.
For instance, the code below prevents all cut/copy/paste events and shows the text we’re trying to cut/copy/paste:

```
<input type="text" id="input">
<script>
  input.onpaste = function(event) {
    alert("paste: " + event.clipboardData.getData('text/plain'));
    event.preventDefault();
  };

  input.oncut = input.oncopy = function(event) {
    alert(event.type + '-' + document.getSelection());
    event.preventDefault();
  };
</script>
```

Please note: inside cut and copy event handlers a call to event.clipboardData.getData(...) returns an empty string. That’s because technically the data isn’t in the clipboard yet. If we use event.preventDefault() it won’t be copied at all. So the example above uses document.getSelection() to get the selected text. You can find more details about document selection in the article Selection and Range.
It’s possible to copy/paste not just text, but everything. For instance, we can copy a file in the OS file manager, and paste it.

That’s because clipboardData implements DataTransfer interface, commonly used for drag’n’drop and copy/pasting. It’s a bit beyond our scope now, but you can find its methods in the DataTransfer specification.
Also, there’s an additional asynchronous API of accessing the clipboard: navigator.clipboard. More about it in the specification Clipboard API and events, not supported by Firefox.
Safety restrictionsThe clipboard is a “global” OS-level thing. A user may switch between various applications, copy/paste different things, and a browser page shouldn’t see all that.
So most browsers allow seamless read/write access to the clipboard only in the scope of certain user actions, such as copying/pasting etc.
It’s forbidden to generate “custom” clipboard events with dispatchEvent in all browsers except Firefox. And even if we manage to dispatch such event, the specification clearly states that such “syntetic” events must not provide access to the clipboard.
Even if someone decides to save event.clipboardData in an event handler, and then access it later – it won’t work.
To reiterate, event.clipboardData works solely in the context of user-initiated event handlers.
On the other hand, navigator.clipboard is the more recent API, meant for use in any context. It asks for user permission, if needed.
SummaryData change events:

Event
Description
Specials

change
A value was changed.
For text inputs triggers on focus loss.

input
For text inputs on every change.
Triggers immediately unlike change.

cut/copy/paste
Cut/copy/paste actions.
The action can be prevented. The event.clipboardData property gives access to the clipboard. All browsers except Firefox also support navigator.clipboard.

TasksDeposit calculatorimportance: 5Create an interface that allows to enter a sum of bank deposit and percentage, then calculates how much it will be after given periods of time.
Here’s the demo:

Any input change should be processed immediately.
The formula is:

```
// initial: the initial money sum
// interest: e.g. 0.05 means 5% per year
// years: how many years to wait
let result = Math.round(initial * (1 + interest) ** years);
```

Open a sandbox for the task.solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + ’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/events-change-input","identifier":"\/events-change-input"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesForms, controls{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Forms, controls","item":"https://javascript.info/forms-controls"}]}July 3, 2019Forms: event and method submitThe 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 to initiate form sending from JavaScript. We can use it to dynamically create and send our own forms to server.
Let’s see more details of them.
Event: submitThere are two main ways to submit a form:

The first – to click <input type="submit"> or <input type="image">.

The second – press Enter on an input field.

Both actions lead to 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:

Go into the text field and press Enter.
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: submitTo submit a form to the server manually, we can call form.submit().
Then the submit event is not generated. It is assumed that if the programmer calls

form.submit(), then the script already did all related processing.
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();
```

TasksModal formimportance: 5Create a function showPrompt(html, callback) that shows a form with the message html, an input field and buttons OK/CANCEL.

A user should type something into a text field and press Enter or the OK button, then callback(value) is called with the value they entered.
Otherwise if the user presses Esc or CANCEL, then callback(null) is called.

In both cases that ends the input process and removes the form.
Requirements:

The form should be in the center of the window.
The form is modal. In other words, no interaction with the rest of the page is possible until the user closes it.
When the form is shown, the focus should be inside the <input> for the user.
Keys Tab/Shift+Tab should shift the focus between form fields, don’t allow it to leave for other page elements.

Usage example:

```
        showPrompt("Enter something<br>...smart :)", function(value) {
  alert(value);
});
```

A demo in the iframe:

P.S. The source document has HTML/CSS for the form with fixed positioning, but it’s up to you to make it modal.
Open a sandbox for the task.solutionA modal window can be implemented using a half-transparent that covers the whole window, like this:

```
        #cover-div {
  position: fixed;
  top: 0;
  left: 0;
  z-index: 9000;
  width: 100%;
  height: 100%;
  background-color: gray;
  opacity: 0.3;
}
```

Because the covers everything, it gets all clicks, not the page below it.
Also we can prevent page scroll by setting body.style.overflowY='hidden'.
The form should be not in the , but next to it, because we don’t want it to have opacity.
Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/forms-submit","identifier":"\/forms-submit"}); };var disqus_shortname = "javascriptinfo";var

disqus_enabled = true;
TutorialBrowser: Document, Events, Interfaces{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"}]}Document and resource loadingPage: DOMContentLoaded, load, beforeunload, unloadScripts: async, deferResource loading: onload and onerrorCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map
TutorialBrowser: Document, Events, InterfacesDocument and resource loading{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document and resource loading","item":"https://javascript.info/loading"}]}October 14, 2022Page: DOMContentLoaded, load, beforeunload, unloadThe 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 not yet have 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.
DOMContentLoadedThe 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 it 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 scriptsWhen 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:

Scripts with the async attribute, that we’ll cover a bit later, don’t block DOMContentLoaded.
Scripts that are generated dynamically with document.createElement('script') and then added to the webpage also don’t block this event.

DOMContentLoaded and stylesExternal 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 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 autofillFirefox, Chrome and Opera autofill forms on DOMContentLoaded.

For instance, if the page has a form with login and password, and the browser remembered 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.onloadThe 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() { // can also use 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.onunloadWhen a visitor leaves the page, the unload event triggers on 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 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.
If we want to cancel the transition to another page, we can’t do it here. But we can use another event – onbeforeunload.
window.onbeforeunloadIf a visitor initiated navigation away from the page or tries to close the window, the beforeunload 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 of that – there’s no way to customize the message shown to the user.

The event.preventDefault() doesn’t work from a beforeunload handler

That may sound weird, but most browsers ignore event.preventDefault(). Which means, following code may not work:

```
window.addEventListener("beforeunload", (event) => {
  // doesn't work, so this event handler doesn't do anything
  event.preventDefault();
});
```

Instead, in such handlers one should set event.returnValue to a string to get the result similar to the code above:

```
window.addEventListener("beforeunload", (event) => {
  // works, same as returning from window.onbeforeunload
  event.returnValue = "There are unsaved changes. Leave now?";
});
```

readyStateWhat 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') {
  // still loading, 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, it appeared long ago. Nowadays, it is rarely used.
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>

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>

<img src="https://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] initial readyState:loading
[2] readyState:interactive
[2] DOMContentLoaded
[3] iframe onload
[4] img onload
[4] readyState:complete
[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.

SummaryPage load events:

The DOMContentLoaded event triggers on document when the DOM is ready. We can apply JavaScript to elements at this stage.

Script such as <script>...</script> or <script src="..."></script> block DOMContentLoaded, the browser waits for them to execute.
Images and other resources may also still continue loading.

The load event on window triggers when the page and all resources are loaded. We rarely use it, because there's usually no need to wait for so long.
The beforeunload event on window triggers when the user wants to leave the page. If we cancel the event, browser asks whether the user really wants to leave (e.g we have unsaved changes).
The unload event on window triggers when the user is finally leaving, in the handler we can only do simple things that do not involve delays or asking a user. Because of that limitation, it's rarely used. We can send out a network request with navigator.sendBeacon.
document.readyState is the current state of the document, changes can be tracked in the readystatechange event:

loading – the document is loading.
interactive – the document is parsed, happens at about the same time as DOMContentLoaded, but before it.
complete – the document and resources are loaded, happens at about the same time as window.onload, but before it.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {};

Object.assign(this.page, {"url":"https:\/\/javascript.info\/onload-ondomcontentloaded","identifier":"\/onload-ondomcontentloaded"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesDocument and resource loading{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Document and resource loading","item":"https://javascript.info/loading"}]}October 25, 2021Scripts: async, deferIn 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 for the script to download, execute the downloaded script, and only then can it process the rest of the page.
That leads to two important issues:

Scripts can’t see DOM elements below them, so they can’t add handlers etc.
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.
deferThe defer attribute tells the browser not to wait for the script. Instead, the browser will continue to process the HTML, build DOM. The script loads “in the background”, and then runs when the DOM is fully built.
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>
```

In other words:

Scripts with defer never block the page.
Scripts with defer always execute when the DOM is ready (but before DOMContentLoaded event).

The following example demonstrates the second part:

```
<p>...content before scripts...</p>

<script>
  document.addEventListener('DOMContentLoaded', () => alert("DOM ready after defer!"));
</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...content after scripts...</p>
```

The page content shows up immediately.
DOMContentLoaded event handler waits for the deferred script. It only triggers when the script is downloaded and executed.

Deferred scripts keep their relative order, just like regular scripts.
Let’s say, we have two deferred scripts: the long.js and then small.js:

```
<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>

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 finishes first.
…But the defer attribute, besides telling the browser “not to block”, ensures that the relative order is kept. So even though small.js loads first, it still waits and runs after long.js executes.
That may be important for cases when we need to load a JavaScript library and then a script that depends on it.

The defer attribute is only for external scripts
The defer attribute is ignored if the <script> tag has no src.

asyncThe async attribute is somewhat like defer. It also makes the script non-blocking. But it has important differences in the behavior.
The async attribute means that a script is completely independent:

The browser doesn’t block on async scripts (like defer).
Other scripts don’t wait for async scripts, and async scripts don’t wait for them.
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)

In other words, async scripts load in the background and run when ready. The DOM and other scripts don’t wait for them, and they don’t wait for anything. A fully independent script that runs when loaded. As simple, as it can get, right?
Here’s an example similar to what we’ve seen with defer: two scripts long.js and small.js, but now with async instead of defer.
They don’t wait for each other. Whatever loads first (probably small.js) – 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>
```

The page content shows up immediately: async doesn’t block it.
DOMContentLoaded may happen both before and after async, no guarantees here.
A smaller script small.js goes second, but probably loads before long.js, so small.js runs first. Although, it might be that long.js loads first, if cached, then it runs first. In other words, async scripts run in the “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>
```

The async attribute is only for external scripts
Just like defer, the async attribute is ignored if the <script> tag has no src.

Dynamic scriptsThere’s one more important way of adding a script to the page.
We can create a script and append it to the document 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).

This can be changed if we explicitly set script.async=false. Then scripts will be executed in the document order, just like defer.
In this example, loadScript(src) function adds a script and also sets async to false.
So long.js always runs first (as it’s added first):

```
        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");
```

Without script.async=false, scripts would execute in default, load-first order (the small.js probably first).
Again, as with the defer, the order matters if we'd like to load a library and then another script that depends on it.
SummaryBoth async and defer have one common thing: downloading of such scripts doesn't block page rendering. So the user can read page content and get acquainted with the page immediately.
But there are also essential differences between them:

Order
DOMContentLoaded

async
Load-first order. Their document order doesn't matter – which loads first runs first
Irrelevant. May load and execute while the document has not yet been fully downloaded. That happens if scripts are small or cached, and the document is long enough.

defer
Document order (as they go in the document).
Execute after the document is loaded and parsed (they wait if needed), right before DOMContentLoaded.

In practice, defer is used for scripts that need the whole DOM and/or their relative execution order is important.
And async is used for independent scripts, like counters or ads. And their relative execution order does not matter.

Page without scripts should be usable
Please note: if you're using defer or async, then user will see the page before the script loads.
In such case, some graphical components are probably not initialized yet.
Don't forget to put "loading" indication and disable buttons that aren't functional yet. Let the user clearly see what he can do on the page, and what's still getting ready.

Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/script-async-defer","identifier":"\/script-async-defer"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

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.

Please note:

For our own scripts we could use JavaScript modules here, but they are not widely adopted by third-party libraries.

script.onload

The main helper is the load event. It triggers after the script was 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 variable "_"
  alert( _.VERSION ); // shows library version
};
```

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.onerrorErrors 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.

Important:
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 resourcesThe 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.

That’s for historical reasons.
Crossorigin policyThere’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:

```
// Ø=ÜÁ 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:

No crossorigin attribute – access prohibited.

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 remote server.

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 remote server.

Please note:

You can read more about cross-origin access in the chapter Fetch: Cross-Origin Requests. It describes the fetch method for network requests, but the policy is exactly the same.

Such thing as “cookies” is out of our current scope, but you can read about them in the chapter Cookies, document.cookie.

In our case, we didn’t have any crossorigin attribute. So the 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.
SummaryImages <img>, external styles, scripts and other resources provide load and error events to track their loading:

load triggers on a successful load,
error triggers on a failed load.

The only exception is <iframe>: for historical reasons it always triggers load, for any load completion, even if the page is not found.
The readystatechange event also works for resources, but is rarely used, because load/error events are simpler.
TasksLoad images with a callbackimportance: 4Normally, images are loaded when they are created. So when we add <img> to the page, the user does not see the picture immediately. The browser needs to load it first.
To show an image immediately, we can create it "in advance", like this:

```
let img = document.createElement('img');
img.src = 'my.jpg';
```

The browser starts loading the image and remembers it in the cache. Later, when the same image appears in the document (no matter how), it shows up immediately.
Create a function preloadImages(sources, callback) that loads all images from the array sources and, when ready, runs callback.
For instance, this will show an alert after the images are loaded:

```
function loaded() {
```

```
  alert("Images loaded")
}

preloadImages(["1.jpg", "2.jpg", "3.jpg"], loaded);
```

In case of an error, the function should still assume the picture “loaded”.
In other words, the callback is executed when all images are either loaded or errored out.
The function is useful, for instance, when we plan to show a gallery with many scrollable images, and want to be sure that all images are loaded.
In the source document you can find links to test images, and also the code to check whether they are loaded or not. It should output 300.
Open a sandbox for the task.solutionThe algorithm:

Make img for every source.
Add onload/onerror for every image.
Increase the counter when either onload or onerror triggers.
When the counter value equals to the sources count – we’re done: callback().

Open the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/onload-onerror","identifier":"\/onload-onerror"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, Interfaces{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"}]}MiscellaneousMutation observerSelection and RangeEvent loop: microtasks and macrotasksCtrl + !•Previous lessonCtrl + !’Next lessonShareTutorial map
TutorialBrowser: Document, Events, InterfacesMiscellaneous{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Miscellaneous","item":"https://javascript.info/ui-misc"}]}September 13, 2020Mutation observerMutationObserver is a built-in object that observes a DOM element and fires a callback when it detects a change.
We’ll first take a look at the syntax, and then explore a real-world use case, to see

where such thing may be useful.
SyntaxMutationObserver is easy to use.
First, we create an observer with a callback-function:

```
let observer = new MutationObserver(callback);
```

And then attach it to a DOM node:

```
observer.observe(node, config);
```

config is an object with boolean options “what kind of changes to react on”:

childList – changes in the direct children of node,
subtree – in all descendants of node,
attributes – attributes of node,
attributeFilter – an array of attribute names, to observe only selected ones.
characterData – whether to observe node.data (text content),

Few other options:

attributeOldValue – if true, pass both the old and the new value of attribute to callback (see below), otherwise only the new one (needs attributes option),
characterDataOldValue – if true, pass both the old and the new value of node.data to callback (see below), otherwise only the new one (needs characterData option).

Then after any changes, the callback is executed: changes are passed in the first argument as a list of MutationRecord objects, and the observer itself as the second argument.
MutationRecord objects have properties:

type – mutation type, one of

"attributes": attribute modified
"characterData": data modified, used for text nodes,
"childList": child elements added/removed,

target – where the change occurred: an element for "attributes", or text node for "characterData", or an element for a "childList" mutation,
addedNodes/removedNodes – nodes that were added/removed,
previousSibling/nextSibling – the previous and next sibling to added/removed nodes,
attributeName/attributeNamespace – the name/namespace (for XML) of the changed attribute,
oldValue – the previous value, only for attribute or text changes, if the corresponding option is set attributeOldValue/characterDataOldValue.

For example, here's a with a contentEditable attribute. That attribute allows us to focus on it and edit.

```
<div contentEditable id="elem">Click and <b>edit</b>, please</div>

<script>
let observer = new MutationObserver(mutationRecords => {
  console.log(mutationRecords); // console.log(the changes)
});

// observe everything except attributes
observer.observe(elem, {
  childList: true, // observe direct children
  subtree: true, // and lower descendants too
  characterDataOldValue: true // pass old data to callback
});
</script>
```

If we run this code in the browser, then focus on the given and change the text inside edit, console.log will show one mutation:

```
mutationRecords = [{
  type: "characterData",
  oldValue: "edit",
  target: <text node>,
  // other properties empty
}];
```

If we make more complex editing operations, e.g. remove the edit, the mutation event may contain multiple mutation records:

```
mutationRecords = [{
  type: "childList",
  target: <div#elem>,
  removedNodes: [<b>],
  nextSibling: <text node>,
  previousSibling: <text node>
  // other properties empty
}, {
  type: "characterData"
  target: <text node>
  // ...mutation details depend on how the browser handles such removal
  // it may coalesce two adjacent text nodes "edit " and ", please" into one node
  // or it may leave them separate text nodes
}];
```

So, MutationObserver allows to react on any changes within DOM subtree.
Usage for integrationWhen such thing may be useful?
Imagine the situation when you need to add a third-party script that contains useful functionality, but also does something unwanted, e.g. shows ads Unwanted ads .
Naturally, the third-party script provides no mechanisms to remove it.
Using MutationObserver, we can detect when the unwanted element appears in our DOM and remove it.
There are other situations when a third-party script adds something into our document, and we'd like to detect, when it happens, to adapt our page, dynamically resize something etc.
MutationObserver allows to implement this.
Usage for architectureThere are also situations when MutationObserver is good from

architectural standpoint.
Let’s say we’re making a website about programming. Naturally, articles and other materials may contain source code snippets.
Such snippet in an HTML markup looks like this:

```
...
<pre class="language-javascript"><code>
  // here's the code
  let hello = "world";
</code></pre>
...
```

For better readability and at the same time, to beautify it, we’ll be using a JavaScript syntax highlighting library on our site, like Prism.js. To get syntax highlighting for above snippet in Prism, Prism.highlightElem(pre) is called, which examines the contents of such pre elements and adds special tags and styles for colored syntax highlighting into those elements, similar to what you see in examples here, on this page.
When exactly should we run that highlighting method? Well, we can do it on DOMContentLoaded event, or put the script at the bottom of the page. The moment our DOM is ready, we can search for elements pre[class*="language"] and call Prism.highlightElem on them:

```
// highlight all code snippets on the page
document.querySelectorAll('pre[class*="language"]').forEach(Prism.highlightElem);
```

Everything’s simple so far, right? We find code snippets in HTML and highlight them. Now let’s go on. Let’s say we’re going to dynamically fetch materials from a server. We’ll study methods for that later in the tutorial. For now it only matters that we fetch an HTML article from a webserver and display it on demand:

```
let article = /* fetch new content from server */
articleElem.innerHTML = article;
```

The new article HTML may contain code snippets. We need to call Prism.highlightElem on them, otherwise they won't get highlighted.
Where and when to call Prism.highlightElem for a dynamically loaded article?
We could append that call to the code that loads an article, like this:

```
let article = /* fetch new content from server */
articleElem.innerHTML = article;

let snippets = articleElem.querySelectorAll('pre[class*="language-"]');
snippets.forEach(Prism.highlightElem);
```

…But, imagine if we have many places in the code where we load our content – articles, quizzes, forum posts, etc. Do we need to put the highlighting call everywhere, to highlight the code in content after loading? That's not very convenient.
And what if the content is loaded by a third-party module? For example, we have a forum written by someone else, that loads content dynamically, and we'd like to add syntax highlighting to it. No one likes patching third-party scripts.
Luckily, there's another option.
We can use MutationObserver to automatically detect when code snippets are inserted into the page and highlight them.
So we'll handle the highlighting functionality in one place, relieving us from the need to integrate it.
Dynamic highlight demoHere's the working example.
If you run this code, it starts observing the element below and highlighting any code snippets that appear there:

```
let observer = new MutationObserver(mutations => {
```

```
  for(let mutation of mutations) {
    // examine new nodes, is there anything to highlight?

    for(let node of mutation.addedNodes) {
      // we track only elements, skip other nodes (e.g. text nodes)
      if (!(node instanceof HTMLElement)) continue;

      // check the inserted element for being a code snippet
      if (node.matches('pre[class*="language-"]')) {
        Prism.highlightElement(node);
      }

      // or maybe there's a code snippet somewhere in its subtree?
      for(let elem of node.querySelectorAll('pre[class*="language-"]')) {
        Prism.highlightElement(elem);
      }
    }
  }

});

let demoElem = document.getElementById('highlight-demo');

observer.observe(demoElem, {childList: true, subtree: true});
```

Here, below, there’s an HTML-element and JavaScript that dynamically fills it using innerHTML.
Please run the previous code (above, observes that element), and then the code below. You’ll see how MutationObserver detects and highlights the snippet.
A demo-element with id="highlight-demo", run the code above to observe it.
The following code populates its innerHTML, that causes the MutationObserver to react and highlight its contents:

```
        let demoElem = document.getElementById('highlight-demo');

// dynamically insert content with code snippets
demoElem.innerHTML = `A code snippet is below:
  <pre class="language-javascript"><code> let hello = "world!"; </code></pre>
  <div>Another one:</div>
  <div>
    <pre class="language-css"><code>.class { margin: 5px; } </code></pre>
  </div>
`;
```

Now we have MutationObserver that can track all highlighting in observed elements or the whole document. We can add/remove code snippets in HTML without thinking about it.
Additional methodsThere's a method to stop observing the node:

observer.disconnect() – stops the observation.

When we stop the observing, it might be possible that some changes were not yet processed by the observer. In such cases, we use

observer.takeRecords() – gets a list of unprocessed mutation records – those that happened, but the callback has not handled them.

These methods can be used together, like this:

```
        // get a list of unprocessed mutations
// should be called before disconnecting,
// if you care about possibly unhandled recent mutations
let mutationRecords = observer.takeRecords();

// stop tracking changes
observer.disconnect();
...
```

Records returned by observer.takeRecords() are removed from the processing queue

The callback won't be called for records, returned by observer.takeRecords().

Garbage collection interaction

Observers use weak references to nodes internally. That is, if a node is removed from the DOM, and becomes unreachable, then it can be garbage collected. The mere fact that a DOM node is observed doesn’t prevent the garbage collection.

SummaryMutationObserver can react to changes in DOM – attributes, text content and adding/removing elements.
We can use it to track changes introduced by other parts of our code, as well as to integrate with third-party scripts.
MutationObserver can track any changes. The config “what to observe” options are used for optimizations, not to spend resources on unneeded callback invocations.
Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/mutation-observer","identifier":"\/mutation-observer"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialBrowser: Document, Events, InterfacesMiscellaneous{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Browser: Document, Events, Interfaces","item":"https://javascript.info/ui"},{"@type":"ListItem","position":3,"name":"Miscellaneous","item":"https://javascript.info/ui-misc"}]}October 30, 2022Selection and RangeIn this chapter we’ll cover selection in the document, as well as selection in form fields, such as <input>.
JavaScript can access an existing selection, select/deselect DOM nodes as a whole or partially, remove the selected content from the document, wrap it into a tag, and so on.
You can find some recipes for common tasks at the end of the chapter, in “Summary” section. Maybe that covers your current needs, but you’ll get much more if you read the whole text.
The underlying Range and Selection objects are easy to grasp, and then you’ll need no recipes to make them do what you want.
RangeThe basic concept of selection is Range, that is essentially a pair of “boundary points”: range start and range end.
A Range object is created without parameters:

```
let range = new Range();
```

Then we can set the selection boundaries using range.setStart(node, offset) and range.setEnd(node, offset).
As you might guess, further we'll use the Range objects for selection, but first let's create few such objects.
Selecting the text partiallyThe interesting thing is that the first argument node in both methods can be either a text node or an element node, and the meaning of the second argument depends on that.
If node is a text node, then offset must be the position in its text.
For example, given the element Hello , we can create the range containing the letters "ll" as follows:

```
    <p id="p">Hello</p>
<script>
  let range = new Range();
  range.setStart(p.firstChild, 2);
  range.setEnd(p.firstChild, 4);

  // toString of a range returns its content as text
  console.log(range); // ll
</script>
```

Here we take the first child of (that's the text node) and specify the text positions inside it:

Selecting element nodesAlternatively, if node is an element node, then offset must be the child number.
That's handy for making ranges that contain nodes as a whole, not stop somewhere

inside their text.
For example, we have a more complex document fragment:

```
<p id="p">Example: <i>italic</i> and <b>bold</b></p>
```

Here’s its DOM structure with both element and text nodes:

%¾ P#text Example:$#%¾ I#text italic#text $#and$#%¾ B#text bold

```
let selectPDomtree = {
  "name": "P",
  "nodeType": 1,
  "children": [{
    "name": "#text",
    "nodeType": 3,
    "content": "Example: "
  }, {
    "name": "I",
    "nodeType": 1,
    "children": [{
      "name": "#text",
      "nodeType": 3,
      "content": "italic"
    }]
  }, {
    "name": "#text",
    "nodeType": 3,
    "content": " and "
  }, {
    "name": "B",
    "nodeType": 1,
    "children": [{
      "name": "#text",
      "nodeType": 3,
      "content": "bold"
    }]
  }]
}

drawHtmlTree(selectPDomtree, 'div.select-p-domtree', 690, 320);
```

Let’s make a range for "Example: italic".
As we can see, this phrase consists of exactly two children of <p>, with indexes 0 and 1:

The starting point has <p> as the parent node, and 0 as the offset.
So we can set it as range.setStart(p, 0).

The ending point also has <p> as the parent node, but 2 as the offset (it specifies the range up to, but not including offset).
So we can set it as range.setEnd(p, 2).

Here’s the demo. If you run it, you can see that the text gets selected:

```
<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
  console.log(range); // Example: italic

  // apply this range for document selection (explained later below)
  document.getSelection().addRange(range);
</script>
```

Here’s a more flexible test stand where you can set range start/end numbers and explore other 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 below
    document.getSelection().removeAllRanges();
    document.getSelection().addRange(range);
  };
</script>
```

E.g. selecting in the same <p> from offset 1 to 4 gives us the range italic and bold:

Starting and ending nodes can be different

We don’t have to use the same node in setStart and setEnd. A range may span across many unrelated nodes. It’s only important that the end is after the start in the document.

Selecting a bigger fragmentLet’s make a bigger selection in our example, like this:

We already know how to do that. 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 "Example: ")
ends at the position 3 in <b> first child (taking first three letters of “bold”, 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);

  console.log(range); // ample: italic and bol

  // use this range for selection (explained later)
  window.getSelection().addRange(range);
</script>
```

As you can see, it’s fairly easy to make a range of whatever we want.
If we’d like to take nodes as a whole, we can pass elements in setStart/setEnd.
Otherwise, we can work on the text level.
Range propertiesThe range object that we created in the example above has following properties:

startContainer, startOffset – node and offset of the start,

in the example above: first text node inside <p> and 2.

endContainer, endOffset – node and offset of the end,

in the example above: first text node inside <b> and 3.

collapsed – boolean, true if the range starts and ends on the same point (so there’s no content inside the range),

in the example above: false

commonAncestorContainer – the nearest common ancestor of all nodes within the range,

in the example above: <p>

Range selection methodsThere are many convenient methods to manipulate ranges.
We’ve already seen setStart and setEnd, here are other similar methods.
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

Technically, setStart/setEnd can do anything, but more methods provide more convenience.
In all these methods, 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.
Even more methods to create ranges:

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

Range editing methodsOnce the range is created, we can manipulate its content using these methods:

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) { console.log(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>
```

There also exist methods to compare ranges, but these are rarely used. When you need them, please refer to the spec or MDN manual.
SelectionRange is a generic object for managing selection ranges. Although, creating a Range doesn't mean that we see a selection on screen.
We may create Range objects, pass them around – they do not visually select anything on their own.
The document selection is represented by 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:

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.
Here's a small demo that shows the current selection (select something and click) as text:
alert(document.getSelection())
Selection propertiesAs said, a selection may in theory contain multiple ranges. We can get these range objects using the method:

getRangeAt(i) – get i-th range, starting from 0. In all browsers except Firefox, only 0 is used.

Also, there exist properties that often provide better convenience.
Similar to a range, a selection object 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/start vs Range

There's an important differences of a selection anchor/focus compared with a Range start/end.

As we know, Range objects always have their start before the end.

For selections, that's not always the case.

Selecting something with a mouse can be done in both directions: either "left-to-right" or "right-to-left".

In other words, when the mouse button is pressed, and then it moves forward in the document, then its end (focus) will be after its start (anchor).

E.g. if the user starts selecting with mouse and goes from "Example" to "italic":

…But the same selection could be done backwards: starting from "italic" to "Example" (backward direction), then its end (focus) will be before the start (anchor):

Selection eventsThere are events on to keep track of selection:

elem.onselectstart – when a selection starts specifically on element elem (or inside it). For instance, when the user presses the mouse button on it and starts to move the pointer.

Preventing the default action cancels the selection start. So starting a selection from this element becomes impossible, but the element is still selectable. The visitor just needs to start the selection from elsewhere.

document.onselectionchange – whenever a selection changes or starts.

Please note: this handler can be set only on document, it tracks all selections in it.

Selection tracking demoHere's a small demo. It tracks the current selection on the document and shows its boundaries:

```
        <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 selection = document.getSelection();

    let {anchorNode, anchorOffset, focusNode, focusOffset} = selection;

    // anchorNode and focusNode are text nodes usually
    from.value = `${anchorNode?.data}, offset ${anchorOffset}`;
    to.value = `${focusNode?.data}, offset ${focusOffset}`;
  };
</script>
```

Selection copying demoThere are two approaches to copying the selected content:

We can use document.getSelection().toString() to get it as text.
Otherwise, to copy the full DOM, e.g. if we need to keep formatting, we can get the underlying ranges with getRangeAt(...). A Range object, in turn, has cloneContents() method that clones its content and returns as DocumentFragment object, that we can insert elsewhere.

Here’s the demo of copying the selected content 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 methodsWe can work with the selection by adding/removing 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.

There are also convenience methods to manipulate the selection range directly, without intermediate Range calls:

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)

For most tasks these methods are just fine, there's no need to access the underlying Range object.
For example, selecting the whole contents of the paragraph <p>:

```
<p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  // select from 0th child of <p> to the last child
  document.getSelection().setBaseAndExtent(p, 0, p, p.childNodes.length);
</script>
```

The same thing using ranges:

```
        <p id="p">Select me: <i>italic</i> and <b>bold</b></p>

<script>
  let range = new Range();
  range.selectNodeContents(p); // or selectNode(p) to select the <p> tag too

  document.getSelection().removeAllRanges(); // clear existing selection if any
  document.getSelection().addRange(range);
</script>
```

To select something, remove the existing selection first
If a document selection already exists, empty it first with removeAllRanges(). And then add ranges. Otherwise, all browsers except Firefox ignore new ranges.
The exception is some selection methods, that replace the existing selection, such as setBaseAndExtent.

Selection in form controlsForm elements, such as input and textarea provide 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 selectionFor 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.
document.onselectionchange event should not trigger for selections inside a form control, according to the spec, as it’s not related to document selection and ranges. Some browsers generate it, but we shouldn’t rely on it.

Example: moving cursorWe 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 is exactly at that place
      area.selectionStart = area.selectionEnd = 10;
    });
  };
</script>
```

Example: modifying selectionTo 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 cursorIf 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 unselectableTo make something unselectable, there are three ways:

Use CSS property user-select: none.

```
        <style>
#elem {
  user-select: none;
}
</style>
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>
```

This doesn’t allow the selection to start at elem. But the user may start the selection elsewhere and include elem into it.
Then elem will become a part of document.getSelection(), so the selection actually happens, but its content is usually ignored in copy-paste.

Prevent default action in onselectstart or mousedown events.

```
<div>Selectable <div id="elem">Unselectable</div> Selectable</div>

<script>
  elem.onselectstart = () => false;
</script>
```

This prevents starting the selection on elem, but the visitor may start it at another element, then extend to elem.
That's convenient when there's another event handler on the same action that triggers the select (e.g. mousedown). So we disable the selection to avoid conflict, still allowing elem contents to be copied.

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.

References
DOM spec: Range
Selection API
HTML spec: APIs for the text control selections

SummaryWe covered two different APIs for selections:

For document: Selection and Range objects.
For input, textarea: additional methods and properties.

The second API is very simple, as it works with text.
The most used recipes are probably:

Getting the selection:

```
let selection = document.getSelection();

let cloned = /* element to clone the selected nodes to */;
```

```
// then apply Range methods to selection.getRangeAt(0)
// or, like here, to all ranges to support multi-select
for (let i = 0; i < selection.rangeCount; i++) {
  cloned.append(selection.getRangeAt(i).cloneContents());
}
```

Setting the selection:

```
        let selection = document.getSelection();

// directly:
selection.setBaseAndExtent(...from...to...);

// or we can create a range and:
selection.removeAllRanges();
selection.addRange(range);
```

And finally, about the cursor. The cursor position in editable elements, like <textarea> is always at the start or the end of the selection. We can use it to get cursor position or to move the cursor by setting elem.selectionStart and elem.selectionEnd.

Ctrl + !•Previous lessonCtrl + !’Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/selection-range","identifier":"\/selection-range"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;