Part 2

Browser: Document, Events, Interfaces

JS

Ilya Kantor

Built at November 26, 2018

The last version of the tutorial is at https://javascript.info.

We constantly work to improve the tutorial. If you find any mistakes, please write at our github .

Learning how to manage the browser page: add elements, manipulate their size and position, dynamically create interfaces and interact with the visitor.

Document

Here we'll learn to manipulate a web-page using JavaScript.

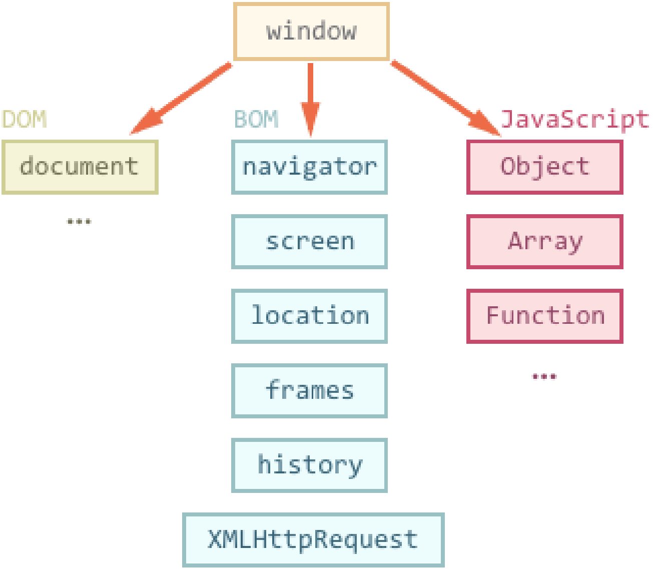
Browser environment, specs

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

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

A host environment provides platform-specific objects and functions additional to the language core. Web browsers give a means to control web pages. Node.JS provides server-side features, and so on.

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



There's a "root" object called `window`. It has two roles:

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

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

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

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

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

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

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

Document Object Model (DOM)

The `document` object gives access to the page content. We can change or create anything on the page using it.

For instance:

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

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

Here we used `document.body.style`, but there's much, much more. Properties and methods are described in the specification. There happen to be two working groups who develop it:

1. W3C – the documentation is at https://www.w3.org/TR/dom .
2. WhatWG , publishing at https://dom.spec.whatwg.org .

As it happens, the two groups don't always agree, so it's like we have two sets of standards. But they are very similar and eventually things merge. The documentation that you can find on the given resources is very similar, with about a 99% match. There are very minor differences that you probably won't notice.

Personally, I find https://dom.spec.whatwg.org more pleasant to use.

In the ancient past, there was no standard at all – each browser implemented however it wanted. Different browsers had different sets, methods, and properties for the same thing, and developers had to write different code for each of them. Dark, messy times.

Even now we can sometimes meet old code that uses browser-specific properties and works around incompatibilities. But, in this tutorial we'll use modern stuff: there's no need to learn old things until you really need to (chances are high that you won't).

Then the DOM standard appeared, in an attempt to bring everyone to an agreement. The first version was "DOM Level 1", then it was extended by DOM Level 2, then DOM Level 3, and now it's reached DOM Level 4. People from WhatWG group got tired of version numbers and are calling it just "DOM", without a number. So we'll do the same.

> ℹ **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 it too.
>
> For instance, server-side tools that download HTML pages and process them use the DOM. They may support only a part of the specification though.

> ℹ **CSSOM for styling**
>
> CSS rules and stylesheets are not structured like HTML. There's a separate specification CSSOM that explains how they are represented as objects, and how to read and write them.
>
> CSSOM is used together with DOM when we modify style rules for the document. In practice though, CSSOM is rarely required, because usually CSS rules are static. We rarely need to add/remove CSS rules from JavaScript, so we won't cover it right now.

BOM (part of HTML spec)

Browser Object Model (BOM) are additional objects provided by the browser (host environment) to work with everything except the document.

For instance:

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

- The location object allows us to read the current URL and can redirect the browser to a new one.

Here’s how we can use the `location` object:

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

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

HTML specification

BOM is the part of the general HTML specification .

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

Summary

Talking 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.

Now we'll get down to learning DOM, because the document plays the central role in the UI.

Please note the links above, as there's so much stuff to learn it's impossible to cover and remember everything.

When you'd like to read about a property or a method, the Mozilla manual at https://developer.mozilla.org/en-US/search is a nice resource, but reading the corresponding spec may be better: it's more complex and longer to read, but will make your fundamental knowledge sound and complete.

DOM tree

The backbone of an HTML document are tags.

According to Document Object Model (DOM), every HTML-tag is an object. Nested tags are called "children" of the enclosing one.

The text inside a tag it is an object as well.

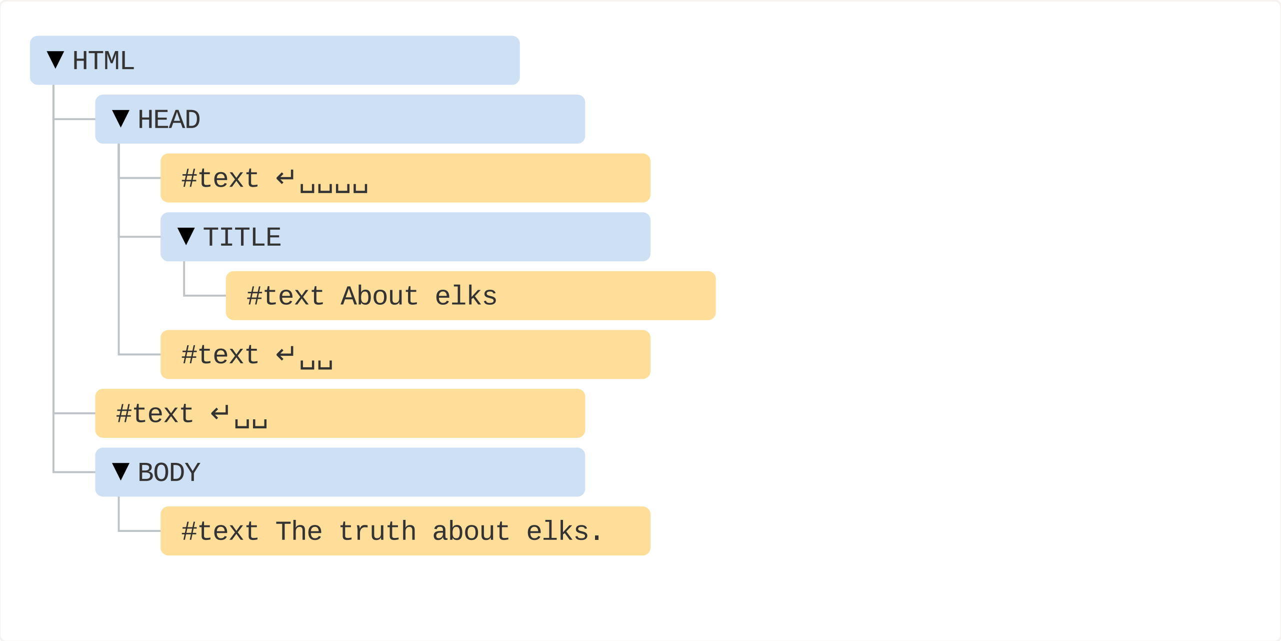
All these objects are accessible using JavaScript.

An example of DOM

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

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

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



Tags are called *element nodes* (or just elements). Nested tags become children of the enclosing ones. As a result we have a tree of elements: `<html>` is at the root, then `<head>` and `<body>` are its children, etc.

The text inside elements forms *text nodes*, 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 elks"`.

Please note the special characters in text nodes:

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

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

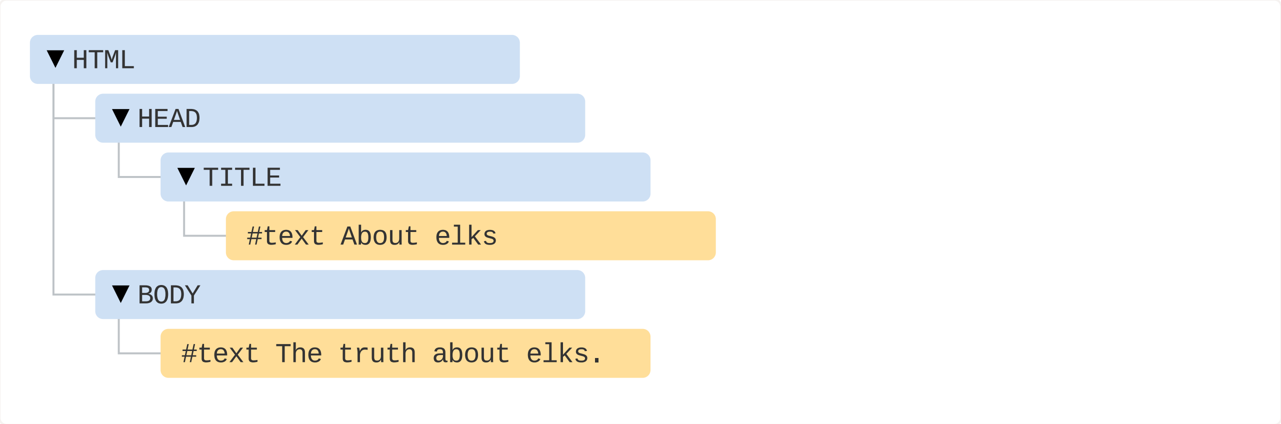
There are only two top-level exclusions:

1. Spaces and newlines before `<head>` are ignored for historical reasons,
2. If we put something after `</body>`, then that is automatically moved inside the `body`, at the end, as the HTML spec requires that all content must be inside `<body>`. So there may be no 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 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 elks</title></head><body>The truth about elks.</body></h
```



Edge spaces and in-between empty text 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.

That’s because they are mainly used to decorate HTML, and do not affect how it is shown (in most cases).

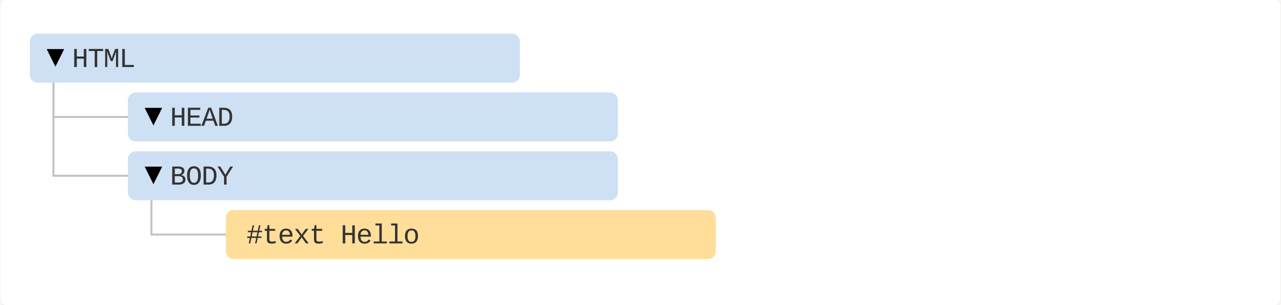
On further DOM pictures we’ll sometimes omit them where they are irrelevant, to keep things short.

Autocorrection

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

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

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

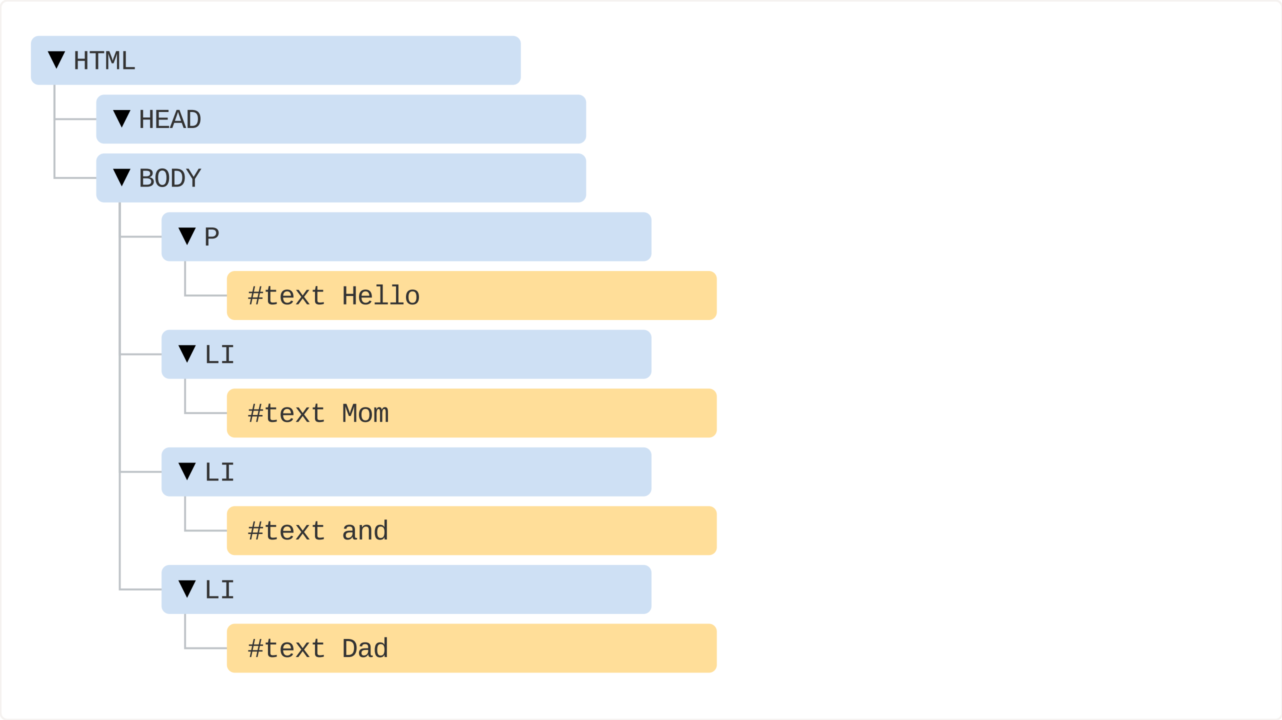


While generating the DOM, browsers automatically process errors in the document, close tags and so on.

Such an “invalid” document:

```
<p>Hello
<li>Mom
<li>and
<li>Dad
```

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



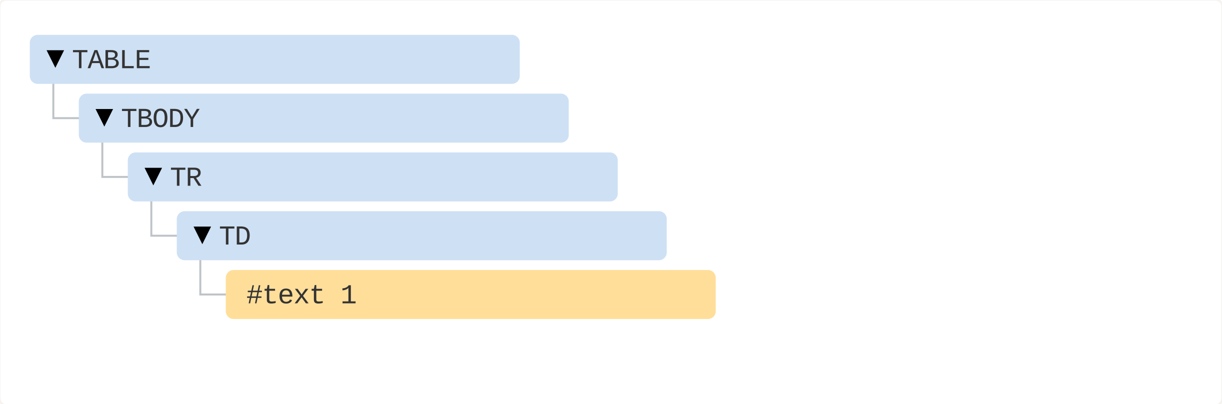
⚠ **Tables always have `<tbody>`**

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

For the HTML:

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

DOM-structure will be:

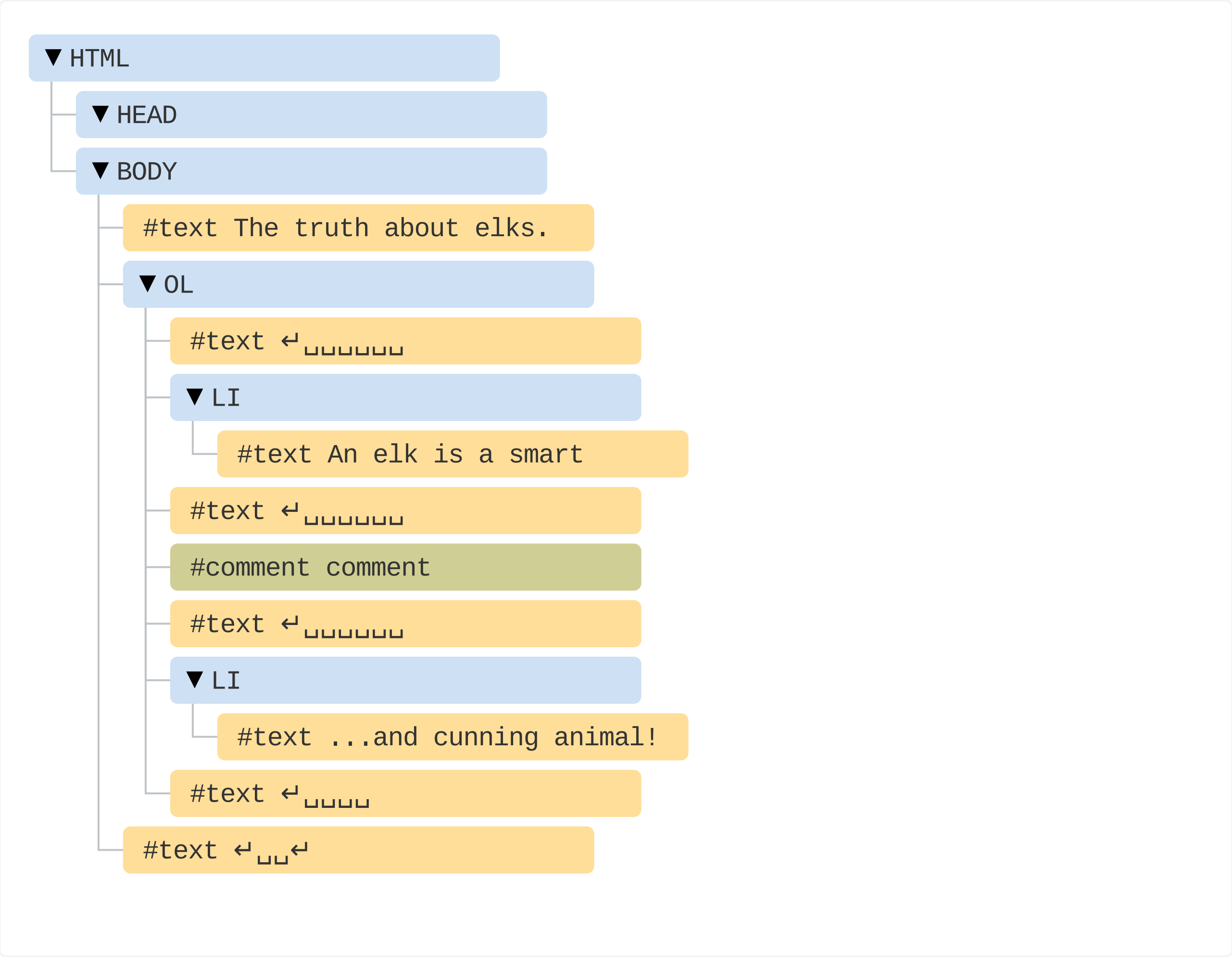


You see? The `<tbody>` appeared out of nowhere. You should keep this in mind while working with tables to avoid surprises.

Other node types

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

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



Here we see a new tree node type – *comment node*, labeled as `#comment`.

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

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

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

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

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

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

See it for yourself

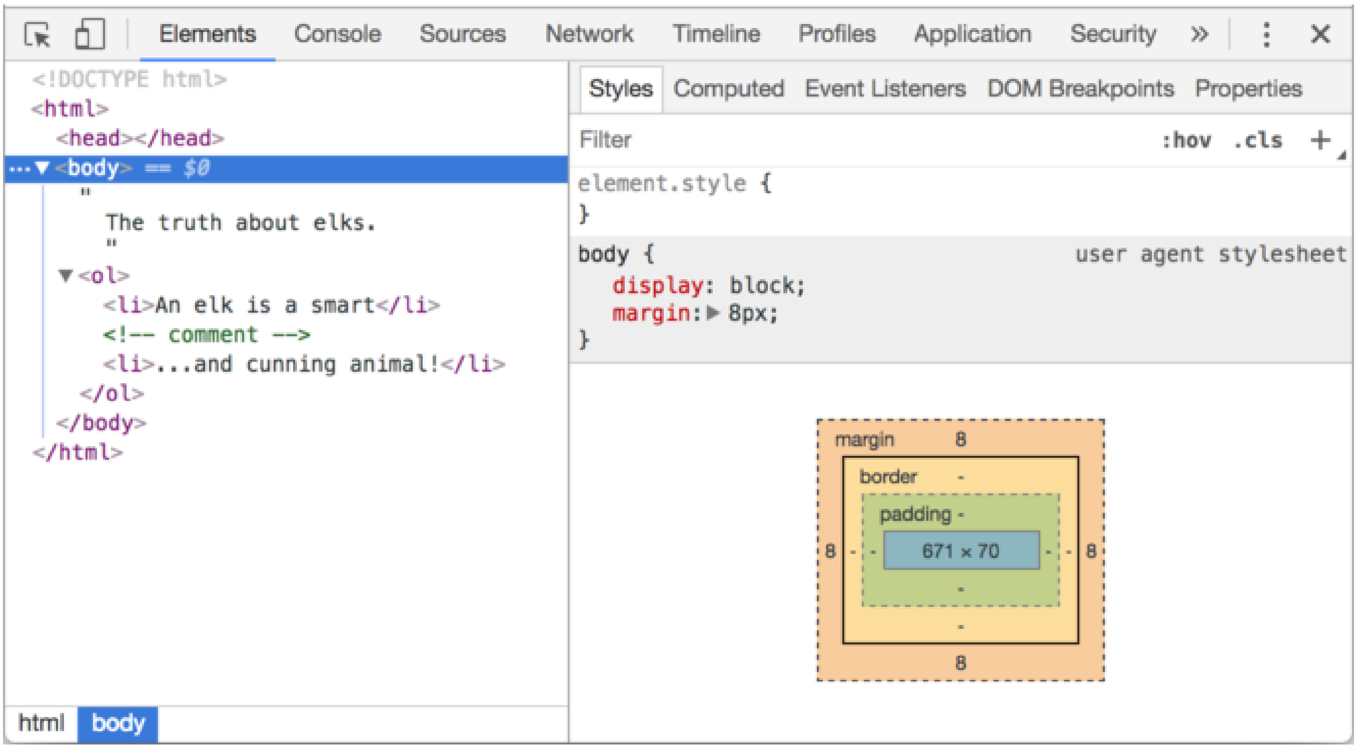
To see the DOM structure in real-time, try Live DOM Viewer . Just type in the document, and it will show up DOM at an instant.

In the browser inspector

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 elks.html, turn on the browser developer tools and switch to the Elements tab.

It should look like this:



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

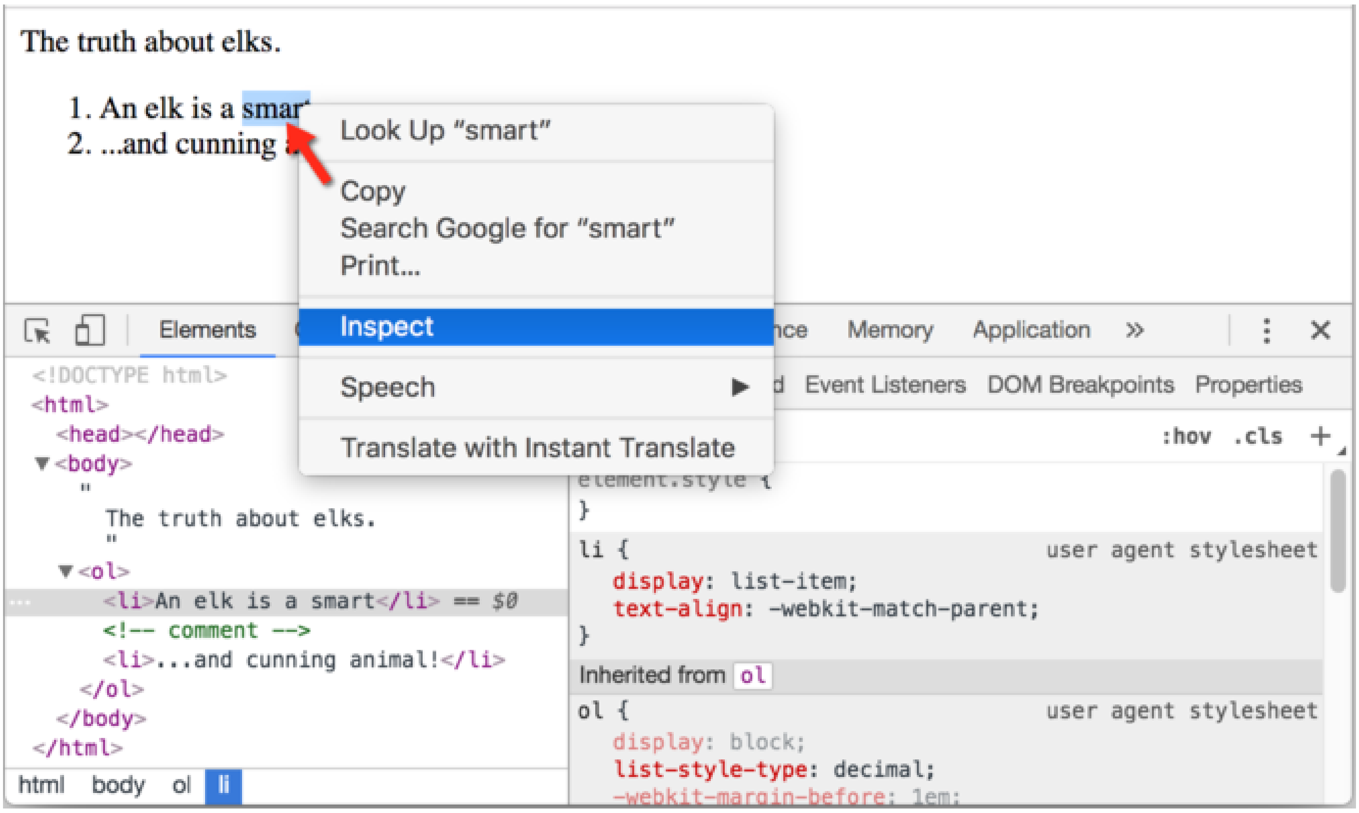
```
element.style {
}
body {                                    user agent stylesheet
   display: block;
   margin:▶ 8px;
}
```

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

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



The truth about elks.

1. An elk is a smar
2. ...and cunning

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

```
element.style {
}
li {                                   user agent stylesheet
  display: list-item;
  text-align: -webkit-match-parent;
}
Inherited from ol
ol {                                   user agent stylesheet
  display: block;
  list-style-type: decimal;
  -webkit-margin-before: 1em;
```

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.

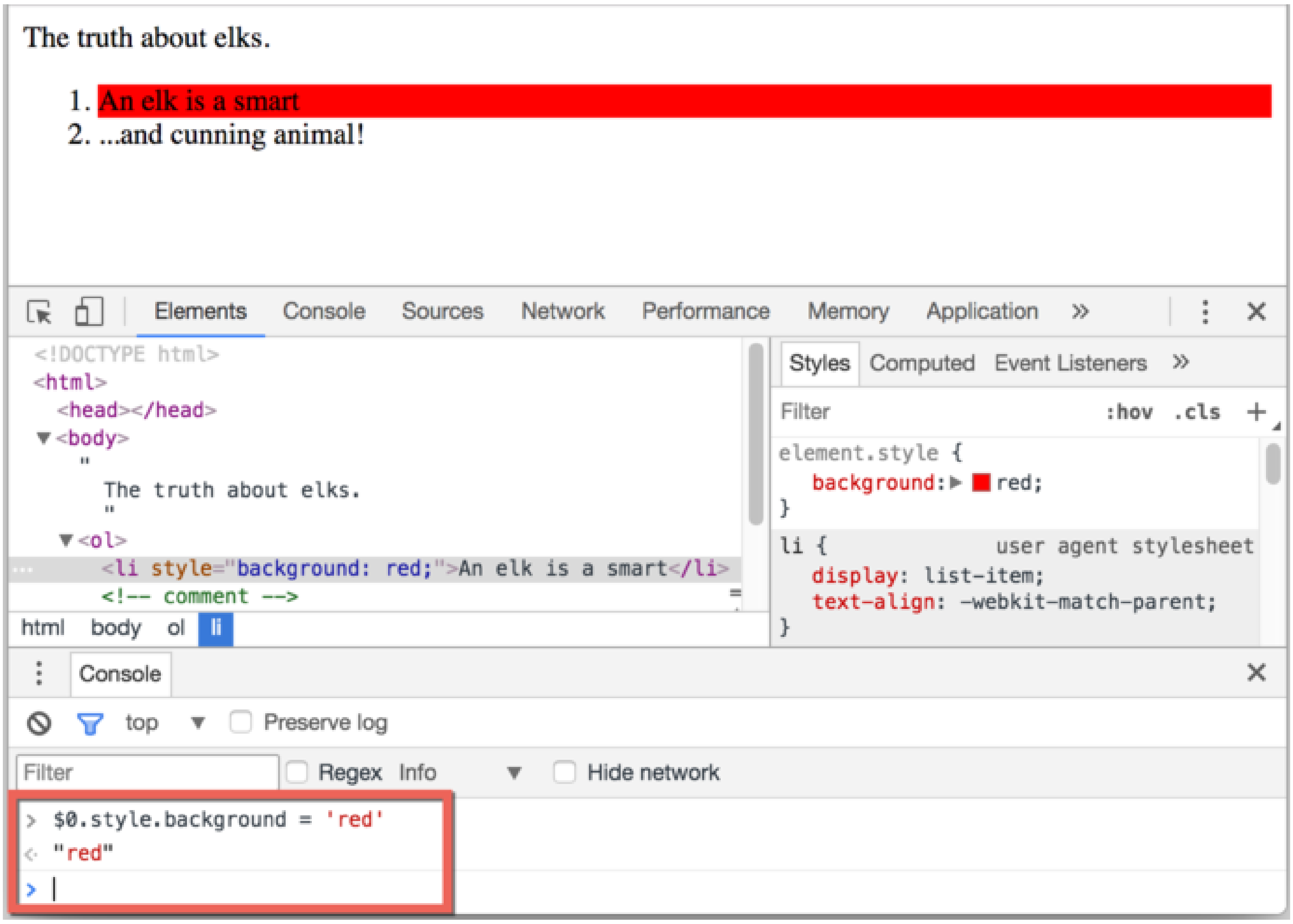
Interaction with console

As we explore the DOM, we also may want to apply JavaScript to it. Like: get a node and run some code to modify it, to see how it looks. Here are few tips to travel between the Elements tab and the console.

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

Now the last selected element is available as `$0`, the previously selected is `$1` etc.

We can run commands on them. For instance, `$0.style.background = 'red'` makes the selected list item red, like this:



The truth about elks.

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

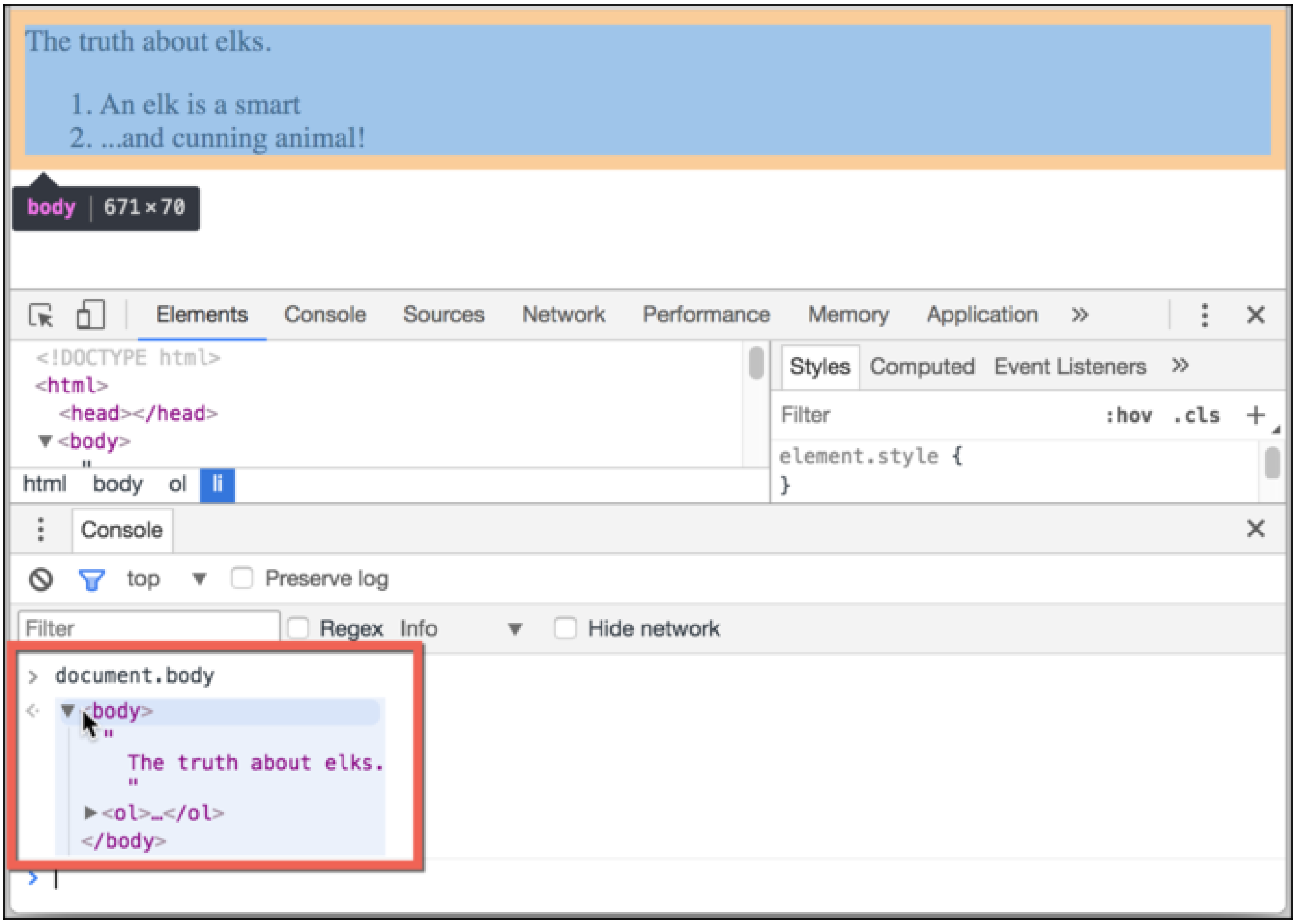
```
<!DOCTYPE html>
<html>
  <head></head>
▼<body>
    "
    The truth about elks.
    "
  ▼<ol>
      <li style="background: red;">An elk is a smart</li>
      <!-- comment -->
```

```
element.style {
    background: ▶ red;
}
li {                    user agent stylesheet
    display: list-item;
    text-align: -webkit-match-parent;
}
```

```
> $0.style.background = 'red'
<· "red"
> |
```

From the other side, if we’re in console and have a variable referencing a DOM node, then we can use the command `inspect(node)` to see it in the Elements pane.

Or we can just output it in the console and explore “at-place”, like `document.body` below:



The truth about elks.

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

```
<!DOCTYPE html>
<html>
  <head></head>
 ▼<body>
```

```
element.style {
}
```

```
> document.body
<· ▼<body>
      "
        The truth about elks.
      "
     ▶<ol>…</ol>
    </body>
>
```

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.

Summary

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

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

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

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.

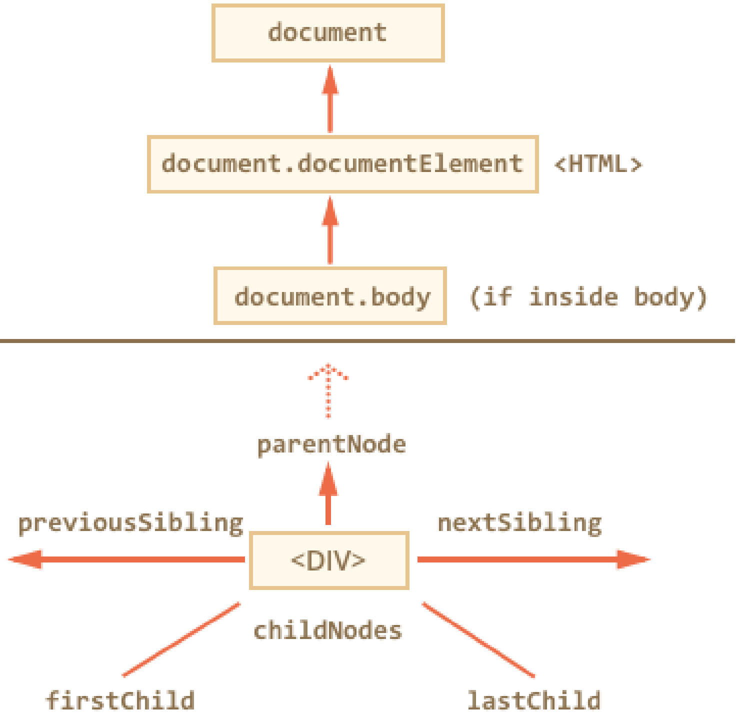
DOM nodes have properties and methods that allow to travel between them, modify, move around the page and more. We'll get down to them in the next chapters.

Walking the DOM

The DOM allows to do anything with elements and their contents, but first we need to reach the corresponding DOM object, get it into a variable, and then we are able to modify it.

All operations on the DOM start with the `document` object. From it we can access any node.

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



Let's discuss them in more detail.

On top: documentElement and body

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

`<html>` = `document.documentElement`

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

`<body>` = `document.body`

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

`<head>` = `document.head`

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

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

A script cannot access an element that doesn’t exist at the moment of running.

In particular, if a script is inside `<head>`, then `document.body` is unavailable, because the browser did not read it yet.

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

```
<html>

<head>
  <script>
    alert( "From HEAD: " + document.body ); // null, there's no <body> yet
  </script>
</head>

<body>

  <script>
    alert( "From BODY: " + document.body ); // HTMLBodyElement, now it exists
  </script>

</body>
</html>
```

In the DOM world `null` means “doesn’t exist”

In the DOM, the `null` value means “doesn’t exist” or “no such node”.

Children: childNodes, firstChild, lastChild

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

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

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

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

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

…And if we ask for all descendants of `<body>`, then we get direct children `<div>`, `<ul>` and also more nested elements like `<li>` (being a child of `<ul>`) and `<b>` (being a child of `<li>`) – the entire subtree.

The `childNodes` collection provides access to all child nodes, including text nodes.

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

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

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

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

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

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

They are just shorthands. If there exist child nodes, then the following is always true:

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

There’s also a special function `elem.hasChildNodes()` to check whether there are any child nodes.

DOM collections

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

There are two important consequences:

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

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

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

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

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

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

```
alert( Array.from(document.body.childNodes).filter ); // now it's there
```

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 assigning `childNodes[i] = ...`.

Changing DOM needs other methods, we’ll see them in the next chapter.

DOM collections are live

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

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

Don’t use `for..in` to loop over collections

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

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

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

Siblings and the parent

Siblings are nodes that are children of the same parent. For instance, `<head>` and `<body>` are siblings:

- `<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 parent is available as `parentNode`.

The next node in the same parent (next sibling) is `nextSibling`, and the previous one is `previousSibling`.

For instance:

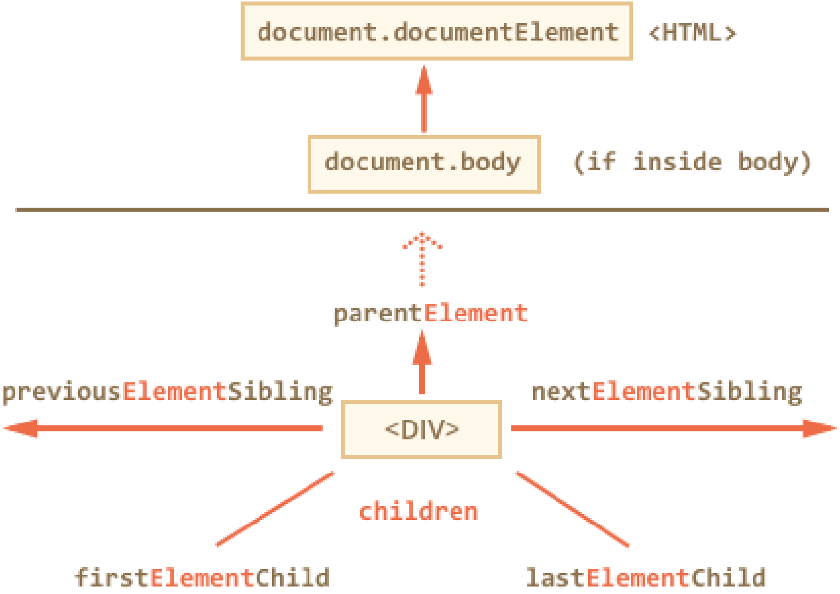
```
<html><head></head><body><script>
  // HTML is "dense" to evade extra "blank" text nodes.

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

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

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

Element-only navigation

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

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

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



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

- `children` – only those children that are element nodes.

- `firstElementChild`, `lastElementChild` – first and last element children.
- `previousElementSibling`, `nextElementSibling` – neighbour 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
```

In other words, the `documentElement` (`<html>`) is the root node. Formally, it has `document` as its parent. But `document` is not an element node, so `parentNode` returns it and `parentElement` does not.

Sometimes that matters when we’re walking over the chain of parents and call a method on each of them, but `document` doesn’t have it, so we exclude it.

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

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

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

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

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

More links: tables

Till now we described the basic navigation properties.

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

Tables are a great example and important particular case of that.

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).

`<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 the content of the first row, second cell
  alert( table.rows[0].cells[1].innerHTML ) // "two"
</script>
```

The specification: tabular data .

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

Summary

Given a DOM node, we can go to its immediate neighbours 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.

✓ Tasks

DOM children

importance: 5

For the page:

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

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

How to access:

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

To solution

The sibling question

importance: 5

If `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` ?

To solution

Select all diagonal cells

importance: 5

Write 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:

1:1	2:1	3:1	4:1	5:1
1:2	2:2	3:2	4:2	5:2
1:3	2:3	3:3	4:3	5:3
1:4	2:4	3:4	4:4	5:4
1:5	2:5	3:5	4:5	5:5

Open a sandbox for the task.

To solution

Searching: getElement* and querySelector*

DOM navigation properties are great when elements are close to each other. What if they are not? How to get an arbitrary element of the page?

There are additional searching methods for that.

document.getElementById or just id

If an element has the `id` attribute, then there's a global variable by the name from that `id`.

We can use it to access the element, like this:

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

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

  // for elem-content things are a bit more complex
  // that has a dash inside, so it can't be a variable name
  alert(window['elem-content']); // ...but accessible using square brackets [...]
</script>
```

That's unless we declare the same-named variable by our own:

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

<script>
  let elem = 5;

  alert(elem); // the variable overrides the element
</script>
```

The 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.

Good for very simple scripts, but there may be name conflicts. Also, when we look in JS and don't have HTML in view, it's not obvious where the variable comes from.

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

For instance:

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

<script>
  let elem = document.getElementById('elem');

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

Here in the tutorial we'll often use `id` to directly reference an element, but that's only to keep things short. In real life `document.getElementById` is the preferred method.

There can be only one

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 corresponding methods is unpredictable. The browser may return any of them at random. So please stick to the rule and keep `id` unique.

Only `document.getElementById`, not `anyNode.getElementById`

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

getElementsBy*

There are also other methods to look for nodes:

- `elem.getElementsByTagName(tag)` looks for elements with the given tag and returns the collection of them. The `tag` parameter can also be a star `"*"` for "any tags".

For instance:

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

This method is callable in the context of any DOM element.

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;
```

There are also other rarely used methods of this kind:

- `elem.getElementsByClassName(className)` returns elements that have the given CSS class. Elements may have other classes too.
- `document.getElementsByName(name)` returns elements with the given `name` attribute, document-wide. Exists for historical reasons, very rarely used, we mention it here only for completeness.

For instance:

```
<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>
```

querySelectorAll

Now goes the heavy artillery.

The call to `elem.querySelectorAll(css)` returns all elements inside `elem` matching the given CSS selector. That’s the most often used and powerful method.

Here we look for all `<li>` elements that are last children:

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

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

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

Can use pseudo-classes as well

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

querySelector

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

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

matches

Previous methods were searching the DOM.

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

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

For instance:

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

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

closest

All elements that are directly above the given one are called its *ancestors*.

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

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

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

For instance:

```
<h1>Contents</h1>

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

<script>
  let chapter = document.querySelector('.chapter'); // LI
  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV

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

Live collections

All methods `"getElementsBy*"` return a *live* collection. Such collections always reflect the current state of the document and “auto-update” when it changes.

In the example below, there are two scripts.

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

Here we used separate scripts to illustrate how the element addition affects the collection, but any DOM manipulations affect them. Soon we'll see more of them.

Summary

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

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

Please note that methods `getElementById` and `getElementsByName` can only be called in the context of the document: `document.getElementById(...)`. But not on an element: `elem.getElementById(...)` would cause an error.

Other methods can be called on elements too. For instance `elem.querySelectorAll(...)` will search inside `elem` (in the DOM subtree).

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:

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

✓ Tasks

Search for elements

importance: 4

Here’s the document with the table and form.

How to find?

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

Open the page table.html in a separate window and make use of browser tools for that.

To solution

Node properties: type, tag and contents

Let’s get a more in-depth look at DOM nodes.

In this chapter we’ll see more into what they are and their most used properties.

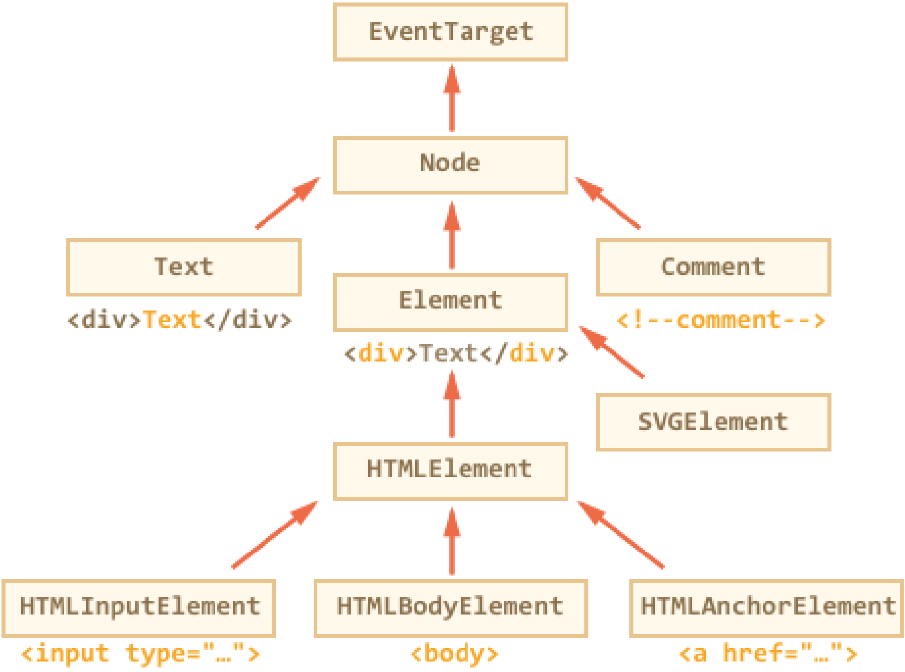
DOM node classes

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

- …and so on, each tag has its own class that may provide specific properties and methods.

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

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:

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

ℹ **`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, classes are described not 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 these properties are strings
  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();
  ...
}
```

Other classes are somewhat similar.

The “nodeType” property

The `nodeType` property provides an old-fashioned way to get the “type” of a DOM node.

It has a numeric value:

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

For instance:

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

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

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

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

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

Tag: nodeName and tagName

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

For instance:

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

Is there any difference between tagName and nodeName?

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

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

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

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

```
<body><!-- comment -->

  <script>
    // for comment
    alert( document.body.firstChild.tagName ); // undefined (no element)
    alert( document.body.firstChild.nodeName ); // #comment

    // for document
    alert( document.tagName ); // undefined (not element)
    alert( document.nodeName ); // #document
  </script>
</body>
```

If we only deal with elements, then `tagName` is the only thing we should use.

The tag name is always uppercase except XHTML

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 contents

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

We can also modify it. So it's one of 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 doesn't execute.

It becomes a part of HTML, just as a script that has already run.

Beware: "innerHTML+=" does a full overwrite

We can append "more HTML" by using `elem.innerHTML+="something"`.

Like this:

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

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

Technically, these two lines do the same:

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

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

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

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

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

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

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

outerHTML: full HTML of the element

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

Here’s an example:

```
<div id="elem">Hello <b>World</b></div>

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

Beware: unlike `innerHTML`, writing to `outerHTML` does not change the element. Instead, it replaces it as a whole in the outer context.

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! The div is still the same!
  alert(div.outerHTML); // <div>Hello, world!</div>
</script>
```

In the line `(*)` we take the full HTML of `<div>...</div>` and replace it by `<p>...</p>`. In the outer document we can see the new content instead of the `<div>`. But the old `div` variable is still the same.

The `outerHTML` assignment does not modify the DOM element, but extracts it from the outer context and inserts a new piece of HTML instead of it.

Novice developers sometimes make an error here: they modify `div.outerHTML` and then continue to work with `div` as if it had the new content in it.

That’s possible with `innerHTML`, but not with `outerHTML`.

We can write to `outerHTML`, but should keep in mind that it doesn’t change the element we’re writing to. It creates the new content on its place instead. We can get a reference to new elements by querying DOM.

nodeValue/data: text node content

The `innerHTML` property is only valid for element nodes.

Other node types 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.

We can read it, like this:

```
<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? Usually, they are not interesting at all, but sometimes developers embed information into HTML in them, like this:

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

…Then JavaScript can read it and process embedded instructions.

textContent: pure text

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

For instance:

```
<div id="news">
  <h1>Headline!</h1>
  <p>Martians attack people!</p>
</div>

<script>
  // Headline! Martians attack people!
  alert(news.textContent);
</script>
```

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

In practice, reading such text is rarely needed.

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

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

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

Compare the two:

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

<script>
```

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

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

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

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

The “hidden” property

The “hidden” attribute and the DOM property specifies whether the element is visible or not.

We can use it in HTML or assign using JavaScript, like this:

```
<div>Both divs below are hidden</div>

<div hidden>With the attribute "hidden"</div>

<div id="elem">JavaScript assigned the property "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

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

Here’s a blinking element:

```
<div id="elem">A blinking element</div>

<script>
  setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

More properties

DOM elements also have additional properties, many of them provided by 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.

Summary

Each 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 get `nodeType` from the DOM object class, but often we need just to see if it is a text or element node. The `nodeType` property is good for that. It has numeric values, most important are: `1` – for elements, `3` – for text nodes. 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, basically 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.

But HTML attributes and DOM properties are not always the same, as we'll see in the next chapter.

✓ Tasks

Count descendants

importance: 5

There's a tree structured as nested `ul/li`.

Write the code that for each `<li>` shows:

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

Demo in new window

Open a sandbox for the task.

To solution

What's in the nodeType?

importance: 5

What does the script show?

```
<html>

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

</html>
```

To solution

Tag in comment

importance: 3

What does this code show?

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

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

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

To solution

Where's the "document" in the hierarchy?

importance: 4

Which class does the `document` belong to?

What’s its place in the DOM hierarchy?

Does it inherit from `Node` or `Element`, or maybe `HTMLElement`?

To solution

Attributes and properties

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

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

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

DOM properties

We’ve already seen built-in DOM properties. There’s a lot. But technically no one limits us, and if it’s not 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 documer
```

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

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

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

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

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

HTML attributes

In HTML, tags may have attributes. When the browser parses the HTML to create DOM objects for tags, it recognizes *standard* attributes and creates DOM properties from them.

So when an element has `id` or another *standard* attribute, the corresponding property gets created. But that doesn't happen if the attribute is non-standard.

For instance:

```
<body id="test" something="non-standard">
  <script>
    alert(document.body.id); // test
    // non-standard attribute does not yield a property
    alert(document.body.something); // undefined
  </script>
</body>
```

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

Here we can see it:

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

    for (let attr of elem.attributes) { // (4) list all
      alert( `${attr.name} = ${attr.value}` );
    }
  </script>
</body>
```

Please note:

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

Property-attribute synchronization

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

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

```
<input>

<script>
  let input = document.querySelector('input');

  // attribute => property
  input.setAttribute('id', 'id');
  alert(input.id); // id (updated)

  // property => attribute
  input.id = 'newId';
  alert(input.getAttribute('id')); // newId (updated)
</script>
```

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

```
<input>

<script>
  let input = document.querySelector('input');

  // attribute => property
  input.setAttribute('value', 'text');
  alert(input.value); // text

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

In the example above:

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

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

DOM properties are typed

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

```
<input id="input" type="checkbox" checked> checkbox

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

There are other examples. The `style` attribute is a string, but the `style` property is an object:

```
<div id="div" style="color:red;font-size:120%">Hello</div>

<script>
  // string
  alert(div.getAttribute('style')); // color:red;font-size:120%

  // object
  alert(div.style); // [object CSSStyleDeclaration]
  alert(div.style.color); // red
</script>
```

That’s an important difference. But even if a DOM property type is a string, it may differ from the attribute!

For instance, the `href` DOM property is always a *full* URL, even if the attribute contains a relative URL or just a `#hash`.

Here’s an example:

```
<a id="a" href="#hello">link</a>
<script>
  // attribute
  alert(a.getAttribute('href')); // #hello

  // property
  alert(a.href ); // full URL in the form http://site.com/page#hello
</script>
```

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

Non-standard attributes, dataset

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

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

Like this:

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

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

  for(let div of document.querySelectorAll('[show-info]')) {
    // insert the corresponding info into the field
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // Pete, then 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 the attribute may be preferable to classes like `.order-state-new`, `.order-state-pending`, `order-state-canceled`?

That's 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, 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 needs, DOM properties can serve us well. 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.

Tasks

Get the attribute

importance: 5

Write the code to select the element with `data-widget-name` attribute from the document and to read its value.

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

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

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

To solution

Make external links orange

importance: 3

Make all external links orange by altering their `style` property.

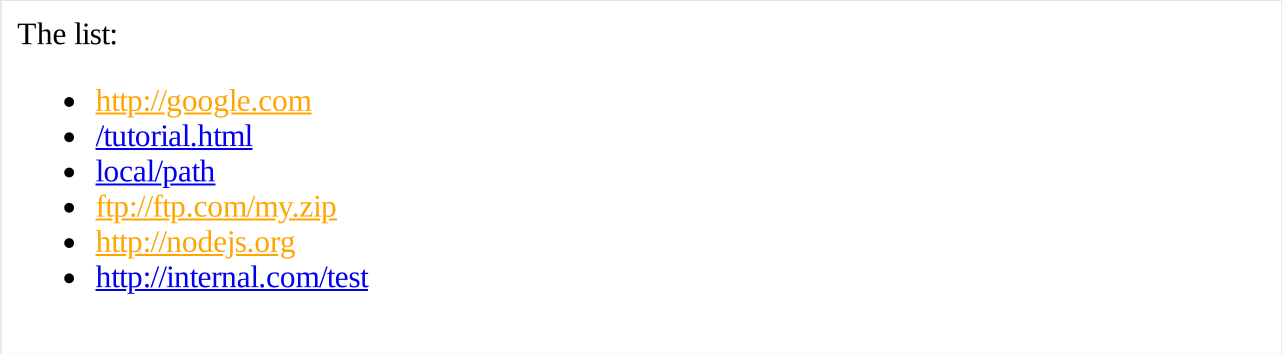
A link is external if:

- It’s `href` has `://` in it
- But doesn’t start with `http://internal.com`.

Example:

```html
<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:



The list:

- http://google.com
- /tutorial.html
- local/path
- ftp://ftp.com/my.zip
- http://nodejs.org
- http://internal.com/test

Open a sandbox for the task.

To solution

Modifying the document

DOM modifications is the key to create “live” pages.

Here we’ll see how to create new elements “on the fly” and modify the existing page content.

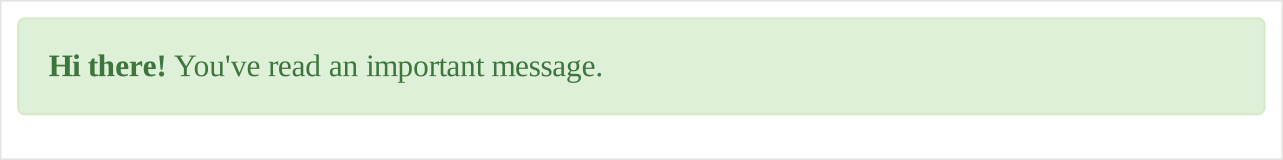
First we’ll see a simple example and then explain the methods.

Example: show a message

For a start, let’s see how to 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>
```



Hi there! You've read an important message.

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

Creating an element

To create DOM nodes, there are two methods:

`document.createElement(tag)`

Creates a new element with the given tag:

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

`document.createTextNode(text)`

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

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

Creating the message

In our case we want to make a `div` with given classes and the message in it:

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

After that, we have our DOM element ready. Right now it is just in a variable and we cannot see it. That is because it's not yet inserted into the page.

Insertion methods

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

There's a special method for that: `document.body.appendChild(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 alert-success";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.appendChild(div);
</script>
```

Here's a brief list of methods to insert a node into a parent element (`parentElem` for short):

`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`.

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

These methods are “old school”: they exist from the ancient times and we can meet them in many old scripts. Unfortunately, there are some tasks that are hard to solve with them.

For instance, how to insert *html* if we have it as a string? Or, given a node, how to insert another node *before* it? Of course, all that is doable, but not in an elegant way.

So there exist two other sets of insertion methods to handle all cases easily.

prepend/append/before/after

This set of methods provides more flexible insertions:

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

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

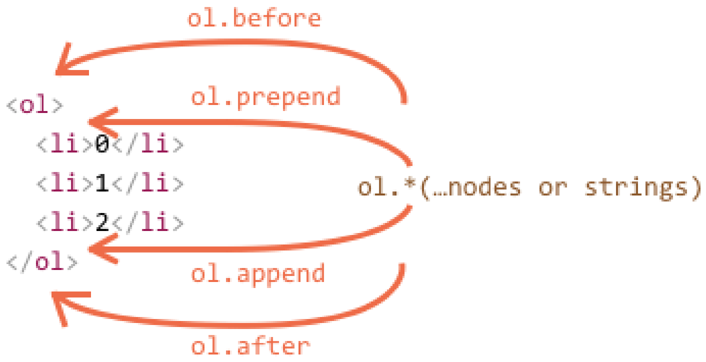
```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before');
  ol.after('after');

  let prepend = document.createElement('li');
  prepend.innerHTML = 'prepend';
  ol.prepend(prepend);

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

before

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

after

Here’s a small picture what methods do:



So the final list will be:

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

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

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

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

All text is inserted *as text*.

So the final HTML is:

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

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

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

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

insertAdjacentHTML/Text/Element

There’s another, pretty versatile method: `elem.insertAdjacentHTML(where, html)`.

The first parameter is a string, specifying where to insert. Must be one of the following:

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

The second parameter is an HTML string, inserted “as is”.

For instance:

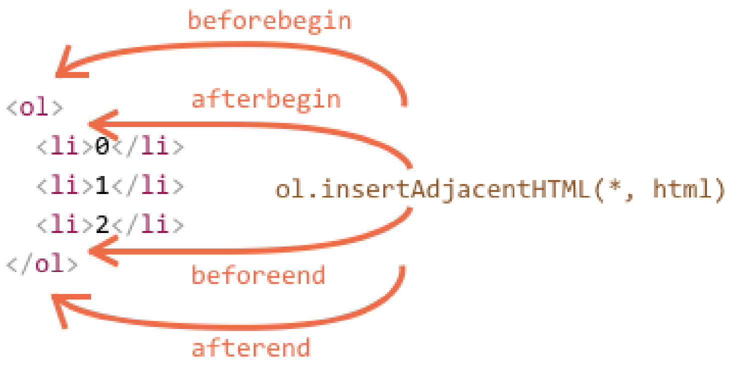
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
  div.insertAdjacentHTML('afterend', '<p>Bye</p>');
</script>
```

…Would lead to:

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

That’s how we can append an arbitrary HTML to our 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 alert-success
    <strong>Hi there!</strong> You've read an important message.
  </div>`);
</script>
```

Cloning nodes: cloneNode

How to insert one more similar message?

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

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

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

An example of copying the message:

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

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

<script>
  let div2 = div.cloneNode(true); // clone the message
  div2.querySelector('strong').innerHTML = 'Bye there!'; // change the clone

  div.after(div2); // show the clone after the existing div
</script>
```

Removal methods

To remove nodes, there are the following methods:

`parentElem.removeChild(node)`

Removes `elem` from `parentElem` (assuming it’s a child).

`node.remove()`

Removes the `node` from its place.

We can easily see that the second method is much shorter. The first one exists for historical reasons.

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>
```

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 alert-success";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.append(div);
  setTimeout(() => div.remove(), 1000);
  // or setTimeout(() => document.body.removeChild(div), 1000);
</script>
```

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 was the downside.

Technically, when `document.write` is called while the browser is still reading HTML, it appends something to it, and the browser consumes it just as it were initially there.

That gives us the upside – it works blazingly fast, because there’s *no DOM modification*. It writes directly into the page text, while the DOM is not yet built, and the browser puts it into DOM at generation-time.

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 of nodes:

- From the parent:
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

  All these methods return `node`.
- Given a list of nodes and strings:
  - `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”.
- Given a piece of HTML: `elem.insertAdjacentHTML(where, html)`, inserts depending on 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`, they 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.

Tasks

createTextNode vs innerHTML vs textContent

importance: 5

We have an empty DOM element `elem` and a string `text`.

Which of these 3 commands do exactly the same?

1. `elem.append(document.createTextNode(text))`
2. `elem.innerHTML = text`
3. `elem.textContent = text`

To solution

Clear the element

importance: 5

Create 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>
```

To solution

Why does "aaa" remain?

importance: 1

Run the example. Why does `table.remove()` not delete the text `"aaa"`?

```
<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>
```

To solution

Create a list

importance: 4

Write an interface to create a list from user input.

For every list item:

1. Ask a user about its content using `prompt`.
2. Create the `<li>` with it and add it to `<ul>`.
3. Continue until the user cancels the input (by pressing `Esc` or CANCEL in prompt).

All elements should be created dynamically.

If a user types HTML-tags, they should be treated like a text.

Demo in new window

To solution

Create a tree from the object

importance: 5

Write a function `createTree` that creates a nested `ul/li` list from the nested object.

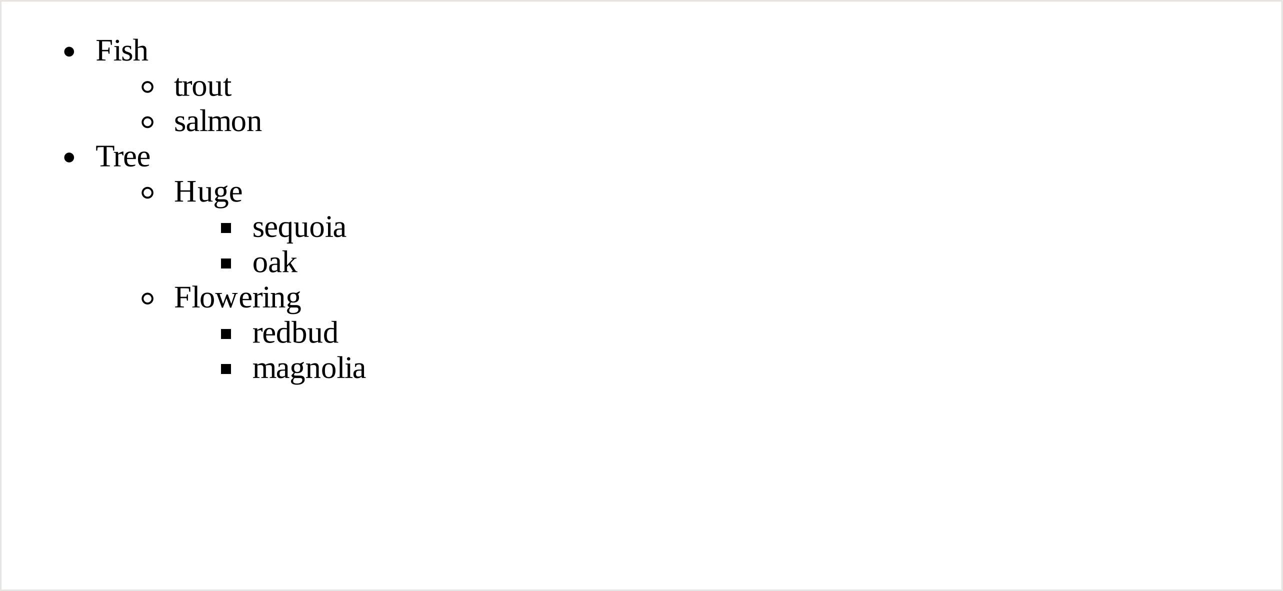
For instance:

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

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

The syntax:

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

The result (tree) should look like this:



- Fish
  - trout
  - salmon
- Tree
  - Huge
    - sequoia
    - oak
  - Flowering
    - redbud
    - magnolia

Choose one of two ways of solving this task:

1. Create the HTML for the tree and then assign to `container.innerHTML`.
2. 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 `<ul></ul>` for the leaves.

Open a sandbox for the task.

To solution

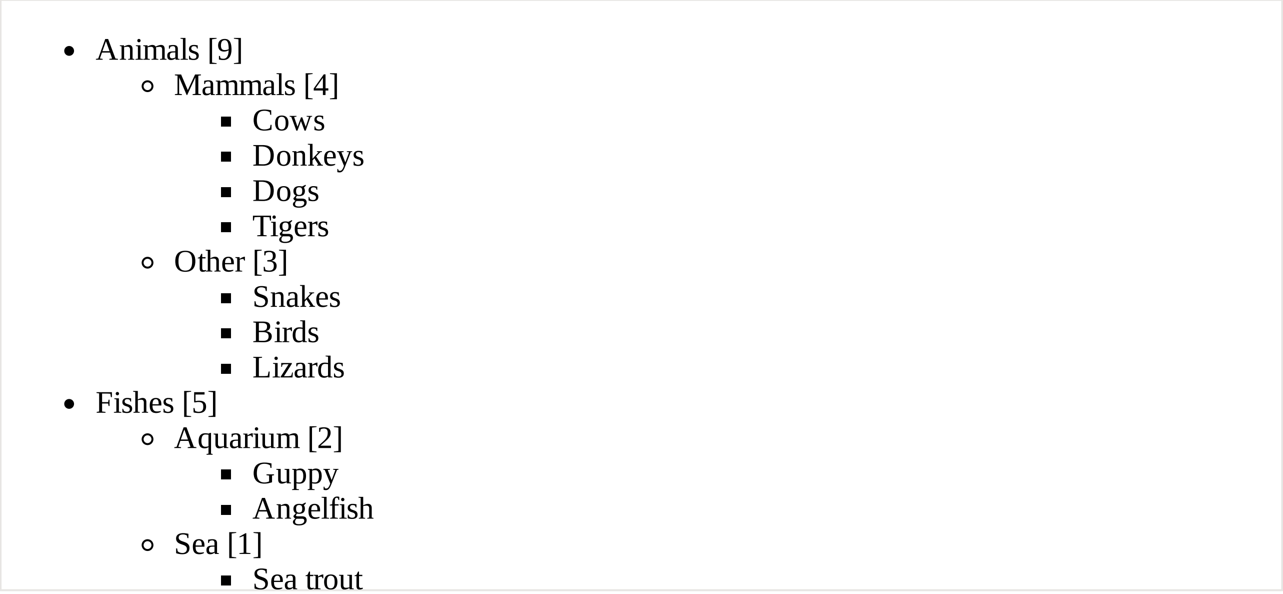
Show descendants in a tree

importance: 5

There’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:



- Animals [9]
    - Mammals [4]
        - Cows
        - Donkeys
        - Dogs
        - Tigers
    - Other [3]
        - Snakes
        - Birds
        - Lizards
- Fishes [5]
    - Aquarium [2]
        - Guppy
        - Angelfish
    - Sea [1]
        - Sea trout

Open a sandbox for the task.

To solution

Create a calendar

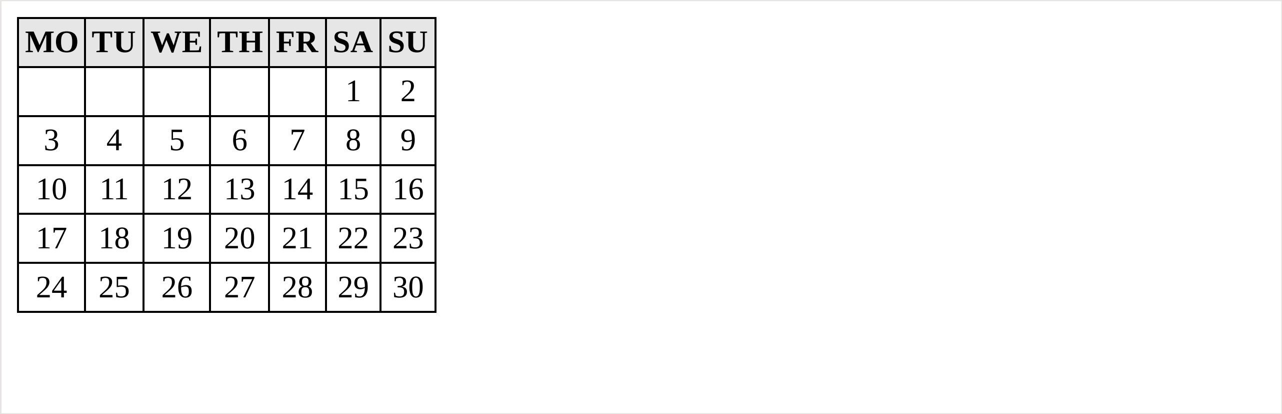
importance: 4

Write a function `createCalendar(elem, year, month)`.

The call should create a calendar for the given year/month and put it inside `elem`.

The calendar should be a table, where a week is `<tr>`, and a day is `<td>`. The table top should be `<th>` with weekday names: 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:



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

P.S. For this task it's enough to generate the calendar, should not yet be clickable.

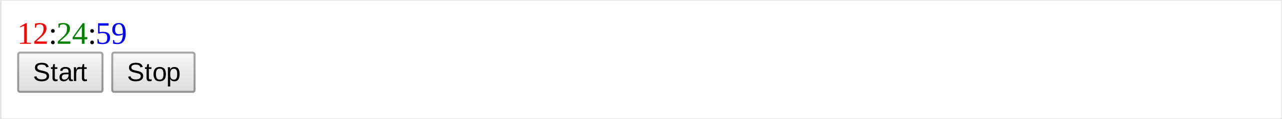
Open a sandbox for the task.

To solution

Colored clock with setInterval

importance: 4

Create a colored clock like here:



12:24:59

Use HTML/CSS for the styling, Javascript only updates time in elements.

Open a sandbox for the task.

To solution

Insert the HTML in the list

importance: 5

Write the code to insert `<li>2</li><li>3</li>` between two `<li>` here:

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

To solution

Sort the table

importance: 5

There's a table:

Name	Surname	Age
John	Smith	10
Pete	Brown	15
Ann	Lee	5
...	...	...

There may be more rows in it.

Write the code to sort it by the `"name"` column.

Open a sandbox for the task.

To solution

Styles and classes

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

There are generally two ways to style an element:

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

CSS is always the preferred way – not only for HTML, but in JavaScript as well.

We should only manipulate the `style` property if classes "can't handle it".

For instance, `style` is acceptable if we calculate coordinates of an element dynamically and want to set them from JavaScript, like this:

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

For other cases, like making the text red, adding a background icon – describe that in CSS and then apply the class. That’s more flexible and easier to support.

className and classList

Changing 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 strings 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` classes.

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")` – if the class exists, then removes it, otherwise adds it.
- `elem.classList.contains("class")` – returns `true/false`, checks for the given class.

Besides that, `classList` is iterable, so we can list all classes like this:

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

Element style

The property `elem.style` is an object that corresponds to what's written in the `"style"` attribute. Setting `elem.style.width="100px"` works as if we had in the attribute `style="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, for instance:

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

That is: a dash `"-"` becomes an uppercase.

Resetting the style property

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

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

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

```
// if we run this code, the <body> "blinks"
document.body.style.display = "none"; // hide

setTimeout(() => document.body.style.display = "", 1000); // back to normal
```

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

Full rewrite with `style.cssText`

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

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

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

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

  alert(div.style.cssText);
</script>
```

We rarely use it, because such assignment removes all existing styles: it does not add, but replaces them. May occasionally delete something needed. But still can be done for new elements when we know we won't delete something important.

The same can be accomplished by setting an attribute: `div.setAttribute('style', 'color: red...')`.

Mind the units

CSS units must be provided in style values.

For instance, we should not set `elem.style.top` to `10`, but rather to `10px`. 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 igr

    // 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 how the browser “unpacks” the property `style.margin` in the last lines and infers `style.marginLeft` and `style.marginTop` (and other partial margins) from it.

Computed styles: getComputedStyle

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 want the current value for the start.

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 style properties, like `elem.style`, but now with respect to all CSS classes.

For instance:

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

  <script>
    let computedStyle = getComputedStyle(document.body);

    // now we can read the margin and the color from it

    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>

</body>
```

ⓘ Computed and resolved values

There are two concepts in CSS :

1. A *computed* style value is the value after all CSS rules and CSS inheritance is applied, as the result of the CSS cascade. It can look like `height:1em` or `font-size:125%`.
2. A *resolved* style value is the one finally applied to the element. Values like `1em` or `125%` are relative. The browser takes the computed value and makes all units fixed and absolute, for instance: `height:20px` or `font-size:16px`. For geometry properties resolved values may have a floating point, like `width:50.5px`.

A long time ago `getComputedStyle` was created to get computed values, but it turned out that resolved values are much more convenient, and the standard changed.

So nowadays `getComputedStyle` actually returns the resolved value of the property.

⚠ `getComputedStyle` requires the full property name

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

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

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

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

“Visited” links styles 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 we may not see the styles applied by `:visited`. And also, there’s a limitation in CSS that forbids to apply geometry-changing styles in `:visited`. That’s to guarantee that there’s no sideway for an evil page to test if a link was visited and hence to break the privacy.

Summary

To 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.

✓ Tasks

Create a notification

importance: 5

Write a function `showNotification(options)` that creates a notification: `<div class="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 window ↗

Use CSS positioning to show the element at given top/right coordinates. The source document has the necessary styles.

Open a sandbox for the task. ↗

To solution

Element size and scrolling

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

We often need them when moving or positioning elements in JavaScript, to correctly calculate coordinates.

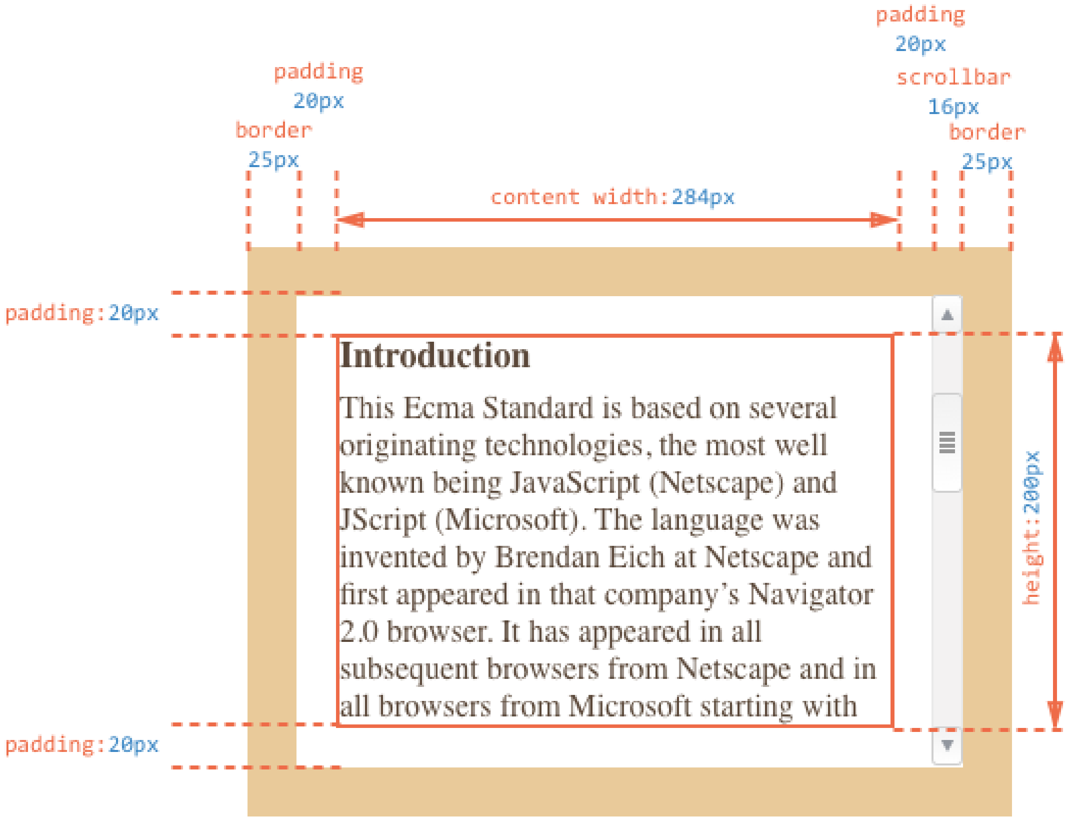
Sample element

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

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

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

The element looks like this:



Introduction

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

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.

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. If there's no scrollbar, then things are just a bit simpler.

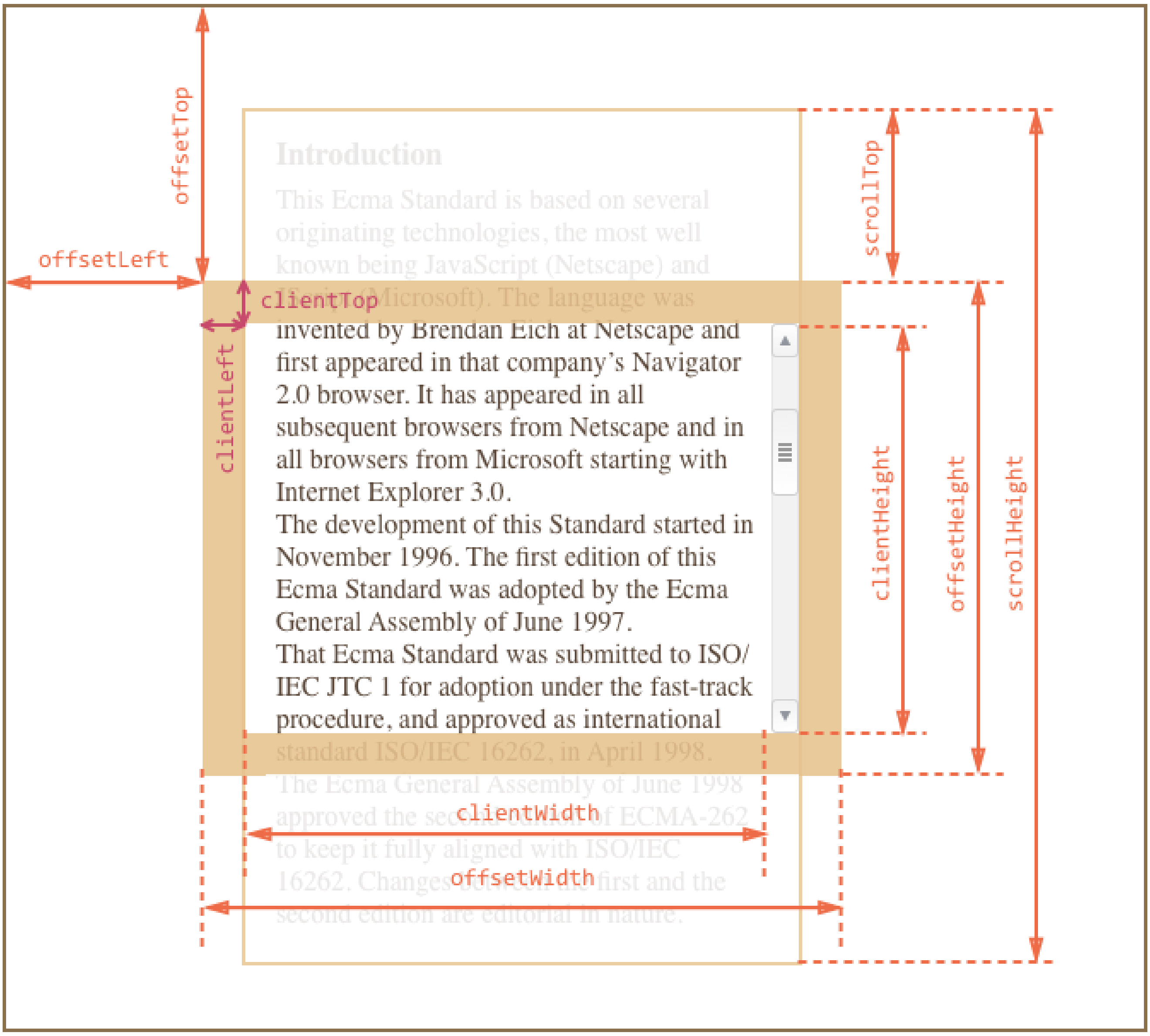
The `padding-bottom` may be filled with text

Usually paddings are shown empty on illustrations, but if there’s a lot of text in the element and it overflows, then browsers show the “overflowing” text at `padding-bottom`, so you can see that in examples. But the padding is still there, unless specified otherwise.

Geometry

Element properties that provide width, height and other geometry are always numbers. They are assumed to be in pixels.

Here’s the overall picture:


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

They are many properties, it’s difficult to fit them all in the single picture, but their values are simple and easy to understand.

Let’s start exploring them from the outside of the element.

offsetParent, offsetLeft/Top

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

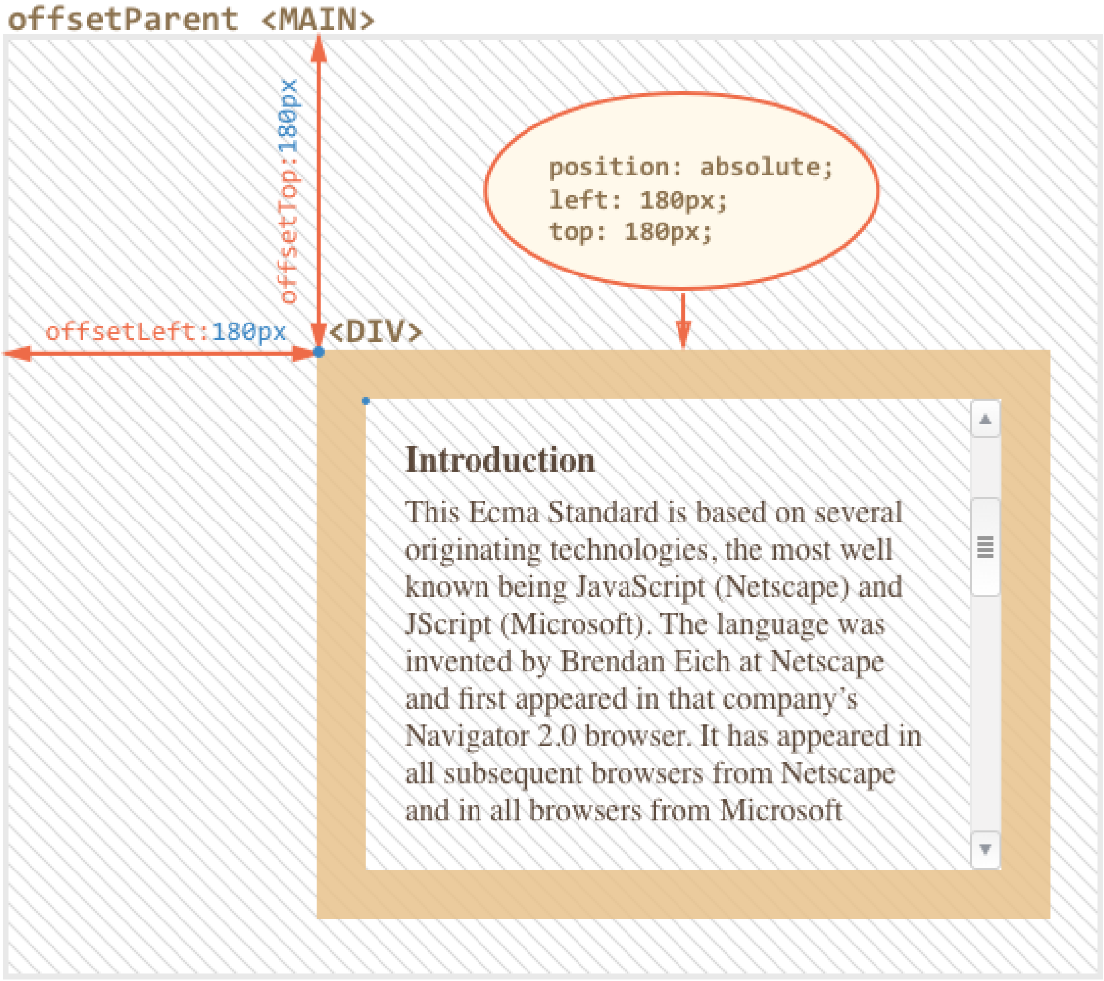
The `offsetParent` is the nearest ancestor that is:

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

In most practical cases we can use `offsetParent` to get the nearest CSS-positioned ancestor. And `offsetLeft/offsetTop` provide x/y coordinates relative to it’s upper-left corner.

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

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


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

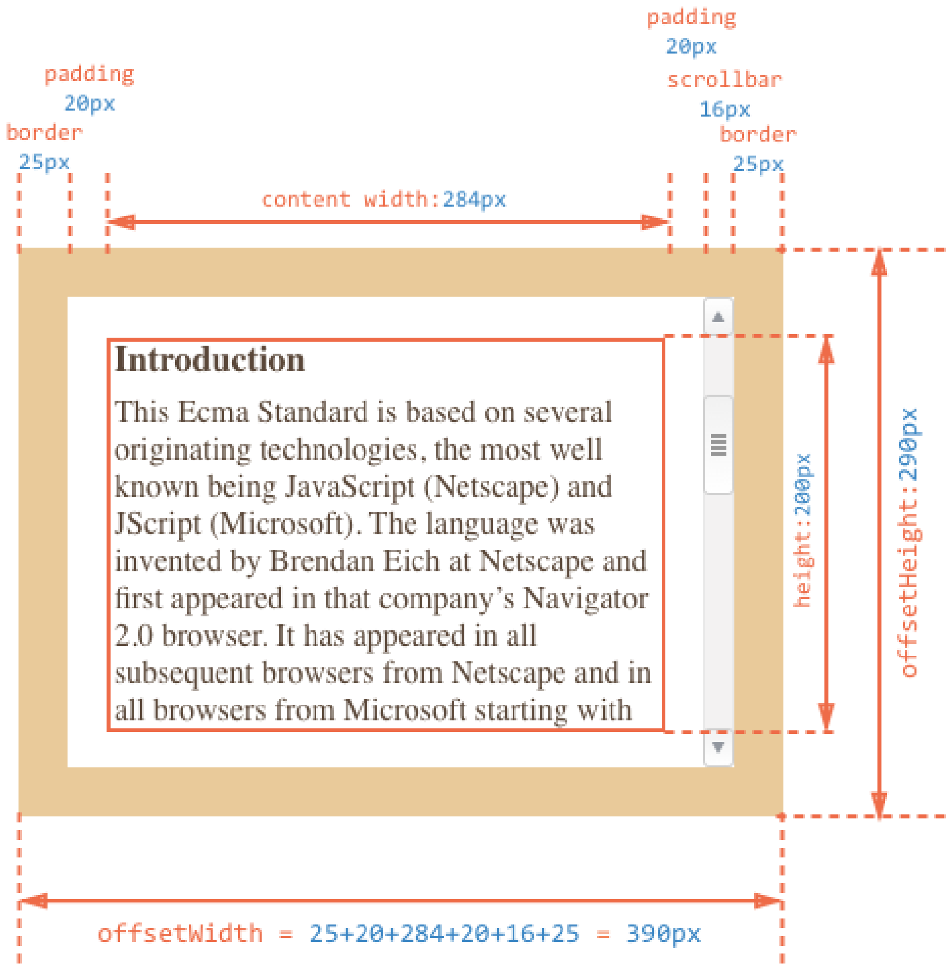
There are several occasions when `offsetParent` is `null`:

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

offsetWidth/Height

Now let’s move on to the element itself.

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



Introduction

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

For our sample element:

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

Geometry properties for not shown elements are zero/null

Geometry properties are calculated only for shown 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` depending on what it is.

For example, `offsetParent` is `null`, and `offsetWidth`, `offsetHeight` are `0`.

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

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

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

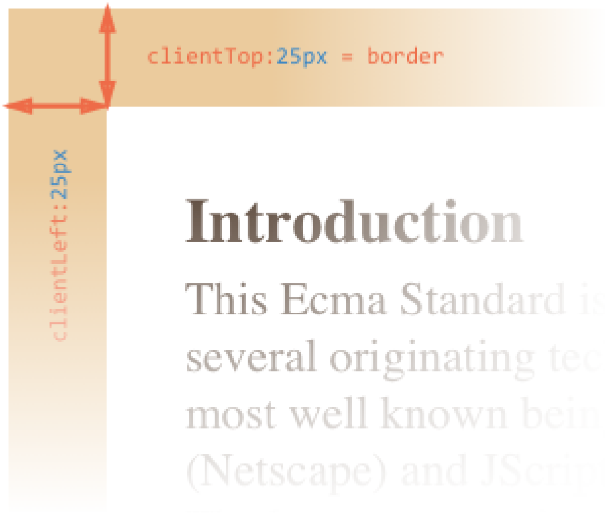
clientTop/Left

Inside the element we have the borders.

To measure them, there are properties `clientTop` and `clientLeft`.

In our example:

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

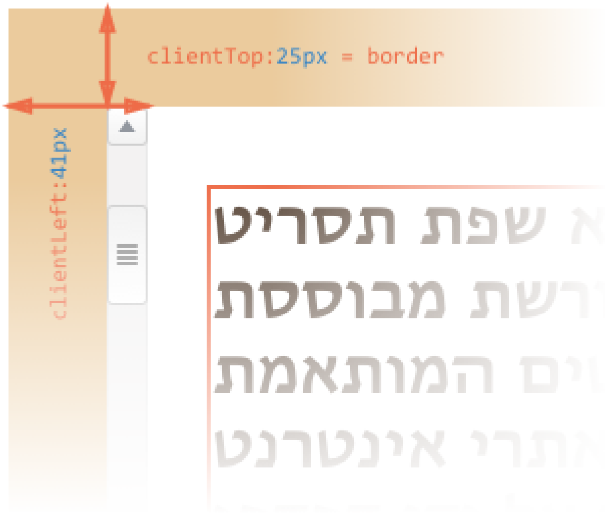


…But to be precise – they are not borders, but relative coordinates of the inner side from the outer side.

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.

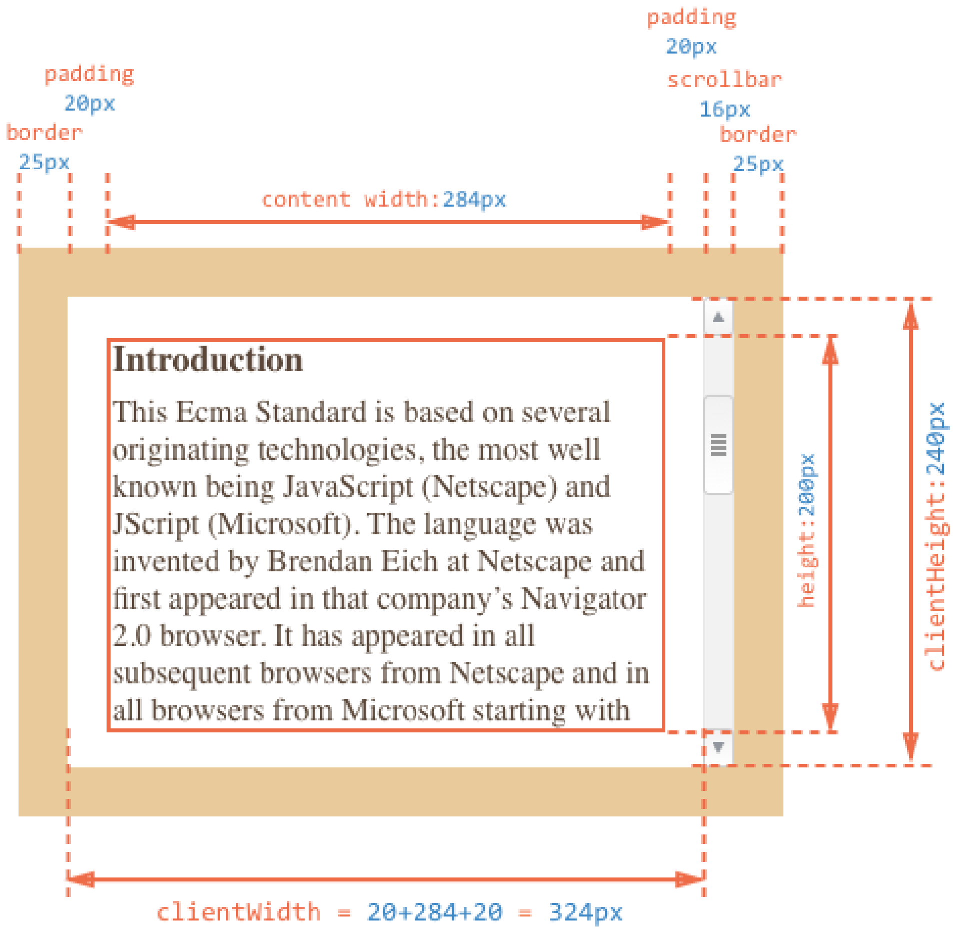
In that case, `clientLeft` would be not `25`, but with the scrollbar width `25 + 16 = 41`:



clientWidth/Height

These properties provide the size of the area inside the element borders.

They include the content width together with paddings, but without the scrollbar:



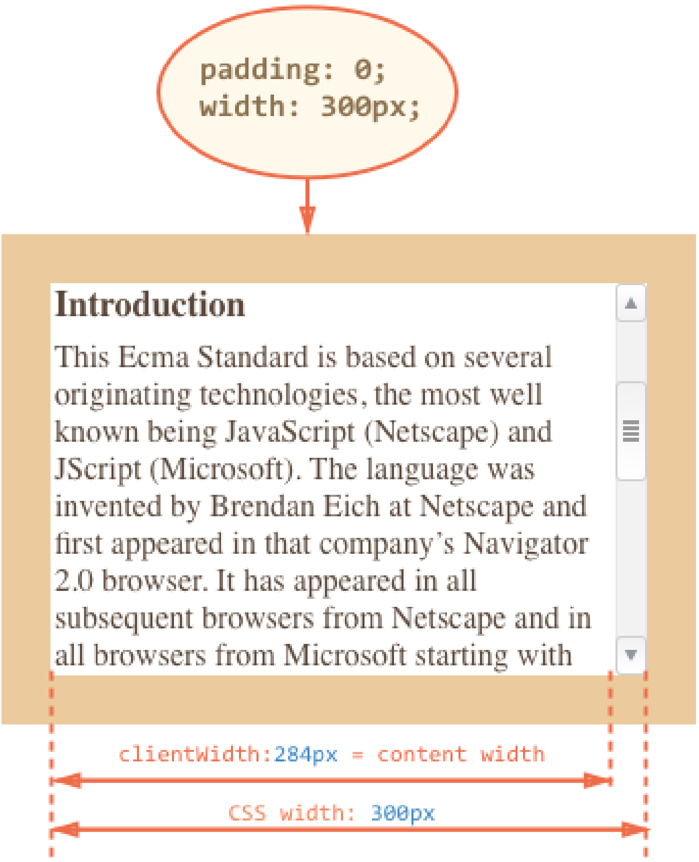
Introduction

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

On the picture above let's first consider `clientHeight` : it's easier to evaluate. 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).



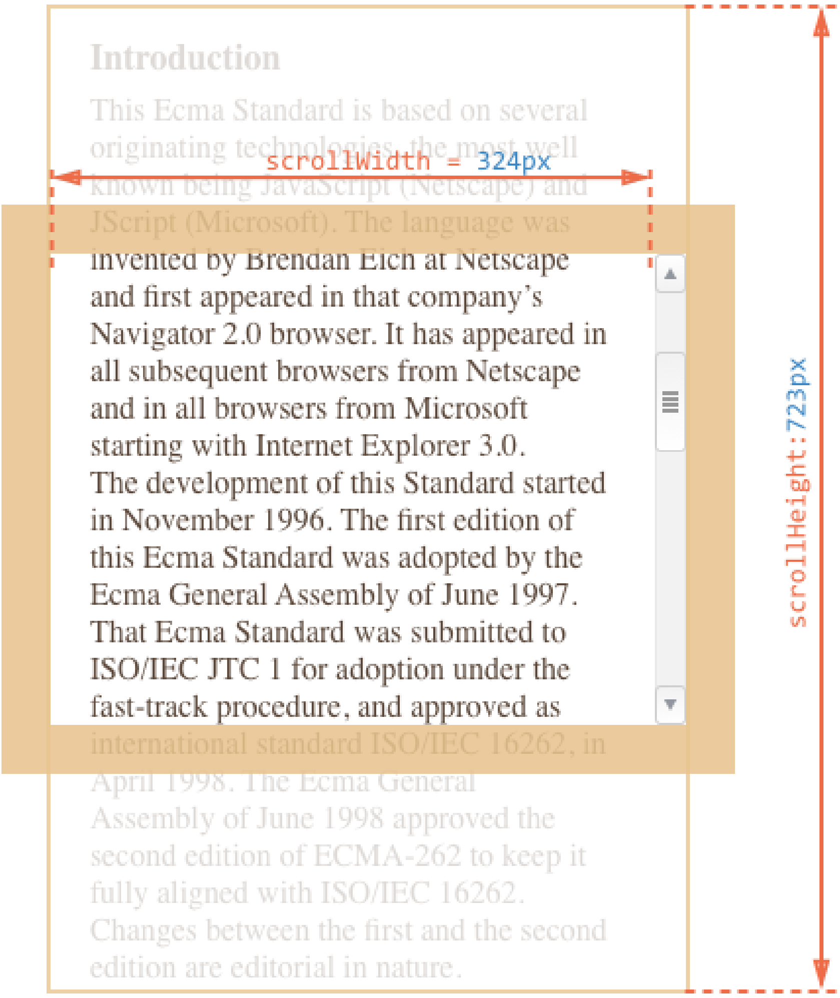
Introduction

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

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

scrollWidth/Height

- Properties `clientWidth/clientHeight` only account for the visible part of the element.
- Properties `scrollWidth/scrollHeight` also include the scrolled out (hidden) parts:


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

On the picture above:

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

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

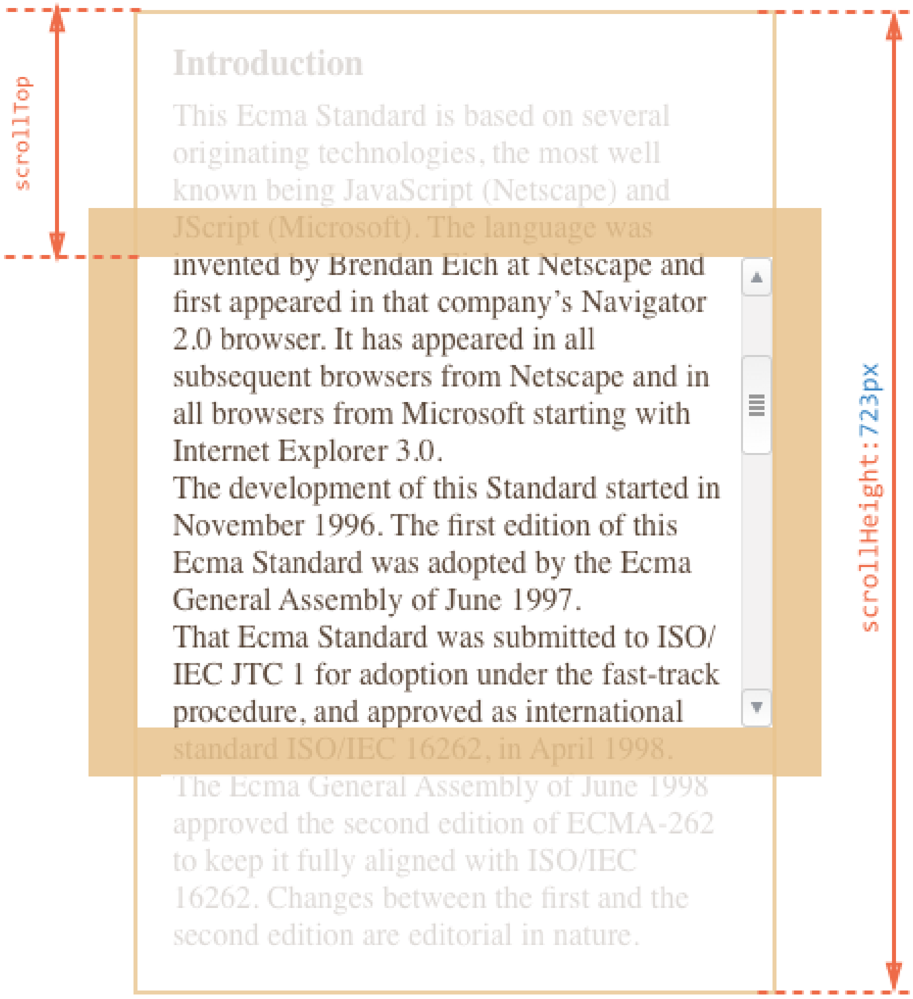
Like this:

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

scrollLeft/scrollTop

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

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



Introduction

This Ecma Standard is based on several originating technologies, the most well known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0.

The development of this Standard started in November 1996. The first edition of this Ecma Standard was adopted by the Ecma General Assembly of June 1997.

That Ecma Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as international standard ISO/IEC 16262, in April 1998. The Ecma General Assembly of June 1998 approved the second edition of ECMA-262 to keep it fully aligned with ISO/IEC 16262. Changes between the first and the second edition are editorial in nature.

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

`scrollLeft/scrollTop` can be modified

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

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

Don't take width/height from CSS

We've just covered geometry properties of DOM elements. They are normally 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 like this?

```
let elem = document.body;

alert( getComputedStyle(elem).width ); // show CSS width for elem
```

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

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

   ```
   <span id="elem">Hello!</span>

   <script>
     alert( getComputedStyle(elem).width ); // auto
   </script>
   ```

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

And there’s one more reason: a scrollbar. Sometimes the code that works fine without a scrollbar starts to bug 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.

Please note that the described difference is only about reading `getComputedStyle(...).width` from JavaScript, visually everything is correct.

Summary

Elements 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 distance from the upper-left outer corner to its upper-left inner 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 including the scrolled out parts. Also includes paddings, but not the scrollbar.
- `scrollLeft/scrollTop` – width/height of the scrolled out part of the element, starting from its upper-left corner.

All properties are read-only except `scrollLeft/scrollTop`. They make the browser scroll the element if changed.

Tasks

What's the scroll from the bottom?

importance: 5

The `elem.scrollTop` property is the size of the scrolled out part from the top. How to get “`scrollBottom`” – the size from the bottom?

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`.

To solution

What is the scrollbar width?

importance: 3

Write the code that returns the width of a standard scrollbar.

For Windows it usually varies between `12px` and `20px`. If the browser doesn't reserves any space for it, then it may be `0px`.

P.S. The code should work for any HTML document, do not depend on its content.

To solution

Place the ball in the field center

importance: 5

Here'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 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.

To solution

The difference: CSS width versus clientWidth

importance: 5

What’s the difference between `getComputedStyle(elem).width` and `elem.clientWidth`?

Give at least 3 differences. The more the better.

To solution

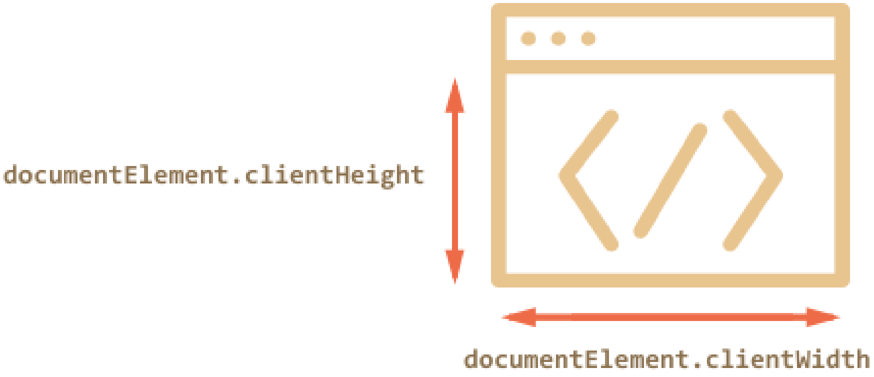
Window sizes and scrolling

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

From the DOM point of view, the root document element is `document.documentElement`. That element corresponds to `<html>` and has geometry properties described in the previous chapter. For some cases we can use it, but there are additional methods and peculiarities important enough to consider.

Width/height of the window

Properties `clientWidth/clientHeight` of `document.documentElement` is exactly what we want here:



⚠ **Not `window.innerWidth/Height`**

Browsers also support properties `window.innerWidth/innerHeight`. They look like what we want. So what’s the difference?

If there’s a scrollbar occupying some space, `clientWidth/clientHeight` provide the width/height inside it. In other words, they return width/height of the visible part of the document, available for the content.

And `window.innerWidth/innerHeight` ignore 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 scro
```

In most cases we need the *available* window width: to draw or position something. That is: inside scrollbars if there are any. So we should use `documentElement.clientHeight/Width`.

⚠ **`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`. Generally that’s not a JavaScript question, but here it affects JavaScript as well.

Width/height of the document

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

These properties work well for regular elements. But 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`! For regular elements that’s a nonsense.

To have a reliable full window size, we should take the maximum of these properties:

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

alert('Full document height, with scrolled out part: ' + scrollHeight);
```

Why so? Better don’t ask. These inconsistencies come from ancient times, not a “smart” logic.

Get the current scroll

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

What’s with the page? Most browsers provide `documentElement.scrollLeft/Top` for the document scroll, but Chrome/Safari/Opera have bugs (like 157855, 106133) and we should use `document.body` instead of `document.documentElement` there.

Luckily, we don’t have to remember these peculiarities at all, because of the special properties `window.pageXOffset/pageYOffset`:

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

These properties are read-only.

Scrolling: scrollTo, scrollBy, scrollIntoView

⚠ **Important:**

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

For instance, if we try to scroll the page from the script in `<head>`, it won’t work.

Regular elements can be scrolled by changing `scrollTop/scrollLeft`.

We can do the same for the page:

- For all browsers except Chrome/Safari/Opera: modify `document.documentElement.scrollTop/Left`.
- In Chrome/Safari/Opera: use `document.body.scrollTop/Left` instead.

It should work, but smells like cross-browser incompatibilities. Not good. Fortunately, there's a simpler, more universal solution: special methods window.scrollBy(x,y) and window.scrollTo(pageX,pageY).

- The method `scrollBy(x,y)` scrolls the page relative to its current position. For instance, `scrollBy(0,10)` scrolls the page `10px` down.
- The method `scrollTo(pageX,pageY)` scrolls the page relative to the document top-left corner. It's like setting `scrollLeft/scrollTop`.

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

These methods work for all browsers the same way.

scrollIntoView

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

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

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

Forbid the scrolling

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

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

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, and if it increased (the scrollbar disappeared) then add `padding` to `document.body` in place of the scrollbar, to keep the content width same.

Summary

Geometry:

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

Coordinates

To move elements around we should be familiar with coordinates.

Most JavaScript methods deal with one of two coordinate systems:

1. Relative to the window(or another viewport) top/left.
2. Relative to the document top/left.

It’s important to understand the difference and which type is where.

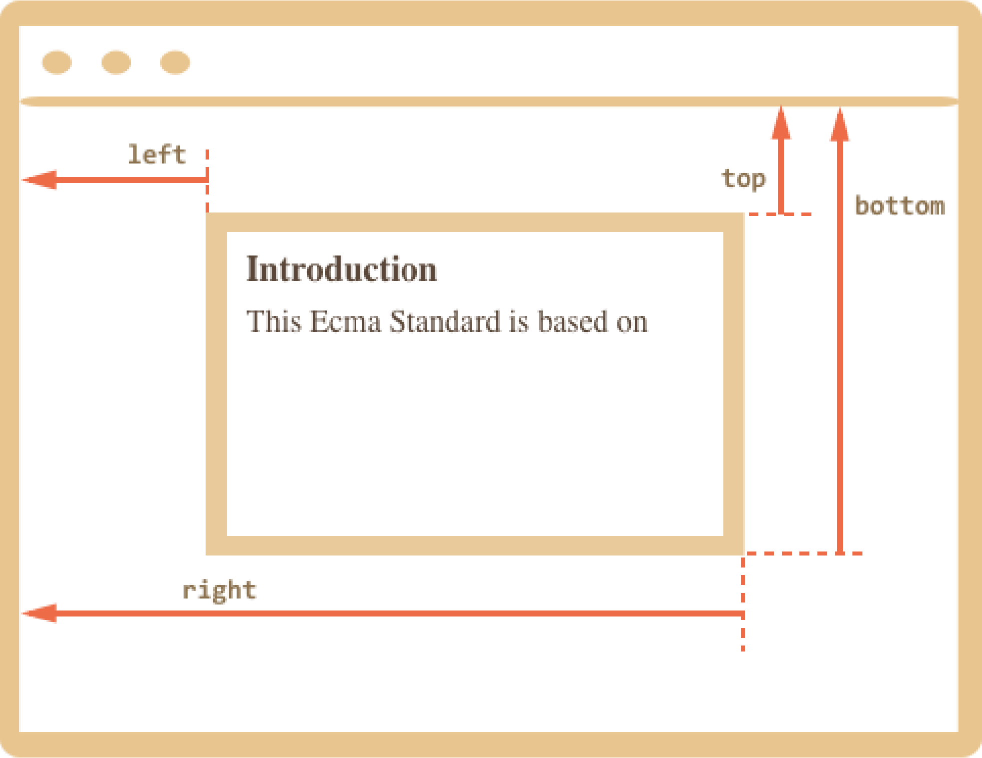
Window coordinates: getBoundingClientRect

Window coordinates start at the upper-left corner of the window.

The method `elem.getBoundingClientRect()` returns window coordinates for `elem` as an object with properties:

- `top` – Y-coordinate for the top element edge,
- `left` – X-coordinate for the left element edge,
- `right` – X-coordinate for the right element edge,
- `bottom` – Y-coordinate for the bottom element edge.

Like this:



Window coordinates do not take the scrolled out part of the document into account, they are calculated from the window's upper-left corner.

In other words, when we scroll the page, the element goes up or down, *its window coordinates change*. That's very important.

Also:

- Coordinates may be decimal fractions. That's normal, internally browser uses them for calculations. We don't have to round them when setting to `style.position.left/top`, the browser is fine with fractions.
- Coordinates may be negative. For instance, if the page is scrolled down and the top `elem` is now above the window. Then,

`elem.getBoundingClientRect().top` is negative.

- Some browsers (like Chrome) provide additional properties (`width` and `height`) to `getBoundingClientRect` as the result. We can also get them by subtraction: `height=bottom-top`, `width=right-left`.

Coordinates right/bottom are different from CSS properties

If we compare window coordinates versus CSS positioning, then there are obvious similarities to `position:fixed`. The positioning of an element is also relative to the viewport.

But in CSS, the `right` property means the distance from the right edge, and the `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 upper-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`.

In most cases such behavior is not a problem, but we should keep that in mind.

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 position:fixed

Most of time we need coordinates to position something. In CSS, to position an element relative to the viewport we use `position:fixed` together with `left/top` (or `right/bottom`).

We can use `getBoundingClientRect` to get coordinates of an element, and then to show something near it.

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

```
let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {
  // create message element
  let message = document.createElement('div');
  // better to use a css class for the style here
  message.style.cssText = "position:fixed; color: red";

  // assign coordinates, don't forget "px"!
  let coords = elem.getBoundingClientRect();

  message.style.left = coords.left + "px";
  message.style.top = coords.bottom + "px";

  message.innerHTML = html;
  return message;
}

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

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

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

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

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

Document coordinates

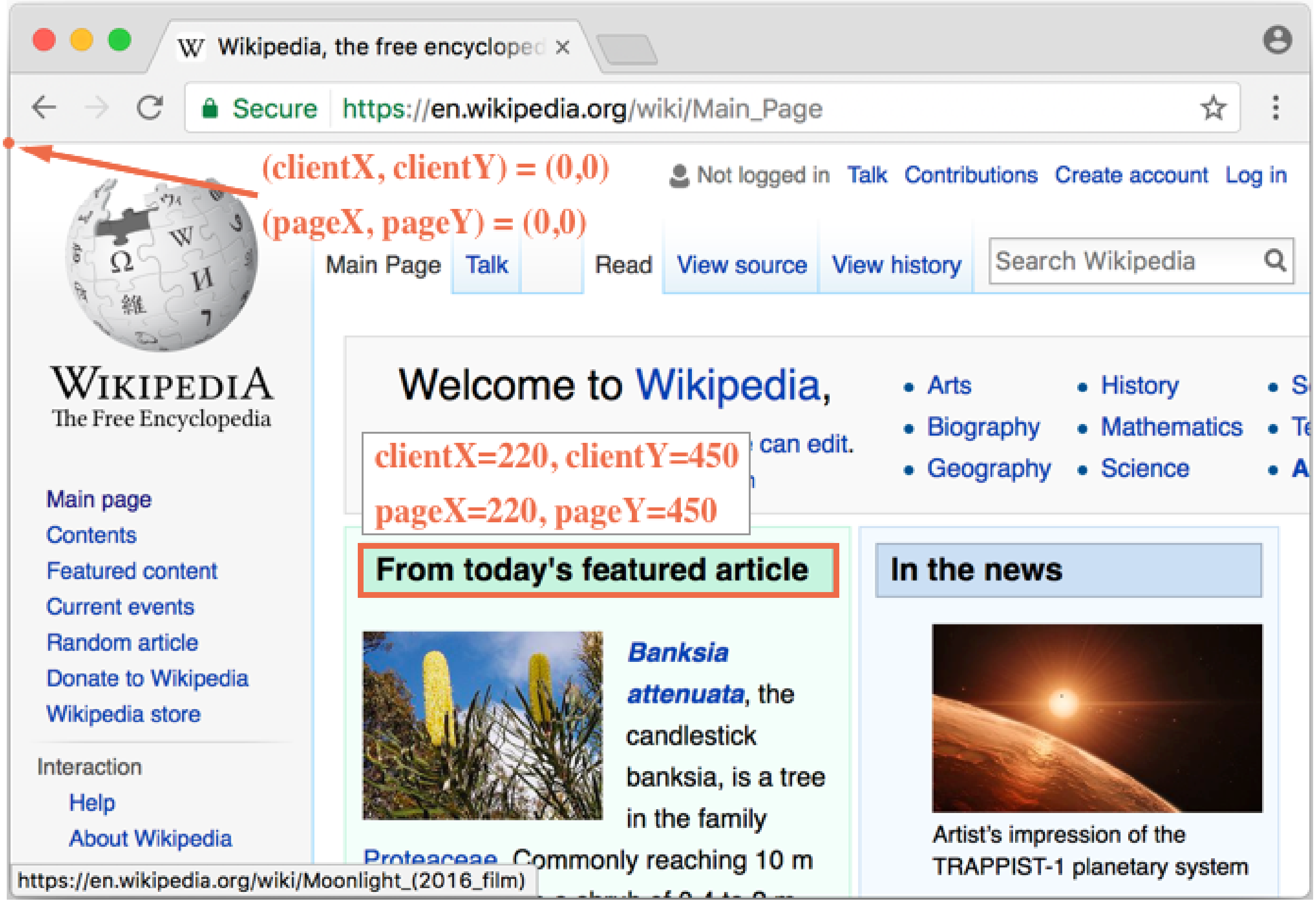
Document-relative coordinates start from the upper-left corner of the document, not the window.

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

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

For clarity we’ll call window coordinates `(clientX,clientY)` and document coordinates `(pageX,pageY)`.

When the page is not scrolled, then window coordinate and document coordinates are actually the same. Their zero points match too:



(clientX, clientY) = (0,0)

(pageX, pageY) = (0,0)

clientX=220, clientY=450

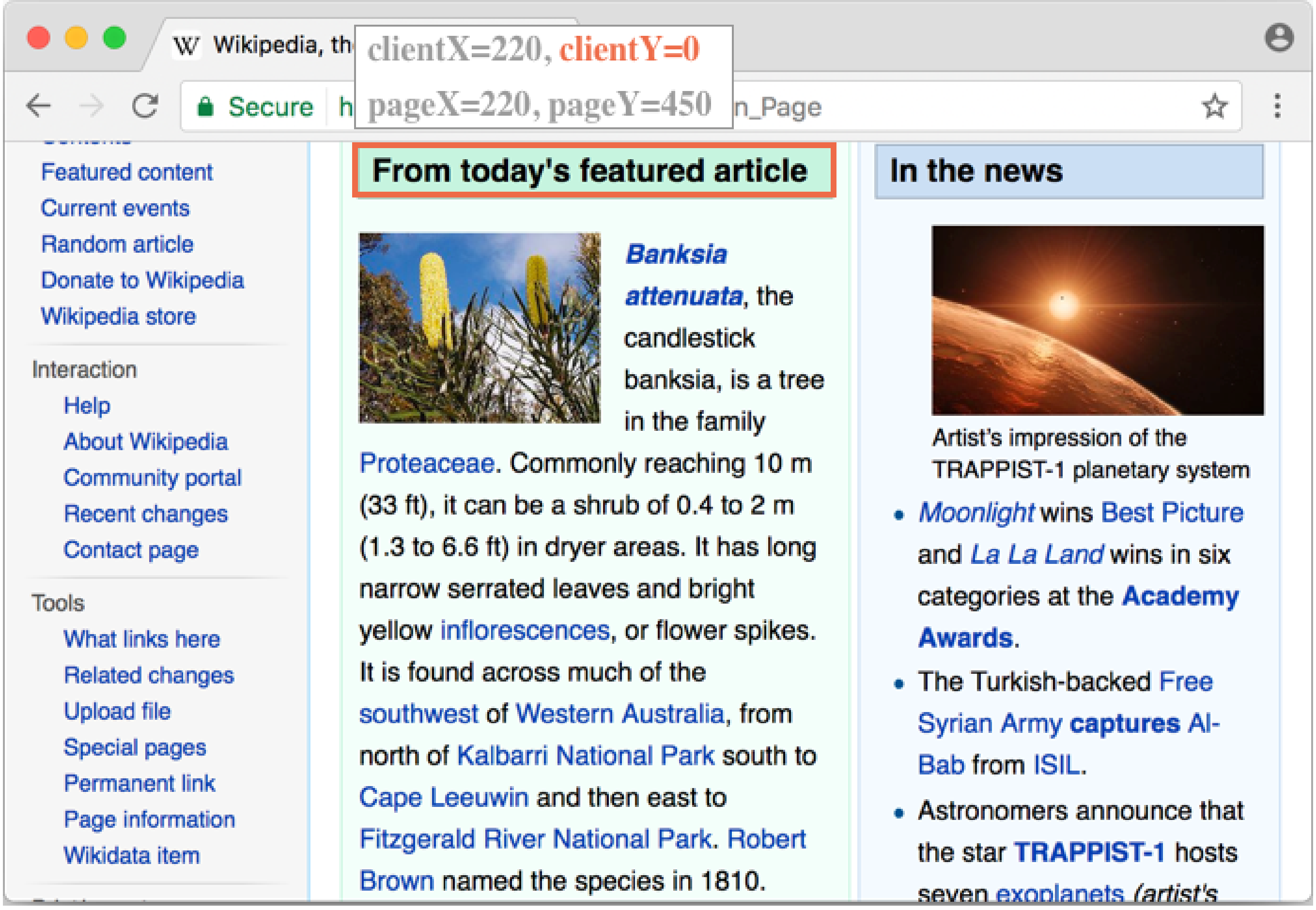
pageX=220, pageY=450

Banksia attenuata, the candlestick banksia, is a tree in the family Proteaceae. Commonly reaching 10 m

Artist's impression of the TRAPPIST-1 planetary system

And if we scroll it, then `(clientX,clientY)` change, because they are relative to the window, but `(pageX,pageY)` remain the same.

Here's the same page after the vertical scroll:



Banksia attenuata, the candlestick banksia, is a tree in the family Proteaceae. Commonly reaching 10 m (33 ft), it can be a shrub of 0.4 to 2 m (1.3 to 6.6 ft) in dryer areas. It has long narrow serrated leaves and bright yellow inflorescences, or flower spikes. It is found across much of the southwest of Western Australia, from north of Kalbarri National Park south to Cape Leeuwin and then east to Fitzgerald River National Park. Robert Brown named the species in 1810.

Artist's impression of the TRAPPIST-1 planetary system

- *Moonlight* wins Best Picture and *La La Land* wins in six categories at the **Academy Awards**.
- The Turkish-backed Free Syrian Army **captures** Al-Bab from ISIL.
- Astronomers announce that the star **TRAPPIST-1** hosts seven exoplanets *(artist's*

- `clientY` of the header `"From today's featured article"` became `0`, because the element is now on window top.
- `clientX` didn't change, as we didn't scroll horizontally.
- `pageX` and `pageY` coordinates of the element are still the same, because they are relative to the document.

Getting document coordinates

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 + pageYOffset,
    left: box.left + pageXOffset
  };
}
```

Summary

Any point on the page has coordinates:

1. Relative to the window – `elem.getBoundingClientRect()`.
2. 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 “pro” and “contra”, there are times we need one or the other one, just like CSS `position` `absolute` and `fixed`.

Tasks

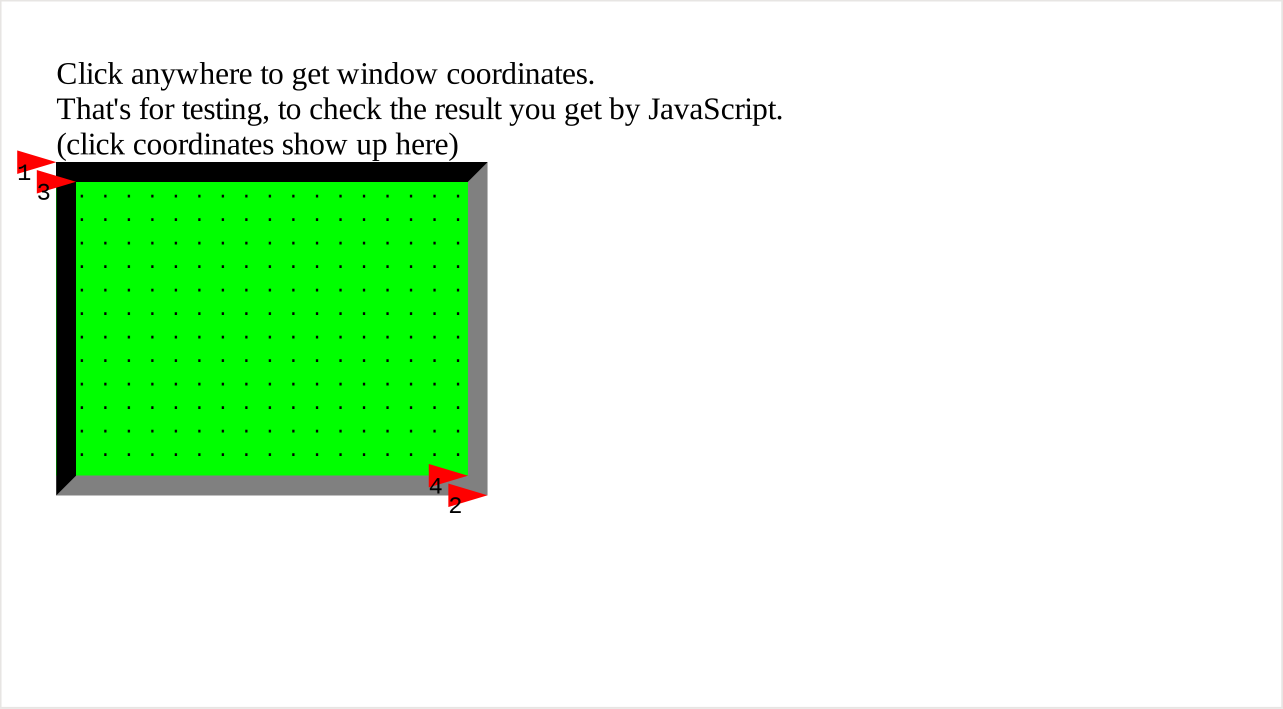
Find window coordinates of the field

importance: 5

In 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.



Click anywhere to get window coordinates.
That's for testing, to check the result you get by JavaScript.
(click coordinates show up here)

Your code should use DOM to get window coordinates of:

1. Upper-left, outer corner (that’s simple).
2. Bottom-right, outer corner (simple too).
3. Upper-left, inner corner (a bit harder).
4. 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.

To solution

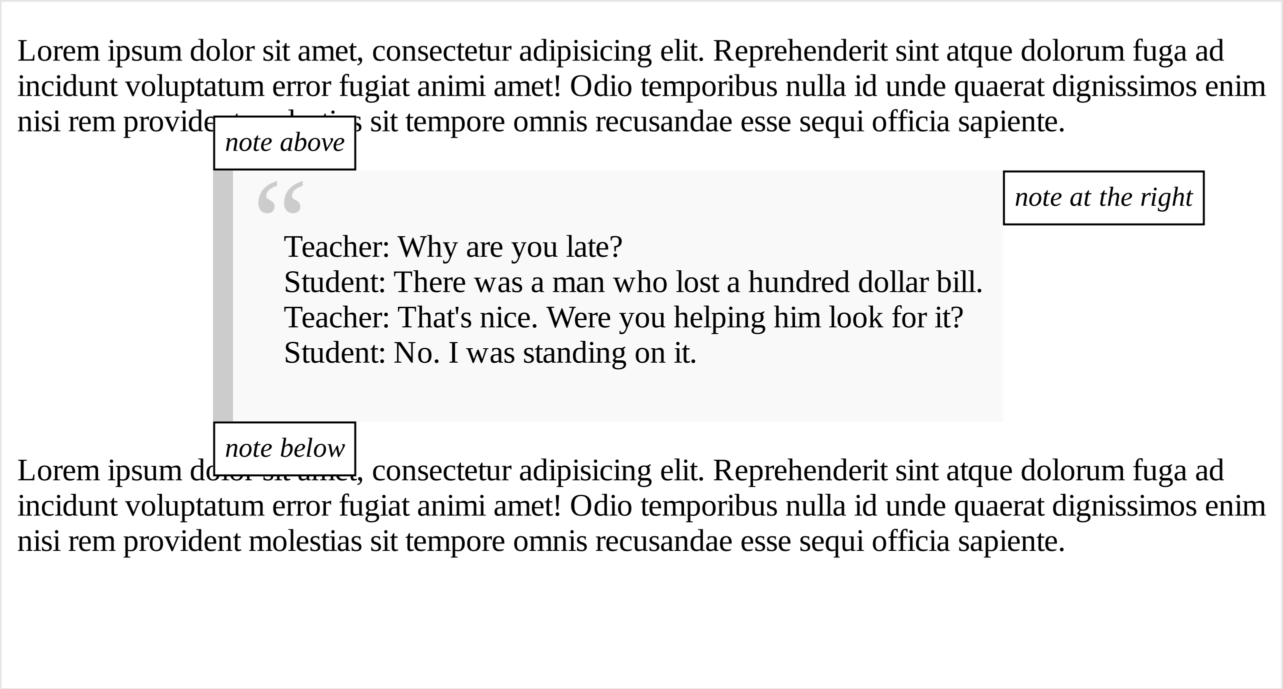
Show a note near the element

importance: 5

Create a function `positionAt(anchor, position, elem)` that positions `elem`, depending on `position` either at the top (`"top"`), right (`"right"`) or bottom (`"bottom"`) of the element `anchor`.

Call it inside the function `showNote(anchor, position, html)` that shows an element with the class `"note"` and the text `html` at the given position near the anchor.

Show the notes like here:



Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

note above

note at the right

> Teacher: Why are you late?
> Student: There was a man who lost a hundred dollar bill.
> Teacher: That's nice. Were you helping him look for it?
> Student: No. I was standing on it.

note below

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

P.S. The note should have `position:fixed` for this task.

Open a sandbox for the task.

To solution

Show a note near the element (absolute)

importance: 5

Modify 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">`.

To solution

Position the note inside (absolute)

importance: 5

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

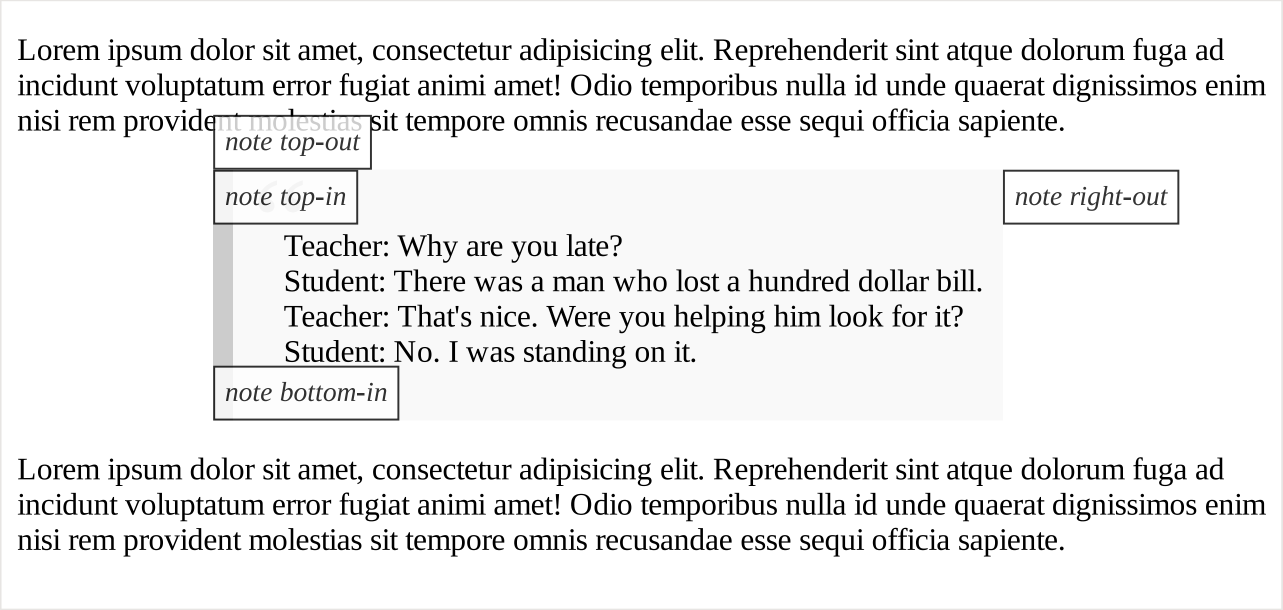
New values for `position`:

- `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:



Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

Teacher: Why are you late?
Student: There was a man who lost a hundred dollar bill.
Teacher: That's nice. Were you helping him look for it?
Student: No. I was standing on it.

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Reprehenderit sint atque dolorum fuga ad incidunt voluptatum error fugiat animi amet! Odio temporibus nulla id unde quaerat dignissimos enim nisi rem provident molestias sit tempore omnis recusandae esse sequi officia sapiente.

As the source code, take the solution of the task Show a note near the element (absolute).

To solution

Introduction into Events

An introduction to browser events, event properties and handling patterns.

Introduction to browser events

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

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

Mouse events:

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

Form element events:

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

Keyboard events:

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

Document events

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

CSS events:

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

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

Event handlers

To react on events we can assign a *handler* – a function that runs in case of an event.

Handlers are a way to run JavaScript code in case of user actions.

There are several ways to assign a handler. Let’s see them, starting from the simplest one.

HTML-attribute

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

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

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

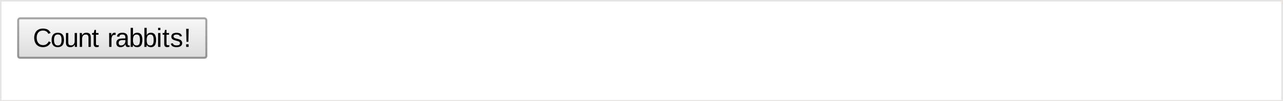
On mouse click, the code inside `onclick` runs.

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

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

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

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Rabbit number " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">
```



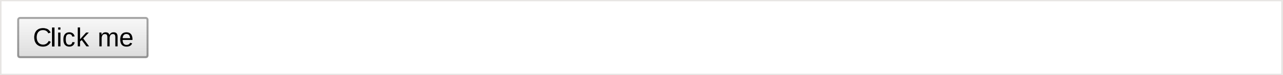
As we know, HTML attribute names are not case-sensitive, so `ONCLICK` works as well as `onClick` and `onCLICK`… But usually attributes are lowercased: `onclick`.

DOM property

We 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.

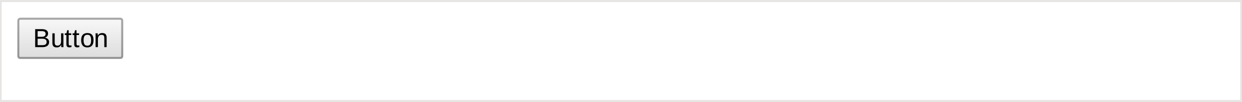
So this way is actually the same as the previous one.

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

These two code pieces work the same:

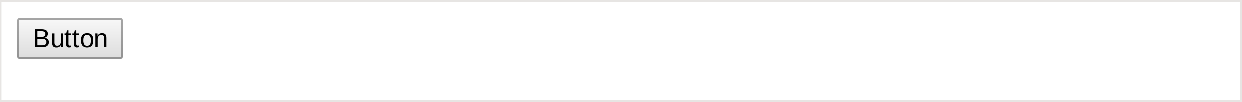
1. Only HTML:

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

2. HTML + JS:

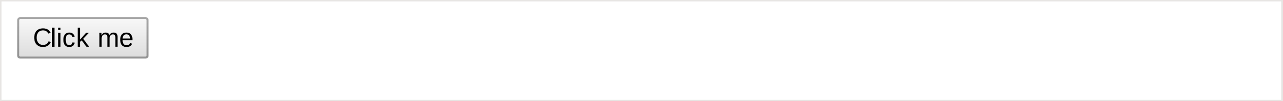
   ```
   <input type="button" id="button" value="Button">
   <script>
     button.onclick = function() {
       alert('Click!');
     };
   </script>
   ```

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

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

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



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

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

elem.onclick = sayThanks;
```

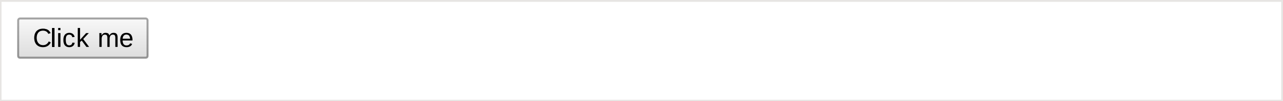
To remove a handler – assign `elem.onclick = null`.

Accessing the element: this

The value of `this` inside a handler is the element. The one which has the handler on it.

In the code below `button` shows its contents using `this.innerHTML`:

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



Possible mistakes

If you’re starting to work with event – please note some subtleties.

The function should be assigned as `sayThanks`, not `sayThanks()`.

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

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

If we add brackets, then `sayThanks()` – will be the *result* of the function execution, so `onclick` in the last code becomes `undefined` (the function returns nothing). That won't work.

…But in the markup we do need the brackets:

```
<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 the body from its content.

So the last example is the same as:

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

Use functions, not strings.

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

Don't use `setAttribute` for handlers.

Such a call won't work:

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

DOM-property case matters.

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

addEventListener

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

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

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

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

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

The syntax to add a handler:

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

`event`

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

`handler`

The handler function.

`phase`

An optional argument, the “phase” for the handler to work. To be covered later. Usually we don’t use it.

To remove the handler, use `removeEventListener`:

```
// exactly the same arguments as addEventListener
element.removeEventListener(event, handler[, phase]);
```

⚠ Removal requires the same function

To remove a handler we should pass exactly the same function as was assigned.

That doesn’t work:

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

The handler won’t be removed, because `removeEventListener` gets another function – with the same code, but that doesn’t matter.

Here’s the right way:

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

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

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

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

```
<input id="elem" type="button" value="Click me"/>

<script>
  function handler1() {
    alert('Thanks!');
  };

  function handler2() {
    alert('Thanks again!');
  }

  elem.onclick = () => alert("Hello");
  elem.addEventListener("click", handler1); // Thanks!
  elem.addEventListener("click", handler2); // Thanks again!
</script>
```

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. Must use `addEventListener`.
>
> For instance, the event `transitionend` (CSS animation finished) is like that.
>
> Try the code below. In most browsers only the second handler works, not the first one.

```
<style>
  input {
    transition: width 1s;
    width: 100px;
  }

  .wide {
    width: 300px;
  }
</style>

<input type="button" id="elem" onclick="this.classList.toggle('wide')" value=

<script>
  elem.ontransitionend = function() {
    alert("DOM property"); // doesn't work
  };

  elem.addEventListener("transitionend", function() {
    alert("addEventListener"); // shows up when the animation finishes
  });
</script>
```

Event object

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

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

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

```
<input type="button" value="Click me" id="elem">

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

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 you bind `this` to something else, and then `event.currentTarget` becomes useful.

`event.clientX / event.clientY`

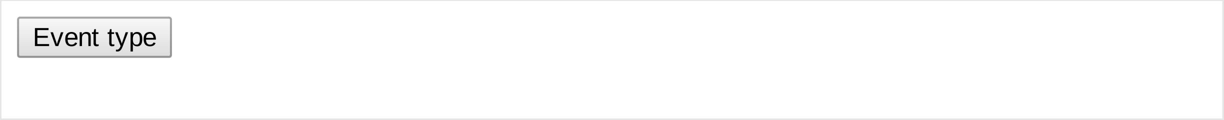
Window-relative coordinates of the cursor, for mouse events.

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

The event object is also accessible from HTML

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

```html
<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: handleEvent

We can assign an object as an event handler using `addEventListener`. When an event occurs, its `handleEvent` method is called with it.

For instance:

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

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

In other words, when `addEventListener` receives an object as the handler, it calls `object.handleEvent(event)` in case of an event.

We could also use a class for that:

```html
<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.

Summary

There are 3 ways to assign event handlers:

1. HTML attribute: `onclick="..."`.
2. DOM property: `elem.onclick = function`.
3. 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 `transtionend` 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.

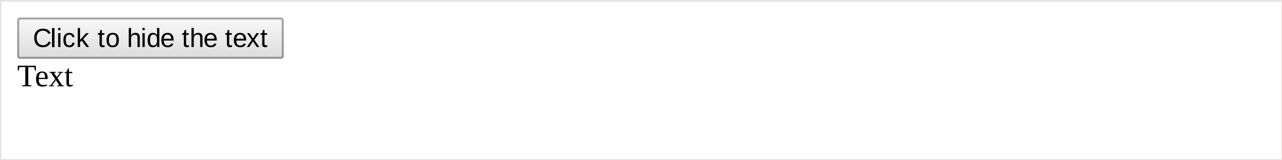
Tasks

Hide on click

importance: 5

Add JavaScript to the `button` to make `<div id="text">` disappear when we click it.

The demo:



Open a sandbox for the task.

To solution

Hide self

importance: 5

Create a button that hides itself on click.

To solution

Which handlers run?

importance: 5

There’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?

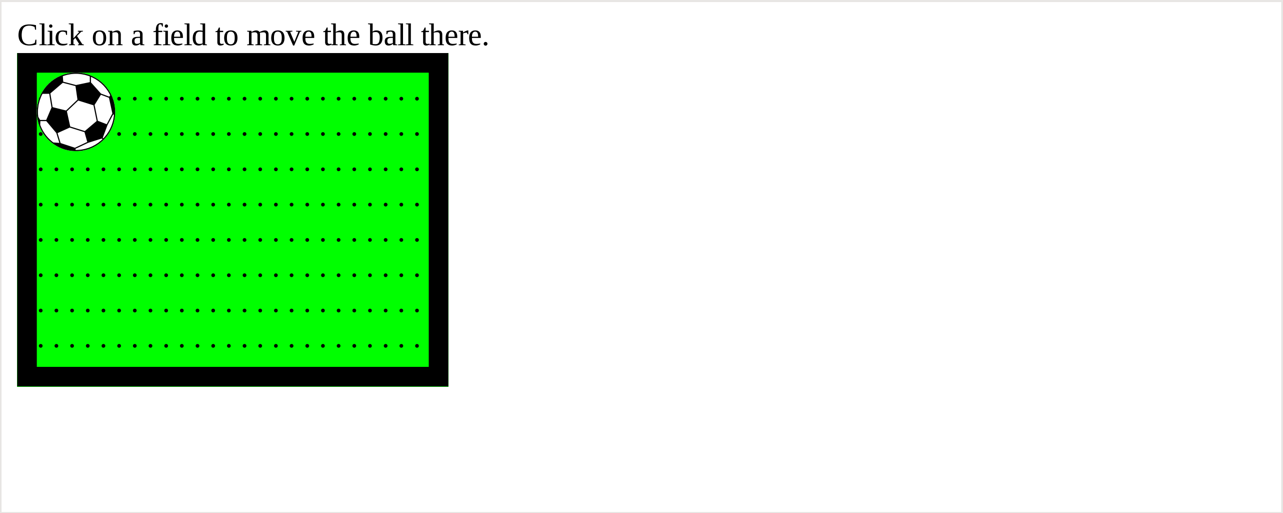
```
button.addEventListener("click", () => alert("1"));

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

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

To solution

Move the ball across the field

importance: 5

Move the ball across the field to a click. Like this:



Click on a field to move the ball there.

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.

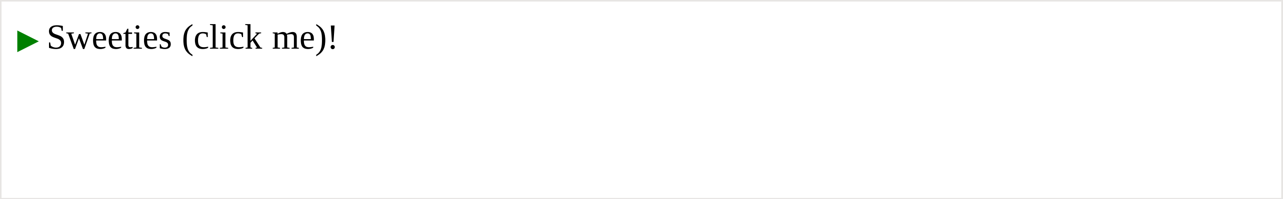
Open a sandbox for the task.

To solution

Create a menu sliding menu

importance: 5

Create a menu that opens/collapses on click:



▶ Sweeties (click me)!

P.S. HTML/CSS of the source document is to be modified.

Open a sandbox for the task.

To solution

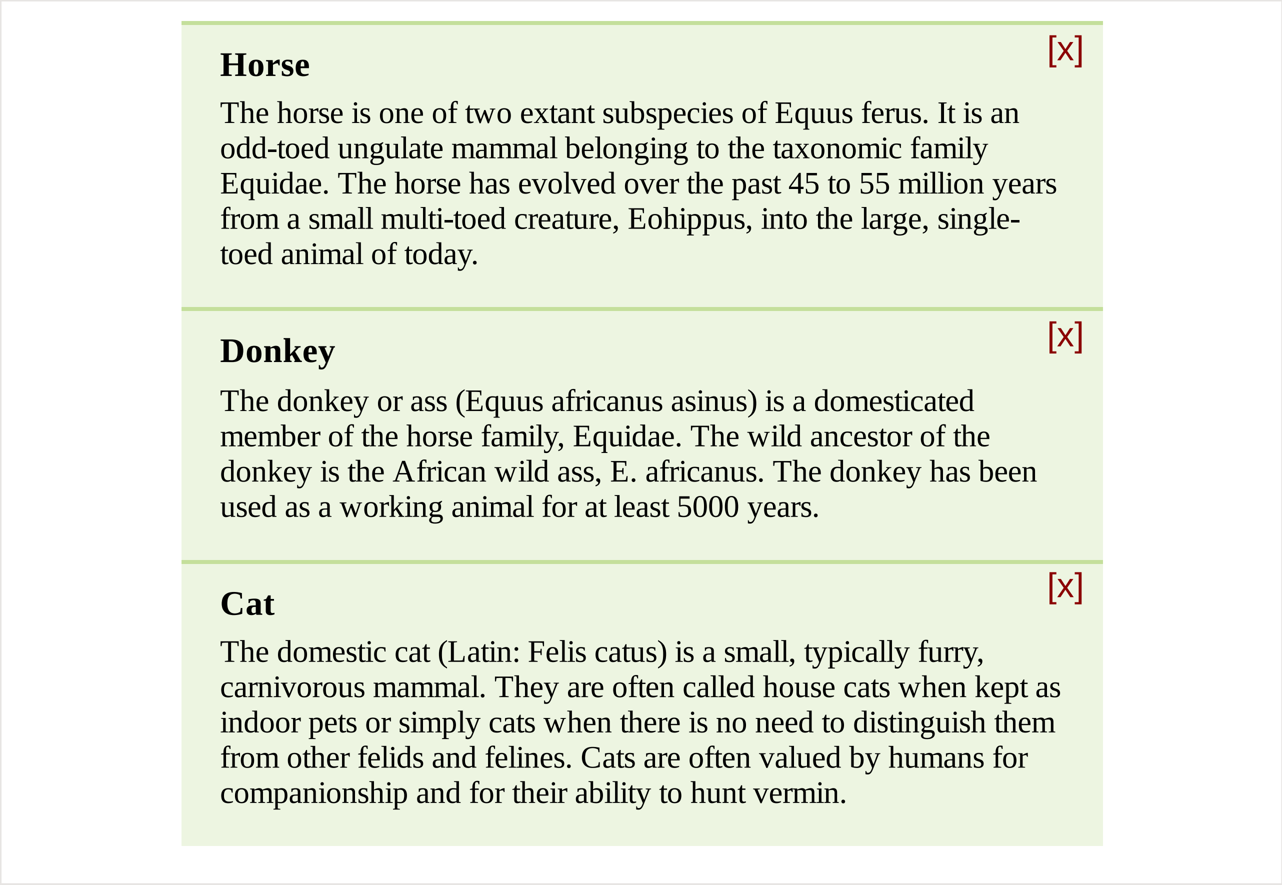
Add a closing button

importance: 5

There’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:



Horse [x]

The horse is one of two extant subspecies of Equus ferus. It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, Eohippus, into the large, single-toed animal of today.

Donkey [x]

The donkey or ass (Equus africanus asinus) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, E. africanus. The donkey has been used as a working animal for at least 5000 years.

Cat [x]

The domestic cat (Latin: Felis catus) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.

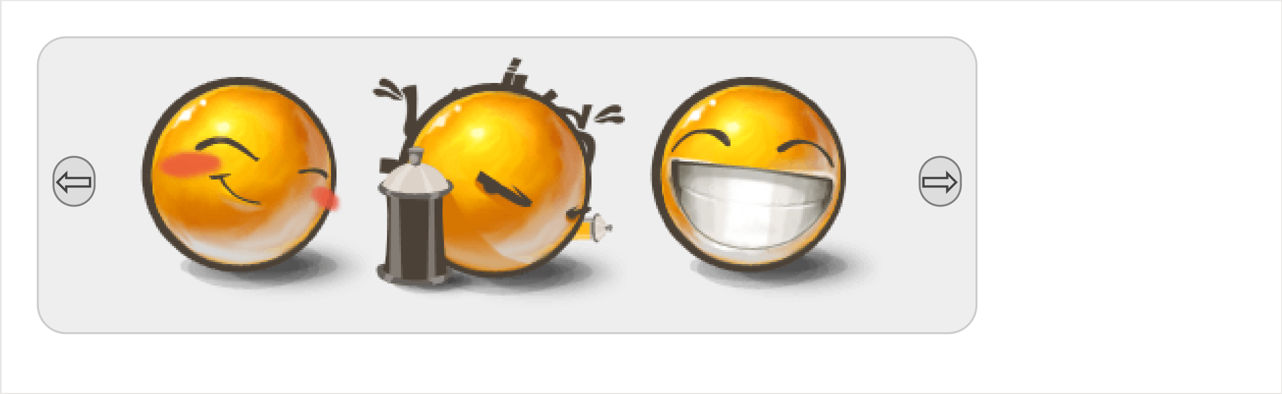
Open a sandbox for the task.

To solution

Carousel

importance: 4

Create 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.

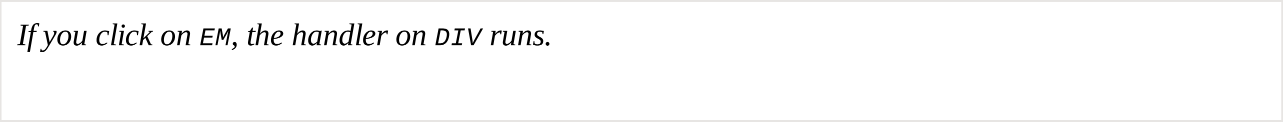
To solution

Bubbling and capturing

Let’s start with an example.

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

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



If you click on `EM`, the handler on `DIV` runs.

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

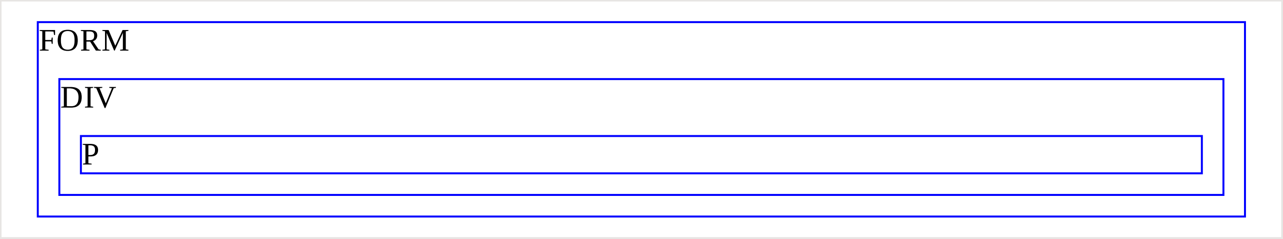
Bubbling

The bubbling principle is simple.

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

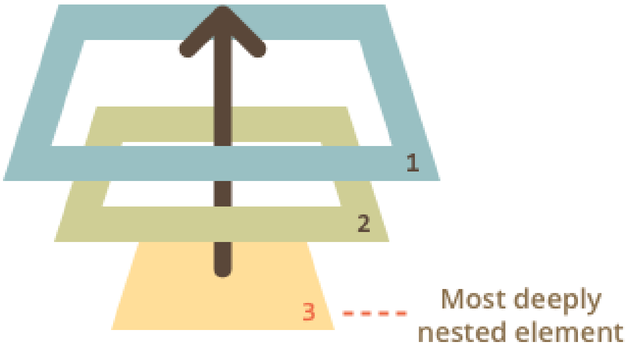
Let’s say we have 3 nested elements `FORM > DIV > P` with a handler on each of them:

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

<form onclick="alert('form')">FORM
  <div onclick="alert('div')">DIV
    <p onclick="alert('p')">P</p>
  </div>
</form>
```



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

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



So if we click on `<p>`, then we'll see 3 alerts: `p` → `div` → `form`.

The process is called "bubbling", because events "bubble" from the inner element up through parents like a bubble in the water.

> ***Almost* all events bubble.**
>
> The key word in this phrase is "almost".
>
> For instance, a `focus` event does not bubble. There are other examples too, we'll meet them. But still it's an exception, rather than a rule, most events do bubble.

event.target

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

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

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

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

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

In `form.onclick` handler:

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

Check it out:

http://plnkr.co/edit/iaPo3qfpDpHzXju0Jita?p=preview

It’s possible that `event.target` equals `this` – when the click is made directly on the `<form>` element.

Stopping bubbling

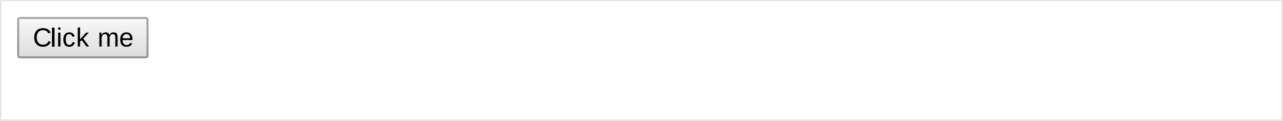
A bubbling event goes from the target element straight up. Normally it goes upwards till `<html>`, and then to `document` object, and some events even reach `window`, calling all handlers on the path.

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

The method for it is `event.stopPropagation()`.

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

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



event.stopImmediatePropagation()

If an element has multiple event handlers on a single event, then even if one of them stops the bubbling, the other ones still execute.

In other words, `event.stopPropagation()` stops the move upwards, but on the current element all other handlers will run.

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

Don't stop bubbling without a need!

Bubbling is convenient. Don't stop it without a real need: obvious and architecturally well-thought.

Sometimes `event.stopPropagation()` creates hidden pitfalls that later may become problems.

For instance:

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

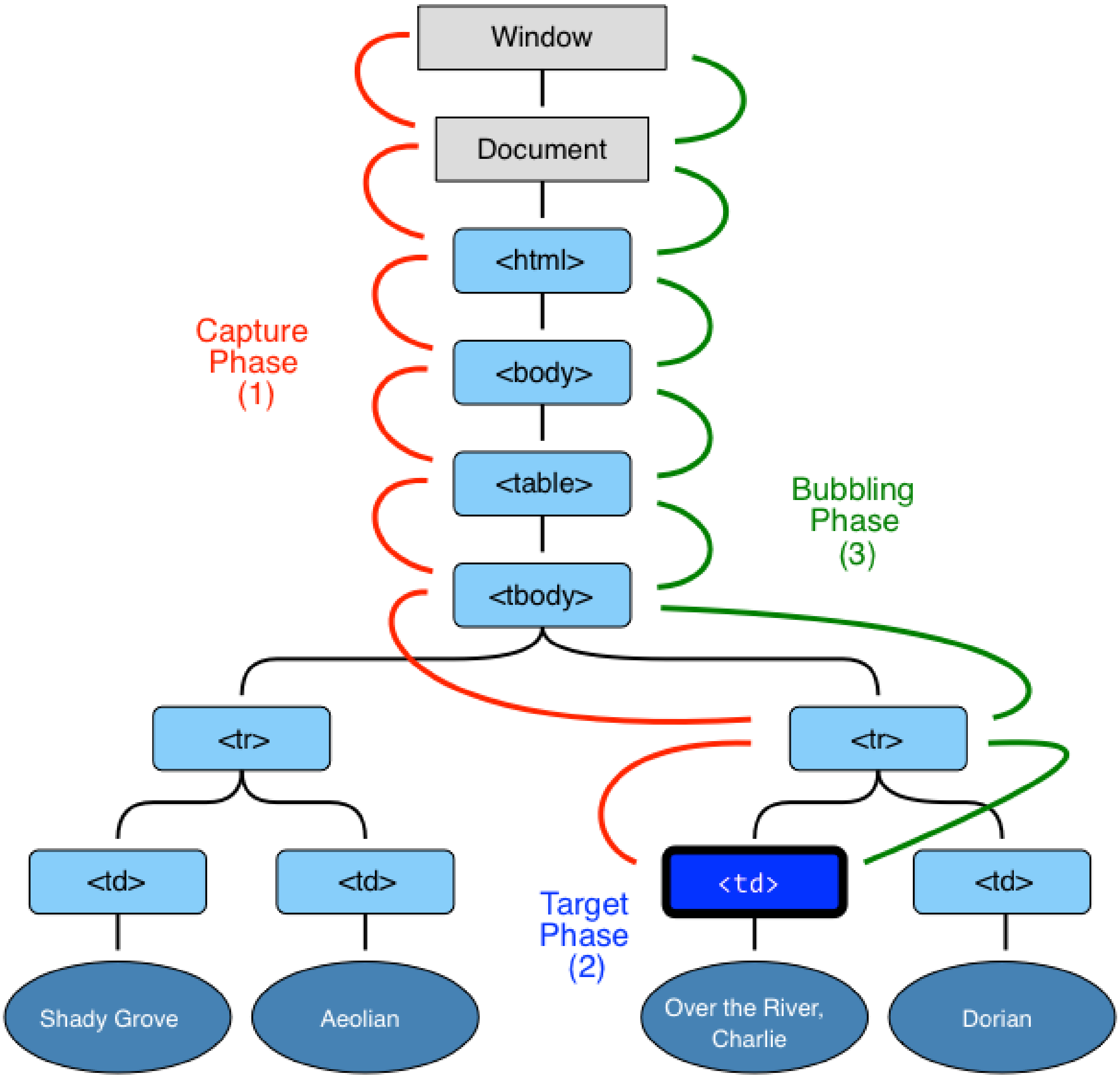
Capturing

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

The standard DOM Events describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

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



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

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

Handlers added using `on<event>`-property or using HTML attributes or using `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 3rd argument of `addEventListener` to `true`.

There are two possible values for that optional last argument:

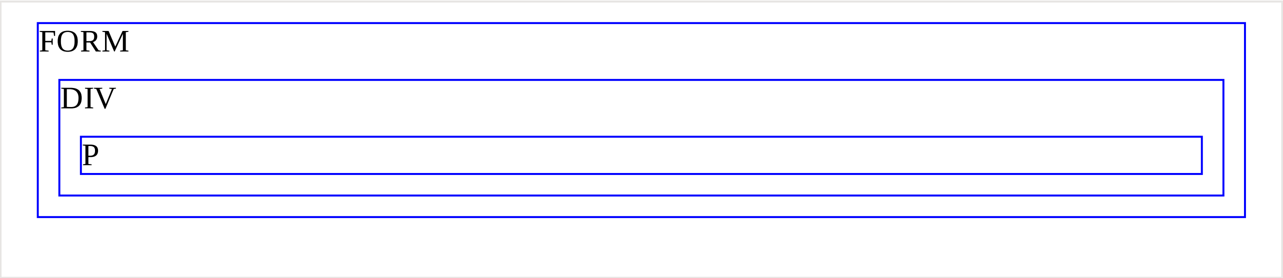
- If it’s `false` (default), then the handler is set on the bubbling phase.
- If it’s `true`, then the handler is set on the capturing phase.

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.

If one puts capturing and bubbling handlers on the target element, the capture handler triggers last in the capturing phase and the bubble handler triggers first in the bubbling phase.

Let’s see it in action:

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

<form>FORM
  <div>DIV
    <p>P</p>
  </div>
</form>

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



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

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

1. `HTML` → `BODY` → `FORM` → `DIV` → `P` (capturing phase, the first listener), and then:
2. `P` → `DIV` → `FORM` → `BODY` → `HTML` (bubbling phase, the second listener).

Please note that `P` shows up two times: at the end of capturing and at the start of bubbling.

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.

Summary

The event handling process:

- When an event happens – the most nested element where it happens gets labeled as the "target element" (`event.target`).
- Then the event first moves from the document root down to the `event.target`, calling handlers assigned with `addEventListener(...., true)` on the way.
- Then the event moves from `event.target` up to the root, calling handlers assigned using `on<event>` and `addEventListener` without the 3rd argument or with the 3rd argument `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, 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 logic behind 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.

Bubbling and capturing lay the foundation for “event delegation” – an extremely powerful event handling pattern that we study in the next chapter.

Event delegation

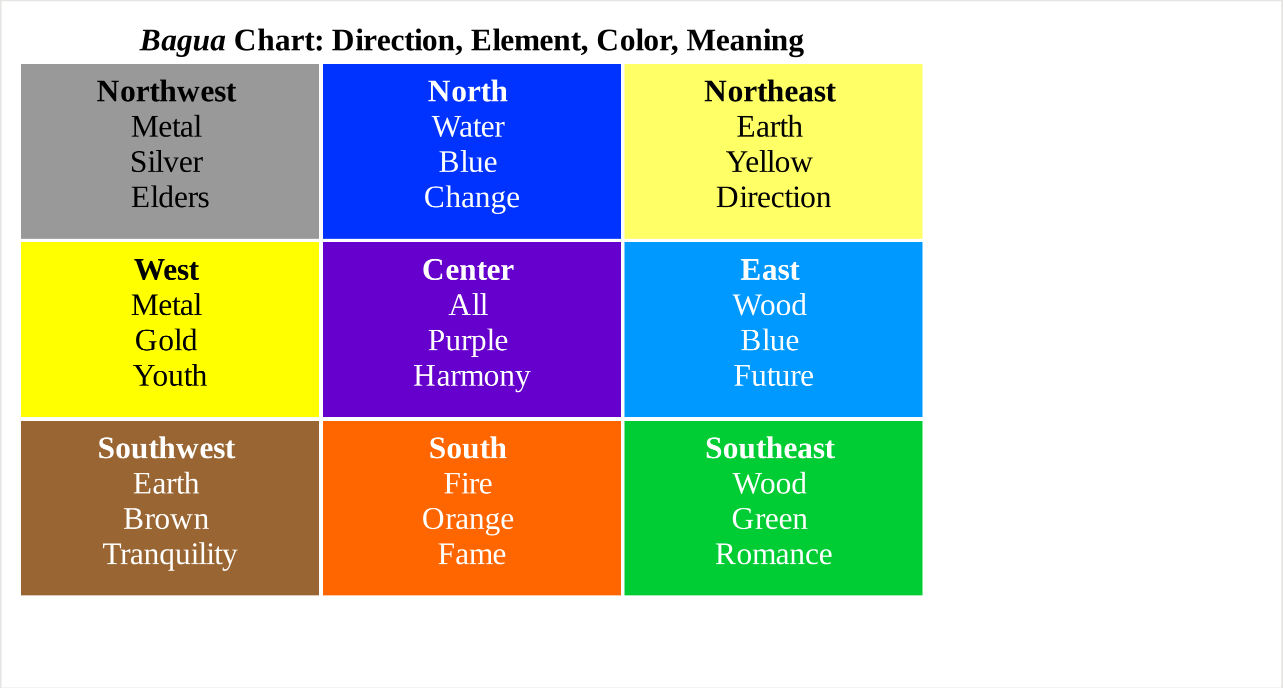
Capturing and bubbling allow us to implement one of most powerful event handling patterns called *event delegation*.

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

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

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

Here it is:



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

The HTML is like this:

```
<table>
  <tr>
    <th colspan="3"><em>Bagua</em> Chart: Direction, Element, Color, Meaning</th>
  </tr>
  <tr>
    <td>...<strong>Northwest</strong>...</td>
    <td>...</td>
    <td>...</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.

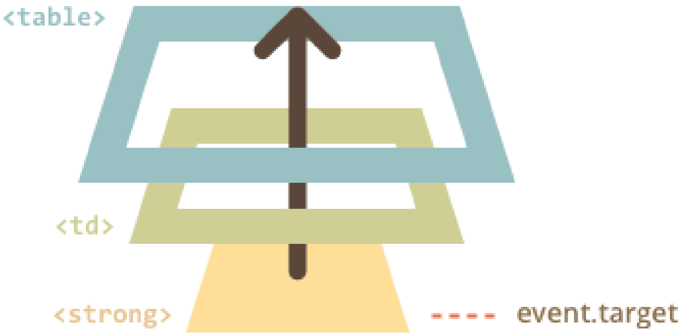
Still, there’s a drawback.

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 `<strong>`:

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

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



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

Here's the improved code:

```
table.onclick = function(event) {
  let td = event.target.closest('td'); // (1)

  if (!td) return; // (2)

  if (!table.contains(td)) return; // (3)

  highlight(td); // (4)
};
```

Explanations:

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

Delegation example: actions in markup

The event delegation may be used to optimize event handling. We use a single handler for similar actions on many elements. Like we did it for highlighting `<td>`.

But we can also use a single handler as an entry point for many different things.

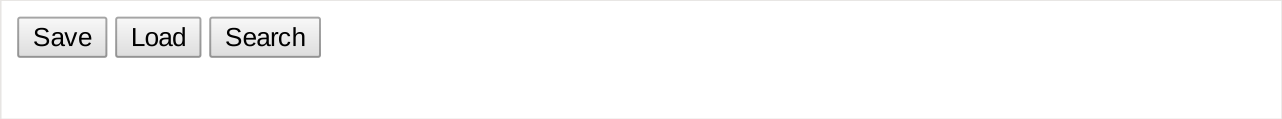
For instance, we want to make a menu with buttons “Save”, “Load”, “Search” and so on. And there’s an object with methods `save`, `load`, `search`….

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 the delegation gives us here?

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

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

The “behavior” pattern

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

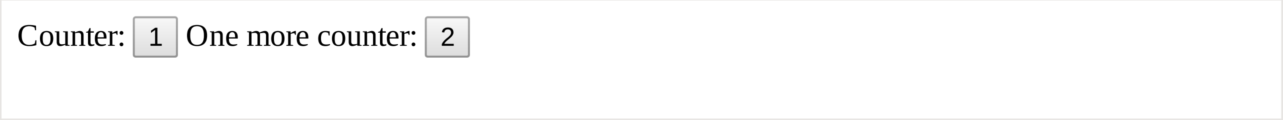
The pattern has two parts:

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

Counter

For instance, here the attribute `data-counter` adds a behavior: “increase 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>
```



Counter: 1 One more counter: 2

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.onclick`, because the latter will cause conflicts: new handlers overwrite old ones.

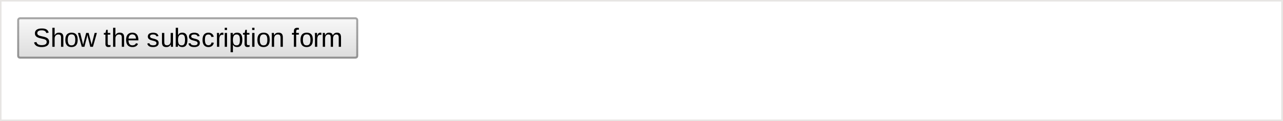
For real projects it’s normal that there are many handlers on `document` set by different parts of the code.

Toggler

One more example. 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 of mini-fragments of JavaScript.

Summary

Event delegation is really cool! It’s one of the most helpful patterns for DOM events.

It’s often used to add same handling for many similar elements, but not only for that.

The algorithm:

1. Put a single handler on the container.
2. In the handler – check the source element `event.target`.
3. 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 alike.

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 if they interest us or not. But usually the load is negligible, so we don’t take it into account.

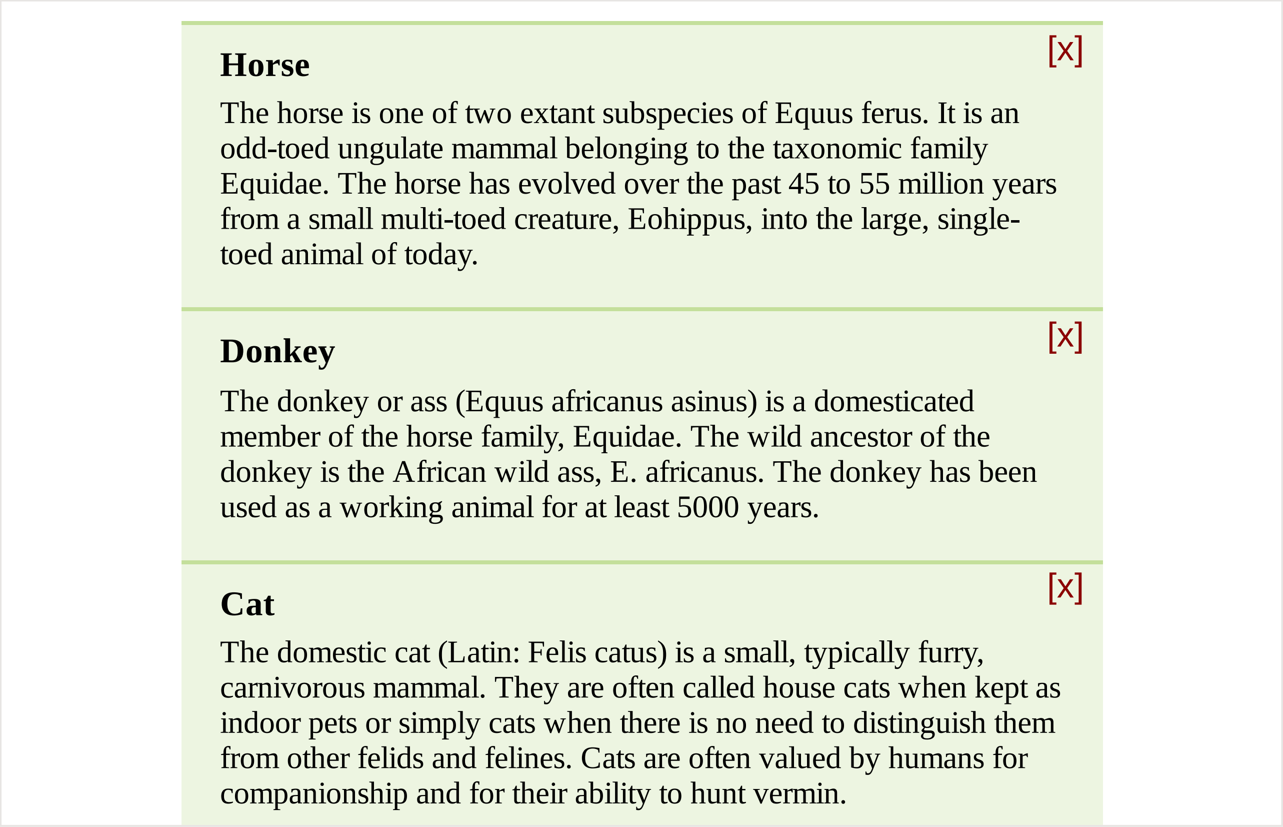
✓ Tasks

Hide messages with delegation

importance: 5

There’s a list of messages with removal buttons `[x]`. Make the buttons work.

Like this:



Horse [x]

The horse is one of two extant subspecies of Equus ferus. It is an odd-toed ungulate mammal belonging to the taxonomic family Equidae. The horse has evolved over the past 45 to 55 million years from a small multi-toed creature, Eohippus, into the large, single-toed animal of today.

Donkey [x]

The donkey or ass (Equus africanus asinus) is a domesticated member of the horse family, Equidae. The wild ancestor of the donkey is the African wild ass, E. africanus. The donkey has been used as a working animal for at least 5000 years.

Cat [x]

The domestic cat (Latin: Felis catus) is a small, typically furry, carnivorous mammal. They are often called house cats when kept as indoor pets or simply cats when there is no need to distinguish them from other felids and felines. Cats are often valued by humans for companionship and for their ability to hunt vermin.

P.S. Should be only one event listener on the container, use event delegation.

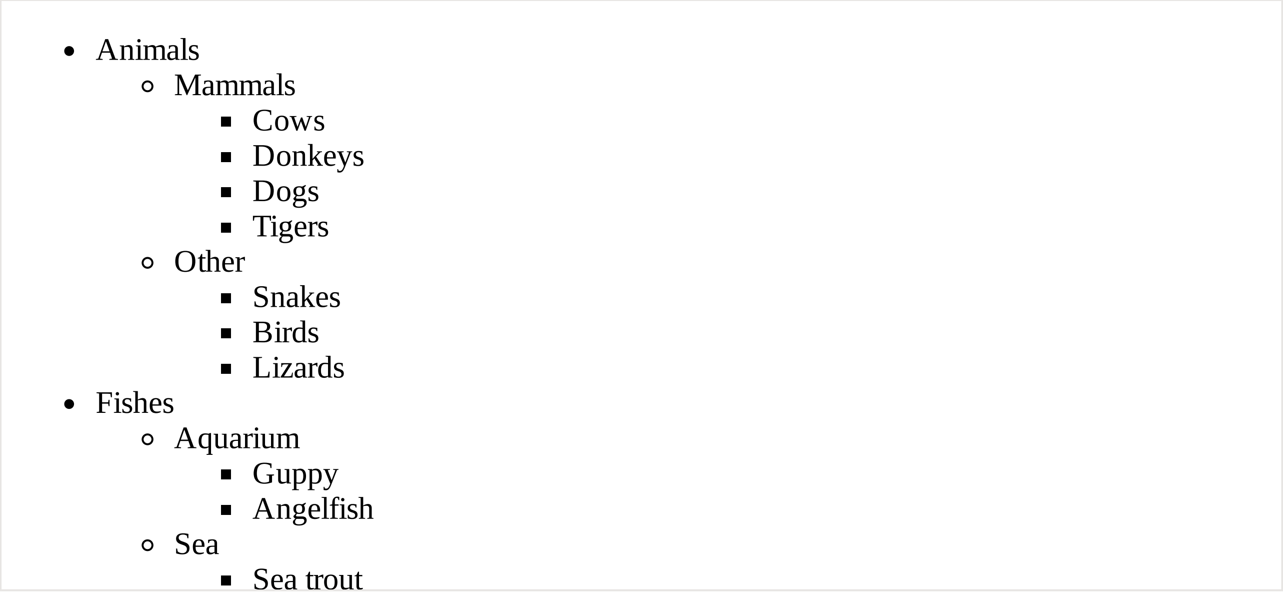
Open a sandbox for the task.

To solution

Tree menu

importance: 5

Create a tree that shows/hides node children on click:



- Animals
  - Mammals
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other
    - Snakes
    - Birds
    - Lizards
- Fishes
  - Aquarium
    - Guppy
    - Angelfish
  - Sea
    - Sea trout

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.

To solution

Sortable table

importance: 4

Make the table sortable: clicks on `<th>` elements should sort it by corresponding column.

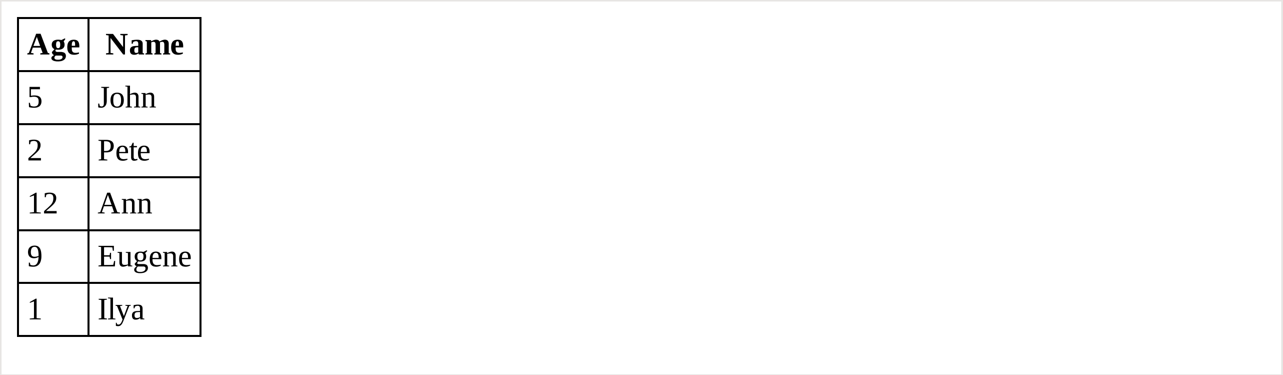
Each `<th>` has the type in the attribute, like this:

```
<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:



Age	Name
5	John
2	Pete
12	Ann
9	Eugene
1	Ilya

P.S. The table can be big, with any number of rows and columns.

Open a sandbox for the task.

To solution

Tooltip behavior

importance: 5

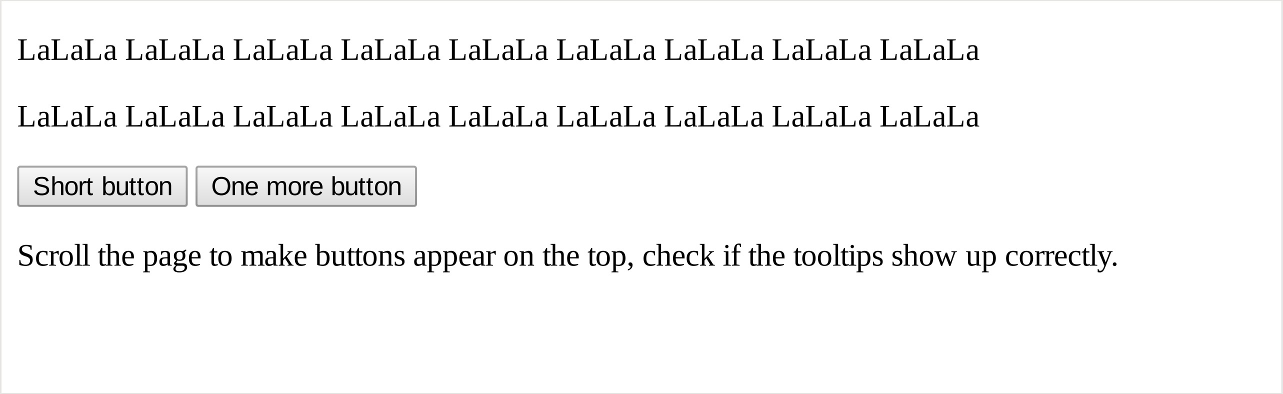
Create JS-code for the tooltip behavior.

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</butto
<button data-tooltip="HTML<br>tooltip">One more button</button>
```

Should work like this:



LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa

LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa LaLaLa

Scroll the page to make buttons appear on the top, check if the tooltips show up correctly.

In this task we assume that all elements with `data-tooltip` have only text inside. No nested tags.

Details:

- 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 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. To keep things natural and simple: only one tooltip may show up at a time.

Open a sandbox for the task. ↗

To solution

Browser default actions

Many events automatically lead to browser actions.

For instance:

- A click on a link – initiates going to its URL.
- A click on submit button inside a form – 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, often we don't want browser actions. Fortunately, it can be prevented.

Preventing browser actions

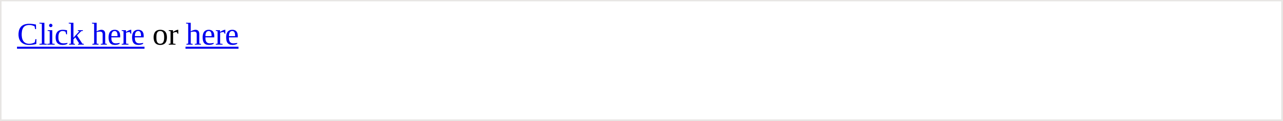
There are two ways to tell the browser we don't want it to act:

- The main way is to use the `event` object. There's a method `event.preventDefault()`.

- If the handler is assigned using `on<event>` (not by `addEventListener`), then we can just return `false` from it.

In the example below a click to links doesn't lead to URL change:

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



Click here or here

Not necessary to return `true`

The value returned by an event handler is usually ignored.

The only exception – is `return false` from a handler assigned using `on<event>`.

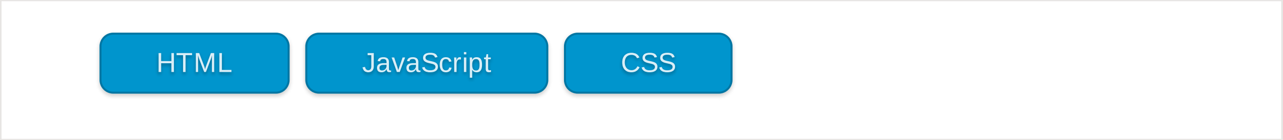
In all other cases, the return is not needed and it's not processed anyhow.

Example: the menu

Consider a site menu, like this:

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

Here's how it looks with some CSS:



Menu items are links `<a>`, not buttons. There are several benefits, for instance:

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

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

Like here:

```
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert( href ); // ...can be loading from the server, UI generation etc

  return false; // prevent browser action (don't go to the URL)
};
```

If we omit `return false`, then after our code executes the browser will do its “default action” – following to the URL in `href`.

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

Prevent further 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. That’s normal.

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">
```

Focus works

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.

event.defaultPrevented

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

There’s an interesting use case for it.

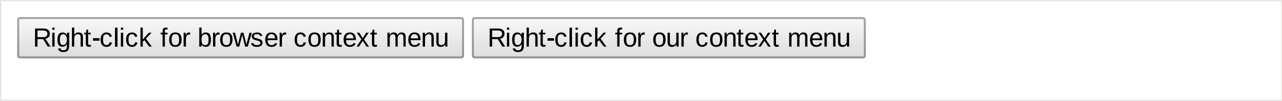
You remember in the chapter Bubbling and capturing we talked about `event.stopPropagation()` and why stopping bubbling is bad?

Sometimes we can use `event.defaultPrevented` instead.

Let’s see a practical example where stopping the bubbling looks necessary, but actually we can do well without it.

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:

```html
<button>Right-click for browser context menu</button>

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


Right-click for browser context menu
Right-click for our context menu

Now let’s say we want to implement our own document-wide context menu, with our options. And inside the document we may have other elements with their own context menus:

```html
<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>
```

Right-click here for the document context menu

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: “We fully handle the event in the button handler, let’s stop it” and use `event.stopPropagation()`:

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

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

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

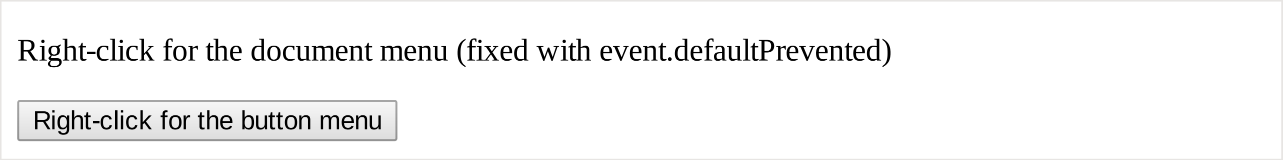
Right-click for the document menu

Right-click for the button menu (fixed with event.stopPropagation)

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 (fixed with 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>
```



Right-click for the document menu (fixed with event.defaultPrevented)

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 special global object with a method that handles `document.oncontextmenu`, and also methods that allow to store various “lower-level” 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.

Summary

There 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.
- `wheel` – rolling a mouse wheel event has scrolling as the default action.
- `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>`.

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 preform 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 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.

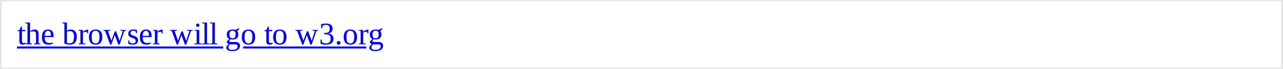
✓ Tasks

Why "return false" doesn't work?

importance: 3

Why in the code below `return false` doesn’t work at all?

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

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



the browser will go to w3.org

The browser follows the URL on click, but we don’t want it.

How to fix?

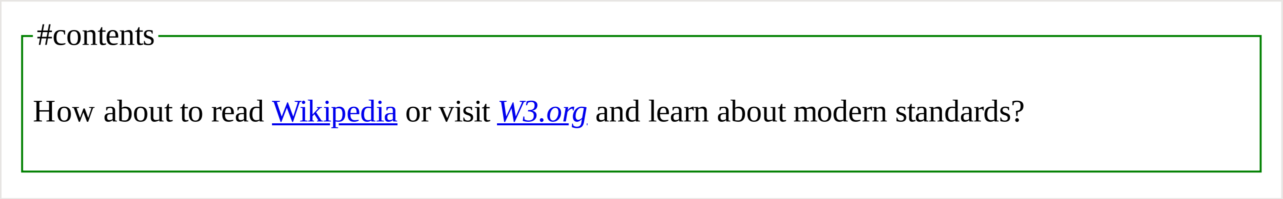
To solution

Catch links in the element

importance: 5

Make 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:



#contents

How about to read Wikipedia or visit *W3.org* and learn about modern standards?

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 the event delegation.
- The content may have nested tags. Inside links too, like `<a href=".."><i>...</i></a>`.

Open a sandbox for the task.

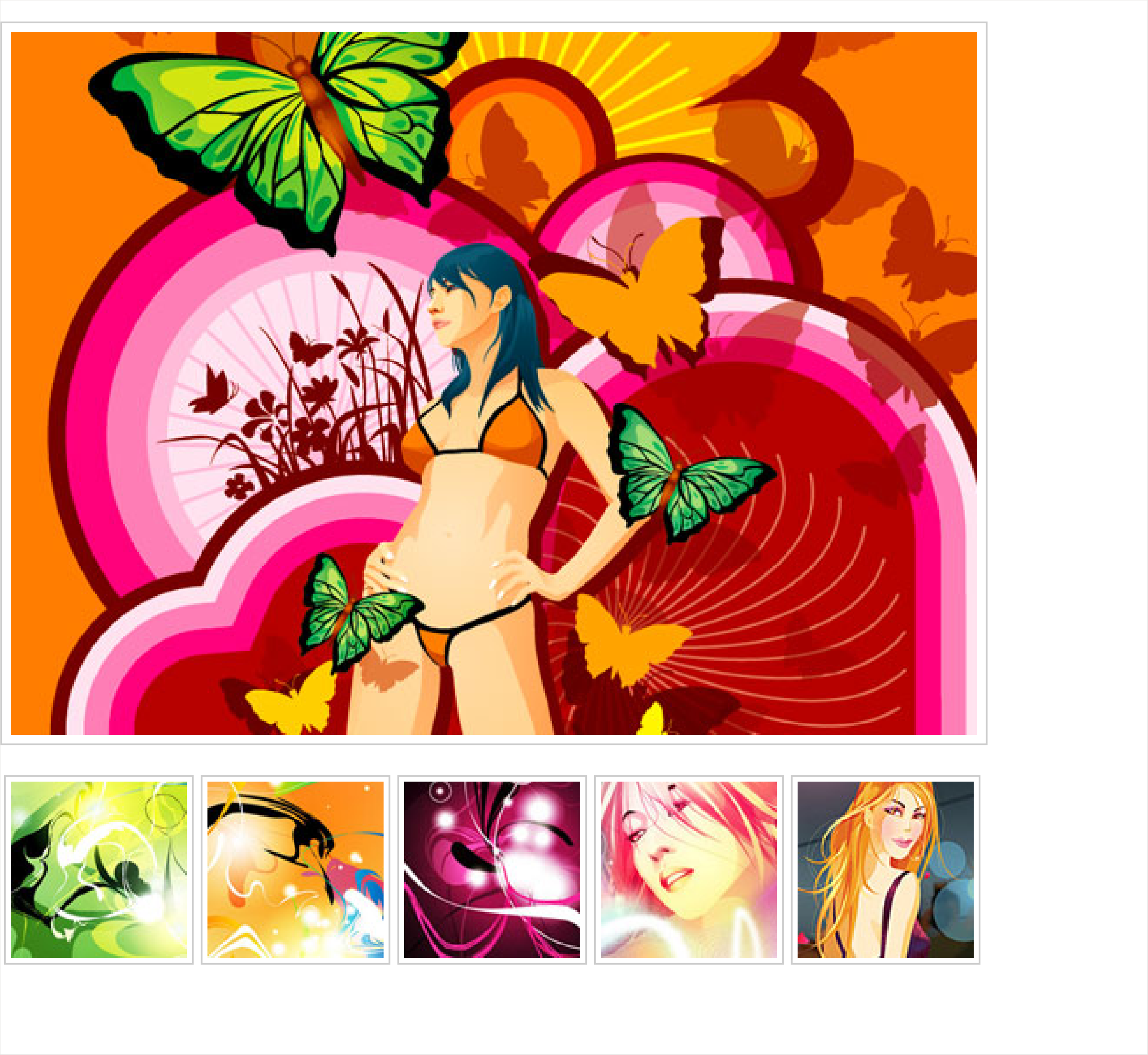
To solution

Image gallery

importance: 5

Create 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.

To solution

Dispatching custom events

We 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 the menu may trigger events telling what happens with the menu: `open` (menu open), `select` (an item is selected) and so on.

Also we can generate built-in events like `click`, `mousedown` etc, that may be good for testing.

Event constructor

Events form a hierarchy, just like DOM element classes. The root is the built-in Event class.

We can create `Event` objects like this:

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

Arguments:

- *event type* – may be any string, like `"click"` or our own like `"hey-ho!"`.
- *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}`.

dispatchEvent

After 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 built-in 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 example

We 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);
</script>
```

Notes:

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

Here’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 events

For our own, custom events 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.

The event class tells something about “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()

We can call `event.preventDefault()` on a script-generated event if `cancelable:true` flag is specified.

Of course, if the event has a non-standard name, then it’s not known to the browser, and there’s no “default browser action” for it.

But the event-generating code may plan some actions after `dispatchEvent`.

The call of `event.preventDefault()` is a way for the handler to send a signal that those actions shouldn’t be performed.

In that case the call to `elem.dispatchEvent(event)` returns `false`. And the event-generating code knows that the processing shouldn’t continue.

For instance, in the example below there's a `hide()` function. It generates the `"hide"` event on the element `#rabbit`, notifying all interested parties that the rabbit is going to hide.

A handler set by `rabbit.addEventListener('hide',...)` will learn about that and, if it wants, can prevent that action by calling `event.preventDefault()`. Then the rabbit won't hide:

```
<pre id="rabbit">
  |\   /|
   \|_|/
   /. .\
  =\_Y_/=
   {>o<}
</pre>

<script>
  // hide() will be called automatically in 2 seconds
  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();
    }
  });

  // hide in 2 seconds
  setTimeout(hide, 2000);

</script>
```

Events-in-events are synchronous

Usually events are processed asynchronously. That is: if the browser is processing `onclick` and in the process a new event occurs, then it awaits till `onclick` processing is finished.

The exception is when one event is initiated from within another one.

Then the control jumps to the nested event handler, and after it goes back.

For instance, here the nested `menu-open` event is processed synchronously, during the `onclick`:

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

<script>
  // 1 -> nested -> 2
  menu.onclick = function() {
    alert(1);

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

    alert(2);
  };

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

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

That's not only about `dispatchEvent`, there are other cases. JavaScript in an event handler can call methods that lead to other events – they are too processed synchronously.

If we don't like it, we can either put the `dispatchEvent` (or other event-triggering call) at the end of `onclick` or, if inconvenient, wrap it in `setTimeout(...,0)`:

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

<script>
  // 1 -> 2 -> nested
  menu.onclick = function() {
    alert(1);

    // alert(2)
    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {
      bubbles: true
    })), 0);
    alert(2);
  };

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

Summary

To generate an event, 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 to generate browser events like `click` or `keydown`, we should use with the great care.

We shouldn’t generate browser events as it’s a hacky way to run handlers. That’s a 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.

Events in details

Here we cover most important events and details of working with them.

Mouse events basics

Mouse events come not only from “mouse manipulators”, but are also emulated on touch devices, to make them compatible.

In this chapter we’ll get into more details about mouse events and their properties.

Mouse event types

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

Simple events

The most used simple events are:

`mousedown/mouseup`

Mouse button is clicked/released over an element.

`mouseover/mouseout`

Mouse pointer comes over/out from an element.

`mousemove`

Every mouse move over an element triggers that event.

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

Complex events

`click`

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

`contextmenu`

Triggers after `mousedown` if the right mouse button was used.

`dblclick`

Triggers after a double click over an element.

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

Events order

An action may trigger multiple events.

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

In cases when a single action initiates multiple events, their order is fixed. That is, the handlers are called in the order `mousedown` → `mouseup` → `click`. Events are handled in the same sequence: `onmouseup` finishes before `onclick` runs.

Getting the button: which

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

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

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

There are the three possible values:

- `event.which == 1` – the left button
- `event.which == 2` – the middle button
- `event.which == 3` – the right button

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

Modifiers: shift, alt, ctrl and meta

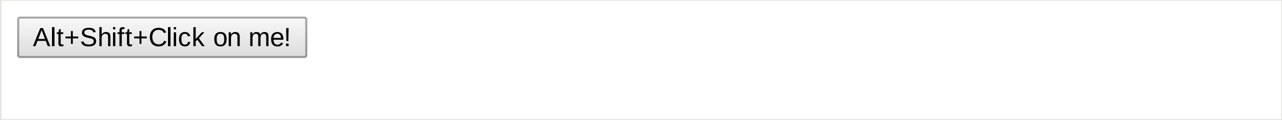
All mouse events include the information about pressed modifier keys.

The properties are:

- `shiftKey`
- `altKey`
- `ctrlKey`
- `metaKey` (`Cmd` for Mac)

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`, it corresponds to the property `metaKey`.

In most cases when Windows/Linux uses `Ctrl`, on Mac people use `Cmd`. So where a Windows user presses `Ctrl+Enter` or `Ctrl+A`, a Mac user would press `Cmd+Enter` or `Cmd+A`, and so on, most apps use `Cmd` instead of `Ctrl`.

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 Mac, and it generates the `contextmenu` event, not `click` like Windows/Linux.

So if we want users of all operational systems to feel comfortable, then together with `ctrlKey` we should use `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 has a keyboard – it works. And if your device doesn’t have it – then there’s another way to do the same.

Coordinates: clientX/Y, pageX/Y

All mouse events have coordinates in two flavours:

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

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

Document-relative coordinates are counted from the left-upper corner of the document, not the window. Coordinates `pageX`, `pageY` are similar to `position:absolute` on the document level.

You can read more about coordinates in the chapter Coordinates.

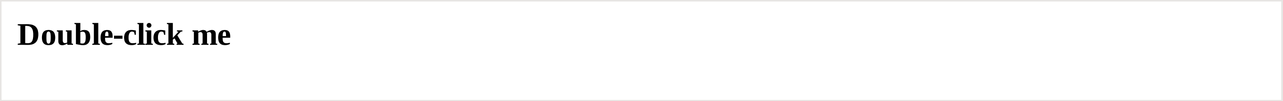
No selection on mousedown

Mouse clicks have a side-effect that may be disturbing. A double click selects the text.

If we want to handle click events ourselves, then the “extra” selection doesn’t look good.

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

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



Double-click me

There’s a CSS way to stop the selection: the `user-select` property from CSS UI Draft.

Most browsers support it with prefixes:

```
<style>
  b {
    -webkit-user-select: none;
    -moz-user-select: none;
    -ms-user-select: none;
    user-select: none;
  }
</style>

Before...
<b ondblclick="alert('Test')">
  Unselectable
</b>
...After
```

Before... **Unselectable** ...After

Now if you double-click on “Unselectable”, it doesn’t get selected. Seems to work.

…But there is a potential problem! The text became truly unselectable. Even if a user starts the selection from “Before” and ends with “After”, the selection skips “Unselectable” part. Do we really want to make our text unselectable?

Most of time, we don’t. A user may have valid reasons to select the text, for copying or other needs. That may be inconvenient if we don’t allow them to do it. So this solution is not that good.

What we want is to prevent the selection on double-click, that’s it.

A text selection is the default browser action on `mousedown` event. So the alternative solution would be to handle `mousedown` and prevent it, like this:

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

Before... **Double-click me** ...After

Now the bold element is not selected on double clicks.

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 though.

Canceling the selection

Instead of *preventing* the selection, we can cancel it “post-factum” in the event handler.

Here’s how:

```html
Before...
<b ondblclick="getSelection().removeAllRanges()">
  Double-click me
</b>
...After
```

Before... **Double-click me** ...After

If you double-click on the bold element, then the selection appears and then is immediately removed. That doesn’t look nice though.

Preventing copying

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

```html
<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 th
</div>
```

Dear user, The copying is forbidden for you. If you know JS or HTML, then you can get everything from the page source though.

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

Surely that can’t stop the user from opening HTML-source, but not everyone knows how to do it.

Summary

Mouse events have the following properties:

- Button: `which`.
- Modifier keys (`true` if pressed): `altKey`, `ctrlKey`, `shiftKey` and `metaKey` (Mac).
  - If you want to handle `Ctrl`, then don't forget Mac users, they use `Cmd`, so it's better to check `if (e.metaKey || e.ctrlKey)`.
- Window-relative coordinates: `clientX/clientY`.
- Document-relative coordinates: `pageX/pageY`.

It's also important to deal with text selection as an unwanted side-effect of clicks.

There are several ways to do this, for instance:

1. The CSS-property `user-select:none` (with browser prefixes) completely disables text-selection.
2. Cancel the selection post-factum using `getSelection().removeAllRanges()`.
3. Handle `mousedown` and prevent the default action (usually the best).

✓ Tasks

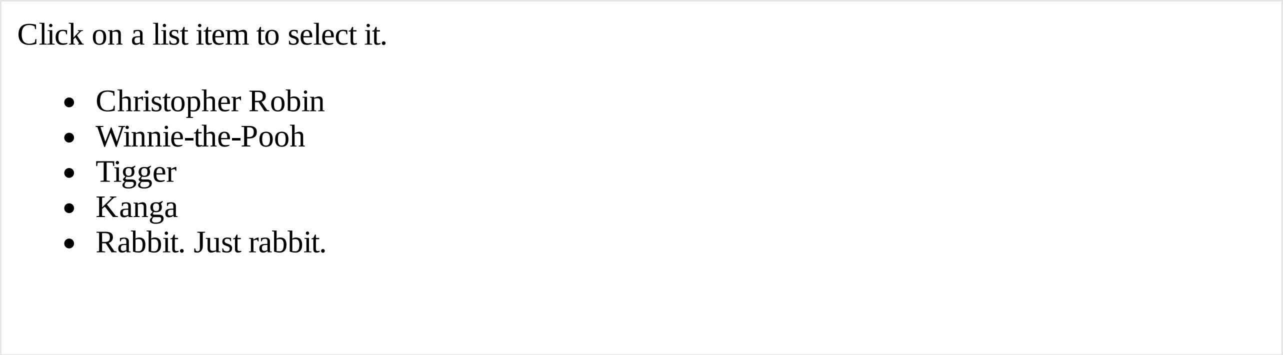
Selectable list

importance: 5

Create 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:



Click on a list item to select it.

- Christopher Robin
- Winnie-the-Pooh
- Tigger
- Kanga
- Rabbit. Just rabbit.

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.

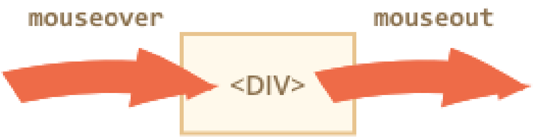
To solution

Moving: mouseover/out, mouseenter/leave

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

Mouseover/mouseout, relatedTarget

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



These events are special, because they have a `relatedTarget`.

For `mouseover`:

- `event.target` – is the element where the mouse came over.
- `event.relatedTarget` – is the element from which the mouse came.

For `mouseout` the reverse:

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

⚠ **`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.

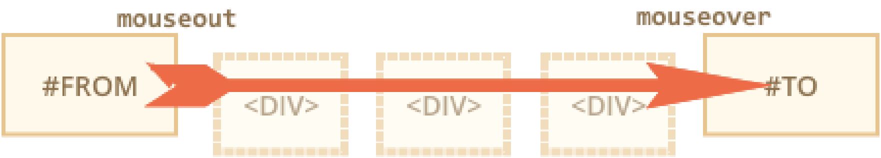
We should keep that possibility in mind when using `event.relatedTarget` in our code. If we access `event.relatedTarget.tagName`, then there will be an error.

Events frequency

The `mousemove` event triggers when the mouse moves. But that doesn't mean that every pixel leads to an event.

The browser checks the mouse position from time to time. And if it notices changes then triggers the events.

That means that if the visitor is moving the mouse very fast then DOM-elements may be skipped:

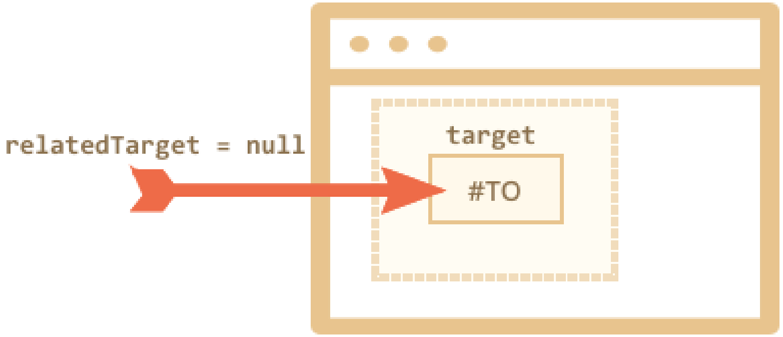


If the mouse moves very fast from `#FROM` to `#TO` elements as painted above, then intermediate `<div>` (or some of them) may be skipped. The `mouseout` event may trigger on `#FROM` and then immediately `mouseover` on `#TO`.

In practice that's helpful, because if there may be many intermediate elements. We don't really want to process in and out of each one.

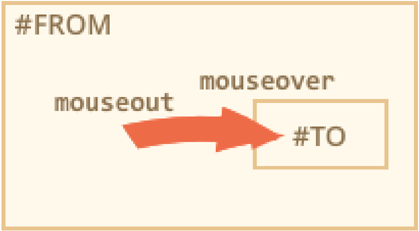
From the other side, we should keep in mind that we can't assume that the mouse slowly moves from one event to another. No, it can "jump".

In particular it's possible that the cursor jumps right inside the middle of the page from out of the window. And `relatedTarget=null`, because it came from "nowhere":



“Extra” mouseout when leaving for a child

Imagine – a mouse pointer entered an element. The `mouseover` triggered. Then the cursor goes into a child element. The interesting fact is that `mouseout` triggers in that case. The cursor is still in the element, but we have a `mouseout` from it!



That seems 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. That simple.

There’s a funny consequence that we can see on the example below.

The red `<div>` is nested inside the blue one. The blue `<div>` has `mouseover/out` handlers that log all events in the textarea below.

Try entering the blue element and then moving the mouse on the red one – and watch the events:

http://plnkr.co/edit/4PtAFMCaPdtyY0SbI3Zf?p=preview

1. On entering the blue one – we get `mouseover [target: blue]`.
2. Then after moving from the blue to the red one – we get `mouseout [target: blue]` (left the parent).
3. …And immediately `mouseover [target: red]`.

So, for a handler that does not take `target` into account, it looks like we left the parent in `mouseout` in `(2)` and returned back to it by `mouseover` in `(3)`.

If we perform some actions on entering/leaving the element, then we'll get a lot of extra "false" runs. For simple stuff may be unnoticeable. For complex things that may bring unwanted side-effects.

We can fix it by using `mouseenter/mouseleave` events instead.

Events mouseenter and mouseleave

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

But there are two differences:

1. Transitions inside the element are not counted.
2. Events `mouseenter/mouseleave` do not bubble.

These events are intuitively very clear.

When the pointer enters an element – the `mouseenter` triggers, and then doesn't matter where it goes while inside the element. The `mouseleave` event only triggers when the cursor leaves it.

If we make the same example, but put `mouseenter/mouseleave` on the blue `<div>`, and do the same – we can see that events trigger only on entering and leaving the blue `<div>`. No extra events when going to the red one and back. Children are ignored.

http://plnkr.co/edit/f5nodDg34O7ctGxOeb8Y?p=preview

Event delegation

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

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

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

Handlers for `mouseenter/leave` on `<table>` only trigger on entering/leaving the whole table. It's impossible to get any information about transitions inside it.

Not a problem – let's use `mouseover/mouseout`.

A simple handler may look like this:

```
// let's highlight cells under mouse
table.onmouseover = function(event) {
  let target = event.target;
  target.style.background = 'pink';
};

table.onmouseout = function(event) {
  let target = event.target;
  target.style.background = '';
};
```

These handlers work when going from any element to any inside the table.

But we'd like to handle only transitions in and out of `<td>` as a whole. And highlight the cells as a whole. We don't want to handle transitions that happen between the children of `<td>`.

One of solutions:

- Remember the currently highlighted `<td>` in a variable.
- On `mouseover` – ignore the event if we're still inside the current `<td>`.
- On `mouseout` – ignore if we didn't leave the current `<td>`.

That filters out "extra" events when we are moving between the children of `<td>`.

The details are in the full example .

Summary

We covered events `mouseover`, `mouseout`, `mousemove`, `mouseenter` and `mouseleave`.

Things that are good to note:

- A fast mouse move can make `mouseover, mousemove, mouseout` to skip intermediate elements.
- Events `mouseover/out` and `mouseenter/leave` have an additional target: `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. They assume that the mouse can be only over one element at one time – the deepest one.
- Events `mouseenter/leave` do not bubble and do not trigger when the mouse goes to a child element. They only track whether the mouse comes inside and outside the element as a whole.

Tasks

Improved tooltip behavior

importance: 5

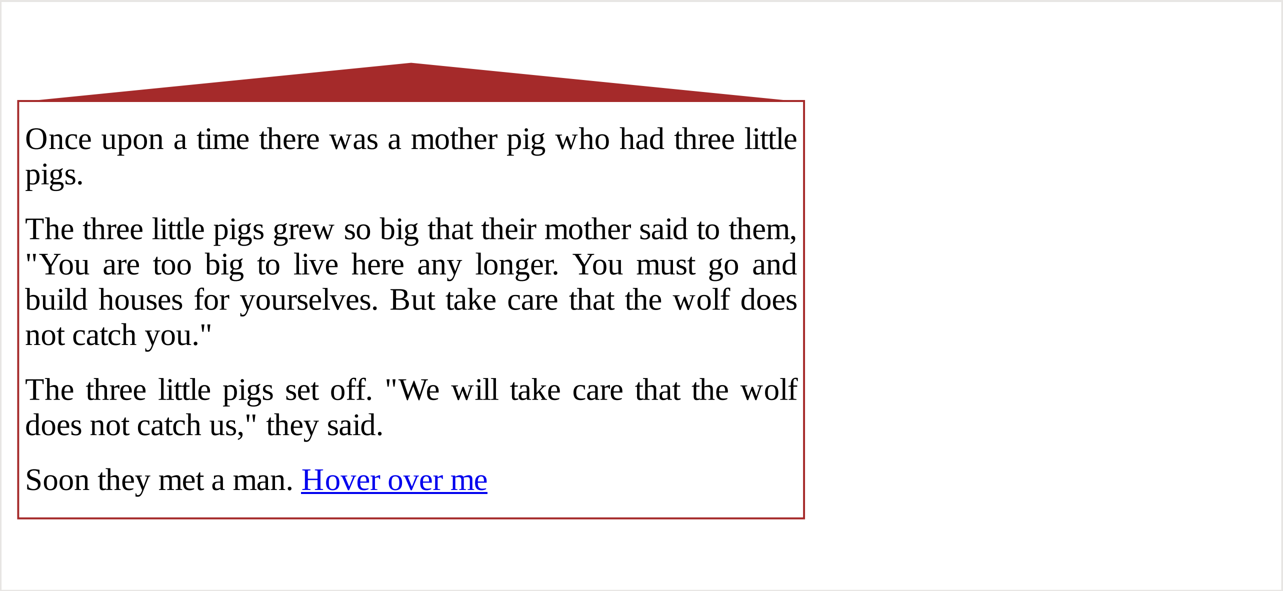
Write JavaScript that shows a tooltip over an element with the attribute `data-tooltip`.

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

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="Rea
</div>
```

The result in iframe:



Once upon a time there was a mother pig who had three little pigs.

The three little pigs grew so big that their mother said to them, "You are too big to live here any longer. You must go and build houses for yourselves. But take care that the wolf does not catch you."

The three little pigs set off. "We will take care that the wolf does not catch us," they said.

Soon they met a man. Hover over me

P.S. Hint: only one tooltip may show up at the same time.

Open a sandbox for the task.

To solution

"Smart" tooltip

importance: 5

Write a function that shows a tooltip over an element only if the visitor moves the mouse *over it*, but not *through it*.

In other words, if the visitor moves the mouse on the element and stopped – show the tooltip. And if they just moved the mouse through fast, 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.

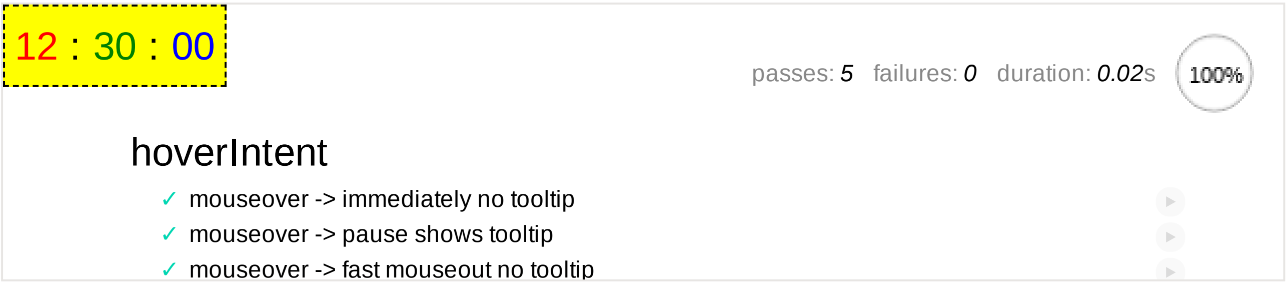
Make a universal object `new HoverIntent(options)` for it. With `options`:

- `elem` – element to track.
- `over` – a function to call if the mouse is slowly moving the element.
- `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:



12 : 30 : 00

passes: 5 failures: 0 duration: 0.02s 100%

hoverIntent

✓ mouseover -> immediately no tooltip

✓ mouseover -> pause shows tooltip

✓ mouseover -> fast mouseout no tooltip

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.

To solution

Drag'n'Drop with mouse events

Drag’n’Drop is a great interface solution. Taking something, dragging and dropping is a clear and simple way to do many things, from copying and moving (see file managers) to ordering (drop into cart).

In the modern HTML standard there’s a section about Drag Events .

They are interesting, because they allow to solve simple tasks easily, and also allow to handle drag’n’drop of “external” files into the browser. So we can take a file in the OS file-manager and drop it into the browser window. Then JavaScript gains access to its contents.

But native Drag Events also have limitations. For instance, we can limit dragging by a certain area. Also we can’t make it “horizontal” or “vertical” only. There are other drag’n’drop tasks that can’t be implemented using that API.

So here we’ll see how to implement Drag’n’Drop using mouse events. Not that hard either.

Drag’n’Drop algorithm

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

1. Catch `mousedown` on a draggable element.

2. Prepare the element to moving (maybe create a copy of it or whatever).
3. Then on `mousemove` move it by changing `left/top` and `position:absolute`.
4. On `mouseup` (button release) – perform all actions related to a finished Drag’n’Drop.

These are the basics. We can extend it, for instance, by highlighting droppable (available for the drop) elements when hovering over them.

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

```
ball.onmousedown = function(event) { // (1) start the process

  // (2) prepare to moving: make absolute and on top by z-index
  ball.style.position = 'absolute';
  ball.style.zIndex = 1000;
  // move it out of any current parents directly into body
  // to make it positioned relative to the body
  document.body.append(ball);
  // ...and put that absolutely positioned ball under the cursor

  moveAt(event.pageX, event.pageY);

  // centers the ball at (pageX, pageY) coordinates
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - ball.offsetWidth / 2 + 'px';
    ball.style.top = pageY - ball.offsetHeight / 2 + 'px';
  }

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

  // (3) move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);

  // (4) drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};
```

If we run the code, we can notice something strange. On the beginning of the drag’n’drop, the ball “forks”: we start to dragging it’s “clone”.

That’s because the browser has its own Drag’n’Drop for images and some other elements that runs automatically and conflicts with ours.

To disable it:

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

Now everything will be all right.

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 cursor 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 positioning

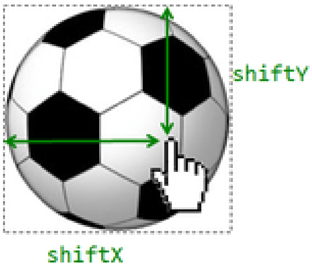
In the examples above the ball is always centered under the pointer:

```
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 can we `mousedown` anywhere on the ball. If do it at the edge, then the ball suddenly “jumps” to become centered.

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 cursor should remain over the edge while dragging.



1. When a visitor presses the button (`mousedown`) – we can remember the distance from the cursor to the left-upper corner of the ball in variables `shiftX/shiftY`. We should 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;
   ```

   Please note that there's no method to get document-relative coordinates in JavaScript, so we use window-relative coordinates here.

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

   ```
   // onmousemove
   // ball has position:absoute
   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);

  // centers the ball at (pageX, pageY) coordinates
  function moveAt(pageX, pageY) {
    ball.style.left = pageX - shiftX + 'px';
    ball.style.top = pageY - shiftY + 'px';
  }

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

  // (3) move the ball on mousemove
  document.addEventListener('mousemove', onMouseMove);
```

```
  // (4) drop the ball, remove unneeded handlers
  ball.onmouseup = function() {
    document.removeEventListener('mousemove', onMouseMove);
    ball.onmouseup = null;
  };

};

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

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

Detecting 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 a user into a trash can or whatever.

Abstractly, we take a "draggable" element and drop it onto "droppable" element.

We need to know the target droppable at the end of Drag'n'Drop – to do the corresponding action, and, preferably, during the dragging process, to highlight it.

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

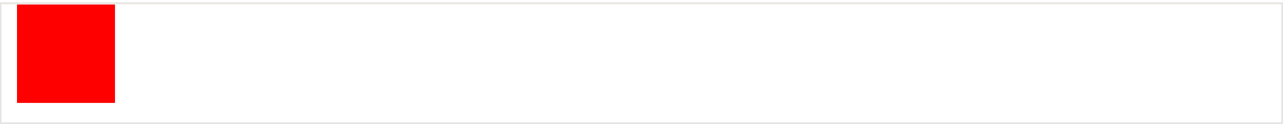
What's the first idea? Probably to put `onmouseover/mouseup` handlers on potential droppables and detect when the mouse pointer appears over them. And then we know that we are dragging/dropping on that element.

But that doesn't work.

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

For instance, below are two `<div>` elements, red on top of blue. 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 in 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 coordinates are out of the window).

So in any of our mouse event handlers we can detect the potential droppable under the pointer like this:

```
// in a mouse event handler
ball.hidden = true; // (*)
let elemBelow = document.elementFromPoint(event.clientX, event.clientY);
ball.hidden = false;
// elemBelow is the element below the ball. If it's droppable, we can handle it.
```

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`.

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

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

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

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
  // 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 lo
  let droppableBelow = elemBelow.closest('.droppable');

  if (currentDroppable != droppableBelow) { // if there are any changes
    // we're flying in or out...
    // note: both values can be null
    //   currentDroppable=null if we were not over a droppable (e.g over an empty
    //   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 gate, the gate is highlighted.

http://plnkr.co/edit/VdbuAVBTO0X3sIR8o66m?p=preview

Now we have the current “drop target” in the variable `currentDroppable` during the whole process and can use it to highlight or any other stuff.

Summary

We considered a basic `Drag'n'Drop` algorithm.

The key components:

1. Events flow: `ball.mousedown` → `document.mousemove` → `ball.mouseup` (cancel native `ondragstart`).
2. At the drag start – remember the initial shift of the pointer relative to the element: `shiftX/shiftY` and keep it during the dragging.

3. Detect droppable elements under the pointer using `document.elementFromPoint`.

We can lay a lot on this foundation.

- On `mouseup` we can 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.

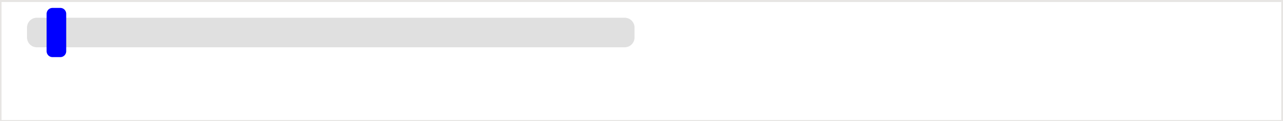
There are frameworks that build architecture over it: `DragZone`, `Droppable`, `Draggable` and other classes. Most of them do the similar stuff to described above, so it should be easy to understand them now. Or roll our own, because you already know how to handle the process, and it may be more flexible than to adapt something else.

✓ Tasks

Slider

importance: 5

Create 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.

To solution

Drag superheroes around the field

importance: 5

This 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.
- Draggable elements 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 window ↗

Open a sandbox for the task. ↗

To solution

Keyboard: keydown and keyup

Before 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 handle 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).

Teststand

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

Keydown and keyup

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

event.code and event.key

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

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

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

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

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

ℹ **“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.

See alphanumeric section of the spec for more codes, or just try 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 – to detect when we have the hotkey.

Please answer the question – in such a listener, should we check the value of `event.key` or `event.code`?

Please, pause and answer.

Made up your mind?

If you’ve got an understanding, then the answer is, of course, `event.code`, as we don’t want `event.key` there. The value of `event.key` can change depending on the language or `CapsLock` enabled. The value of `event.code` is strictly bound to the key, so here we go:

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

Auto-repeat

If a key is being pressed for a long enough time, it starts to 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 all repeating keys the event object has `event.repeat` property set to `true`.

Default actions

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

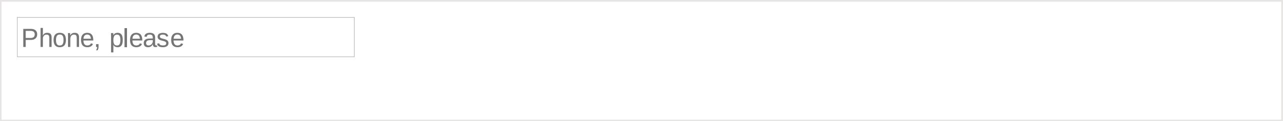
For instance:

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

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

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

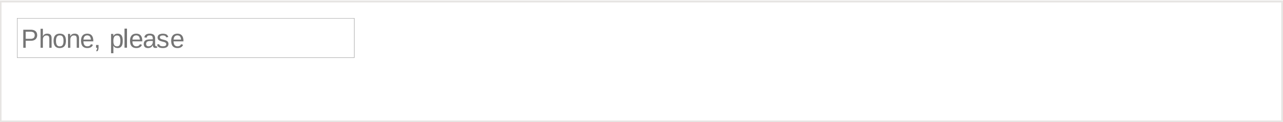
```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' ||
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please" ty
```



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

Let’s relax it a little bit:

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



Now arrows and deletion works well.

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

Legacy

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

There were so many browser incompatibilities that developers of the specification decided to deprecate all of them. The old code still works, as the browser keep supporting them, but there’s totally no need to use those any more.

There was time when this chapter included their detailed description. But as of now we can forget about those.

Summary

Pressing 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 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 input, including mouse or speech recognition.

We should use keyboard events when we really want keyboard. For example, to react on hotkeys or special keys.

✓ Tasks

Extended hotkeys

importance: 5

Create 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 window

To solution

Scrolling

Scroll events allow to react on a page or element scrolling. There are quite a few good things we can do here.

For instance:

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

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

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

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

Prevent scrolling

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

But we can prevent scrolling by `event.preventDefault()` on an event that causes the scroll.

For instance:

- `wheel` event – a mouse wheel roll (a "scrolling" touchpad action generates it too).
- `keydown` event for `pageUp` and `pageDown`.

Sometimes that may help. But there are more ways to scroll, so it's quite hard to handle all of them. So it's more reliable to use CSS to make something unscrollable, like `overflow` property.

Here are few tasks that you can solve or look through to see the applications on `onscroll`.

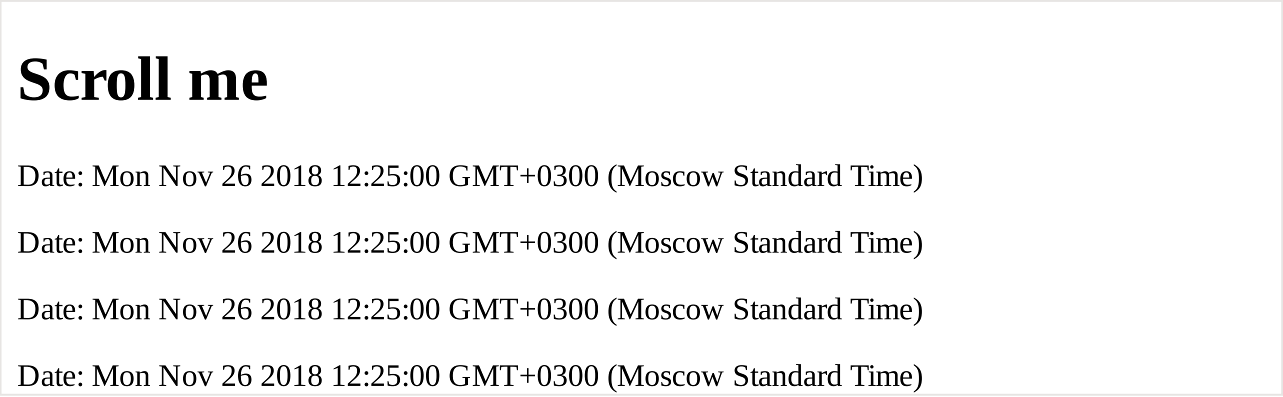
Tasks

Endless page

importance: 5

Create 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:



Scroll me

Date: Mon Nov 26 2018 12:25:00 GMT+0300 (Moscow Standard Time)

Date: Mon Nov 26 2018 12:25:00 GMT+0300 (Moscow Standard Time)

Date: Mon Nov 26 2018 12:25:00 GMT+0300 (Moscow Standard Time)

Date: Mon Nov 26 2018 12:25:00 GMT+0300 (Moscow Standard Time)

Please note two important features of the scroll:

1. **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).
2. **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.

To solution

Up/down button

importance: 5

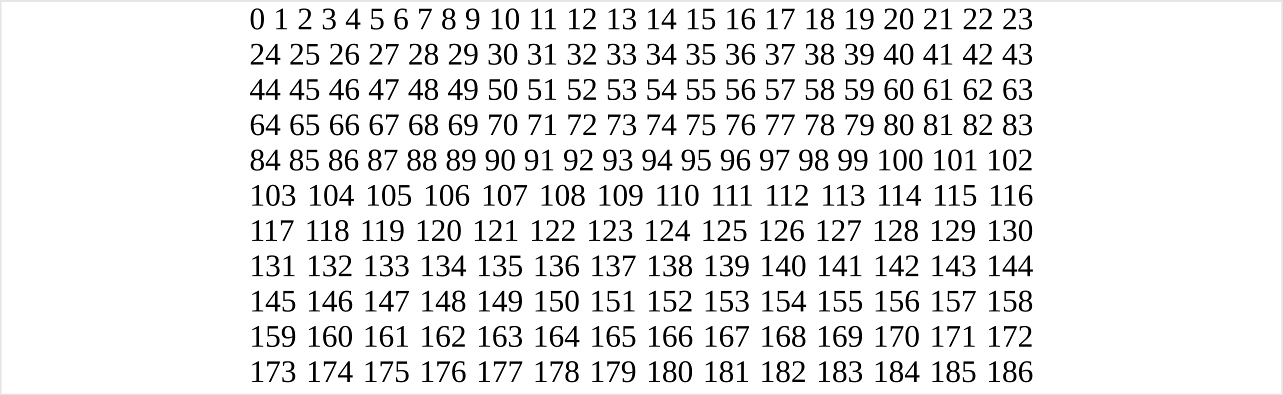
Create a “to the top” button to help with page scrolling.

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:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115 116
117 118 119 120 121 122 123 124 125 126 127 128 129 130
131 132 133 134 135 136 137 138 139 140 141 142 143 144
145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172
173 174 175 176 177 178 179 180 181 182 183 184 185 186

Open a sandbox for the task.

To solution

Load visible images

importance: 4

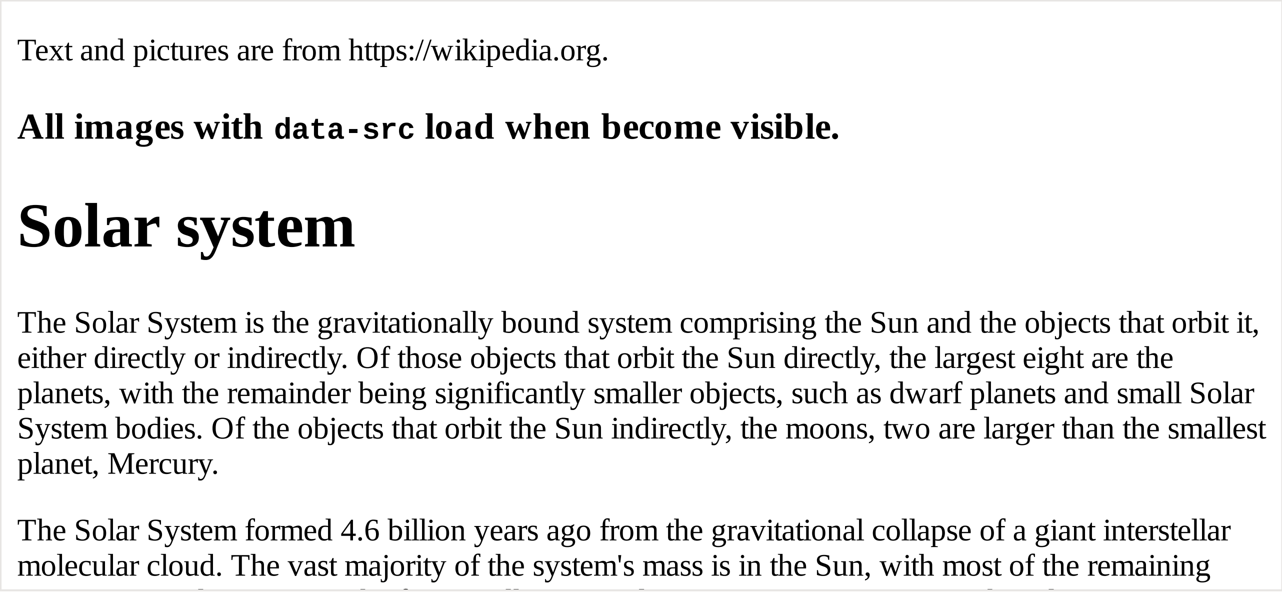
Let’s say we have a slow-speed client and want to save their mobile traffic.

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`:



Text and pictures are from https://wikipedia.org.

All images with `data-src` load when become visible.

Solar system

The Solar System is the gravitationally bound system comprising the Sun and the objects that orbit it, either directly or indirectly. Of those objects that orbit the Sun directly, the largest eight are the planets, with the remainder being significantly smaller objects, such as dwarf planets and small Solar System bodies. Of the objects that orbit the Sun indirectly, the moons, two are larger than the smallest planet, Mercury.

The Solar System formed 4.6 billion years ago from the gravitational collapse of a giant interstellar molecular cloud. The vast majority of the system's mass is in the Sun, with most of the remaining

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.

To solution

Page lifecycle: DOMContentLoaded, load, beforeunload, unload

The lifecycle of an HTML page has three important events:

- `DOMContentLoaded` – the browser fully loaded HTML, and the DOM tree is built, but external resources like pictures `<img>` and stylesheets may be not yet loaded.
- `load` – the browser loaded all resources (images, styles etc).

- `beforeunload/unload` – when 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 – additional resources are loaded, we can get image sizes (if not specified in HTML/CSS) etc.
- `beforeunload/unload` event – the user is leaving: we can check if the user saved the changes and ask them whether they really want to leave.

Let's explore the details of these events.

DOMContentLoaded

The `DOMContentLoaded` event happens on the `document` object.

We must use `addEventListener` to catch it:

```
document.addEventListener("DOMContentLoaded", ready);
```

For instance:

```
<script>
  function ready() {
    alert('DOM is ready');

    // image is not yet loaded (unless was cached), so the size is 0x0
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  }

  document.addEventListener("DOMContentLoaded", ready);
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

In the example the `DOMContentLoaded` handler runs when the document is loaded and does not wait for the image to load. So `alert` shows zero sizes.

At the first sight `DOMContentLoaded` event is very simple. The DOM tree is ready – here's the event. But there are few peculiarities.

DOMContentLoaded and scripts

When the browser initially loads HTML and comes across a `<script>...</script>` in the text, it can't continue building DOM. It must execute the script right now. So `DOMContentLoaded` may only happen after all such scripts are executed.

External scripts (with `src`) also put DOM building to pause while the script is loading and executing. So `DOMContentLoaded` waits for external scripts as well.

The only exception are external scripts with `async` and `defer` attributes. They tell the browser to continue processing without waiting for the scripts. This lets the user see the page before scripts finish loading, which is good for performance.

Scripts with `async` and `defer`

Attributes `async` and `defer` work only for external scripts. They are ignored if there's no `src`.

Both of them tell the browser that it may go on working with the page, and load the script "in background", then run the script when it loads. So the script doesn't block DOM building and page rendering.

There are two differences between them.

	`async`	**`defer`**
Order	Scripts with `async` execute *in the load-first order*. Their document order doesn't matter – which loads first runs first.	Scripts with `defer` always execute *in the document order* (as they go in the document).
`DOMContentLoaded`	Scripts with `async` 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.	Scripts with `defer` execute after the document is loaded and parsed (they wait if needed), right before `DOMContentLoaded`.

So `async` is used for independent scripts, like counters or ads, that don't need to access page content. And their relative execution order does not matter.

While `defer` is used for scripts that need DOM and/or their relative execution order is important.

DOMContentLoaded and styles

External style sheets don't affect DOM, and 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 for the stylesheet to execute:

```
<link type="text/css" rel="stylesheet" href="style.css">
<script>
  // the script doesn't not execute until the stylesheet is loaded
  alert(getComputedStyle(document.body).marginTop);
</script>
```

The reason is that the script may want to get coordinates and other style-dependent properties of elements, like in the example above. Naturally, it has to wait for styles to load.

As `DOMContentLoaded` waits for scripts, it now waits for styles before them as well.

Built-in browser autofill

Firefox, Chrome and Opera autofill forms on `DOMContentLoaded`.

For instance, if the page has a form with login and password, and the browser 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.

One of minor benefits in using `async` and `defer` for external scripts – they don't block `DOMContentLoaded` and don't delay browser autofill.

window.onload

The `load` event on the `window` object triggers when the whole page is loaded including styles, images and other resources.

The example below correctly shows image sizes, because `window.onload` waits for all images:

```
<script>
  window.onload = function() {
    alert('Page loaded');

    // image is loaded at this time
    alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
  };
</script>
```

```html
<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

window.onunload

When a visitor leaves the page, the `unload` event triggers on `window`. We can do something there that doesn't involve a delay, like closing related popup windows. But we can't cancel the transition to another page.

For that we should use another event – `onbeforeunload`.

window.onbeforeunload

If a visitor initiated navigation away from the page or tries to close the window, the `beforeunload` handler asks for additional confirmation.

It may return a string with the question. Historically browsers used to show it, but as of now only some of them do. That's because certain webmasters abused this event handler by showing misleading and hackish messages.

You can try it by running this code and then reloading the page.

```js
window.onbeforeunload = function() {
  return "There are unsaved changes. Leave now?";
};
```

readyState

What happens if we set the `DOMContentLoaded` handler after the document is loaded?

Naturally, it never runs.

There are cases when we are not sure whether the document is ready or not, for instance an external script with `async` attribute loads and runs asynchronously. Depending on the network, it may load and execute before the document is complete or after that, we can't be sure. So we should be able to know the current state of the document.

The `document.readyState` property gives us information about it. 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') {
  document.addEventListener('DOMContentLoaded', work);
} else {
  work();
}
```

There’s a `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.readySta
```

The `readystatechange` event is an alternative mechanics of tracking the document loading state, it appeared long ago. Nowadays, it is rarely used, but let’s cover it for completeness.

What is the place of `readystatechange` among other events?

To see the timing, here’s a document with `<iframe>`, `<img>` and handlers that log events:

```
<script>
  function log(text) { /* output the time and message */ }
  log('initial readyState:' + document.readyState);

  document.addEventListener('readystatechange', () => log('readyState:' + documer
  document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));

  window.onload = () => log('window onload');
</script>

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>
```

```
<img src="http://en.js.cx/clipart/train.gif" id="img">
<script>
  img.onload = () => log('img onload');
</script>
```

The working example is in the sandbox .

The typical output:

1. [1] initial readyState:loading
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

The numbers in square brackets denote the approximate time of when it happens. The real time is a bit greater, but events labeled with the same digit happen approximately at the same time (± a few ms).

- `document.readyState` becomes `interactive` right before `DOMContentLoaded`. These two events 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.

Lifecycle events summary

Page lifecycle events:

- `DOMContentLoaded` event triggers on `document` when DOM is ready. We can apply JavaScript to elements at this stage.
  - All inline scripts and scripts with `defer` are already executed.
  - Async scripts may execute both before and after the event, depends on when they load.
  - Images and other resources may also still continue loading.

- `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.
- `beforeunload` event on `window` triggers when the user wants to leave the page. If it returns a string, the browser shows a question whether the user really wants to leave or not.
- `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.
- `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.

Resource loading: onload and onerror

The browser allows 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 call a function that resides in an external script.

We can load it dynamically, like this:

```
let script = document.createElement('script');
script.src = "my.js";

document.head.append(script);
```

…But how to run the function that is declared inside that script? We need to wait until the script loads, and only then we can call it.

script.onload

The main helper is the `load` event. It triggers after the script 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 helper function "_"
  alert(_); // the function is available
};
```

So in `onload` we can use script variables, run functions etc.

…And what if the loading failed? For instance, there's no such script (error 404) or the server or the server is down (unavailable).

script.onerror

Errors that occur during the loading (but not execution) of the script can be tracked on `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 error details here. We don't know was it error 404 or 500 or something else. Just that the loading failed.

Other resources

The `load` and `error` events also work for other resources. There may be minor differences though.

For instance:

`<img>`, `<link>` (external stylesheets)

Both `load` and `error` events work as expected.

`<iframe>`

Only `load` event when the iframe loading finished. It triggers both for successful load and in case of an error. That’s for historical reasons.

Summary

Pictures `<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.

Tasks

Load images with a callback

importance: 4

Normally, images are loaded when they are created. So i 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.

To solution

Forms, controls

Special properties and events for forms `<form>` and controls: `<input>`, `<select>` and other.

Form properties and methods

Forms and control elements, such as `<input>` have a lot of special properties and events.

Working with forms can be much more convenient if we know them.

Navigation: form and elements

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

That’s a *named* collection: we can use both the name and the number to get the form.

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

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

For instance:

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
</form>

<script>
  // get the form
  let form = document.forms.my; // <form name="my"> element

  // get the element
  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1
</script>
```

There may be multiple elements with the same name, that’s often the case with radio buttons.

In that case `form.elements[name]` is a collection, for instance:

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

<script>
let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0].value); // 10, the first input value
</script>
```

These navigation properties do not depend on the tag structure. All 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 support the `elements` property.

For instance:

```html
<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 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]`.

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:

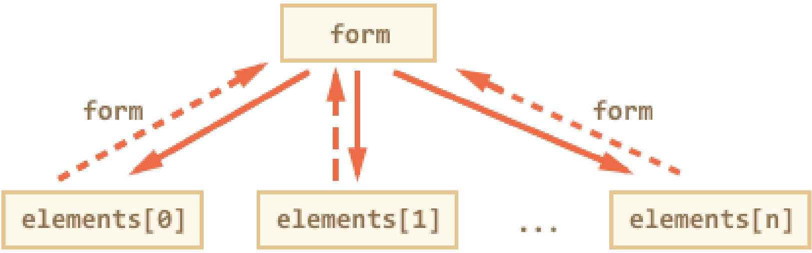
```
<form id="form">
  <input name="login">
</form>

<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

  // the direct access now can use both names: the new one and the old one
  alert(form.username == form.login); // true
</script>
```

That’s usually not a problem, because we rarely change names of form elements.

Backreference: element.form

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

Here’s the picture:



For instance:

```
<form id="form">
  <input type="text" name="login">
</form>

<script>
  // form -> element
  let login = form.login;

  // element -> form
  alert(login.form); // HTMLFormElement
</script>
```

Form elements

Let’s talk about form controls, pay attention to their specific features.

input and textarea

Normally, we can access the value as `input.value` or `input.checked` for checkboxes.

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 we should never use `textarea.innerHTML`: it stores only the HTML that was initially on the page, not the current value.

select and option

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

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

So we have three ways to set the value of a `<select>`:

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

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

Here is an example:

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

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

Unlike most other controls, `<select multiple>` allows multiple choice. In that case we need to walk over `select.options` to get all selected values.

Like this:

```
<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 at https://html.spec.whatwg.org/multipage/forms.html#the-select-element .

new Option

In the specification of the option element there's a nice short syntax to create `<option>` elements:

```
option = new Option(text, value, defaultSelected, selected);
```

Parameters:

- `text` – the text inside the option,
- `value` – the option value,
- `defaultSelected` – if `true`, then `selected` attribute is created,
- `selected` – if `true`, then the option is selected.

For instance:

```
let option = new Option("Text", "value");
// creates <option value="value">Text</option>
```

The same element selected:

```
let option = new Option("Text", "value", true, true);
```

Additional properties of `<option>`

Option elements have additional properties:

`selected`

Is the option selected.

`index`

The number of the option among the others in its `<select>`.

`text`

Text content of the option (seen by the visitor).

Summary

Form 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, or `input.checked` for checkboxes and radio buttons.

For `<select>` we can also get the value by the index `select.selectedIndex` or through the options collection `select.options`. The full specification of this and other elements is at https://html.spec.whatwg.org/multipage/forms.html .

These are the basics to start working with forms. In the next chapter we’ll cover `focus` and `blur` events that may occur on any element, but are mostly handled on forms.

Tasks

Add an option to select

importance: 5

There’s a `<select>`:

```html
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>
```

Use JavaScript to:

1. Show the value and the text of the selected option.
2. Add an option: `<option value="classic">Classic</option>`.
3. Make it selected.

To solution

Focusing: focus/blur

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

Focusing generally means: "prepare to accept the data here", so that's the moment when we can run the code to initialize or load something.

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 here.

Events focus/blur

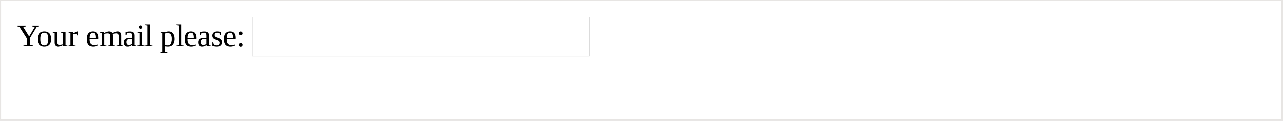
The `focus` event is called on focusing, and `blur` – when the element loses the focus.

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

In the example below:

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

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

Your email please: <input type="email" id="input">

<div id="error"></div>

<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // not email
    input.classList.add('invalid');
    error.innerHTML = 'Please enter a correct email.'
  }
};

input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // remove the "error" indication, because the user wants to re-enter somethir
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```


Your email please:

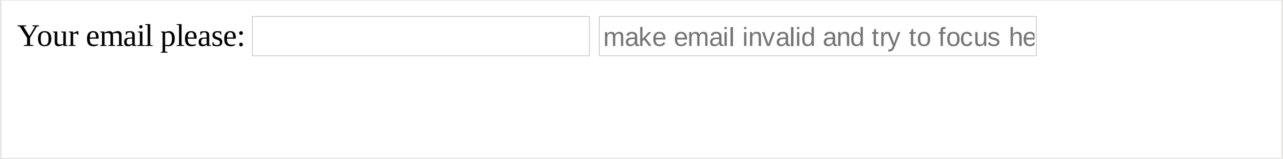
Modern HTML allows to do many validations using input attributes: `required`, `pattern` and so on. And sometimes they are just what we need. JavaScript can be used when we want more flexibility. Also we could automatically send the changed value on the server if it’s correct.

Methods focus/blur

Methods `elem.focus()` and `elem.blur()` set/unset the focus on the element.

For instance, let’s make the visitor unable to leave the input if the value is invalid:

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

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

<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>
```


Your email please: make email invalid and try to focus he

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.

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 evade causing it by ourselves.

Allow focusing on any element: tabindex

By default many elements do not support focusing.

The list varies between browsers, but one thing is always correct: `focus/blur` support is guaranteed for elements that a visitor can interact with: `<button>`, `<input>`, `<select>`, `<a>` and so on.

From the other hand, elements that exist to format something like `<div>`, `<span>`, `<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`.

The purpose of this attribute is to specify the order number of the element when `Tab` 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 us to the second one.

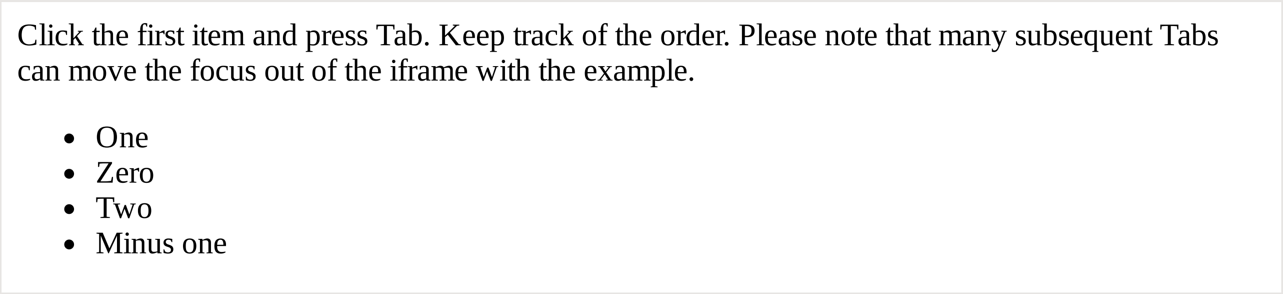
There are two special values:

- `tabindex="0"` makes the element the last one.
- `tabindex="-1"` means that `Tab` should ignore that element.

Any element supports focusing if it has `tabindex`.

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 mar
<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>
```



Click the first item and press Tab. Keep track of the order. Please note that many subsequent Tabs can move the focus out of the iframe with the example.

- One
- Zero
- Two
- Minus one

The order is like this: `1 - 2 - 0` (zero is always the last). Normally, `<li>` does not support focusing, but `tabindex` full enables it, along with events and styling with `:focus`.

`elem.tabIndex` works too

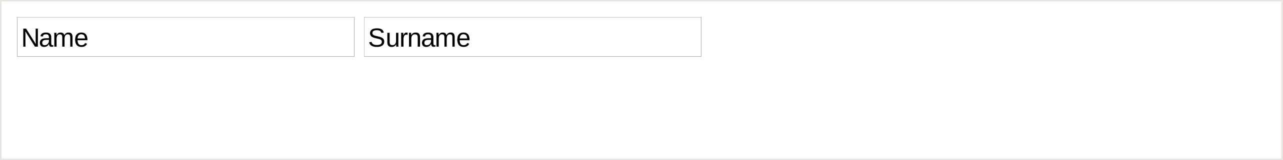
We can add `tabindex` from JavaScript by using the `elem.tabIndex` property. That has the same effect.

Delegation: focusin/focusout

Events `focus` and `blur` do not bubble.

For instance, we can't put `onfocus` on the `<form>` to highlight it, like this:

```
<!-- on focusing in the form -- add the class -->
<form onfocus="this.className='focused'">
  <input type="text" name="name" value="Name">
  <input type="text" name="surname" value="Surname">
</form>

<style> .focused { outline: 1px solid red; } </style>
```



Name

Surname

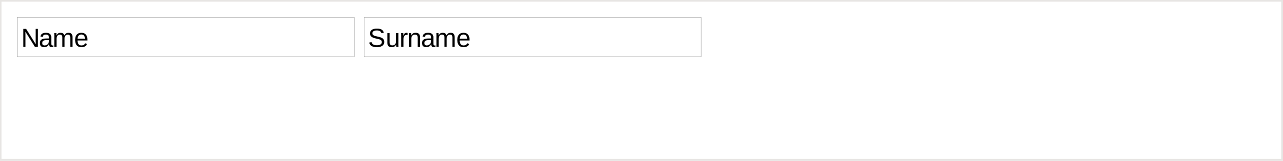
The example above doesn't work, because when user focuses on an `<input>`, the `focus` event triggers on that input only. It doesn't bubble up. So `form.onfocus` never triggers.

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>
```



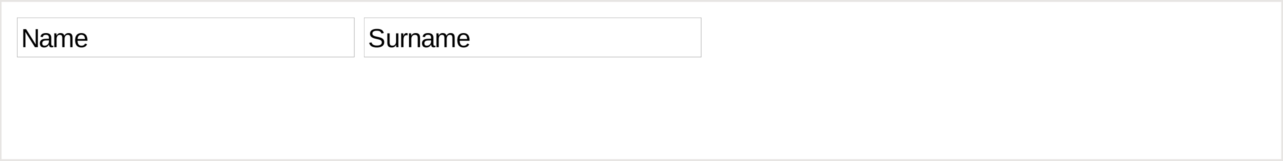
Name Surname

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>
  // put the handler on capturing phase (last argument true)
  form.addEventListener("focusin", () => form.classList.add('focused'));
  form.addEventListener("focusout", () => form.classList.remove('focused'));
</script>
```



Name Surname

Summary

Events `focus` and `blur` trigger on focusing/losing focus on the element.

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`.

✓ Tasks

Editable div

importance: 5

Create a `<div>` that turns into `<textarea>` when clicked.

The textarea allows to edit the HTML in the `<div>`.

When the user presses `Enter` or it looses focus, the `<textarea>` turns back into `<div>`, and its content becomes HTML in `<div>`.

Demo in new window

Open a sandbox for the task.

To solution

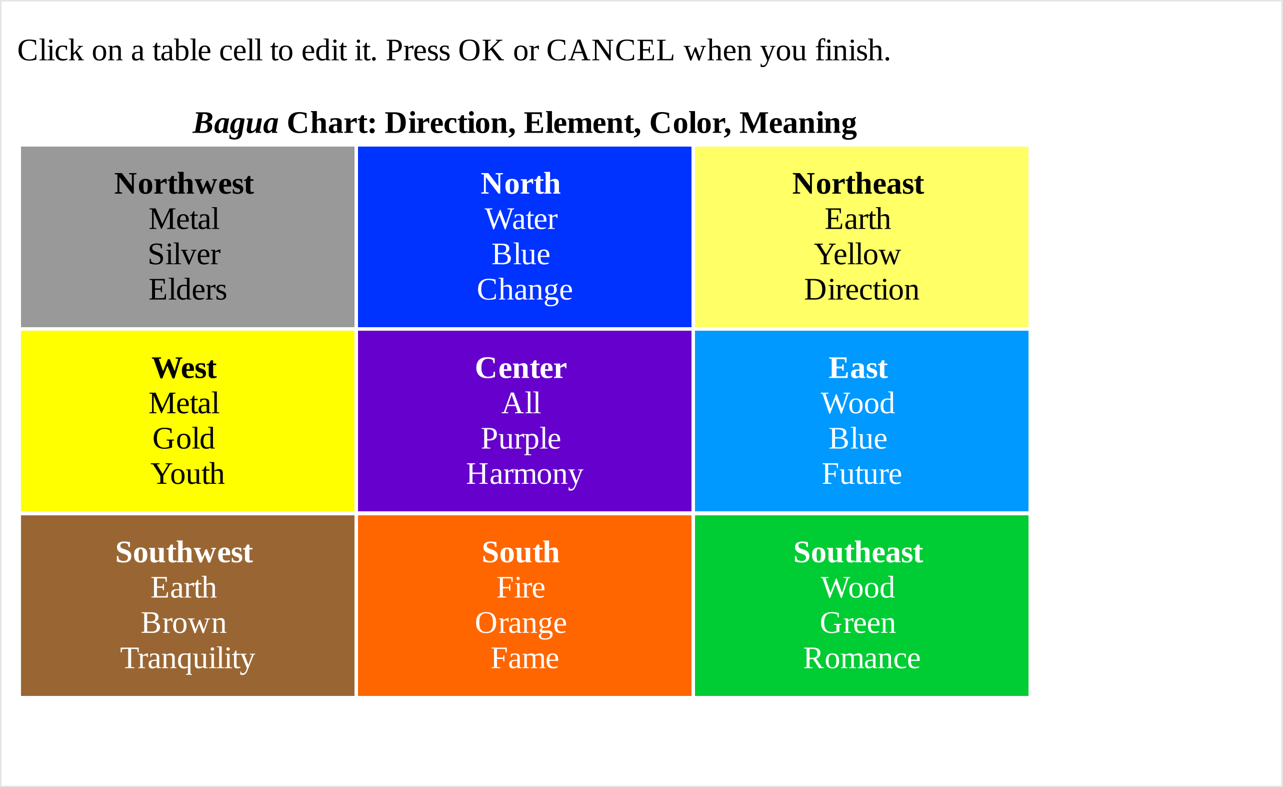
Edit TD on click

importance: 5

Make table cells editable on click.

- On click – the cell should became “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:



Click on a table cell to edit it. Press OK or CANCEL when you finish.

***Bagua* Chart: Direction, Element, Color, Meaning**

Northwest Metal Silver Elders	**North** Water Blue Change	**Northeast** Earth Yellow Direction
West Metal Gold Youth	**Center** All Purple Harmony	**East** Wood Blue Future
Southwest Earth Brown Tranquility	**South** Fire Orange Fame	**Southeast** Wood Green Romance

Open a sandbox for the task.

To solution

Keyboard-driven mouse

importance: 4

Focus on the mouse. Then use arrow keys to move it:

Demo in new window

P.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.

To solution

Events: change, input, cut, copy, paste

Let's discuss various events that accompany data updates.

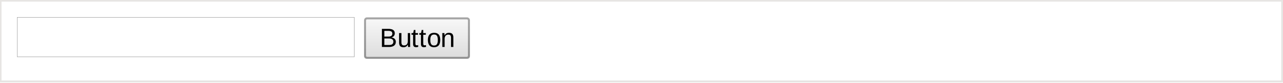
Event: change

The change event triggers when the element has finished changing.

For text inputs that means that the event occurs when it looses 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">
```



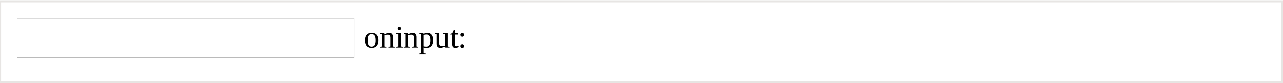
For other elements: `select`, `input type=checkbox/radio` it triggers right after the selection changes.

Event: input

The `input` event triggers every time a value is modified.

For instance:

```
<input type="text" id="input"> oninput: <span id="result"></span>
<script>
  input.oninput = function() {
    result.innerHTML = input.value;
  };
</script>
```


oninput:

If we want to handle every modification of an `<input>` then this event is the best choice.

Unlike keyboard events it works on any value change, even those that does not involve keyboard actions: pasting with a mouse or using speech recognition to dictate the text.

Can’t prevent anything in `oninput`

The `input` event occurs after the value is modified.

So we can’t use `event.preventDefault()` there – it’s just too late, there would be no effect.

Events: cut, copy, paste

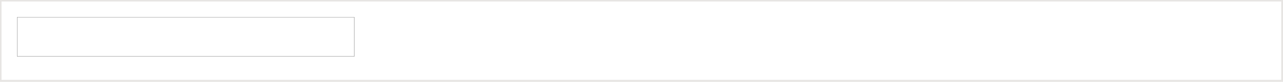
These events occur on cutting/copying/pasting a value.

They belong to ClipboardEvent class and provide access to the data that is copied/pasted.

We also can use `event.preventDefault()` to abort the action.

For instance, the code below prevents all such events and shows what we are trying to cut/copy/paste:

```html
<input type="text" id="input">
<script>
  input.oncut = input.oncopy = input.onpaste = function(event) {
    alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));
    return false;
  };
</script>
```



Technically, we can copy/paste everything. For instance, we can copy and file in the OS file manager, and paste it.

There’s a list of methods in the specification to work with different data types, read/write to the clipboard.

But please note that clipboard is a “global” OS-level thing. Most browsers allow read/write access to the clipboard only in the scope of certain user actions for the safety. Also it is forbidden to create “custom” clipboard events in all browsers except Firefox.

Summary

Data 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 read/write access to the clipboard.

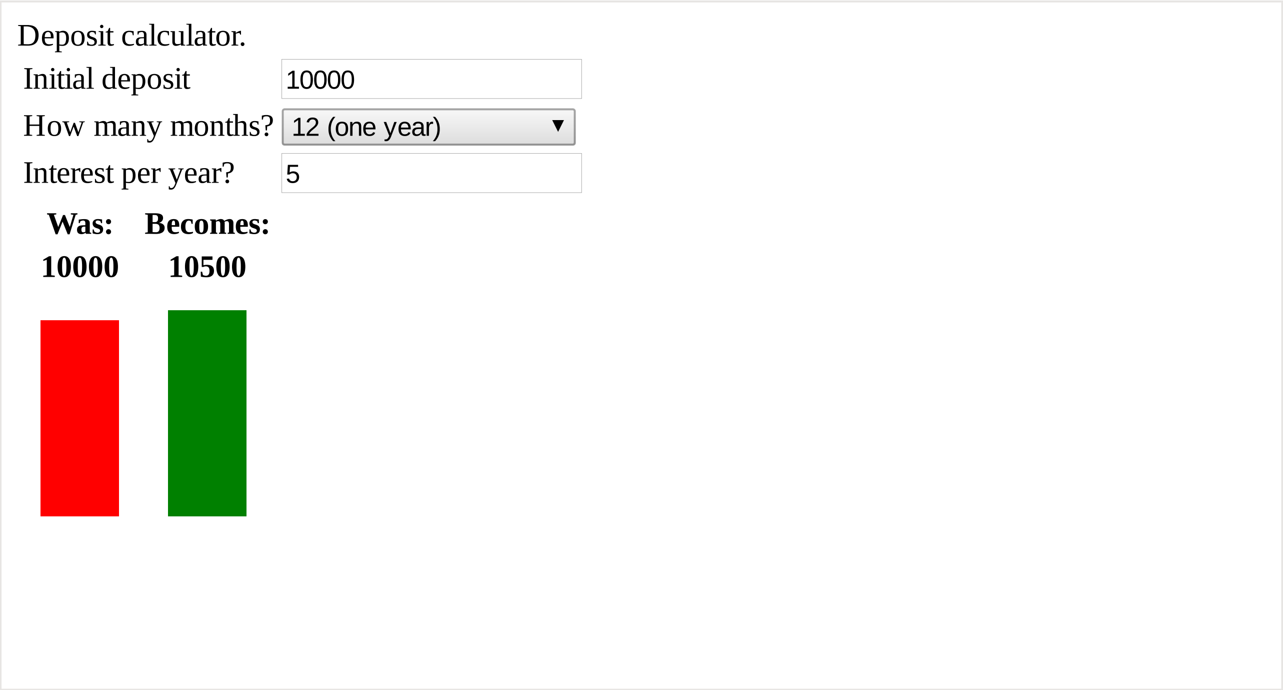
Tasks

Deposit calculator

importance: 5

Create 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:



Deposit calculator.

Initial deposit	10000
How many months?	12 (one year) ▼
Interest per year?	5

Was:	Becomes:
10000	10500

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.

To solution

Form submission: event and method submit

The `submit` event triggers when the form is submitted, it is usually used to validate the form before sending it to the server or to abort the submission and process it in JavaScript.

The method `form.submit()` allows 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: submit

There are two main ways to submit a form:

1. The first – to click `<input type="submit">` or `<input type="image">`.
2. The second – press `Enter` on an input field.

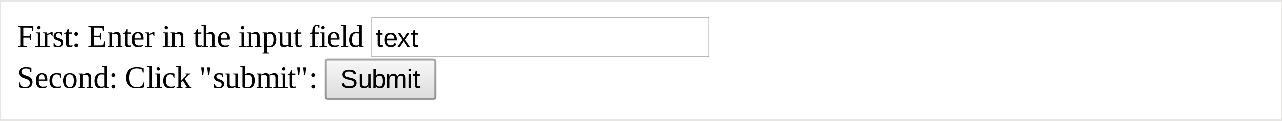
Both actions lead to `submit` event on the form. The handler can check the data, and if there are errors, show them and call `event.preventDefault()`, then the form won't be sent to the server.

In the form below:

1. Go into the text field and press `Enter`.
2. Click `<input type="submit">`.

Both actions show `alert` and the form is not sent anywhere due to `return false`:

```
<form onsubmit="alert('submit!');return false">
  First: Enter in the input field <input type="text" value="text"><br>
  Second: Click "submit": <input type="submit" value="Submit">
</form>
```



First: Enter in the input field text
Second: Click "submit": Submit

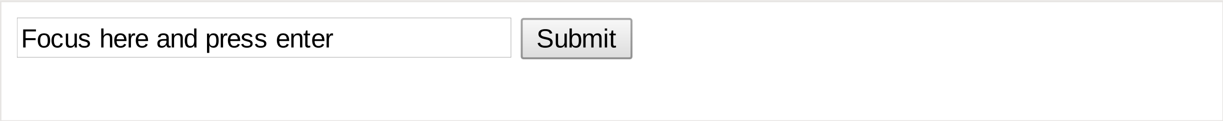
Relation between `submit` and `click`

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:

```html
<form onsubmit="return false">
 <input type="text" size="30" value="Focus here and press enter">
 <input type="submit" value="Submit" onclick="alert('click')">
</form>
```



Focus here and press enter Submit

Method: submit

To 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:

```javascript
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();
```

✓ Tasks

Modal form

importance: 5

Create 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:

Click the button below

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.

To solution

Solutions

Walking the DOM

DOM children

There are many ways, for instance:

The `<div>` 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
```

To formulation

The sibling question

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

Please note that for both cases if there are no children, then there will be an error. For instance, if `elem.lastChild` is `null`, we can't access `elem.lastChild.nextSibling`.

To formulation

Select all diagonal cells

We'll be using `rows` and `cells` properties to access diagonal table cells.

Open the solution in a sandbox.

To formulation

Searching: getElement* and querySelector*

Search for elements

There 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"
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.
// there's no direct query for that
let inputs = form.querySelectorAll('input') // search all
inputs[inputs.length-1] // take last
```

To formulation

Node properties: type, tag and contents

Count descendants

Let’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 it directly from the first child node, 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 `li.getElementsByTagName('li')`.

Open the solution in a sandbox.

To formulation

What's in the nodeType?

There’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>
```

To formulation

Tag in comment

The answer: **`BODY`**.

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

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

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

What’s going on step by step:

1. 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.
2. The comment is now the only child node, so we get it in `body.firstChild`.
3. The `data` property of the comment is its contents (inside `<!--...-->`): `"BODY"`.

To formulation

Where's the "document" in the hierarchy?

We 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
```

For built-in classes in all prototypes there’s a `constructor` reference, and we can get `constructor.name` to see the name of the class. Let’s do it for all objects in the `document` prototype chain:

```
alert(HTMLDocument.prototype.constructor.name); // HTMLDocument
alert(HTMLDocument.prototype.__proto__.constructor.name); // Document
alert(HTMLDocument.prototype.__proto__.__proto__.constructor.name); // No
```

We also could examine the object using `console.dir(document)` and see these names by opening `__proto__`. The console takes them from `constructor` internally.

To formulation

Attributes and properties

Get the attribute

```
<!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>
```

To formulation

Make external links orange

First, 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.

To formulation

Modifying the document

createTextNode vs innerHTML vs textContent

Answer: **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.textContent = text;
  elem3.innerHTML = text;
</script>
```

To formulation

Clear the element

First, 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 = '';
}
```

To formulation

Why does "aaa" remain?

The 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 adds `"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. It shows `"aaa"` before the `<table>`.

The HTML standard specifies in detail how to process bad HTML, and such behavior of the browser is correct.

To formulation

Create a list

Please note the usage of `textContent` to assign the `<li>` content.

Open the solution in a sandbox.

To formulation

Create a tree from the object

The easiest way to walk the object is to use recursion.

1. The solution with innerHTML .
2. The solution with DOM .

To formulation

Show descendants in a tree

To append text to each `<li>` we can alter the text node `data`.

Open the solution in a sandbox.

To formulation

Create a calendar

We'll create the table as a string: `"<table>...</table>"`, and then assign it to `innerHTML`.

The algorithm:

1. Create the table header with `<th>` and weekday names.
2. 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`).

3. First few cells till the first day of the month `d.getDay()` may be empty. Let's fill them in with `<td></td>`.
4. 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>”`.
5. 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.

To formulation

Colored clock with setInterval

First, let's make HTML/CSS.

Each component of the time would look great in its own `<span>`:

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

Open the solution in a sandbox.

To formulation

Insert the HTML in the list

When 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>');
```

To formulation

Sort the table

The solution is short, yet may look a bit tricky, so here I provide it with extensive comments:

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

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

1. Get all `<tr>`, like `table.querySelectorAll('tr')`, then make an array from them, cause we need array methods.

2. The first TR (`table.rows[0]`) is actually a table header, so we take the rest by `.slice(1)`.

3. Then sort them comparing by the content of the first `<td>` (the name field).

4. Now insert nodes in the right order by `.append(...sortedRows)`.

   Tables always have an implicit element, so we need to take it and insert into it: a simple `table.append(...)` would fail.

   Please note: we don't have to remove them, just "re-insert", they leave the old place automatically.

Open the solution in a sandbox.

To formulation

Styles and classes

Create a notification

Open the solution in a sandbox.

To formulation

Element size and scrolling

What's the scroll from the bottom?

The 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.

To formulation

What is the scrollbar width?

To 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:

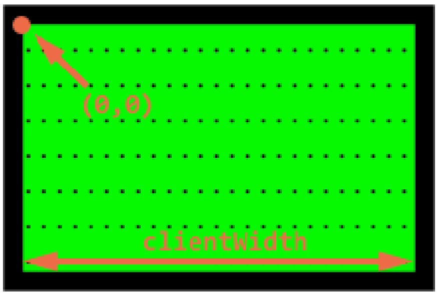
```
// create a div with the scroll
let div = document.createElement('div');

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);
```

To formulation

Place the ball in the field center

The 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:

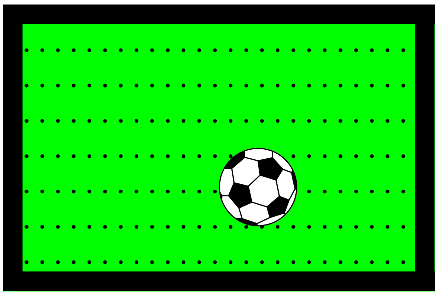


The inner field width/height is `clientWidth/clientHeight`. So the field center has coordinates `(clientWidth/2, clientHeight/2)`.

…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
ball.style.top = Math.round(field.clientHeight / 2 - ball.offsetHeight /
```

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.

In real life after the first load browser usually caches the image, and on next loads it will have the size immediately.

But on the first load the value of `ball.offsetWidth` is `0`. That leads to wrong coordinates.

We should fix that by adding `width/height` to `<img>`:

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

…Or provide the size in CSS:

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

Open the solution in a sandbox.

To formulation

The difference: CSS width versus clientWidth

Differences:

1. `clientWidth` is numeric, while `getComputedStyle(elem).width` returns a string with `px` at the end.
2. `getComputedStyle` may return non-numeric width like `"auto"` for an inline element.
3. `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*.
4. 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.

To formulation

Coordinates

Find window coordinates of the field

Outer corners

Outer 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 corner

That 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.clientT
```

Right-bottom inner corner

In 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.

To formulation

Show a note near the element

In 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.

To formulation

Show a note near the element (absolute)

The 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.

To formulation

Position the note inside (absolute)

Open the solution in a sandbox.

To formulation

Introduction to browser events

Hide on click

Open the solution in a sandbox.

To formulation

Hide self

Can use `this` in the handler to reference “itself” here:

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

To formulation

Which handlers run?

The 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`.

To formulation

Move the ball across the field

First 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.

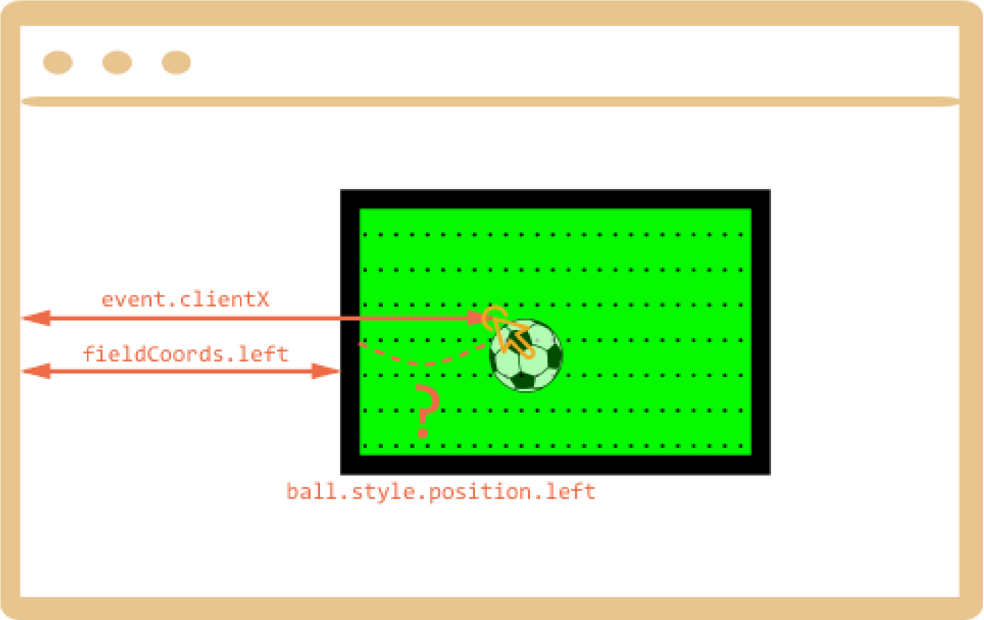
So we should use `position:absolute` and, to make the positioning really solid, make `field` itself positioned.

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.position.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 - fieldInnerCoords.left - field.clientLeft;
```

Normally, `ball.style.position.left` means the “left edge of the element” (the ball). So if we assign that `left`, then the ball edge 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 - fieldInnerCoords.left - field.clientLeft - bal
```

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.

To formulation

Create a menu sliding menu

HTML/CSS

First let’s create HTML/CSS.

A menu is a standalone graphical component on the page, so its better to put it into a single DOM element.

A list of menu items can be layed out as a list `ul/li`.

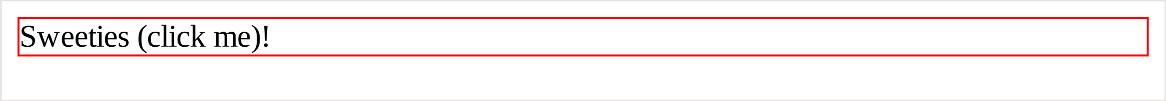
Here’s the example structure:

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

We use `<span>` for the title, because `<div>` has an implicit `display:block` on it, and it will occupy 100% of the horizontal width.

Like this:

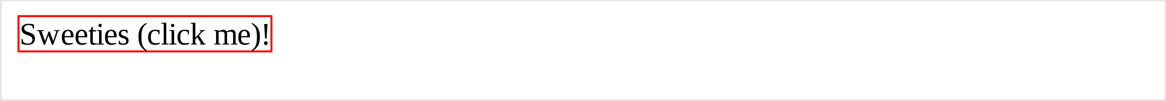
```
<div style="border: solid red 1px" onclick="alert(1)">Sweeties (click me)
```


Sweeties (click me)!

So if we set `onclick` on it, then it will catch clicks to the right of the text.

…but `<span>` has an implicit `display: inline`, so it occupies exactly enough place to fit all the text:

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


Sweeties (click me)!

Toggling the menu

Toggling 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: '▼ ';
}

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

Open the solution in a sandbox.

To formulation

Add a closing button

To 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]<
```

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

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

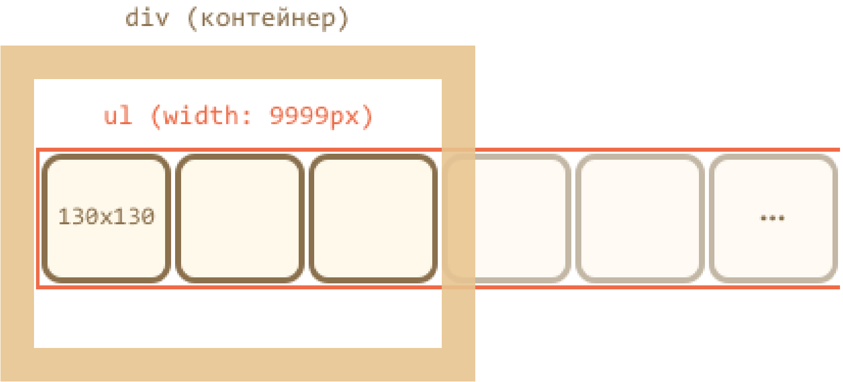
Open the solution in a sandbox.

To formulation

Carousel

The images ribbon can be represented as `ul/li` list of images `<img>`.

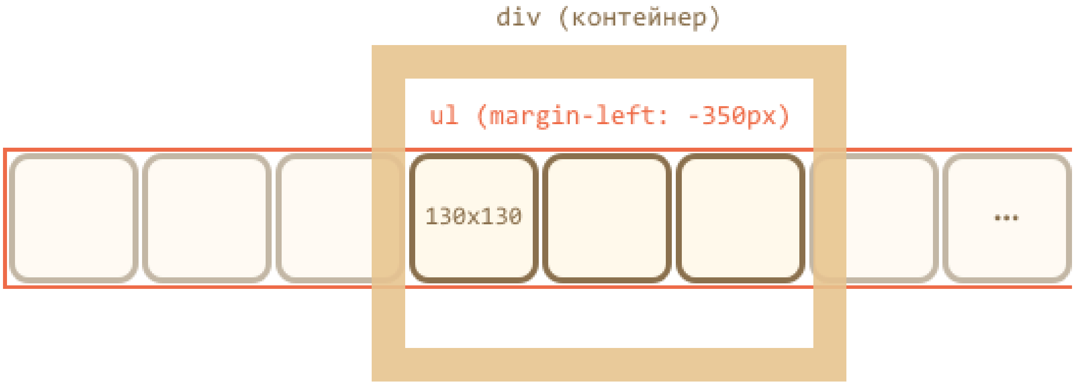
Normally, such a ribbon is wide, but we put a fixed-size `<div>` around to "cut" it, so that only a part of the ribbon is visibble:



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 `<div>` 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 `<div class="carousel">` and style things inside it.

Open the solution in a sandbox.

To formulation

Event delegation

Hide messages with delegation

Open the solution in a sandbox.

To formulation

Tree menu

The solution has two parts.

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

Open the solution in a sandbox.

To formulation

Sortable table

Open the solution in a sandbox.

To formulation

Tooltip behavior

Open the solution in a sandbox.

To formulation

Browser default actions

Why "return false" doesn't work?

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

To formulation

Catch links in the element

That’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.

To formulation

Image gallery

The 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.

To formulation

Mouse events basics

Selectable list

Open the solution in a sandbox.

To formulation

Moving: mouseover/out, mouseenter/leave

Improved tooltip behavior

Open the solution in a sandbox.

To formulation

"Smart" tooltip

The algorithm looks simple:

1. 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.
2. When a mouse cursor entered the element, start measuring the speed on `mousemove`.
3. If the speed is slow, then run `over`.
4. Later if we're out of the element, and `over` was executed, run `out`.

The question is: "How to measure the speed?"

The first idea would be: to run our 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 to mouse events, like `mousemove`.

So we can set a handler on `mousemove` to track coordinates and remember them. Then we can 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.

To formulation

Drag'n'Drop with mouse events

Slider

We have a horizontal Drag'n'Drop here.

To position the element we use `position:relative` and slider-relative coordinates for the thumb. Here it's more convenient here than `position:absolute`.

Open the solution in a sandbox.

To formulation

Drag superheroes around the field

To 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`.

Then, 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.

To formulation

Keyboard: keydown and keyup

Extended hotkeys

We should use two handlers: `document.onkeydown` and `document.onkeyup`.

The set `pressed` should 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.

To formulation

Scrolling

Endless page

The 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()`. And the `bottom` property will be window-relative coordinate of the document end.

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 can’t scroll it any more up.

And the window height is `document.documentElement.clientHeight`.

We want the document bottom be no more than `100px` away from it.

So here’s the function:

```
function populate() {
  while(true) {
    // document bottom
    let windowRelativeBottom = document.documentElement.getBoundingClient

    // if it's greater than window height + 100px, then we're not at the
    // (see examples above, big bottom means we need to scroll more)
    if (windowRelativeBottom > document.documentElement.clientHeight + 10

    // otherwise let's add more data
    document.body.insertAdjacentHTML("beforeend", `<p>Date: ${new Date()}
  }
}
```

Open the solution in a sandbox.

To formulation

Up/down button

Open the solution in a sandbox.

To formulation

Load visible images

The `onscroll` handler should check which images are visible and show them.

We also may want to run it when the page loads, to detect immediately visible images prior to any scrolling and load them.

If we put it at the `<body>` bottom, then it runs when the page content is loaded.

```
// ...the page content is above...

function isVisible(elem) {

  let coords = elem.getBoundingClientRect();

  let windowHeight = document.documentElement.clientHeight;

  // top elem edge is visible OR bottom elem edge is visible
  let topVisible = coords.top > 0 && coords.top < windowHeight;
  let bottomVisible = coords.bottom < windowHeight && coords.bottom > 0;

  return topVisible || bottomVisible;
}

showVisible();
window.onscroll = showVisible;
```

For visible images we can take `img.dataset.src` and assign it to `img.src` (if not did it yet).

P.S. The solution also has a variant of `isVisible` that "pre-loads" images that are within 1 page above/below (the page height is `document.documentElement.clientHeight`).

Open the solution in a sandbox.

To formulation

Resource loading: onload and onerror

Load images with a callback

The algorithm:

1. Make `img` for every source.
2. Add `onload/onerror` for every image.
3. Increase the counter when either `onload` or `onerror` triggers.

4. When the counter value equals to the sources count – we’re done: `callback()`.

Open the solution in a sandbox.

To formulation

Form properties and methods

Add an option to select

The 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>
```

To formulation

Focusing: focus/blur

Editable div

Open the solution in a sandbox.

To formulation

Edit TD on click

1. 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.
2. Set `textarea.value` to `td.innerHTML`.
3. Focus on the textarea.
4. Show buttons OK/CANCEL under the cell, handle clicks on them.

Open the solution in a sandbox. ↗

To formulation

Keyboard-driven mouse

We can use `mouse.onclick` to handle the click and make the mouse “moveable” with `position:fixed`, then 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. ↗

To formulation

Events: change, input, cut, copy, paste

Deposit calculator

Open the solution in a sandbox. ↗

To formulation

Form submission: event and method submit

Modal form

A modal window can be implemented using a half-transparent `<div id="cover-div">` 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 `<div>` 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 `<div>`, but next to it, because we don’t want it to have `opacity`.

Open the solution in a sandbox.

To formulation