16th July 2019

Searching: getElement*, querySelector*

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

There are additional searching methods for that.

document.getElementById or just id

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

We can use it to immediately access the element no matter where it is:

```
<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>
```

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.

If we declare a variable with the same name, it takes precedence:

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

<script>
  let elem = 5;

  alert(elem); // 5
</script>
```

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.

querySelectorAll

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

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

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

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

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

Can use pseudo-classes as well

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

querySelector

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

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

matches

Previous methods were searching the DOM.

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

The method comes in handy when we are iterating over elements (like in 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

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

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

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

For instance:

```
<h1>Contents</h1>

```

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

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

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

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

getElementsBy*

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

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

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

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

For instance:

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

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

```
<table id="table">
  <tr>
    <td>Your age:</td>

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

```
<script>
  let inputs = table.getElementsByTagName('input');

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

Don't forget the "s" letter!

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

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

It returns a collection, not an element!

Another widespread novice mistake is to write:

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

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

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

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

Looking for `.article` elements:

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

<script>
  // find by name attribute
  let form = document.getElementsByName('my-form')[0];

  // find by class inside the form
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, found two elements with class "article"
</script>
```

Live collections

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

In the example below, there are two scripts.

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

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

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

<div>Second div</div>

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

In contrast, `querySelectorAll` returns a *static* collection. It's like a fixed array of elements.

If we use it instead, then both scripts output `1`:

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

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

<div>Second div</div>

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

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

Summary

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

Method	Searches by...	Can call on an element?	Live?
`querySelector`	CSS-selector	✔	-
`querySelectorAll`	CSS-selector	✔	-
`getElementById`	`id`	-	-

Method	Searches by...	Can call on an element?	Live?
`getElementsByName`	`name`	-	✔
`getElementsByTagName`	tag or `'*'`	✔	✔
`getElementsByClassName`	class	✔	✔

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

Besides that:

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

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

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

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.

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)