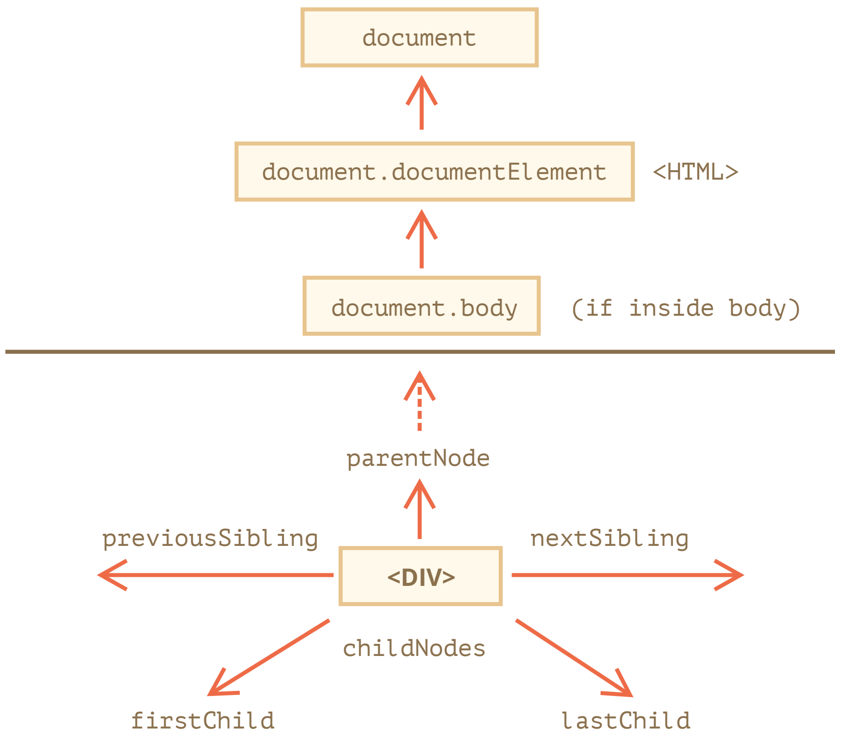
30th November 2019

# Walking the DOM

The DOM allows us to do anything with elements and their contents, but first we need to reach the corresponding DOM object.

All operations on the DOM start with the `document` object. That's the main "entry point" to DOM. From it we can access any node.

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



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 the DOM node of the `<html>` tag.

**`<body> = document.body`**

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

### `<head>` = `document.head`

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

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

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

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

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

```
<html>

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

<body>

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

</body>
</html>
```

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

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

## Children: childNodes, firstChild, lastChild

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

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

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

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

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

…And descendants of `<body>` are not only direct children `<div>`, `<ul>` but also more deeply nested elements, such as `<li>` (a child of `<ul>`) and `<b>` (a child of `<li>`) – the entire subtree.

**The `childNodes` collection lists all child nodes, including text nodes.**

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

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

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

  <div>End</div>

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

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

```
alert( Array.from(document.body.childNodes).filter ); // function
```

### DOM collections are read-only

DOM collections, and even more – *all* navigation properties listed in this chapter are read-only.

We can't replace a child by something else by assigning `childNodes[i] = ...` .

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

### DOM collections are live

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

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

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

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

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

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

## Siblings and the parent

*Siblings* are nodes that are children of the same parent.

For instance, here `<head>` and `<body>` are siblings:

```html
<html>
  <head>...</head><body>...</body>
</html>
```

- `<body>` is said to be the “next” or “right” sibling of `<head>`,
- `<head>` is said to be the “previous” or “left” sibling of `<body>`.

The next sibling is in `nextSibling` property, and the previous one – in `previousSibling`.

The parent is available as `parentNode`.

For example:

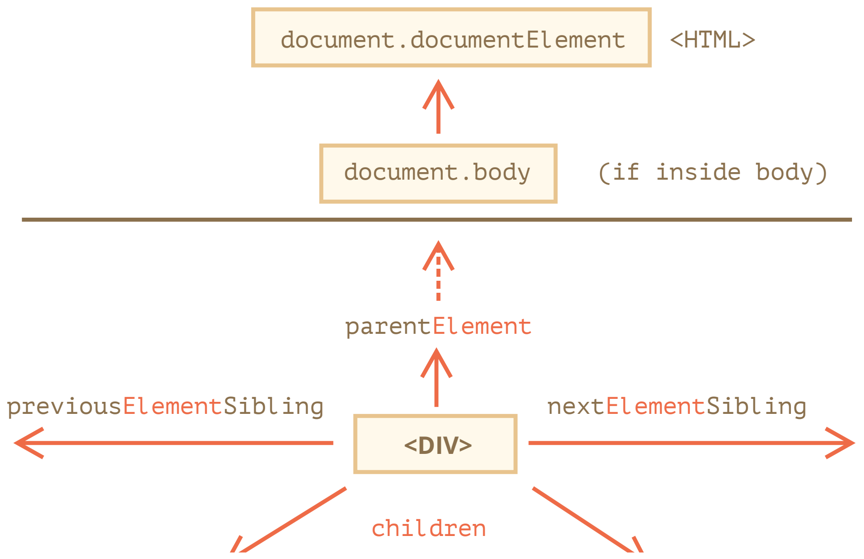
```javascript
// parent of <body> is <html>
alert( document.body.parentNode === document.documentElement ); // true

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

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

## Element-only navigation

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

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

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



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

- `children` – only those children that are element nodes.
- `firstElementChild`, `lastElementChild` – first and last element children.
- `previousElementSibling`, `nextElementSibling` – neighbor elements.
- `parentElement` – parent element.

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

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

With the one exception of `document.documentElement`:

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

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

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

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

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

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

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

  <div>End</div>

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

## More links: tables

Till now we described the basic navigation properties.

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

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

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

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

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

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

**`<tr>`:**

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

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

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

An example of usage:

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

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

The specification: tabular data.

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

## Summary

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

There are two main sets of them:

- For all nodes: `parentNode` , `childNodes` , `firstChild` , `lastChild` , `previousSibling` , `nextSibling` .
- For element nodes only: `parentElement` , `children` , `firstElementChild` , `lastElementChild` , `previousElementSibling` , `nextElementSibling` .

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

## ✓ Tasks

### DOM children

importance: 5

Look at this page:

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

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

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

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` ?

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.

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…)