26th December 2019

Modifying the document

DOM modification is the key to creating “live” pages.

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

Example: show a message

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

Here’s how it will look:

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

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

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 in the HTML or an external CSS file).

Creating an element

To create DOM nodes, there are two methods:

`document.createElement(tag)`

Creates a new *element node* with the given tag:

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

`document.createTextNode(text)`

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

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

Creating the message

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

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

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

Insertion methods

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

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

Here's the full code:

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

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

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

This set of methods provides more ways to insert:

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

- `node.after(...nodes or strings)` –- insert nodes or strings after `node`,
- `node.replaceWith(...nodes or strings)` –- replaces `node` with the given nodes or strings.

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

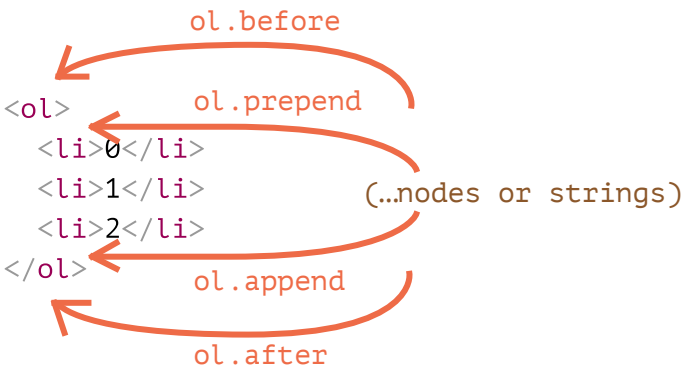
```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // insert string "before" before <ol>
  ol.after('after'); // insert string "after" after <ol>

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

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

before

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

after

Here's a visual picture what methods do:



So the final list will be:

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

```
  <li>2</li>
  <li>append</li>
</ol>
after
```

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

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

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

All text is inserted *as text*.

So the final HTML is:

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

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

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

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

insertAdjacentHTML/Text/Element

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

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

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

The second parameter is an HTML string, that is inserted "as HTML".

For instance:

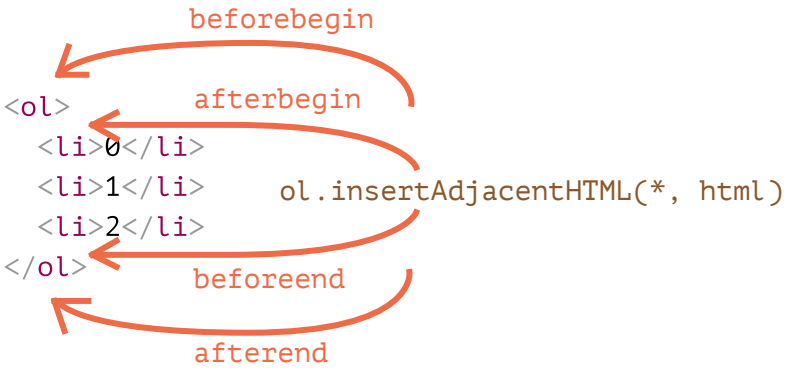
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
  div.insertAdjacentHTML('afterend', '<p>Bye</p>');
</script>
```

…Would lead to:

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

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

Here's the picture of insertion variants:



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

The method has two brothers:

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

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

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

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

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

Node removal

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

Let's make our message disappear after a second:

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

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

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

Please note: if we want to *move* an element to another place – there's no need to remove it from the old one.

All insertion methods automatically remove the node from the old place.

For instance, let's swap elements:

```
<div id="first">First</div>
<div id="second">Second</div>
<script>
  // no need to call remove
  second.after(first); // take #second and after it insert #first
</script>
```

Cloning nodes: 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>
```

DocumentFragment

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

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

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

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>
```

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

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

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

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append + "..." operator = friends!
</script>
```

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

Old-school insert/remove methods

> **Old school**
>
> This information helps to understand old scripts, but not needed for new development.

There are also "old school" DOM manipulation methods, existing for historical reasons.

These methods come from really ancient times. Nowadays, there's no reason to use them, as modern methods, such as `append`, `prepend`, `before`, `after`, `remove`, `replaceWith`, are more flexible.

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

`parentElem.appendChild(node)`

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

The following example adds a new `<li>` to the end of `<ol>`:

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

<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.appendChild(newLi);
</script>
```

`parentElem.insertBefore(node, nextSibling)`

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

The following code inserts a new list item before the second `<li>`:

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.insertBefore(newLi, list.children[1]);
</script>
```

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

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

`parentElem.replaceChild(node, oldChild)`

Replaces `oldChild` with `node` among children of `parentElem`.

`parentElem.removeChild(node)`

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

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

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

<script>
  let li = list.firstElementChild;
  list.removeChild(li);
</script>
```

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

A word about “document.write”

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

The syntax:

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

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

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

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

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

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

For instance:

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

So it's kind of unusable at "after loaded" stage, unlike other DOM methods we covered above.

That's the downside.

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

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

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

Summary

- Methods to create new nodes:
  - `document.createElement(tag)` – creates an element with the given tag,
  - `document.createTextNode(value)` – creates a text node (rarely used),
  - `elem.cloneNode(deep)` – clones the element, if `deep==true` then with all descendants.
- Insertion and removal:
  - `node.append(...nodes or strings)` – insert into `node`, at the end,

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

Text strings are inserted "as text".

- There are also "old school" methods:
  - `parent.appendChild(node)`
  - `parent.insertBefore(node, nextSibling)`
  - `parent.removeChild(node)`
  - `parent.replaceChild(newElem, node)`

  All these methods return `node`.
- Given some HTML in `html`, `elem.insertAdjacentHTML(where, html)` inserts it depending on the value of `where`:
  - `"beforebegin"` – insert `html` right before `elem`,
  - `"afterbegin"` – insert `html` into `elem`, at the beginning,
  - `"beforeend"` – insert `html` into `elem`, at the end,
  - `"afterend"` – insert `html` right after `elem`.

  Also there are similar methods, `elem.insertAdjacentText` and `elem.insertAdjacentElement`, that insert text strings and elements, but they are rarely used.
- To append HTML to the page before it has finished loading:
  - `document.write(html)`

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

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`

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>
```

solution

Why does "aaa" remain?

importance: 1

In the example below, the call `table.remove()` removes the table from the document.

But if you run it, you can see that the text `"aaa"` is still visible.

Why does that happen?

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

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

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

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

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": {
      "apple tree": {},
      "magnolia": {}
    }
  }
};
```

The syntax:

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

The result (tree) should look like this:

- Fish
  - trout
  - salmon
- Tree
  - Huge
    - sequoia
    - oak
  - Flowering
    - apple tree
    - 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.

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.

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.

solution

Colored clock with setInterval

importance: 4

Create a colored clock like here:

13:16:33
Start Stop

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

Open a sandbox for the task.

solution

Insert the HTML in the list

importance: 5

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

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

solution

Sort the table

importance: 5

There's a table:

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

```
    <td>Ann</td><td>Lee</td><td>5</td>
  </tr>
  <tr>
    <td>...</td><td>...</td><td>...</td>
  </tr>
</tbody>
</table>
```

There may be more rows in it.

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

Open a sandbox for the task.

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…)