WDD330 WK2 READING NOTES

1.7 Modifying the document

* 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 a visual picture of what the methods do:
* So the final list will be:
* before
* <ol id="ol">
* <li>prepend</li>
* <li>0</li>
* <li>1</li>
* <li>2</li>
* <li>append</li>
* </ol>
* After

What if we’d like to insert an HTML string “as html”, with all tags and stuff working, in the same manner as elem.innerHTML does it?

elem.insertAdjacentHTML(where, 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.
* "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.

To remove a node, there’s a method node.remove()

IE. setTimeout(() => div.remove(), 1000);

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.**

**IE.** second.after(first); // take #second and after it insert #first

CLONE NODES

* 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.

IE. <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>

4.2 LocalStorage, sessionStorage

Web storage objects localStorage and sessionStorage allow to save key/value pairs in the browser.  data survives a page refresh (for sessionStorage) and even a full browser restart (for localStorage).

Both storage objects provide the same methods and properties:

* setItem(key, value) – store key/value pair.
* getItem(key) – get the value by key.
* removeItem(key) – remove the key with its value.
* clear() – delete everything.
* key(index) – get the key on a given position.
* length – the number of stored items.

As you can see, it’s like a Map collection (setItem/getItem/removeItem), but also allows access by index with key(index).

how to get all saved values or keys? Unfortunately, storage objects are **not iterable**.

let keys = Object.keys(localStorage);

for(let key of keys) {

alert(`${key}: ${localStorage.getItem(key)}`);

}

We can use JSON to store objects though:

localStorage.user = JSON.stringify({name: "John"});

// sometime later

let user = JSON.parse( localStorage.user );

alert( user.name ); // John

Also it is possible to stringify the whole storage object, e.g. for debugging purposes:

// added formatting options to JSON.stringify to make the object look nicer

alert( JSON.stringify(localStorage, null, 2) );

sessionStorage

* The sessionStorage exists only within the current browser tab.
  + Another tab with the same page will have a different storage.
  + But it is shared between iframes in the same tab (assuming they come from the same origin).
* The data survives page refresh, but not closing/opening the tab.
* Triggers on setItem, removeItem, clear calls.
* Contains all the data about the operation (key/oldValue/newValue), the document url and the storage object storageArea.
* Triggers on all window objects that have access to the storage except the one that generated it (within a tab for sessionStorage, globally for localStorage).

13.2 Modules: Import and Export

Can label any declaration as exported:

Ie. // export an array

export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// export a constant

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// export a class

export class User {

constructor(name) {

this.name = name;

}

}

**No semicolons after export class/function**

Ie. export function sayHi(user) {

alert(`Hello, ${user}!`);

} // no ; at the end

Import\*

 if there’s a lot to import, we can import everything as an object using import \* as <obj>, for instance:

// 📁 main.js

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

Can import and export as:

Ie. // 📁 say.js

...

export {sayHi as hi, sayBye as bye};

// 📁 main.js

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

Export default

1. Modules that contain a library, pack of functions, like say.js above.
2. Modules that declare a single entity, e.g. a module user.js exports only class User.
3. Put export default before the entity to export:
4. // 📁 user.js
5. export default class User { // just add "default"
6. constructor(name) {
7. this.name = name;
8. }
9. }
10. And then import it without curly braces:
11. // 📁 main.js
12. import User from './user.js'; // not {User}, just User
13. new User('John');

import needs curly braces for named exports and doesn’t need them for the default one.

| **Named export** | **Default export** |
| --- | --- |
| export class User {...} | export default class User {...} |
| import {User} from ... | import User from ... |

Most of today’s readings were all new to me. Java Script has not been an easy learn for me. I need to be able to code to better grasp the concepts. I’m sure that we will be putting this knowledge into practice soon enough. Is there a better way to grasp these concepts?