{POWER.CODERS}

HTML and the DOM

Contents

HTML & the Document Object Model

DOM

Document Object Model

Anything found in an HTML or XML document can be accessed, changed, deleted, or added by a programmer using the DOM.

The DOM of an HTML document can be represendeted as a nested set of boxes, called DOM Tree.

DOM Tree

The DOM represents a document as a tree structure. HTML elements become **nodes** in that tree.

All nodes in the tree have some kind of relation to each other:

- parent
- child
- sibling

`document`

The `document` object is globally available in your browser.

It allows you to access and manipulate the DOM of the current web page:

1. Find the DOM node you want to change
2. Store this DOM node as a variable
3. Manipulate the DOM node
    - Change its attributes
    - Modify its styles
    - Give it new inner HTML
    - Append new nodes to it

Finding a node

The `document` object is the **root** of the DOM.

```
document.body.textContent = "New text";
```

body is a node inside the DOM and can be accessed using the `document` object.

All HTML elements are **objects** with **properties** and **methods.**

Selecting elements

```
> document.getElementById(id)
> document.getElementsByClassName(classname)
> document.getElementsByTagName(tagname)
```

`document.getElementsByClassName` and `document.getElementsByTagName` return all of the elements as a **list**.

Thus, you can call single elements by their index, e.g. `[0]` or loop through them.

Storing it as a variable

```
let elm = document.getElementById("demo");
```

Each element in the DOM has a set of properties and methods we can use to determine its relationships in the DOM.

Working with DOM

- `elm.childNodes` returns an array of an element's children
- `elm.firstChild` returns the first child node of an element
- `elm.lastChild` returns the last child node of an element
- `elm.hasChildNodes` returns true if there are children
- `elm.nextSibling` returns the next node at the same level
- `elm.previousSibling` returns the previous node
- `elm.parentNode` returns the parent node of an element

Example

```
let elm = document.getElementById("demo");
let arr = elm.childNodes;
arr.forEach(function(el){
  el.textContent = "new text";
});
```

querySelector

- `document.querySelector()` returns the **first element** that matches the specified CSS selector(s)
- `document.querySelectorAll()` returns **all elements** that match the specified CSS selector(s)

```
let arr = document.querySelectorAll("#demo > *");
arr.forEach(function(el){
  el.textContent = "new text";
});
```

Changing attributes

```
<img src="orange.png" id="myimg">
<script>
  let el = document.querySelector("#myimg");
  el.src="apple.png";
  el.className="landscape";
</script>
```

Practicallly all attributes of an element can be changed using JavaScript, e.g. `src`, `href` or `value`.

Changing style

```
<p id="demo">Some text.</p>
<script>
  let el = document.querySelector("#demo");
  el.style.color="#6600FF";
  el.style.width="100px";
  el.style.backgroundColor="red";
</script>
```

All CSS properties can be set and modified using JavaScript. Just remember that you cannot use dashes (-) in the propertey names, e.g. `backgroundColor`, `textDecoration` or `paddingTop`.

Creating new elements

```
<p id="demo">Some <strong>text</strong>.</p>
<script>
  let span = document.createElement("span");
  let node = document.createTextNode("Some new text");
  let parent = document.querySelector("#demo");
  span.appendChild(node);
  parent.appendChild(span);
</script>
```

This creates a new span-tag, appending content to it, and afterwards appending the new element to the existing paragraph.

INSERT METHODS

- `node.appendChild(new_node)` adds a node at the end of the list of children
- `parent.insertBefore(new_node,node)` adds a node right before a child you specify
- `node.insertAdjacentElement(position,new_node)` adds a node into a specified position: `afterbegin`, `afterend`, `beforebegin`, `beforeend`

Removing elements

```
<p id="demo">Some <strong>text</strong>.</p>
<script>
  let parent = document.querySelector("#demo");
  let child = parent.querySelector("strong");
  parent.removeChild(child);
</script>
```

Notice that `querySelector` can also be used as an element method, not only on the `document` object.

Replacing elements

```
<p id="demo">Some <strong>text</strong>.</p>
<script>
  let span = document.createElement("span");
  let node = document.createTextNode("Some new text");
  let parent = document.querySelector("#demo");
  let strong = parent.querySelector("strong");
  span.appendChild(node);
  parent.replaceChild(span, strong);
</script>
```

The code above creates a new `span` including text that replaces the existing `strong`.

DOM SELECTORS

- getElementsByTagName
- getElementsByClassName
- getElementById

- querySelector
- querySelectorAll

- getAttribute
- setAttribute

Adding, replacing, removing

- innerHTML
- innerText
- textContent

- createElement
- createTextNode

- appendChild
- removeChild
- replaceChild

Changing styles

- style.{property} //ok

- className //recommended
- classList //recommended

- classList.add //check canisue.com
- classList.remove //check canisue.com
- classList.toggle //check canisue.com

Debugging

Use of the debugger

A debugger is a tool that runs your program and allows you to pause it and execute it step by step, and inspect variable values at anytime during execution.

It helps you understand what's really going on in your program and reduces guesswork. It' a great tool to learn as it allows to visualize the flow of your program. But it's also very useful for advanced programmers, during developpement of complex applications.

VS Code comes with a way to debug .js files using node (so your script can't call alert() for example as they only exist in the context of a browser). Learn more about it here.

Browser dev tools also include a debugger.

Debugging

To detect and remove exisiting and potential errors (aka bugs) in your code.

It is a very important step in the developer's work. There is no program or application without any bug.

```
console.log("Is often used for debugging");
```

debugger; // stops the execution of JavaScript, like setting a breakpoint

Recommend tutorials on debugging

- Use VS Code: How to set up debugging
- Use `console.log`: Boost your debugging skills
- Use dev tools: Tutorial on YouTube
- Chrome Dev Tools: JavaScript Debugging Reference

JavaScript Events

What are events?

Events are notable things in the DOM that JavaScript detects and can **react** to.

When an event occurs on a target element, a **handler** is executed.

Events are an essential part of a dynamic website.

You can find a complete list of events on w3schools.com.

Common events

Event	Description
`onclick`	occurs when the user clicks on an element
`onload`	occurs when an object has loaded
`onunload`	occurs once a page has unloaded (for `body`)
`onchange`	occurs when the content of a form element changed (for `input`, `select`, `textarea`)
`onmouseover`	occurs when the pointer is moved over an element or its children
`onfocus`	occurs when an element gets focus
`onblur`	occurs when an element loses focus

Handling events

Events can be added to HTML elements as attributes.

```
<h2 onclick="this.innerHTML = 'These are events!'">What are events?</h2>
```

With `onclick` we define that the function we want will be executed. In this case we defined the function directly.

But you can also call functions. Like I did on this title.

Simple event handler

Events handlers can be assigned to elements in the JS file.

```
var h2 = querySelector("h2");
h2.onclick = function() { writeIntoConsole() };

function writeIntoConsole(){
console.log(writeIntoConsole);
alert("Open console!");
}
```

You can attach events to almost all HTML elements.

`onload`

`onload` events can be used if you want to perform actions after the page is loaded.

```
<body onload="writeInConsole()">
```

```
window.onload = function{
writeInConsole();
}
```

Event Listener

Adding events as HTML attributes as well as simple event handlers have one disadvantage:

You can only add **one event handler** to the target element.

The `addEventListener` method in JavaScript allows you to add **many event handlers** (even of the same type) to one element.

An example

```
document.querySelector("h2").addEventListener("click", writeInConsole);
```

- The first parameter is the **event type**, e.g. `click` or `mouseover`. Note that there is no **on**-prefix anymore.
- The second parameter is the **function** that gets called when the event occurs.
- The third paramter (optional) is the **useCapture** boolean defining the order of event handling. Video

Event propagation

Imagine you have a child node `<p>` and a parent node `<section>`. Both have `onclick` events. Now you click on the child. Which function should be executed first?

This is called **event propagation** and it defines which order the event handlers have.

- Bubbling: the inner most event is handled first. This is **default**.
  Bubbling goes UP the DOM.
- Capturing: the outer most event is handled first.
  Capturing goes DOWN the DOM.

Event handling order

If you want to use capturing, set the optional third parameter `useCapture` to **True**.

```
document.querySelector("h2").addEventListener("click", writeInConsole,true);
```

When to use `useCapture`

As the default state is **False** and bubbling propagation is used, the question you probably ask yourself is:

What is capturing propagation good for?

- If you want the click handler on the parent to be executed first.
- If you have the same events handled for multiple elements, e.g. all `input` elements on `focus`.
- On events which do not support bubbling, e.g. `load` and `blur`. More info on MDN.

Removing events

The `removeEventListener` method will remove an event handler which was set using `addEventListener`

```
document.querySelector("h2").removeEventListener("click", writeInConsole);
```

It's exactly the same, just replace `add` with `remove`.

Looking forward to more?

We'll go further with objects, arrays, DOM manipulation and events next week.

But first, let's practice what we learned this week!

Useful hints for your JS project

Built-in objects

- `Math` to perform mathematical tasks
- `Date` to work with dates

Math

```
let pi = Math.PI;
document.write(Math.floor(pi));
document.write(Math.round(pi));
document.write(Math.ceil(pi));

let randomNumber = Math.ceil(Math.random() * 10);
document.write(randomNumber);
```

Date

```
function printTime(){
let currentDate = new Date();
let hours = currentDate.getHours();
let mins = currentDate.getMinutes();
let secs = currentDate.getSeconds();

document.write(hours + ":" + mins + ":" + secs +"\n");
}

setInterval(printTime, 1000); // prints current time each second
```

Online resources

Check what we learned so far on w3schools.com and mdn.com, e.g.

- functions
- objects
- arrays
- DOM manipulation
- events
- Data structures

JavaScript local storage

`localStorage`

`localStorage` allows JavaScript sites and apps to store and access data right **in the browser** with no expiration date.

The data stored in the browser will persist **even after** the browser window has been closed.

5 METHODS

- `setItem()` : Add key and value to `localStorage`
- `getItem()` : Retrieve a value by the key
- `removeItem()` : Remove an item by key
- `clear()` : Clear all `localStorage`
- `key()` : Passed a number to retrieve nth key

Examples

```
window.localStorage.setItem('name', 'Susanne König');
```

```
window.localStorage.getItem('name');
```

```
window.localStorage.removeItem('name');
```

Work with JSON

`localStorage` can only store **strings**. If you want to store an object or an array, use `JSON.stringify()` and `JSON.parse()`

```
const person = {
name: "Susanne König",
location: "Zürich",
}

window.localStorage.setItem('user', JSON.stringify(person));
```

```
JSON.parse(window.localStorage.getItem('user'));
```

When to use

`localStorage` is not that very secure, so do not store sensitive data. It is not a substitute for a database.

- Set flag if overlay / popup was shown once and closed
- Store data in form wizard with several steps. Better to use `sessionStorage`

Online resources

- Can I use localStorage
- localStorage vs sessionStorage
- Guide to using localStorage

Javascript can be dangerous

- innerHTML → Inserts HTML
- outerHTML → Inserts HTML
- insertAdjacentHTML → Inserts HTML
- eval → Evaluates Javascript(!)

  Never use this function
- document.write() → Inserts HTML
- document.writeln() → Inserts HTML

Let's check out some examples, shall we?

Can you spot the issue?

```
let urlDisplayElement = document.getElementById('urlDisplay');
let currentUrl = window.location;
urlDisplayElement.innerHTML = `Current url is: ${currentUrl}`
```

Try and add

```
?<h3>this would be rendered as html</h3>
```

to the URL before you execute the code

What about this?

```
let object = {
  "someProperty": "something"
  "someOtherProperty": "somethingElse"
}

// User shall be able to print the property they want
let userInput = document.getElementById('input').value;

document.getElementById('output').innerText = eval("object." + userInput);
```

- Eval evaluates everything as JavaScript
- `.something; alert('hacked');`
- The line above would generate an alert.

Best practice

Use `innerText` or `textContent`

Always sanitize your inputs

Let's break some things!

Disclaimer: Everything we do shall only be used for testing your **OWN** projects. Do not attempt those things on other websites as this is illegal and may lead to you having to pay fines and even serve jail time.

- Calculator
- Say my name
- What was my URL?

Online resources

- W3schools about DOM
- MDN event reference
- Javascript Key codes
- Can i use ...: always check which browsers are supported
- Codepen: find HTML/CSS and JS snippets online
- Understanding XSS and preventing it using Pure JavaScript
- XSS Prevention Cheat Sheet

JavaScript on the web

JavaScript on the web

Up to now you learned how to do so called static web sites. They are called static because the content cannot be changed.

With JavaScript we make the web sites interactive.

Web use cases

- AJAX loaded content (loading parts of the content without refreshing the site)
- Include external content (e.g. add Twitter feed)
- Form validation and process data
- Overlay elements and lightboxes
- Sliders, tabs and accordions
- Website tracking
- Drawing and animation

Let's have a first try

Code along with me

Internal vs. External

Internal

```
<body>
...
  <script>
    window.alert("Hello World!");
  </script>
...
</body>
```

`<script>` tags can be inserted in head and body.

Each instruction in JS is a **statement**.

Statements are separated by **semicolons**.

Internal vs. External

External

```
<body>
...
  <script src="myScript.js"></script>
...
</body>
```

Normally you place the javascript code into a **external file** and load the file at the **end of the body.**

- It's easier to maintain
- It's faster to load