Programming JS

Margit Tennosaar

DOM

What is the DOM?

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. By learning the DOM, you can make your web pages more interactive and responsive to user actions.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)

HTML events

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

https://www.w3schools.com/jsref/dom_obj_event.asp

JavaScript can change all the HTML elements in the page

JavaScript can change all the HTML attributes in the page

JavaScript can change all the CSS styles in the page

JavaScript can remove existing HTML elements and attributes

JavaScript can add new HTML elements and attributes

JavaScript can react to all existing HTML events in the page

JavaScript can create new HTML events in the page

Invoking functions in HTML

```
<element event="some JavaScript">

<button onclick="alert('Button clicked!')">Click Me!</button>
<button onclick="displayDate()">The time is?</button>

<img src="./image.jpg" onclick="displayDate()"/>
<div onclick="displayDate()">The time is?</div>
```

```
<button onclick="showAlert()">Click Me</button>

function showAlert() {
    alert('Button clicked!');
  }
```

Invoking functions in JavaScript

HTML selectors in JavaScript are used to select and manipulate HTML elements on a webpage.

- getElementById()
- getElementsByClassName()
- getElementsByTagName()
- querySelector()
- querySelectorAll()

With seletors you can easily identify elements based on their unique IDs, classes, tag names, or even complex CSS selectors.

```
// Selects an element with the ID 'myElement’
const element = document.getElementById('myElement’);

// Selects all elements with the class 'myClass’
const elements = document.getElementsByClassName('myClass’);

// Selects all <div> elements
const divs = document.getElementsByTagName('div’);

// Selects the first element that has the class 'myClass’
const firstElementWithMyClass = document.querySelector(‘.myClass');

// Selects the first <p> element inside a <div>
const firstPInDiv = document.querySelector('div > p’);

// Selects all elements with the class 'myClass’
const allElementsWithMyClass = document.querySelectorAll('.myClass');

// Selects all <p> elements inside <div>s
const allPInDivs = document.querySelectorAll('div > p');
```

```
document.getElementById('myButton').addEventListener('click', function () {
  alert('Button clicked!');
});
```

```
document.getElementById('myButton').addEventListener('click’, myFunction);
});
```

Events

1. click
2. mouseover
3. keydown
4. submit
5. load

https://www.w3schools.com/tags/ref_eventattributes.asp

Manipulating content and attributes

✗ The innerText property returns just the text, without spacing and inner element tags.

✗ The innerHTML property returns the text, including all spacing and inner element tags.

✓ The textContent property returns the text with spacing, but without inner element tags.

✗ document.write()

✗ window.alert()

✓ console.log()

Additional DOM manipulation methods

appendChild: Adds a new child element.

removeChild: Removes an existing child element.

setAttribute: Sets or modifies an attribute.

classList Methods: Add, remove, toggle, or check classes.

JavaScript can change all the HTML elements in the page

JavaScript can change all the HTML attributes in the page

JavaScript can change all the CSS styles in the page

JavaScript can remove existing HTML elements and attributes

JavaScript can add new HTML elements and attributes

JavaScript can react to all existing HTML events in the page

JavaScript can create new HTML events in the page

Changing HTML elements

```
element.innerHTML = '<p>New Content</p>’;

element.textContent = 'New Text Content’;

element.innerText = 'Visible Text Content';
```

Changing HTML attributes

```
element.setAttribute('class', 'newClass’);

let className = element.getAttribute('class’);

element.removeAttribute('class');
```

Changing CSS styles

```
element.style.color = 'red';
element.style.fontSize = '16px';


element.className = 'newClassName';

// classList

element.classList.add('newClass');

element.classList.remove('oldClass');
element.classList.toggle('active');

element.classList.contains('active');
```

Removing HTML elements and attributes

```
element.remove();

parentElement.removeChild(childElement);
```

Adding new HTML elements and attributes

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

parentElement.appendChild(newElement);

parentElement.insertBefore(newElement, referenceElement);

parentElement.append(newElement, 'Some text’);

parentElement.prepend(newElement, 'Some text');
```

Forms and user input

Getting Input from a Text Field

```
<input type="text" id="username" placeholder="Enter your name" />
<button id="submitBtn">Submit</button>
<p id="output"></p>




function displayUsername() {
  const inputValue = document.getElementById("username").value;
  document.getElementById("output").textContent = "User entered: " + inputValue;
}

document.getElementById("submitBtn").addEventListener("click", displayUsername);
```

Live Input Tracking

```
<input type="text" id="liveInput" placeholder="Start typing..." />
<p id="liveOutput"></p>




function showLiveText(event) {
  document.getElementById("liveOutput").textContent =
    "You typed: " + event.target.value;
}

document.getElementById("liveInput").addEventListener("input", showLiveText);
```

Handling form submission in JavaScript

```
<form id="userForm">
  <input type="text" id="name" placeholder="Your Name" />
  <input type="email" id="email" placeholder="Your Email" />
  <button type="submit">Submit</button>
</form>
<p id="formOutput"></p>

function processForm(event) {
  event.preventDefault();
  const name = document.getElementById("name").value;
  const email = document.getElementById("email").value;
  document.getElementById("formOutput").textContent =
   `Submitted Name: ${name}, Email: ${email}`;
}

document.getElementById("userForm").addEventListener("submit", processForm);
```

Scope

Scope in JavaScript refers to the accessibility of variables and functions at various parts of your code. It dictates where variables and functions can be accessed or referenced.

- Global Scope: Variables defined outside any function or block are in the global scope and are accessible from anywhere in the code.
- Local (Function) Scope: Variables declared within a function are in the local scope and are only accessible within that function.
- Block Scope (ES6): Introduced in ES6, let and const declarations are block-scoped, meaning they are only accessible within the block they are defined in.

Global scope

```
let globalVar = 'I am a global variable';

function exampleFunction() {
  console.log(globalVar); // Accessible here
}

console.log(globalVar); // Also accessible here
```

Local scope

```
let localVar = 'I am THE local variable’;

function exampleFunction1() {
  localVar = 'I am a local variable one’;
  console.log(localVar); // Accessible here
}

function exampleFunction2() {
  console.log(localVar); // Accessible here
  localVar = 'I am a local variable two’;
  console.log(localVar); // Accessible here
}

exampleFunction1()
console.log(localVar);
exampleFunction2()
console.log(localVar);
exampleFunction1()
console.log(localVar);
```

Block scope

```
if (true) {
  let blockVar = 'I am a block-scoped variable';
  console.log(blockVar); // Accessible here
}

console.log(blockVar); // Uncaught ReferenceError: blockVar is not defined
```

Best practices

- Prefer let and const over var to avoid unintended consequences of hoisting.
- Declare variables at the top of their scope for clarity.
- Keep global variables to a minimum to avoid cluttering the global namespace.

Common pitfalls

- Not understanding the difference between var, let, and const.
- Accidentally overwriting global variables due to scope misunderstandings.
- Assuming that block-scoped variables (with let and const) will behave like function-scoped (var) variables.

```
console.log(myLetVar); // ReferenceError: Cannot access 'myLetVar' before initialization
let myLetVar = 5;

console.log(myConstVar); // ReferenceError: Cannot access 'myConstVar' before initialization
const myConstVar = 10;
```