Javascript Notes

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Getting started

1.1 Javascript directly in HTML

Here is an example of how to write a very simple web page that will give a pop-up box saying Hi there!:

```
<html>
    <script type="text/javascript"> alert("Hi there!");
    </script>
</html>
```

1.2 External file

First, we are going to create a separate JavaScript file. These files have the postfix .js. I'm going to call it ch1_alert.js. This will be the content of our file:

```
alert("Saying hi from a different file!");
```

Then, in an html file

```
<html>
    <script type="text/javascript" src="ch1_alert.js"></script>
</html>
```

1.3 Running javascript like a script with node

You can run JavaScript code in the terminal using Node.js.

```
console.log("Hello world")
```

```
node script.js
```

1.4 Semicolons

After every statement, you should insert a semicolon. JavaScript is very forgiving and will understand many situations in which you have forgotten one

1.5 Comments

Js has two types of comments, single line // and multi line /* */

1.6 Prompts

Another thing we would like to show you here is also a command prompt. It works very much like an alert, but instead, it takes input from the user

```
prompt("Enter something")
```

1.7 Random

We can generate random reals between 0 and 1 with

```
Math.random()
```

Multiply by 100 to get reals between 0 and 100

```
Math.random() * 100
```

Use the floor function to truncate to integers

```
Math.floor(Math.random() * 100)
```

Essentials

2.1 Variables

2.1.1 let, var, and const

A variable definition consists of three parts: a variable-defining keyword (let, var, or const), a name, and a value. Let's start with the difference between let, var, or const. Here you can see some examples of variables using the different keywords:

```
let nr1 = 12;
var nr2 = 8;
const PI = 3.14159;
```

let and var are both used for variables that might have a new value assigned to them somewhere in the program. The difference between let and var is complex. It is related to scope.

var has global scope and let has block scope. var's global scope means that you can use the variables defined with var in the entire script. On the other hand, let's block scope means you can only use variables defined with let in the specific block of code in which they were defined

On the other hand, const is used for variables that only get a value assigned once— for example, the value of pi, which will not change. If you try reassigning a value declared with const, you will get an error:

2.2 Types

2.2.1 Strings

- Double quotes
- Single quotes
- **Backticks:** special template strings in which you can use variables directly

  Aka string interpolation

```
name = "Nate"
console.log(`Hello ${name}`)
```

2.2.2 Number

The number data type is used to represent, well, numbers. In many languages, there is a very clear difference between different types of numbers. The developers of JavaScript decided to go for one data type for all these numbers: number. To be more precise, they decided to go for a 64-bit floating-point number. This means that it can store rather large numbers and both signed and unsigned numbers, numbers with decimals, and more.

2.2.3 BigInt

A BigInt data type can be recognized by the postfix n:

```
let big = 123n
```

Cannot mix BigInt and other types, use explicit conversions

2.2.4 Boolean

true or false

2.2.5 Undefined

It has a special data type for a variable that has not been assigned a value. And this data type is undefined:

```
let unassigned;
console.log(unassigned);
```

We can also purposefully assign an undefined value. It is important you know that it is possible, but it is even more important that you know that manually assigning undefined is a bad practice:

```
let x = undefined
```

2.2.6 null

```
let empty = null;
```

2.2.7 Getting type

We use typeof

2.2.8 Converting data types

We have String(), Number(), and Boolean()

Operators

- +,-,/,*, **, %
- **Postfix, Prefix**
- ==: Compares value and does type conversions
- ===: Compares value and type, does not do any conversions
- !=
- !==
- <, >, <=, >=
- &&: And
- ||: Or
- !: Not
- []: Index operator

Arrays and their properties

Arrays are lists of values. These values can be of all data types and one array can even contain different data types

4.1 Creating arrays

```
let arr = [1,2,3]
let arr = new Array(3)  // Array of three undefined values
```

```
let arr = [1,2,3,4]
console.log(arr[-1]) // Undefined

arr[-1] = "What?"
console.log(arr[-1]) // What?
```

4.1.1 Methods and member variables

Member variables:

- **.length**

Methods

- **push(...items)**: Adds one or more elements to the end of the array.
- **pop()**: Removes the last element from the array and returns that element.
- **shift()**: Removes the first element from the array and returns that element.
- **unshift(...items)**: Adds one or more elements to the beginning of the array.
- **splice(start, deleteCount, ...items)**: Adds or removes items from the array at the specified index.
- **sort(compareFunction?)**: Sorts the elements of the array in place and returns the sorted array.
- **reverse()**: Reverses the order of the array elements in place.
- **fill(value, start?, end?)**: Fills elements in the array with the specified value.
- **copyWithin(target, start, end?)**: Copies a sequence of elements within the array to another position within the same array.
- **concat(...arrays)**: Returns a new array that is the result of merging two or more arrays.
- **slice(begin?, end?)**: Returns a shallow copy of a portion of an array into a new array.

- **join(separator?)**: Joins all elements of an array into a string using the specified separator.
- **toString()**: Returns a string representing the specified array and its elements.
- **toLocaleString()**: Returns a localized string representing the array and its elements.
- **forEach(callback(currentValue, index, array))**: Executes a provided function once for each array element.
- **map(callback(currentValue, index, array))**: Creates a new array with the results of calling a provided function on every element.
- **filter(callback(currentValue, index, array))**: Creates a new array with all elements that pass the test implemented by the provided function.
- **reduce(callback(accumulator, currentValue, index, array), initialValue?)**: Applies a function against an accumulator and each element to reduce the array to a single value.
- **reduceRight(callback(accumulator, currentValue, index, array), initialValue?)**: Same as reduce(), but processes the array from right-to-left.
- **every(callback(currentValue, index, array))**: Tests whether all elements pass the provided test function.
- **some(callback(currentValue, index, array))**: Tests whether at least one element passes the provided test function.
- **find(callback(currentValue, index, array))**: Returns the first element that satisfies the provided testing function.
- **findIndex(callback(currentValue, index, array))**: Returns the index of the first element that satisfies the provided testing function.
- **indexOf(searchElement, fromIndex?)**: Returns the first index at which a given element can be found.
- **lastIndexOf(searchElement, fromIndex?)**: Returns the last index at which a given element can be found.
- **includes(searchElement, fromIndex?)**: Determines whether an array includes a certain element, returning true or false.
- **flat(depth?)**: Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
- **flatMap(callback(currentValue, index, array))**: First maps each element using a mapping function, then flattens the result into a new array.
- **at(index)**: Returns the item at the given index, supporting negative indices to count back from the end (introduced in ES2022).

Iterators

- **entries()**: Returns a new Array Iterator object that contains key/value pairs.
- **keys()**: Returns a new Array Iterator object that contains the keys for each index.
- **values()**: Returns a new Array Iterator object that contains the values for each index.

Array Static Methods: These methods are called on the Array constructor itself rather than on an instance:

- **Array.from(iterable, mapFunction?, thisArg?)**: Creates a new, shallow-copied Array instance from an array-like or iterable object.
- **Array.isArray(value)**: Determines whether the passed value is an Array.
- **Array.of(...elements)**: Creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

String properties

Objects

Objects in javascript can be created with curly braces, for example

```
let dog = { dogName: "JavaScript",
    weight: 2.4,
    color: "brown",
    breed: "chihuahua",
    age: 3,
    burglarBiter: true
};
```

There are two main ways to index objects,

```
console.log(dog["weight"]);
console.log(dog.weight);
```

Logic statements

7.1 If, else if, else

Uses C syntax

```
if (...) {

} else if (...) {

} else {

}
```

7.2 Ternary statements (conditional operator)

Also uses C syntax

```
flag = true

console.log(flag ? "True!" : "False!")
```

7.3 Switch

Also uses *C* syntax

```
switch (expression) {
    case value1:
        ...
        break;
    ...
    default:
        ...
        break;
}
```

Loops

8.1 While and do while loops

Uses *C* syntax

```
let tt = 5, t =0;
while (tt--) {
    console.log(t++);
}
```

And

```
let flag = true
do {
    console.log("Hello world");
    flag = false;
} while (flag);
```

8.2 For loops

Uses *C* syntax

```
for (let i=0; i<5; ++i) {
    console.log(i);
}
```

8.3 For of loop

There is another loop we can use to iterate over the elements of an array: the for of loop. It cannot be used to change the value associated with the index as we can do with the regular loop, but for processing values it is a very nice and readable loop.

```
let A = [1,2,3]
for (let e of A) {
    console.log(e)
}
```

```
let A = [1,2,3]
for (let e of A) {
    ++e;
}

for (let e of A) {
    console.log(e);
}

/*
1
2
3
*/
```

8.4 For in loop

The `for in` loop gives us the indices of an array, or in the case of an object, the keys

```
let A = {
    name: "Nate",
    lang: "js",
    x: 10
};

for (let e in A) {
    console.log(e, ":", A[e]);
}
```

```
let A = [1,2,3]

for (let e in A) {
    console.log(e)
}

/*
0
1
2
*
```

8.5 Looping over objects with C++ structured binding syntax

```
let A = {
    name: "Nate",
    lang: "js",
    x: 10
};


for (let [k,v] of Object.entries(A)) {
    console.log(k, ":", v);
}
```

8.6 Labels with breaking

In js we can give labels to an area in the program, then specify the name of those labels as the argument of a break statement.

```
outer:
for (let i=0; i<5; ++i) {
    inner:
    console.log("i:", i);
    for (let j=0; j<5; ++j) {
        console.log("j:", j);
        if (j == 2) break outer;
    }
}

/*
i: 0
j: 0
j: 1
j: 2
*/
```

Functions

9.1 Simple functions

```
function f(x,y) {
    console.log(x+y);
}
f(1,2); // 3
```

We can define functions inside of functions

```
function f(x,y) {
    function g() {
        console.log(x+y);
    }
    g();
}
f(1,2); // 3
```

9.2 Assigning functions to variables

We can also initialize a variable with a function definition, for example

```
let f = function() {
    console.log("Hello world");
}
f()
```

9.3 Functions missing arguments

Javascript does not complain, it simply gives NaN

```
function f(x,y) {
    console.log(x+y);
}
f(1) // NaN
```

9.4 Arrow functions (Lambdas)

For one line lambdas

```
let f = x => x + 5; // One parameter
let g = (x,y) => x+y; // Multiple parameters
let h = () => console.log("Hello World"); // No parameters

(() => console.log("Hello world"))();
```

For multi line lambdas

```
(x => {
    console.log("Some function");
    console.log("x is", x);
})(5);

let f = x => {
    return x + 10;
};
let x = f(5);
console.log(x); // 15
```

Notice the set of parenthesis around the top function, this is required for immediately invoked lambdas.

9.5 The spread operator

```
function f(x,y,z) {
    return x + y + z;
}

let A = [1,2,3];
console.log(f(...A)) // 6;
```

or

```
function f(x,y,a,b) {
    return (x*y) + (a*b);
}

let A = [1,2];
let B = [3,4]
console.log(f(...A, ...B)) // 14;
```

9.6 Rest parameter

```
function f(...args) {
    for (let arg of args) {
        console.log(arg);
    }
    console.log(typeof(args)); // Object
}
f(1,2,3,4);
```

```
function f(arg1, ...other) {
    console.log("Arg1 is", arg1);
    for (let arg of other) {
        console.log(arg);
    }
}
f(1,2,3,4);
```

9.7 Hoisting

Hoisting is a process during the compilation phase where JavaScript moves the declarations of functions (and variables, if using var) to the top of their scope. For functions, both the name and the function body are hoisted.

```
f(); // Works because f is hoisted

function f() {
    console.log("Hello world");
}
```

In the above code, the entire function f is hoisted. That means the JavaScript engine is aware of the function f from the very start, allowing it to be called even before the line where f is defined.

9.8 setTimeout and setInterval

`setTimeout(callback, delay, param1?, param2?, ... paramN)` takes a function and a time in milliseconds, it calls the callback, but only after the specified time has elapsed. The optional arguments at the end are passed to the function after the delay.

`setInterval(callback, delay, param1, param2?, ..., paramN)` works similarly to setTimeout, repeatedly calls a function or executes a code snippet, with a fixed time delay between each call.

9.9 Restructuring function return values

For a function that returns an array,

```
function f() {
    return [1,2,3];
}

let [a,b,c] = f();
console.log(a,b,c);
```

We can also retrieve only some of the values

```
function f() {
    return [1,2,3];
}

let [a,b] = f();
console.log(a,b);
```

For objects,

```
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name, lang} = f();
console.log(name, lang);
/*
Nate Javascript
*/
```

and we can also alias the variables

```
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name: n, lang: l} = f();
console.log(n, l);
/*
Nate Javascript
*/
```

We can even get only some of the values

```
function f() {
    return {name: "Nate", lang: "Javascript"};
}

let {name} = f();
console.log(name);
/*
Nate
*/
```

Classes

We can create a class with the syntax

```
class className {
    ...
}
```

We can create constructors with

```
class name {
    constructor(a1,a2,...an) {
        ...
    }
}
```

and create instances of our class with the *new* keyword

```
let obj = new className(a1,a2,...,an)
```

To give a class member variables, we can define them in the constructor, and with the *this* keyword

```
class C {
    constructor() {
        this.x = 15;
    }
};
let c = new C();
console.log(c.x);
```

To create methods, we use the normal function syntax but drop the *function* keyword. We can also define member variables inside methods.

```
class foo {
    constructor() {
        this.x = 15;
    }

    g() {
        this.y = 20;
    }
};

let f = new foo();
console.log(f.x, f.y); // 15 Undefined
f.g();
console.log(f.x, f.y); // 15 20
```

We can only declare member variables (fields) outside of a constructor or method, we cannot define them.

```
class foo {
    x;
    constructor() {
        this.x = 15;
    }
};
let f = new foo();
console.log(f.x);
```

We cannot use *let, var,* or *const* in declaring or defining class fields.

10.1 Private

We can create private fields with the # syntax

```
class foo {
    #x;
    constructor() {
        this.#x = 15;
    }
};
let f = new foo();
console.log(f.x); // Undefined
```

We can also use this syntax to make methods private

```
class foo {
    #print() {
        console.log("Hello world");
    }
};
let f = new foo();
f.print(); // Error
```

10.2 Getters and setters

Getters and setters are special properties that we can use to get data from a class and to set data fields on the class. Getters and setters are computed properties. So, they are more like properties than they are like functions. We call them accessors. They do look a bit like functions, because they have () behind them, but they are not! These accessors start with the get and set keywords.

```
class foo {
    #name;

    set name(name) {
        this.#name = name;
    }

    get name() {
        return this.#name;
    }
};
let f = new foo();
f.x = "Nate";
console.log(f.x); // Nate
```

10.3 Inheritence

We implement inheritance with the *extends* keyword

```
class A {
    x;
    constructor() {
        this.x = 20;
    }

    print() {
        console.log("Hello world");
    }
};

class B extends A {
};

let b = new B();
console.log(b.x); // 20
b.print(); // Hello world
```

Private fields and methods in A are not visible in B

```
class A {
    #x;
    constructor() {
        this.#x = 20;
    }

    print() {
        console.log("Hello world");
    }
};

class B extends A {
};

let b = new B();
console.log(b.x); // Undefined
b.print(); // Hello world
```

10.4 Prototypes

A prototype is the mechanism in JavaScript that makes it possible to have objects. When nothing is specified when creating a class, the objects inherit from the Object.prototype prototype. This is a rather complex built-in JavaScript class that we can use. We don't need to look at how this is implemented in JavaScript, as we can consider it the base object that is always on top of the inheritance tree and therefore always present in our objects

There is a *prototype* property available on all classes, and it is always named "prototype."

Here is how to add a function to this class using prototype

```
class Person {
    constructor(firstname, lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
    }
    greet() {
        console.log("Hi there!");
    }
}

Person.prototype.introduce = function () {
    console.log("Hi, I'm", this.firstname);
};

let p1 = new Person("Bob", "Smith");
p1.introduce(); // Hi, I'm Bob
```

prototype is a property holding all the properties and methods of an object. So, adding a function to prototype is adding a function to the class. You can use prototype to add properties or methods to an object, like we did in the above example in our code with the introduce function. You can also do this for properties

```
Person.prototype.favoriteColor = "green";
```

So the methods and properties defined via prototype are really as if they were defined in the class.

This is something you should not be using when you have control over the class code and you want to change it permanently. In that case, just change the class. However, you can expand existing objects like this and even expand existing objects conditionally. It is also important to know that the JavaScript built-in objects have prototypes and inherit from Object.prototype. However, be sure not to modify this prototype since it will affect how our JavaScript works.

The Document Object Model

11.1 The BOM

The BOM, sometimes also called the window browser object, is the amazing ”magic” element that makes it possible for your JavaScript code to communicate with the browser.

The window object contains all the properties required to represent the window of the browser, so for example, the size of the window and the history of previously visited web pages. The window object has global variables and functions, and these can all be seen when we explore the window object. The exact implementation of the BOM depends on the browser and the version of the browser. This is important to keep in mind while working your way through these sections.

Some of the most important objects of the BOM we will look into in this chapter are:

- History
- Navigator
- Location

You can type the following command in the browser console and press Enter to get information about the window object:

```
0  console.dir(window);
```

The console.dir() method shows a list of all the properties of the specified object. You can click on the little triangles to open the objects and inspect them even more.

The BOM contains many other objects. We can access these like we saw when we dealt with objects, so for example, we can get the length of the history (in my browser) accessing the history object of the window and then the length of the history object, like this

```
0  window.history.length;
```

11.1.1 BOM Objects

We have

- **window**: The window object is the top-level object in a browser and serves as the global context for JavaScript running in the browser. All global variables and functions become properties of the window object.
- **history**: The history object allows you to navigate back and forth through the user's session history.
- **location**: The location object provides information about the current URL and allows you to redirect the browser to a new URL or reload the current page.

- **navigator**: The navigator object exposes properties that provide information about the browser and the underlying operating system.
- **screen**: The navigator object exposes properties that provide information about the browser and the underlying operating system.The screen object contains information about the physical screen of the device. This includes properties like screen.width, screen.height, and screen.availHeight.
- **frames**: Although not an independent object per se, window.frames is a collection that refers to the child windows (or frames) if your document is split into different frames or iframes.

11.1.2 Common window properties and methods

Properties:

- **window.document**: Provides access to the DOM (Document Object Model) of the current page.
- **window.location**: References the location object, which provides details about the current URL (see Location section below).
- **window.history**: Gives access to the history object to navigate user history (see History section below).
- **window.navigator**: Provides details about the browser and the operating system (see Navigator section below).
- **window.innerWidth & window.innerHeight**: Represents the viewport's width and height. Useful for responsive designs and adapting layouts dynamically.
- **window.outerWidth & window.outerHeight**: Offers the total width and height of the browser window including interface elements like toolbars and scrollbars.
- **window.localStorage & window.sessionStorage**: Allow you to store data on the client side persistently or for the session duration, respectively.
- **window.console**: Provides access to the browser's debugging console for logging purposes.
- **window.frames**: Holds a collection of all the frames (or iframes) embedded in the window (further discussed under Frames).

Common Methods:

- **window.alert(message)**: Displays an alert dialog with the specified message.
- **window.confirm(message)**: Opens a modal dialog with OK and Cancel buttons, returning a Boolean based on the user's choice.
- **window.prompt(message, defaultText)**: Shows a dialog prompting the user for input.
- **window.setTimeout(callback, delay)**: Executes a function once after a specified delay (in milliseconds).
- **window.setInterval(callback, interval)**: Repeatedly executes a function with a fixed time delay between each call.

- **window.clearTimeout(timeoutID) & window.clearInterval(intervalID)** Cancel pending timeouts or intervals.
- **window.open(url, name, specs)**: Opens a new browser window or tab.
- **window.close()**: Closes the current window (typically works only if the window was opened by a script).
- **window.addEventListener(event, handler)**: Attaches an event handler to the window (and similarly, removeEventListener to detach).

11.1.3 History Objecti properties and methods

Properties:

- **history.length**: Returns the number of URL entries in the session history.

Methods:

- **history.back()**: Equivalent to clicking the browser's Back button. It moves backward by one in the history stack.
- **history.forward()**: Moves forward by one, similar to clicking the Forward button.
- **history.go(delta)**: Loads a specific page relative to the current page. For example, history.go(-1) is like calling history.back().
- **history.pushState(state, title, url)**: Adds a new state to the browser's session history without reloading the page. This is essential in Single Page Applications (SPAs) for creating navigable states.
- **history.replaceState(state, title, url)**: Modifies the current history entry instead of creating a new one, which is useful for updating the URL or state without affecting the history stack.

11.1.4 Location Object properties and methods

Properties:

- **location.href**: A string containing the full URL of the current page.
- **location.protocol**: The protocol scheme of the URL (e.g., "http:" or "https:").
- **location.host**: Combines the hostname and port (if specified).
- **location.hostname**: The domain name of the web host.
- **location.port**: The port number used by the URL.
- **location.pathname**: The path (directory and file name) part of the URL.
- **location.search**: Contains the query string part of the URL (the portion following the ?).
- **location.hash**: Holds the anchor or fragment identifier of the URL (the portion following the).

Methods:

- **location.assign(url)**: Loads the resource at the specified URL, similar to clicking a link.
- **location.replace(url)**: Loads a new document, replacing the current document in the history. This means the current page won't be available via the Back button.
- **location.reload(forceReload)**: Reloads the current page. Using a truthy value for forceReload forces the page to reload from the server rather than the cache.

11.1.5 Navigator Object properties and methods

Properties:

- **navigator.appName**: Provides the name of the browser (though its use is generally discouraged due to its lack of reliability).
- **navigator.appVersion**: A version string for the browser.
- **navigator.userAgent**: A string that contains details about the browser, its version, and the operating system. This is often used for browser detection.
- **navigator.platform**: Indicates the operating system platform (e.g., "Win32", "MacIntel").
- **navigator.language**: The default language of the browser.
- **navigator.languages**: An array that lists the user's preferred languages.
- **navigator.onLine**: A Boolean indicating whether the browser is online.
- **navigator.cookieEnabled**: Indicates whether cookies are enabled in the browser.

Methods:

- **navigator.javaEnabled()**: Returns a Boolean indicating if the Java plugin is enabled in the browser.
- **navigator.sendBeacon(url, data)**: Used to asynchronously send data to a server, often employed for sending analytics data without blocking unloading of the document.

11.1.6 Screen Object

Properties

- **screen.width**: The total width of the visitor's screen in pixels.
- **screen.height**: The total height of the visitor's screen in pixels.
- **screen.availWidth**: The width of the screen available for the window (excluding interface elements like the taskbar).
- **screen.availHeight**: The height of the screen available for the window.
- **screen.colorDepth**: The number of bits used to display one color (usually 24 or 32).
- **screen.pixelDepth**: Similar to colorDepth, representing the actual color resolution.

11.2 The DOM

The DOM is actually not very complicated to understand. It is a way of displaying the structure of an HTML document as a logical tree

We can inspect the DOM in a similar fashion as we did the others. We execute the following command in the console of our website (again, the document object is globally accessible, so accessing it through the window object is possible but not necessary):

```
0  console.dir(document)
```

11.2.1 Selecting page elements

The document object contains many properties and methods. In order to work with elements on the page, you'll first have to find them. If you need to change the value of a certain paragraph, you'll have to grab this paragraph first. We call this selecting the paragraph. After selecting, we can start changing it.

To select page elements to use within your JavaScript code and in order to manipulate elements, you can use either the querySelector() or querySelectorAll() method. Both of these can be used to select page elements either by tag name, ID, or class

The document.querySelector() method will return the first element within the document that matches the specified selectors. If no matching page elements are found, the result null is returned. To return multiple matching elements, you can use the method document.querySelectorAll().

The querySelectorAll() method will return a static NodeList, which represents a list of the document's elements that match the specified group of selectors. We will demonstrate the usage of both querySelector() and querySelectorAll() with the following HTML snippet:

```
1  <!doctype html>
2  <html>
3      <head>
4          <title>JS Tester</title>
5      </head>
6      <body>
7          <h1 class="output">Hello World</h1>
8          <div class="output">Test</div>
9      </body>
10 </html>
```

We are going to select the h1 element with querySelector(). Therefore, if there is more than one, it will just grab the first

```
    const ele1 = document.querySelector("h1");
    console.dir(ele1);
\end{jscod}
\bigbreak \noindent
If you want to select multiple elements, you can use
 ↪  querySelectorAll(). This
method is going to return all the elements that match the
 ↪  selector in an array. In this
example, we are going to look for instances of the output class,
 ↪  which is done by
prepending the class name with a dot.
\bigbreak \noindent
\begin{jscode}
const eles = document.querySelectorAll(".output");
console.log(eles);
```

After selecting, you can start using the dynamic features of the DOM: you can manipulate the elements using JavaScript. Content can be changed in the same way a variable's contents can be, elements can be removed or added, and styles can be adjusted. This can all be done with JavaScript and the way the user interacts with the page can affect this. We have seen the two most common methods to select in the DOM here, querySelector() and querySelectorAll(). You can actually select any element you might need with these

11.2.2 Basic DOM traversing

We can traverse the DOM using the document object that we saw in the previous chapter. This document object contains all the HTML and is a representation of the web page. Traversing over these elements can get you to the element you need in order to manipulate it

Even for a simple HTML piece there are already multiple ways to traverse the DOM

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Let's find the treasure</h1>
        <div id="forest">
            <div id="tree1">
                <div id="squirrel"></div>
                <div id="flower"></div>
            </div>
            <div id="tree2">
                <div id="shrubbery">
                    <div id="treasure"></div>
                </div>
                <div id="mushroom">
                    <div id="bug"></div>
                </div>
            </div>
        </div>
    </body>
</html>
```

We now want to traverse the DOM of this snippet to find the treasure. We can do this by stepping into the document object and navigating our way from there onwards

We can start by using the body property from the document. This contains everything that's inside the body element

```
console.dir(document.body);
```

We should get a really long object. There are a few ways from this object to get to our treasure. To do so, let's discuss the children and childNodes property.

To get to the treasure using children you would have to use:

```
console.dir(document.body.children.forest.children.tree2.childre
↪ n.
↪ shrubbery.children.treasure);
```

As you can see, on every element we select, we have to select the children again. So, first, we grab the children from the body, then we select forest from these children. Then from forest, we want to grab its children again, and from these children we want to select tree2. From tree2 we want to grab the children again, from these children we need shrubbery. And then finally, we can grab the children from shrubbery and select treasure.

To get to the treasure using childNodes you would have to use your console a lot because text and comment nodes are also in there. childNodes is an array, so you will have to select the right index to select the right child. There is one advantage here: it is a lot shorter because you won't need to select the name separately.

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
childNodes[1]);
```

You could also combine them:

```
console.dir(document.body.childNodes[3].childNodes[3].childNodes[1].
children.treasure);
```

So far, we have seen how we can move down the DOM, but we can also move up. Every element knows its parent. We can use the parentElement property to move back up. For example, if we use the treasure HTML sample and type this into the console:

```
document.body.children.forest.children.tree2.parentElement;
```

We are back at forest, since that is the parent element of tree2. This can be very useful, in particular when combined with functions such as getElementById(), which we will see later in more detail

Not only can we move up and down, we can also move sideways. For example, if we select tree2 like this:

```
document.body.children.forest.children.tree2;
```

We can get to tree1 using:

```
document.body.children.forest.children.tree2.previousElementSibling;
```

And from tree1 we can get to tree2 using:

```
document.body.children.forest.children.tree1.nextElementSibling;
```

As an alternative to nextElementSibling, which returns the next node that is an element, you could use nextSibling, which will return the next node of any type.

11.2.3 Selecting elements as objects

Now we know how to traverse the DOM, we can make changes to the elements. Instead of using console.dir(), we can just type in the path to the element we want to change. We now have the element as a JavaScript object, and we can make changes to all its properties. Let's use a simpler HTML page for this one

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Welcome page</h1>
        <p id="greeting"> Hi! </p>
    </body>
</html>
```

We can traverse to the *p* element, for example, by using this code:

```
document.body.children.greeting;
```

11.2.4 Changing innerText

The innerText property focuses on the text between the opening and closing of the element, like so:

```
<element>here</element>
```

The retrieved value would be here as plain text. For example, if we go to the console and we type

```
document.body.children.greeting.innerText = "Bye!";
```

The message that is displayed on the page changes from Hi! to Bye! immediately. innerText returns the content of the element as plain text, which is not a problem in this case because there is only text in there. However, if there is any HTML inside the element you need to select, or if you want to add HTML, you cannot use this method. It will interpret the HTML as text and just output it on the screen. So if we executed this:

```
document.body.children.greeting.innerText = "<p>Bye!</p>";
```

It will output to the screen Bye! , with the HTML around it, as if it was intended as a text string. To get around this, you need to use innerHTML.

11.2.5 Changing innerHTML

If you did not only want to work with plain text, or perhaps specify some HTML formatting with your value, you could use the innerHTML property instead. This property doesn't just process be plain text, it can also be inner HTML elements

```
document.body.children.greeting.innerHTML = "<b>Bye!</b>";
```

11.2.6 Accessing elements in the DOM

There are multiple methods to select elements from the DOM. After getting the elements, we are able to modify them. In the following sections, we will discuss how to get elements by their ID, tag name, and class name, and by CSS selector.

Instead of traversing it step by step as we just did, we are going to use built-in methods that can go through the DOM and return the elements that match the specifications

Consider

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Just an example</h1>
        <div id="one" class="example">Hi!</div>
        <div id="two" class="example">Hi!</div>
        <div id="three" class="something">Hi!</div>
    </body>
</html>
```

Let's start by accessing elements by ID.

11.2.7 Accessing elements by ID

We can grab elements by ID with the getElementById() method. This returns one element with the specified ID. IDs should be unique, as only one result will be returned from the HTML document

If we want to select the element with an ID of two right away, we could use:

```
document.getElementById("two");
```

This would return the full HTML element:

```
<div id="two" class="example">Hi!</div>
```

To reiterate, if you have more than one element with the same ID, it will just give you back the first one it encounters. You should avoid this situation in your code though.

11.2.8 Accessing elements by tag name

If we ask for elements by tag name, we get an array as a result. This is because there could be more than one element with the same tag name. It will be a collection of HTML elements, or HTMLCollection, which is a special JavaScript object. It's basically just a list of nodes

```
document.getElementsByTagName("div");
```

It returns

```
// HTMLCollection(3) [div#one.example, div#two.example,
↪  div#three. something, one: div#one.example, two:
↪  div#two.example, three: div#three.something]
```

As you can see, all the elements in the DOM with the div tag are returned. You can read what the ID is and what the class is from the syntax. The first ones in the collection are the objects: div is the name, specifies the ID, and . specifies the class. If there are multiple dots, there are multiple classes. Then you can see the elements again (namedItems), this time as key-value pairs with their ID as the key

We can access them using the item() method to access them by index, like this:

```
document.getElementsByTagName("div").item(1);
```

This will return:

```
<div id="two" class="example">Hi!</div>
```

We can also access them by name, using the namedItem() method, like this:

```
document.getElementsByTagName("div").namedItem("one");
```

And this will return:

```
<div id="one" class="example">Hi!</div>
```

When there is only one match, it will still return an HTMLCollection. There is only one h1 tag, so let's demonstrate this behavior:

```
document.getElementsByTagName("h1");
```

This will output:

```
// HTMLCollection [h1]
```

Since h1 doesn't have an ID or class, it is only h1. And since it doesn't have an ID, it is not a namedItem and is only in there once.

11.2.9 Accessing elements by class name

We can do something very similar for class names. In our example HTML, we have two different class names: example and something. If you get elements by class name, it gives back an HTMLCollection containing the results. The following will get all the elements with the class example:

```
document.getElementsByClassName("example");
```

This returns:

```
// HTMLCollection(2) [div#one.example, div#two.example, one:
↪  div#one. example, two: div#two.example]
```

As you can see, it only returned the div tags with the example class. It left out the div with the something class.

11.2.10 Accessing elements with a CSS selector

We can also access elements using a CSS selector. We do this with querySelector() and querySelectorAll(). We then give the CSS selector as an argument, and this will filter the items in the HTML document and only return the ones that satisfy the CSS selector.

The CSS selector might look a bit different than you might think at first. Instead of looking for a certain layout, we use the same syntax as we use when we want to specify a layout for certain elements. We haven't discussed this yet, so we will cover it here briefly.

If we state p as a CSS selector, it means all the elements with tag p. This would look like this:

```
document.querySelectorAll("p");
```

If we say p.example, it means all the p tag elements with example as the class. They can also have other classes; as long as example is in there, it will match. We can also say one, which means select all with an ID of one.

This method is the same result as getElementById(). Which one to use is a matter of taste when all you really need to do is select by ID—this is great input for a discussion with another developer. querySelector() allows for more complicated queries, and some developers will state that getElementById() is more readable. Others will claim that you might as well use querySelector() everywhere for consistency. It doesn't really matter at this point, but try to be consistent.

11.2.11 Using querySelectorAll()

Sometimes it is not enough to return only the first instance, but you want to select all the elements that match the query. For example when you need to get all the input boxes and empty them. This can be done with querySelectorAll():

```
document.querySelectorAll("div");
```

This returns

```
// NodeList(3) [div#one.example, div#two.example,
↪  div#three.something]
```

As you can see, it is of object type NodeList. It contains all the nodes that match the CSS selector. With the item() method we can get them by index, just as we did for the HTMLCollection

11.2.12 Element click handler

HTML elements can do something when they are clicked. This is because a JavaScript function can be connected to an HTML element. Here is one snippet in which the JavaScript function associated with the element is specified in the HTML

```
<!DOCTYPE html>
<html>
    <body>
        <div id="one" onclick="alert('Ouch! Stop it!')">Don't
↪  click here! </div>
    </body>
</html>
```

The onclick attribute is an HTML attribute used to attach JavaScript code to an element. While the alert itself is a JavaScript function, the onclick attribute is an HTML event handler that embeds JavaScript code.

This method of linking HTML and JavaScript is known as using an ”inline event handler.” Although it’s common and simple for small snippets, in larger applications it’s often recommended to separate JavaScript from HTML (for example, using the addEventListener method in JavaScript) for better maintainability and code organization.

Other HTML event handlers are

- **ondblclick:** Fires when an element is double-clicked.
- **onmouseover:** Triggers when the mouse pointer moves over an element.
- **onmouseout:** Occurs when the mouse pointer moves off an element.
- **onmousedown:** Executes when a mouse button is pressed down over an element.

- **onmouseup:** Runs when a mouse button is released over an element.
- **onmousemove:** Fires as the mouse pointer moves within an element.
- **onkeydown:** Activates when the user presses a key down.
- **onkeyup:** Triggers when a key is released.
- **onkeypress:** Similar to onkeydown, but fires for keys that produce a character value (though it's generally less recommended than the keydown/keyup events).
- **oninput:** Fires every time the value of an input or textarea changes—ideal for capturing real-time changes.
- **onchange:** Occurs when the value of an element (such as an input, select, or textarea) has been altered and the element loses focus.
- **onfocus:** Fires when an element, like an input field, gains focus (such as when clicked or tabbed into).
- **onblur:** Triggers when an element loses focus.
- **onload:** Executes when an object (often the window or an image) has completely loaded.
- **onunload:** Occurs when the user leaves the document (commonly used to clean up or save state).
- **onsubmit:** Specifically used with forms, triggering when the form is submitted.
- **onresize:** Fires when the document or an element is resized (this is typically applied to the window)

Next, consider the html code

```
1  <div id="one">Don't click here!</div>
```

This code is at the moment not doing anything if you click it. If we want to dynamically add a click handler to the div element, we can select it and specify the property via the console

```
0  document.getElementById("one").onclick = function () {
1      alert("Auch! Stop!");
2  }
```

11.2.13 This and the DOM

The this keyword always has a relative meaning; it depends on the exact context it is in. In the DOM, the special this keyword refers to the element of the DOM it belongs to. If we specify an onclick to send this in as an argument, it will send in the element the onclick is in.

```
<div class="ysm" onclick="console.log(this)"> Don't click me
↪ </div>
```

logs in the console


```
<div class="ysm" onclick="console.log(this)"> Don't click me
↪ </div>
```


If we instead did the log via a function

```
<div class="ysm" onclick="f()"> Don't click me </div>

<script>
    function f() {
        console.log(this);
    }
</script>
```

We just get the *window* object.

We instead need to pass *this* to the function

```
<div class="ysm" onclick="f(this)"> Don't click me </div>
<script>
    function f(el) {
        console.log(el);
    }
</script>
```

So, the this keyword is referring to the element, and from this element we can traverse the DOM like we just learned. This can be very useful, for example, when you need to get the value of an input box. If you send this, then you can read and modify the properties of the element that triggered the function.

11.2.14 Manipulating element style

After selecting the right element from the DOM, we can change the CSS style that applies to it. We can do this using the style property.

We are going to make a button that will toggle the appearing and disappearing of a line of text. To hide something using CSS, we can set the display property of the element to none.

```
<div class="ysm" id="someDiv" onclick="f()"> Don't click me
↪ </div>
<script>
    function f() {
        let div = document.getElementById("someDiv");
        if (div.style.display === "none") {
            div.style.display = "block";
        } else {
            div.style.display = "none";
        }
}
</script>
```

11.2.15 Changing the classes of an element

HTML elements can have classes, and as we have seen, we can select elements by the name of the class. As you may remember, classes are used a lot for giving elements a certain layout using CSS

With JavaScript, we can change the classes of HTML elements, and this might trigger a certain layout that is associated with that class in CSS. We are going to have a look at adding classes, removing classes, and toggling classes.

11.2.16 Adding and removing classes to elements

we use the classList property with the .add(classname) method.

```
<style> .big { transform: scale(2); } </style>

<div class="ysm" id="someDiv" onclick="f()"> Click to make big
↪ </div>
<script>
function f() {
    let p =
↪ document.querySelector("#someDiv").classList.add("big");
}
</script>
```

To remove, we use .remove(classname)

```
<style> .big { transform: scale(2); } </style>

<div class="ysm" id="someDiv" onclick="f()"> Click to make big
↪ </div>
<script>
    function f() {
        let p =
↪ document.querySelector("#someDiv").classList.toggle("big");
    }
</script>
```

11.2.17 Toggling classes

We can streamline the above system by using the .toggle method, which adds a class if it doesn't have it, and removes it if it does.

```
<style> .big { transform: scale(2); } </style>

<div class="ysm" id="someDiv" onclick="f()"> Click to make big
↪ </div>
<script>
    function f() {
        let p =
↪ document.querySelector("#someDiv").classList.toggle("big");
    }
</script>
```

11.2.18 Manipulating attributes

With the setAttribute() method, we can add or change attributes on an element. This will change the HTML of the page

```
<div class="ysm" id="someDiv">
    <input id="button" type="button" onclick="f()">
</div>
<script>
    function f() {
        let el = document.querySelector("#button");
        el.setAttribute("type", "text");
    }
</script>
```

11.2.19 Event listeners on elements

Events are things that happen on a web page, like clicking on something, moving the mouse over an element, changing an element, and there are many more. We have seen how to add an onclick event handler already. In the same way, you can add an onchange handler, or an onmouseover handler. There is one special condition, though; one element can only have one event handler as an HTML attribute. So, if it has an onclick handler, it cannot have an onmouseover handler as well. At this point, we have only seen how to add event listeners using HTML attributes

There is a way to register event handlers using JavaScript as well. We call these event listeners. Using event listeners, we can add multiple events to one element. This way, JavaScript is constantly checking, or listening, for certain events to the elements on the page. Adding event listeners is a two-step process:

1. Select the element you want to add an event to
2. Use the addEventListener("event", function) syntax to add the event

```
<div class="ysm" id="someDiv">
    <input id="button" type="button">
</div>
<script>
    document.getElementById("button").addEventListener("click",
    ↪ change);

    function change() {
        this.setAttribute("type", "text");
    }
</script>
```

11.2.20 Creating new elements

We use createElement(), then use appendChild() to add to the document tree.

```
<div class="ysm" id="someDiv">
    <input id="button" type="button">
</div>
<script>
    window.onload = function() {

    ↪ document.getElementById("button").addEventListener("click",
    ↪ change);
    }

    function change() {
        let other = document.createElement("p");
        other.innerText = "Hello world";
        document.body.appendChild(other);
    }
</script>
```

Interactive Content and Event Listeners

12.1 Specifying events with JavaScript

Here is the first way to do it using JavaScript.

```
document.getElementById("unique").onclick = function() {
    magic(); };
```

12.2 Specifying events with event listeners

The last method is using the addEventListener() method to add an event to an element. With this, we can specify multiple functions for the same event, for example, when an element gets clicked.

```
document.getElementById("unique").addEventListener("click",
    magic);
```

12.3 The onload event handler

We briefly saw this event handler in the previous chapter. The onload event gets fired after a certain element is loaded. This can be useful for a number of reasons. For example, if you want to select an element using getElementById, you'll have to be sure this element is loaded in the DOM already. This event is most commonly used on the window object, but it can be used on any element. When you use it on window, this event gets started when the window object is done loading

```
window.onload = function() {
    // whatever needs to happen after the page loads goes here
}
```

onload is similar, but it's different for the window and document objects. The difference depends on the web browser you are using. The load event fires at the end of the document loading process. Therefore, you will find that all the objects in the document are in the DOM and the assets have finished loading

You can also use the addEventListener() method on any element to handle any event. And it can also be used for the event that all the content in the DOM is loaded. There is a special built-in event for this: DOMContentLoaded(). This event can be used to handle the event of the DOM loading, which will get fired immediately after the DOM for the page has been constructed when the event is set. Here is how to set it:

```
document.addEventListener("DOMContentLoaded", (e) => {
    console.log(e);
});
```

This will log to the console when all the DOM content has been loaded. As an alternative, you will also often see it in the body tag

```
<body onload="unique()"></body>
```

This is assigning a function called unique() to the body, and it will fire off when the body is done loading. You cannot combine addEventListener() and the HTML by using them together. One will overwrite the other, depending on the order of the web page. If you need two events to happen when the DOM is loaded, you will need two addEventListener() calls in your JavaScript.

12.4 Mouse event handlers

There are different mouse event handlers. Mouse events are actions of the mouse. These are the ones we have

- **ondblclick:** when the mouse is double-clicked
- **onmousedown:** when the mouse clicks on top of an element without the click being released
- **onmouseup:** when the mouse click on top of an element is released
- **onmouseenter:** when the mouse moves onto an element
- **onmouseleave:** when the mouse leaves an element and all of its children
- **onmousemove:** when the mouse moves over an element
- **onmouseout:** when the mouse leaves an individual element
- **onmouseover:** when the mouse hovers over an element

12.5 The event target property

Whenever an event gets fired, an event variable becomes available. It has many properties, and you can check it out by using this command in the function that gets fired for the event

This will show many properties. One of the most interesting properties for now is the target property. The target is the HTML element that fired the event. So, we can use it to get information from a web page

```
<div class="ysm" id="someDiv">
    <input id="button" type="button">
</div>
<script>
    window.onload = function() {

    document.getElementById("button").addEventListener("click",
    change);
    }

    function change() {
        let p = document.createElement("p");
        p.innerText = "Hello world";
        // event.target is <input id="button" type="button">
        // parent->parent is document
        event.target.parentNode.parentNode.appendChild(p);

    }
</script>
```

12.6 DOM event flow