JavaScript with jQuery

Course overview

1. Course introduction

1. Kursplan

2. Introduction to Programming

1. Introduction to Programming

3. JavaScript Introduction

1. JavaScript history
2. The DOM
3. Inclusion

4. JavaScript Syntax

1. Identifiers
2. Keywords
3. Operators
4. Values and Types
5. Numbers
6. Strings
7. Date object
8. Variables and Blocks
9. Flow Control Statements
10. Functions
11. Scope
12. Closure

5. Retrieving data with JavaScript

1. XMLHttpRequest

6. jQuery

1. Introduction to jQuery
2. Why use jQuery?
3. Getting Started with jQuery
4. jQuery plugins
5. Selecting elements
6. Handling events
7. Working with Ajax

7. More ES6 features

1. ES2015 aka ES6
2. Destructuring Assignment
3. Extended Parameter Handling
4. Modules
5. Classes
6. Promises

8. Templating

1. Why Templating?
2. HandlebarsJS
3. Handlebars Helpers

Course introduction

Sections in this chapter:

1. Kursplan

1-1. Kursplan

Kursens innehåll

1-1-1

Kursen syftar till att den studerande utvecklar specialiserade kunskaper genom teori och praktiska övningar. Den får utveckla sina kunskaper om, färdigheter i, och kompetenser för att skapa och utveckla enklare front-end tillämpningar.

Kursen omfattar följande moment:

1-1-2

- Introduktion i JavaScript
- Versionshantering i Git
- JavaScript med jQuery
- Event-hantering med JavaScript
- AJAX
- Manipulera DOM
- Dynamiskt uppdatera HTML-sidor med jQuery

Kursens mål

1-1-3

Målet med kursen är att den studerande ska genom teori och praktiska övningar, utveckla sina kunskaper och färdigheter i att använda programmeringsspråket JavaScript med jQuery; hur språket skrivs, hur det exekverar samt dess olika möjligheter och begränsningar.

Kursen introducerar de olika datatyperna i JavaScript och ger en inblick i hur språket fungerar och kan användas i en front-end miljö. De studerande får öva att hantera händelser, göra AJAX- anrop, manipulera DOM och dynamiskt förändra webbsidor.

1-1-4

Efter genomförd kurs ska den studerande ha kunskaper i/om: 1-1-5

- Om jQuery
- Om versionshantering, Git
- Om AJAX-anrop
- Om manipulering av DOM
- Om dynamisk förändring av CSS med jQuery

Efter genomförd kurs ska den studerande ha färdigheter i att: 1-1-6

- hantera jQuery biblioteket för att effektivisera kod och göra asynkrona AJAX-anrop för att hämta data
- kunna hämta, ändra och skapa olika HTML-element med JavaScript och jQuery
- versionshantering med Git

Efter genomförd kurs ska den studerande ha kompetens för att: 1-1-7

- arbeta med versionshantering
- självständigt använda JavaScript för frontendlösningar
- utveckla interaktiva webbhemsidor med JavaScript och jQuery

Former för undervisning 1-1-8

Kursen kommer att genomföras med traditionell undervisning i form av föreläsningar varvat med tid för praktisk träning på övningsuppgifter, med handledning av läraren.

I kursen ingår också att genomföra inlämningsuppgifter på självstudietiden med efterföljande redovisningar. 1-1-9

Undervisningsspråk 1-1-10

Svenska

Förkunskapskrav 1-1-11

Inga

Examination och former av kunskapskontroll 1-1-12

Kunskapskontroller görs under kursen genom fyra obligatoriska laborationer, en större inlämningsuppgift i grupp och en muntlig redovisning. Kursen avslutas med praktisk tentamen.

Betygsskala

1-1-13

Följande betygsskala tillämpas: VG = Väl Godkänd, G = Godkänd, IG = Icke Godkänd

Principer för betygssättning

1-1-14

För betyget Godkänd ska den studerande:

- Den studerande har uppnått samtliga mål med kursen

För betyget Väl Godkänd ska den studerande:

1-1-15

- Den studerande har uppnått samtliga mål med kursen.
- Självständigt skapa webbformulär inklusive validering

Icke Godkänd ges till studerande som inte bedöms uppfylla kraven för betyget Godkänd.

1-1-16

Introduction to Programming

Sections in this chapter:

1. Introduction to Programming

2-1. Introduction to Programming

Writing code is essentially typing text into a text file. 2-1-1

That text is referred to as your **source code**.

Source Code 2-1-2

The **source code** is a text document combined of a set of special words, phrases and operators.

It is designed to instruct the computer **what to do**.

```
if(age > 0) {
    console.log(age + " is greater than 0!");
}
```

However, these words, phrases and operators are not in a form the computer can understand. 2-1-3

It is human readable but the computer needs some **assistance** to read it.

So, we need a step for **converting** the source code into **instructions that the computer can understand**. 2-1-4

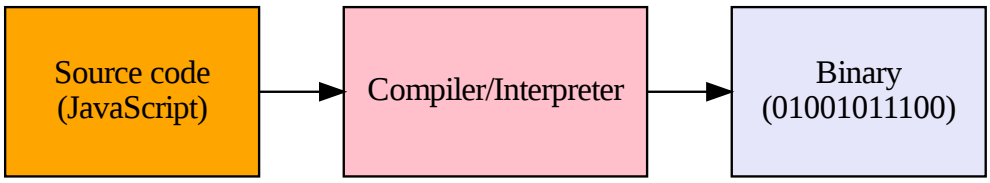
These instructions are in the form of **binary code**, `0's` and `1's`.

A	100 0001	E	100 0101
B	100 0010	F	100 0110
C	100 0011	G	100 0111
D	100 0100	H	100 1000

Your computer **doesn't care** about the source code itself. 2-1-5

Source code can be written in many **different ways** and **different languages**, but in the end it makes **no difference** to your computer how you write it.

The **compiler** or **interpreter** in your computer or browser, whose job it is to convert the source code into **binary code** that the processor can understand, may care. 2-1-6



Compiler 2-1-7

A **compiler** is a special program that processes **source code** and turns it into **code** that a computer's **processor** can read and execute.

Interpreter 2-1-8

An **interpreter** does pretty much the same thing as a compiler, but it will read the source code **line by line** and then convert and execute it.

Compiler	Interpreter
Translates whole source code at once	Translates and executes source code line by line
Slow code analyze	Shorter analyze time
Quicker execution time	Slower execution time
C, C#, C++, Java	JavaScript, Python, Ruby

Compiler

Translates whole source code at once - Takes whole source and translates it to machine code in one step

Slow code analyze - Since the whole code base is being analyzed at once this requires more time

Quicker execution time - Since the whole code base is already compiled to machine code the code can be executed right away

Interpreter

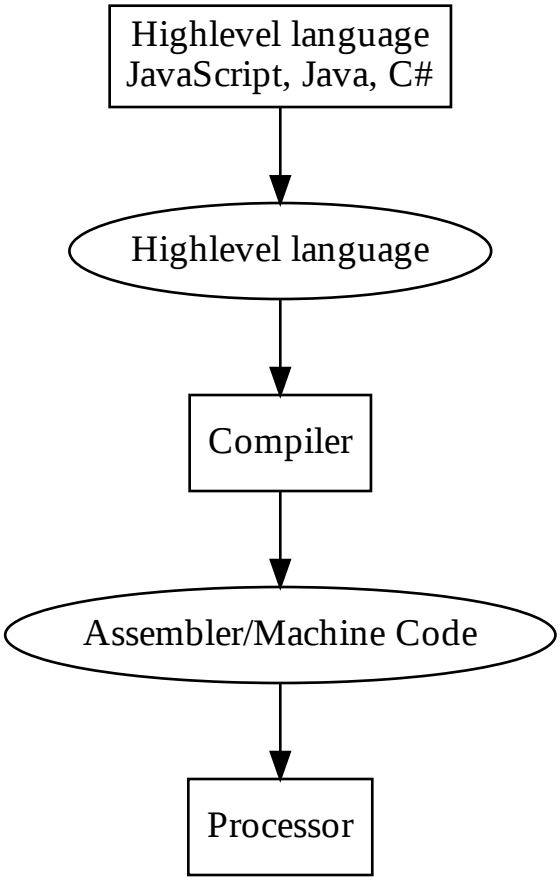
Translates source code line by line - Translates line by line within the source code as it is about to be executed

Shorter analyze time - Shorter analyze time since it only analyzes one statement at a time

Slower execution time - Slower execution time since not all source code is ready as machine code when it is about to be executed

---

The browser takes JavaScript code and compiles it to code that the processor can understand and execute.



Programmers mostly write programs with a high-level language such as JavaScript or C#.

The high-level code has to be translated to machine code for the computer to be able to execute the code or program.

The output of the compiler or interpreter **depends on the language of the source code** and **what compiler or interpreter** that is used to convert the code.

In Chrome, it is by default the **V8 engine** that handles the compilation of JavaScript code.

Different browsers can have different compilers or interpreters.

High-level code

High-level code, source code, is written by humans with a high-level language such as JavaScript.

The code is human read-able and uses English based statements and words. 2-1-15

```
if(x > 0) {
    console.log(x + " is greater than 0!");
}
```

Above example is written in JavaScript

Popular high-level languages: 2-1-17

- JavaScript

```
var x = 3;
if(x > 0) {
    console.log(x + " is greater than 0!");
}
```

- Java

```
int x = 3;
if (x > 0)
   {
       Console.WriteLine(x + " is greater than 0!");
   }
```

- C#

```
int x = 3;
if (x > 0)
   {
       Console.WriteLine(x + " is greater than 0!");
   }
```

- Ruby

```
x = 1
if x > 2
   puts "x is greater than 2"
```

All high-level code must be interpreted or compiled into code that the processor can understand before being executed. 2-1-18

Assembly code / Machine code 2-1-19

Assembly code, also known as *symbolic machine code*, is a low-level language.

```
0013           RESETA  EQU    %00010011
0011           CTLREG  EQU    %00010001

C003 86 13     INITA   LDA A  #RESETA   RESET ACIA
C005 B7 80 04          STA A  ACIA
C008 86 11             LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04          STA A  ACIA

C00D 7E C0 F1          JMP    SIGNON    GO TO START OF MONITOR
```

Assembly code is somewhat human read-able but **closer to machine language**. 2-1-20

JavaScript code written on one line could result in hundreds lines of assembler code.

Machine code is close to impossible for humans to either read of write, since it **consists of numbers only**, `0's` and `1's`. 2-1-21

```
0000 1001 1100 0110 1010 1111 0101 1000 0000 1001 1100 0110 1010
1010 1111 0101 1000 0000 1001 1100 0110 1010 1111 0101 1000 0000
1100 0110 1010 1111 0101 1000 0000 1001 1100 0110 1010 1111 0101
```

Machine code is being handled by the *processor*, the **CPU** (**C**entral **P**rocessing **U**nit). 2-1-22

The processor is also known as the brain of the computer.

You have written a program with the following functions. 2-1-23

Machine Instructions	Machine Operations
0000	Stop Program
0001	Turn light on
0010	Turn light off
0100	Dim light

Translated, the combinations of `0's` and `1's` will represent some functionality.

Syntax 2-1-24

When we talk about writing source code, there is a word that is often referred to, which is **syntax**.

The set of rules that puts syntax together is called the **grammar**. 2-1-25

The **syntax** and the **grammar** together tells you how to write for instance a JavaScript program.

It basically tells you the **valid combinations of characters and words** that will make sure your program does what you want it to do. 2-1-26

```
// Syntax for declaring a variable in JavaScript
var speed = 0;
```

```
// Syntax for declaring a variable in Java
int speed = 0;
```

This is much like the syntax for Swedish or English. 2-1-27

There are punctuation marks, like the comma, the period and the exclamation mark. 2-1-28

There are parts of **phrases that work together**, like a word used as a verb versus a word used as a noun.

The way you put these things together, first into a phrase and then phrases into sentences, is called **grammar**. 2-1-29

A similar concept exists in programming, where you first learn the syntax, and then the rules of how to put the syntax together to make a coherent statement. 2-1-30

Many IDEs have support for **catching syntax errors**. 2-1-31

If the code does not follow the correct syntax an error will be displayed.

2-1-32

```
int speed = 0;
```

Writing this in JavaScript would give you an error since JavaScript is **dynamically typed**, meaning we don't define data types.

Pseudo code 2-1-33

Pseudo code is a detailed yet readable description of what a computer program or algorithm does.

It is many times used in the process of creating a program, since it is an easy way for non-programmers to express the functionality they need. 2-1-34

Programmers can then use the pseudo code as a template to write code in whatever programming language they are most comfortable with. 2-1-35

Let's say you wanted to write the pseudo code to print "Hello World!". It could look like any of the following: 2-1-36

```
Display Hello World!
Print Hello World
Write Hello World
```

As you may have noticed, pseudo code is essentially a mix between natural language, like how most humans speak to each other, and logical statements that will be understood by programmers. 2-1-37

The pseudo code for the following code: 2-1-38

```
if(age > 0) {
    console.log("Greater than 0!");
}
else {
    console.log("Zero or less!");
}
```

could look like this:

```
If age is greater than zero
    Print "Greater that 0!"
Else
    Print "Zero or less!"
```

JavaScript Introduction

Sections in this chapter:

1. JavaScript history
2. The DOM
3. Inclusion

3-1. JavaScript history

Originally and (in)famously created in 10 days by Brendan Eich, who wanted to build a LISP interpreter for the browser. 3-1-1

Name is still highly confusing, and it still gets confused with Java every now and then.

Timeline 3-1-2

... Finding a name is hard

- 1995 - Created by Brendan Eich. Was initially called **Mocha**
- 1995 - Later same year, renamed to **LiveScript**
- 1995 - Hey, still same year. Renamed to **JavaScript** after licensing from Sun (Now Oracle)

- 1996 - JavaScript taken to the ECMA committee. Official name: ECMAScript 3-1-3
- 1998 - ECMAScript 2 released
- 1999 - ECMAScript 3 released
- 2000 - Work on ECMAScript 4 begins, but gets cancelled for various reasons

Then, nothing happens for a few years.

During 2005: 3-1-4

- Work restarts on ES4. Lots of things happening.
- Later this year... Crockford, Microsoft and a few other players opposes ES4. Forms their own subcommittee and designs ES3.1
- Bickering between ES3.1 and ES4 goes on for quite some time...

During 2008: 3-1-5

- Finally, an agreement. ES3.1 prevails but gets renamed to ES5.

So, to summarize: ES5 used to be ES3.1, but shortly before that it was ES4 which got cancelled twice. 3-1-6

Awesome.

In 2015 and 2016, respectively, ES6 and ES7 specifications are finalized. Officially known as ECMAScript 2015 and ECMAScript 2016. 3-1-7

... *And here we are*

3-2. The DOM

The **D**ocument **O**bject **M**odel is the bridge between the JavaScript and HTML. *3-2-1*

It is the **browsers internal representation of the HTML**.

A web page is a document. *3-2-2*

This document can be displayed visually in the browser or as the HTML source, the markup we write.

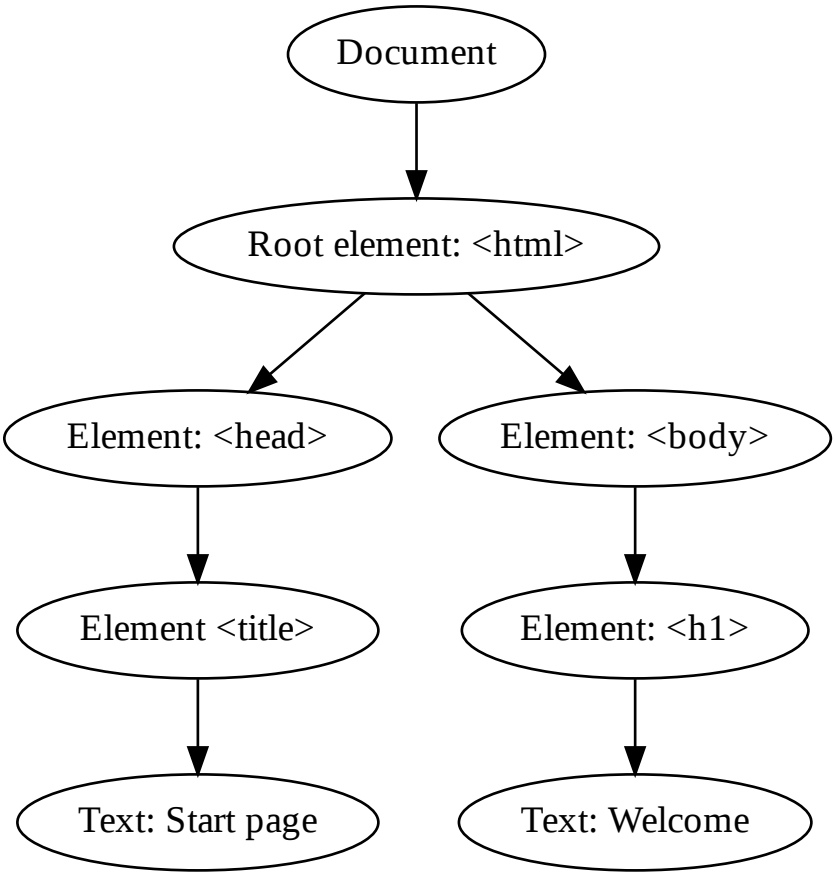
The DOM is this same document represented in **objects and nodes**. *3-2-3*

We can read and mutate these objects and nodes with JavaScript.

There are two major ways we can mutate the DOM: *3-2-4*

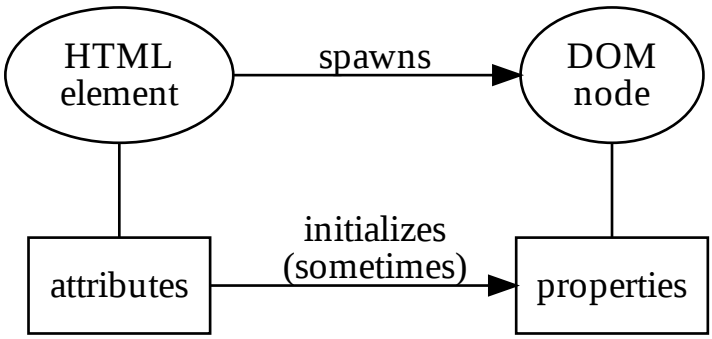
1. Insert nodes into the DOM, take them out, move them, etc.
2. Change content or properties of DOM nodes themselves.

The DOM is structured like a tree: 3-2-5



HTML elements have **attributes** but DOM nodes have **properties**. 3-2-6

Usually, but not always, the **former initializes the latter**:



Showcase 3-2-7

We will now **showcase the difference** of attributes and properties using this simple page:

It is published at http://blog.krawaller.se/attrvsprops.

If you **`view source`** you'll see this: *3-2-8*

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <input value="John Doe"> <button>Submit</button>
  </body>
</html>
```

Note that the **`input`** has the **initial value `"John Doe"`**.

In the console we can **get a reference to the input field node** like this: *3-2-9*

```
var field = document.querySelector("input");
```

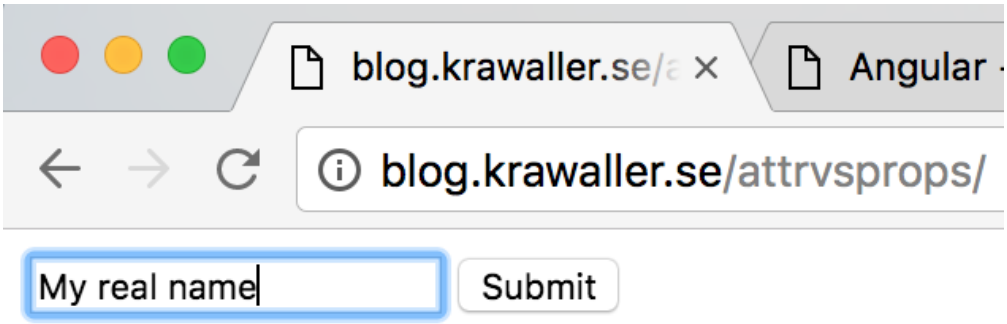
Using that reference we can **confirm the value of the `value` attribute**: *3-2-10*

```
field.getAttribute("value") // "John Doe";
```

And we can also **read the `value` property**: *3-2-11*

```
field.value // "John Doe";
```

Now **type in the field** to say something else: *3-2-12*



If we **query the attribute again**, we see that it is still `John Doe`: *3-2-13*

```
field.getAttribute("value") // "John Doe";
```

But the **property is updated**: 3-2-14

```
field.value // "My real name";
```

This reflects the fact that **attributes are characteristics of the original HTML elements**, while **properties belong to the live DOM nodes**. 3-2-15

To wit: 3-2-16

- **HTML elements have attributes**, and **DOM nodes have properties**.
- **Attributes often initialize properties** (but not always)
- **Attributes never change**, but **properties can change**.

We traverse through the DOM to access children or parents of an element, or the element itself. 3-2-17

document object 3-2-18

We **interact with the DOM** from JavaScript space through the global `document` object.

The `document` object represents the actual web page, our document. 3-2-19

Go to Google and try this in the console: 3-2-20

```
document.body.setAttribute("style","background-color:red;")
```

We *write* JavaScript but *use* the DOM to access elements.

Also, try `document.designMode = "on"`. 3-2-21

Now you can edit anything on the page! *Think about the power inherent in this. Never trust a screenshot.*

Through the `document` object we can access and see all nodes in the DOM. 3-2-22

```
document.childNodes; // [<!DOCTYPE html>, html.gr__google_se]
```

`childNodes` gives you the nodes in the current node.

We get an array of objects which in turn contains more arrays and objects. 3-2-23

The **child nodes** of `document` is usually the `doctype` and `html` elements.

These objects of the array contains a lot of properties 3-2-24

```
assignedSlot : null
attributes : NamedNodeMap {0: lang, 1: class, length: 2}
baseURI : "https://www.google.se/_/chrome/newtab?espv=2&ie=UTF-8"
childElementCount : 2
childNodes : NodeList(2)
   length : 2
   0 : head
     accessKey: ""
     assignedSlot : null
     attributes : NamedNodeMap {length: 0}
     baseURI : "https://www.google.se/_/chrome/newtab?espv=2&ie=UTF-8"
     childElementCount : 12
     childNodes : (12) [style, style, link, style, link, ...]
   1 : body
     accessKey: ""
```

Shortened for convenience

Each node contains `childNodes` of itself. 3-2-25

Child nodes of `html` can be `head` and `body`

We could do this: 3-2-26

```
let html = document.childNodes[1];
let body = html.childNodes[2];
body.style.backgroundColor = "green";
```

But there is an easier ways if we want to modify nodes in the DOM. 3-2-27

The `document` object serves properties and methods for common access, such as `body` for example.

You can find a list of available properties and methods of the `document` object at MDN 3-2-28

Through `document.body` we get an **HTML node**, an **object that represents the element**. 3-2-29

```
document.body;
```

We can chain further and further since it is an object, so `body` in turn have a lot of properties and methods just like `document`. 3-2-30

```
document.body.onresize = () => {
    console.log("RESIZED")
};
```

On `body` we could for example 3-2-31

- call the **`setAttribute`** method
- query attributes using the **`getAttribute`** method
- change its content by assigning to **`innerHTML`** or **`textContent`**
- iterate over child nodes using the **`children`** property, allowing us to walk down the tree.
- go back up the tree using the **parentNode** property.

We don't have to walk the tree from `body` or trough the `childNodes` property to get access to children or grand children and so on. 3-2-32

We are served **several methods** to **find a specific node or set of nodes** on `document`: 3-2-33

- `getElementById("someId")`
- `getElementsByClassName("someClass")`
- `getElementsByTagName("div")`

You can find a list of all available methods at MDN 3-2-34

3-2-35

```
<button id="myButton">Click me!</button>
```

```
let button = document.getElementById("myButton");
```

Some of these methods returns one element while others returns several.

Notice the `getElement`**`s`**`ByTagName` for example 3-2-36

```
document.getElementsByTagName('div');
// (46) [div#modal-collapse.modal-background, div.navbar-inner, ...]
```

It is plural and will always return an array with element; zero, one or more.

The `getElementById` for example only returns one specific element. 3-2-37

An id is unique and should only be present on one element in your document.

```
document.getElementById('modal-collapse');
// <div id="modal-collapse" class="modal-background"></div>
```

And, there are two methods which **gets elements using CSS selectors**: 3-2-38

- `querySelector("#someId")`

  Returns one element

- `querySelectorAll("article > p")`

  Returns list of elements

querySelector(target) 3-2-39

The `querySelector()` functions returns only one element.

```
document.querySelector("#modal-collapse");
// <div id="modal-collapse" class="modal-background"></div>
```

The passed argument to the function can be what css selector you want, preferably an id since it returns only one element.

If you use the class selector for example as an argument to the `querySelector()` 3-2-40

```
document.querySelector(".col-sm-4");
// <div class="col-sm-4">...</div>
```

It will only return the first occurring element and then stop searching.

querySelectorAll(target) 3-2-41

The `querySelectorAll()` will always return an array, with none, one or several elements.

```
document.querySelectorAll(".col-sm-4");
// (3) [div.col-sm-4, div.col-sm-4, div.col-sm-4]

document.querySelectorAll(".col-sm-423");
// []
```

window object

3-2-42

The window object **represents an open window in a browser**.

To access information regarding the browser rather than the document, you use the window object.

Scrolling the whole browser window to a specific point for example

3-2-43

```
window.scrollTo(0, 1000);
```

This will scroll the window based on the given X- and Y-coordinates

console

3-2-44

We have seen the following in a few places

```
console.log("...")
```

This an object to access the browsers debugging console.

It is exposed as `window.console` but can be accessed simply as `console`

3-2-45

```
window.console.log("...");

// same as...

console.log("...");
```

We use `.log()` to log a message or value to the console.

3-2-46

But there are more methods on the console object

3-2-47

- `error()`
- `warn()`

You can find the full list of methods available at MDN

The window object has several properties and methods, amongst many can be: 3-2-48

- `.alert()` - Alert box showing a message
- `.confirm()` - Confirmation box
- `.location` - Current location, url
- `.sessionStorage` - Object used to store data, in session
- `.localStorage` - Object used to store data, no expiration

You can find a full list of available properties and methods at MDN 3-2-49

Get and set values 3-2-50

We have seen how to access elements, but how do we access the **value of an element**.

Input elements 3-2-51

The following gives us an element that the user can use to input a value.

```
<input type="text" id="fname">
```

To get the value of the input we use the `.value` property. 3-2-52

```
let firstName = document.getElementById("fname");
console.log(firstName);
// <input id="input" value="My input">

console.log(firstName.value);
// "The value in the input field"
```

Other elements 3-2-53

We can get the content of other elements that are not input elements, such as paragraphs.

```
<p id="paragraph">My little paragraph</p>
```

For this we can use `innerHTML` 3-2-54

```
let paragraph = document.getElementById("paragraph");

console.log(paragraph.innerHTML);
// "My little paragraph"
```

We could also use `innerText` 3-2-55

```
console.log(paragraph.innerText);
// "My little paragraph"
```

Q Is there any difference between them? 3-2-56

A Yes, if we would have another HTML element within our paragraph, a `span` for example. 3-2-57

```
<p id="paragraph">My little paragraph <span>with span</span></p>
```

Using the `innerText` we would get only the text content 3-2-58

```
let paragraph = document.getElementById("paragraph");

console.log(paragraph.innerText);
// "My little paragraph with span"
```

While `innerHTML` we would get the HTML elements included in the string also. 3-2-59

```
console.log(paragraph.innerHTML);
// "My little paragraph <span>with span</span>"
```

We can use the same properties to set the value or content of an element. 3-2-60

```
let paragraph = document.getElementById("paragraph");
let input = document.getElementsByTagName('input')[0];

paragraph.innerHTML = "Text <span>span</span>";
paragraph.innerText = "Text";
input.value = "Some text";
```

Adding new nodes 3-2-61

We have seen how we can access and operate on existing elements in the DOM.

With JavaScript we can also insert new nodes.

createElement()

3-2-62

Lets say we have a page looking like this

```
<body>
    <div>
        <p>Hello</p>
    </div>
</body>
```

Now we want to add a new paragraph to the div with JavaScript.

We need to create the `p` element using `createElement()`.

3-2-63

```
let paragraph = document.createElement("p");
```

Next we might want to add some content to our paragraph.

3-2-64

Since it is not yet a node in the DOM we cannot use the `innerText` or similar methods.

In this case we need to use `createTextNode()`

3-2-65

```
let text = document.createTextNode("My paragraph text");
```

appendChild()

3-2-66

Now the actual paragraph and the text is two separate things, and there is nothing that indicates that they belong together.

To add the text to the paragraph we use `appendChild()`

3-2-67

```
paragraph.appendChild(text);
```

The paragraph is still not included in the DOM, though.

3-2-68

We need to append our created element to an existing one.

```
let existingElement = document.getElementsByTagName("div")[0];

existingElement.appendChild(paragraph);
```

So by doing these steps 3-2-69

```
let paragraph = document.createElement("p");
let text = document.createTextNode("My paragraph text");
paragraph.appendChild(text);

let existingElement = document.getElementsByTagName("div")[0];
existingElement.appendChild(paragraph);
```

We get a new paragraph within our div.

```
<body>
    <div>
        <p>Hello</p>
    </div>
</body>
```

DOM events 3-2-70

With CSS, the only interaction we could offer was some animations using `:hover` and `:active`.

But now is the time for some **true user interaction**!

The DOM lets us **add event listeners to elements**. 3-2-71

These are **functions that will be called whenever that particular event happen** on that element.

Say we **have this button** in our document: 3-2-72

```
<button id="doomsdaybtn">Don't click me!</button>
```

And a **reference** to the corresponding node:

```
var btn = document.getElementById("doomsdaybtn");
```

We now **create a function to be used as an event listener**... 3-2-73

```
var listener = function(){
  alert("BOOM!");
}
```

...and **attach it using the `addEventListener` method** on the node:

```
btn.addEventListener("click",listener);
```

We could also **add the anonymous function directly**: 3-2-74

```
btn.addEventListener("click",function(){
  alert("BOOM!");
});
```

Now when the user clicks the button, the event handler function will run. 3-2-75

There are three ways to register event handlers for an element in the DOM. 3-2-76

- EventTarget.addEventListener
- HTML attribute
- DOM element properties

Using the `EventTarget.addEventListener` is the preferred way with pure JavaScript to register event handlers.

EventTarget.addEventListener 3-2-77

As we saw in previous example, we select the element we want to target with the event listener.

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    alert("Button is clicked");
});
```

When the button is clicked, the event will be triggered and perform the given task.

In this case it would log *Button is clicked* to the console.

On this element we call the `addEventListener` method into which we pass: 3-2-78

- **what event it should listen to**
- a **callback function**, or **handler function**, what should be done when the event is triggered

The event we pass into the `addEventListener` method can vary. 3-2-79

Common events to listen to are:

- `click` - element gets clicked
- `change` - element changes
- `mouseover` - hover element
- `keydown` - any key down

There is a **full list of events** at MDN:

https://developer.mozilla.org/en-US/docs/Web/Events

HTML attribute 3-2-80

We can also apply event handlers as an attribute on an element.

```
<button onclick="alert('Button is clicked');">Click me!</button>
```

This approach should be **avoided due to separation of concerns**. 3-2-81

Including JavaScript within our HTML makes the markup bigger and a bit messy.

The key is to separate our JavaScript from the HTML-file into a separate JS-file. 3-2-82

The same way we want to separate CSS from our HTML.

DOM element properties 3-2-83

We can register event handlers via properties on a given element.

```
let button = document.getElementById("myButton");

myButton.onclick = event => {
    alert('Hello world');
};
```

This approach is not that usual, it also limits us to only set one handler per element and per event. 3-2-84

Event object

3-2-85

We have seen the passed argument `event` into the event handler.

```
button.addEventListener("click", event => {
    // handle the event
});
```

This is an object that is created when the event occurs.

The `event` object can be used within the handler function.

3-2-86

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    console.log(event);
    // MouseEvent {isTrusted: true, screenX: 39,
    // screenY: 144, clientX: 39, clientY: 24, …}
});
```

The `event` object contains a bunch of information about the event.

3-2-87

```
let button = document.getElementById("myButton");

button.addEventListener("click", event => {
    console.log(event.target);
    // <button id="myButton">Click me!</button>
});
```

For example `target` gives us information about what element triggered the event.

3-3. Inclusion

Including JavaScript is **extremely similar** to how **CSS inclusion** works. 3-3-1

There are **three ways to include JS** 3-3-2

- script element containing code
- script element referencing a file
- inline event handler in an element attribute

Script element with code 3-3-3

A **script element with code** might look like this:

```
<script type="text/javascript">
    /* ...javascript code here... */
</script>
```

This is a script element in which you write your JavaScript code **within the HTML document**. 3-3-4

This comes handy if it is only a **small amount of code**.

Script element referencing a file 3-3-5

Usually though, we want to **separate our files**, having HTML in a `html` file and JavaScript code in a `js` file.

The same way we want to have separate `css` files.

A **script element referencing a separate file** might look like this: 3-3-6

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

Several script elements can be included in your HTML document 3-3-7

```
<script type="text/javascript" src="js/math.js"></script>
<script type="text/javascript" src="js/order.js"></script>
<script type="text/javascript" src="js/employees.js"></script>
```

Note that you **should not self-close the script element**. 3-3-8

This is because the script element **may** contain code, and thus needs to be closed in that case.

The script element is placed within your HTML document. 3-3-9

It can be placed in the head section, within the body or everywhere actually.

But what to keep in mind is that HTML documents is **read from top to bottom**. 3-3-10

This means that the included JavaScript code will be **executed in the order it is placed**.

Due to this behavior, the following would result in an error 3-3-11

```
<script>
    let input = document.getElementById("input");
    console.log(input.getAttribute('value'));
    // Uncaught TypeError: Cannot read property 'getAttribute' of null
</script>
<input id="input" value="My Input">
```

This is because the JavaScript code is executed before the actual element is created, therefore we cannot access it. 3-3-12

For our JavaScript code to work we have to make sure that the **document has loaded with the elements** that the script use. 3-3-13

We can do this simply by placing the script at the **end of the body** element, right before `</body>`. 3-3-14

```
<body>
<input id="input" value="My Input">
<script>
    let input = document.getElementById("input");
    console.log(input.getAttribute('value'));
</script>
</body>
```

This way the elements have loaded before the script executes and can be accessed in the JavaScript code.

Inline event handler

An **inline event handler** looks like this:

```
<button onclick="alert('You clicked, OMG!');">Don't click me</button>
```

As you might have guessed, **inline handlers are a bad idea**, for pretty much the same reasons as inline CSS.