OWL
OPEN WEB LEARNING

Step-by-Step Guides

Latest DOM
for modern browsers

Brand new and complete in ES6+/
ECMAScript 2018 syntax

Includes many exercises

Recommendations for
clean code

Marco Emrich
Christin Marit

EDITION 2018

Learning JavaScript

The non-boring beginner's guide to modern ES6+ JavaScript programming

Vol. 2: DOM Manipulation

OWL
OPEN WEB LEARNING

Step-by-Step Guides

Accompanying files online!

Visit our website *https://owl.institute* and download the accompanying material such as exercise files and solutions:

https://owl.institute/files
Download Code: 1210767621f4

OWL
OPEN WEB LEARNING
Step-by-Step Guides

Marco Emrich and Christin Marit

Learning JavaScript: The non-boring beginner's guide to modern (ES6+) JavaScript programming Vol 2: DOM manipulation

Version 2.0.18 of June 18, 2018
https://owl.institute

About the Authors

Marco Emrich

Marco Emrich holds a university degree in computer science, is a passionate trainer and an advocate of the software craftsmanship movement. He has wide experience as a software architect and developer in a variety of sectors. Marco heads the Web Engineering department of the Webmasters Akademie in Nuremberg, Germany. He also lectures regularly, holds workshops at leading software conferences and writes reference books as well as articles for technical journals. In his spare time, if he's not organizing meetings of the Softwerkskammer software craftsmanship community, you'll probably find Marco teaching his son how to program robot turtles.

https://github.com/marcoemrich
https://twitter.com/marcoemrich

Christin Marit

Christin Marit is a qualified social pedagogue, web developer, photographer, minimalist, world traveler and professional in the art of life. She works for the Webmasters Akademie in Nuremberg as a course developer, author and e-tutor. Christin feels that learning JavaScript should be as easy, wonderful and fascinating as life itself. She also maintains a blog on the lightness of being, which you can find at simplewinke.de.

https://twitter.com/christinmarit

Published by **The Open Web Learning Institute** at
Webmasters Akademie Nürnberg GmbH
Neumeyerstr. 22–26
90411 Nuremberg, Germany

Originially published in 2016 in Germany by Webmasters Press, Nürnberg, under the title *JavaScript: HTML mühelos manipuliert*

Translated from German by Roland Galibert
Cover design: Frank Schad
Item number: 1210767621f4
Version 2.0.18 of June 18, 2018

Printed books made with Prince

For general information about our other products and services, please visit https://owl.institute.

Table of Contents

Preface

1 Everything You've Always Wanted to Know About the DOM…
- 1.1 DOM Manipulation in the Context of Web Applications
- 1.2 A Life Without jQuery
- 1.3 JavaScript in Vanilla
- 1.4 Prerequisites
- 1.5 What This Class Is
- 1.6 What This Class is NOT
- 1.7 Why Firefox?
- 1.8 "NerdWorld" Project

2 Your First Challenge in DOM Manipulation
- 2.1 "Hello DOM" Using querySelector and innerHTML
- 2.2 A Look Behind the Scenes
- 2.3 Using the DOM to Search For and Find Elements
- 2.4 One Function To Find Them All — querySelectorAll
- 2.4.1 Attribute Selectors
- 2.4.2 Combinator Selectors
- 2.4.3 Pseudo Classes
- 2.5 Document vs. Element
- 2.6 Short and Sweet — $
- 2.7 Summary
- 2.8 Exercises

3 And Down into the Depths of HTML
- 3.1 Delays Without Waiting: defer & async
- 3.2 In Strict Isolation: Blocks & use strict
- 3.3 Just a Little Refactoring …

3.4 Array.from was So Yesterday — These Days, it's Array Methods for Everything!
3.5 One More Tiny Improvement

4 Events (Not the Kind You Celebrate)
4.1 Browser Events
4.2 Dealing With Events as They Come Up: Event Handlers
4.2.1 Preliminary Measures
4.2.2 Putting Everything Together
4.3 Guard Clauses: Protection for Your Functions
4.4 Making a Stunning Entrance Using init
4.5 Take a Look Behind the Facade With the Event Object
4.6 preventDefault or, Who Always Wants to be Just Standard?
4.7 Brief and To The Point — on
4.8 Summary
4.9 Exercises

5 Using Multiple JS Files
5.1 Happy File Hacking
5.2 The Terror of Global Pollution
5.3 Not Just Another Dusty Old Library

6 Attributes: Retrieving, Changing and Manipulating Expressly Permitted
6.1 IDL Attribute vs. Content Attribute
6.2 Long Live Progress (Bars)!
6.3 Attention! Boolean Attributes at Work
6.4 It's All in How You Ask the Question — The value Attribute
6.5 Summary
6.6 Exercises

7 A Question of Style: The Style Object
7.1 Completely Calculating and Manipulative: getComputedStyle

7.2 Even More Stylish with Tooltips
7.3 Exercises

8 Undermining HTML Using Data Attributes
8.1 Summary
8.2 Exercise

9 Of New Elements and Appended Children: createElement & appendChild
9.1 Generating HTML Using document.createElement
9.2 Finally Attached: appendChild
9.3 Once Again, a Little Refactoring
9.4 And Now a Little JSON
9.5 Comparison of Various DOM Creation Methods
9.6 Summary

10 Tossing Elements in the Trash with remove
10.1 Classification
10.2 Summary
10.3 Exercises

11 Going On a Family Trip: siblings & insertBefore
11.1 Back to Your Old Friend Refactoring
11.2 Oops! We Still Need to Do a Little Better
11.3 Summary
11.3.1 Example for insertBefore
11.3.2 DOM traversal
11.4 Exercises

12 Appendix A: Reference
12.1 Elements
12.1.1 Element vs. Node
12.1.2 Document Node
12.1.3 DOM Selection

12.1.4 DOM Manipulation
12.1.5 DOM Traversal
12.1.6 DOM Creation
12.2 Attributes
12.3 CSS
12.4 Events

13 Appendix B: Sources & References
13.1 APA Style
13.2 Sources

Solutions (Exercises)

Preface

JavaScript, HTML and CSS

…are the three pillars upon which the modern web is built. No matter what other technologies you use, there's no way you'll avoid working with that dream web trio in some way.

Our first volume (*JavaScript — The First Step is the Easiest*) covered the basics of the JavaScript language. In this volume, we take things to the next level and show you how to use JavaScript to manipulate HTML with the greatest of ease. Manipulating HTML will allow us to build web applications which are much more than merely static pages — we're talking about web applications which flow smoothly and are easy to use, as opposed to web applications in which every page needs to be refreshed anytime you click on something.

After this course, you'll be able to do things like generate dynamic tooltips, program interfaces to manage products, develop your own image galleries...we could easily name a hundred other possible applications, but what we really want to do is to give you the tools which will allow you to develop applications we haven't even thought of yet.

Once you've mastered the dream combo of HTML and JavaScript (and of course CSS), you'll be able to turn your ideas into reality. Where pre-packaged web generators only tie you down, JavaScript is a flexible tool which will unleash your creativity and let you implement things exactly the way you want — your only limit is your imagination.

This volume doesn't cover frameworks like AngularJS and React. Instead, we want to show you that the very browser you're using right now already has everything you need to create dynamic front end web

applications.

We do want to push you towards using just the standard tools available right in your browser to solve small problems instead of resorting to a heavyweight, complex framework. On the other hand, you may need to use such a framework at a later point when you're developing larger applications. In that case, the basic knowledge you gain from this course will still be invaluable to you. It'll help you understand how the framework works "under the hood", so you can solve problems even when they push the framework to its limits.

Just as we did in volume 1, we'll continue to use the latest JavaScript standard in the present course. You should already be very familiar with the basic language features of the ECMAScript2015 specification. If not, it might be worth your while to go through volume 1 again to refresh your memory. In addition to using current language constructs, we'll also continue the principle we started in volume 1 of using the most up-to-date browser implementations. We'll show you the latest APIs, as well as a modern DOM which brings the fun back to developing web applications. What lies behind that mysterious term "DOM"? You'll find the answer right in the first lesson. And on that note, happy reading and coding!

Christin & Marco

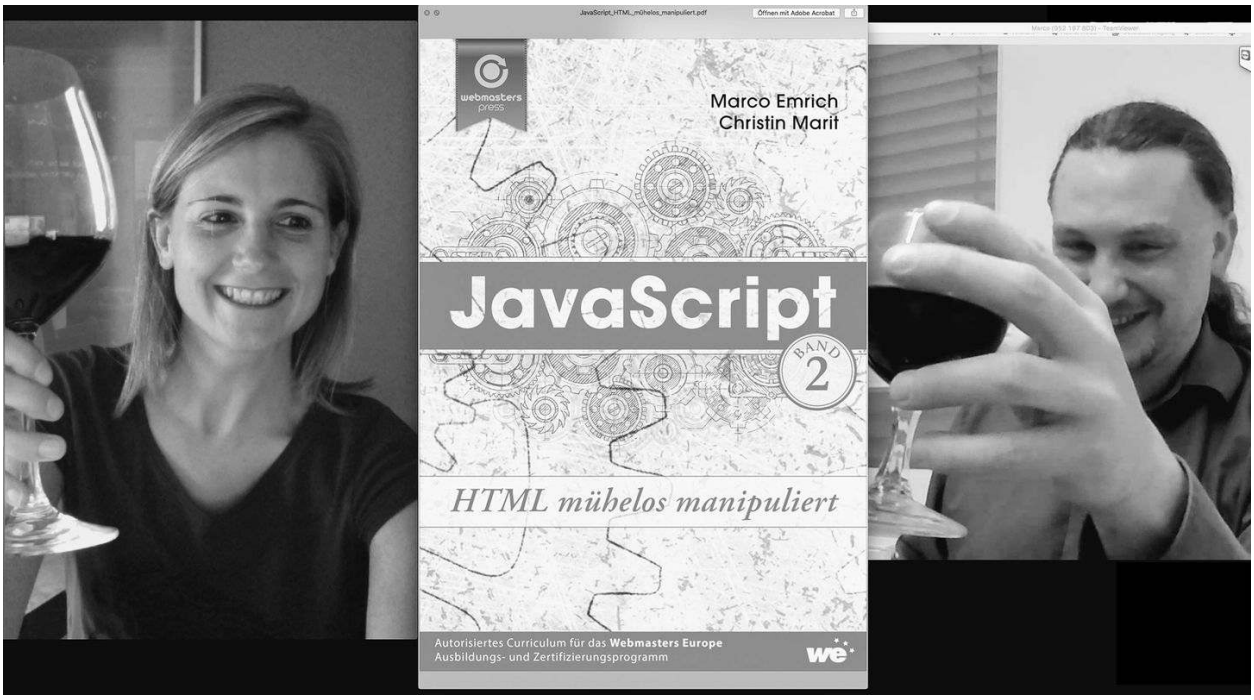


Fig. 1 *Remote programming…remote writing…and remote partying :)*

1 Everything You've Always Wanted to Know About the DOM…

> *And then there is the DOM — I'm pretty confident saying this: I think, it is the worst API ever invented.*
>
> Douglas Crockford talking about "Upgrading the Web"

So the first thing you'll probably want to know is, just exactly what is this DOM we keep talking about? No, it's not Internet shorthand for expensive champagne, it's an abbreviation for the ***Document Object Model*** — the star (you love to hate) of this class.

The DOM is actually a so-called ***API*** (***application programming interface***). In plain English, this means a collection of objects, methods, functions and attributes which allow you to program applications (you'll soon find out exactly what that means).

Through the DOM, a browser provides us the ability to change HTML pages dynamically. All familiar web applications which don't need to be constantly reloaded as you're using them (like Google's Gmail, Twitter, Facebook, etc.) use the DOM to provide you with interfaces which are easy to operate (although there's undoubtedly disagreement on this point). JavaScript (JS for short) in your browser is worthless without the ability to manipulate HTML — and to do that, you need the DOM. Even modern frameworks like AngularJS and ReactJS rely on the DOM "under the hood".

But why do we call the DOM a star you love to hate? Douglas Crockford (the man behind JSON and the author of *JavaScript: The Good Parts*), in his 2015 talk *Upgrading the Web*, called the DOM the worst API of all

time.

Although it's difficult to say how accurate that statement is, we do have to admit that, as tormented developers, our struggles with the API have already given us a couple of gray hairs (they're especially noticeable on Marco). The fact is, the API is complex and not easy to use.

The good news is that the main reason the DOM is so complex is because it's overloaded with many, many features which are rarely used in practice.

As a result, we'll pick out those features of the API which you'll get the most benefit out of, and just teach you those. That, in combination with the latest ECMAScript6 syntax and a couple of tricks we'll show you soon will turn the worst API of all time into …an API which still isn't too good. But at least you'll be able to achieve your goals, and you'll also produce code which will continue to be readable and maintainable over the long run.

1.1 DOM Manipulation in the Context of Web Applications

If you have plans to develop larger JS applications or even ***single page applications***, DOM manipulation alone will definitely not be your method of choice. Specialized frameworks such as *AngularJS* or *ReactJS* are much better suited to those kinds of applications.

So why do we feel you should still learn DOM manipulation *first*?

There are a number of good reasons you should learn DOM manipulation before learning an actual framework. For a start, it may be that you're not actually developing a single page app, but just want to give a conventional web application (e. g. one you developed using ROCA style) a modern face. Or maybe you just want to improve the usability of your app, or just enhance its "fanciness" factor. In such cases, a ***DOM scripting*** approach which is targeted to the right areas and not used too excessively will often be the right choice. You won't need another framework, you won't need to reload huge amounts of data and you won't need to create complicated setup code — your browser alone will contain everything you need.

Speaking of mountains of data, maybe it's also important for you that your website loads quickly, because your mobile users are a big part of your success. Or you realize that speed is a key ranking factor for Google (one aspect of websites which continues to be criminally neglected by many SEO, or ***search engine optimization***, experts). In such cases, it's even more essential that you make use of as little additional code as possible.

If at some later point you do wind up working with a more extensive web framework, you'll discover that not every framework can do everything, or that many "simple" things aren't simple at all in some framework, simply because they just don't fit into that framework's concept.

So when that happens, you'll have to go back to taking things into your own hands, or at the very least, you'll need to understand what the framework does "under the hood".

> You can't master browsers without mastering the DOM.

1.2 A Life Without jQuery

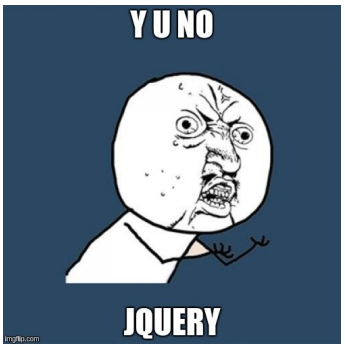


Fig. 2 *Y U No jQuery meme, created with the ImgFlip meme generator*

"*Why the hell aren't you using jQuery?*" is a rough translation of the expression in the meme in fig. 2, written in meme language (very popular in the web development scene).

Actually, we originally intended for this class to be a class about jQuery, and even completed three lessons on it. However, as we worked through our own projects and talked with dozens of developers at conferences, our experiences confirmed that the time is indeed ripe for a web without jQuery.

Don't get us wrong, we're not opposed to jQuery by any means! jQuery is a fantastic library and we really enjoyed using it in a number of projects. jQuery's primary achievement was to provide a stable platform which evens out the differences between browsers. jQuery managed to hide any and all browser eccentricities, no matter how weird they were, saving developers like us a lot of annoyance and thousands of development hours.

Not only that, jQuery also brought with it higher-order functions such as `map` and `each`. These were implemented in jQuery around the times of EcmaScript 3.1, which presumably inspired the EcmaScript standard to also integrate similar functionality directly into that language.

> I can't even begin to describe how crazy the web world was before libraries like jQuery and Prototype came along to handle the differences between browsers. I spent many a night poring over

documentation and debugging JS code just to find out why something which would work perfectly in all other browsers would behave completely differently in IE5/IE6.

At any rate, the web is continuing to evolve. Assuming that *Internet Explorer 9* is the oldest browser you'll need to support, the differences between browsers are no longer as great as they once were. Of course, you still need to consider your specific target group and your actual application, but the need to use jQuery (or comparable libraries) is no longer as essential as it was a few years ago.

By this time, an entire community has formed around the subject of #nojquery (Twitter hashtag). Here are a few more links on the subject you can browse:

- http://youmightnotneedjquery.com
- http://blog.garstasio.com/you-dont-need-jquery
- http://www.sitepoint.com/do-you-really-need-jquery
- http://lea.verou.me/2015/04/jquery-considered-harmful
- http://tutorialzine.com/2014/06/10-tips-for-writing-javascript-without-jquery
- http://programmers.stackexchange.com/questions/166273/advantage of-using-pure-javascript-over-jquery

The website http://youmightnotneedjqueryplugins.com even provides a list of jQuery-free widgets and plugins. Finally, you can also find direct

comparisons of jQuery and vanilla code (which we'll explain shortly) in Ray Nicholus' book (2015) *Beyond jQuery*.

1.3 JavaScript in Vanilla

Our general idea in this class is to work with ***vanilla JS*** for as long as this is feasible. The term *vanilla JS* just means basic JavaScript, without any additional bells, whistles or other accessories. Your reward for developing in vanilla JS will be small file sizes and high performance.

FYI, another term with the same meaning as vanilla JS is *naked JS* (unfortunately our editors deleted all the jokes on this we were going to use ;)) If you visit http://vanilla-js.com, you'll find a satirical page created by Eric Wastl, who extols *vanilla JS* as a library or framework which he offers as a download — a completely empty file :)

> *The Vanilla JS team takes pride in the fact that it is the most lightweight framework available anywhere; using our production-quality deployment strategy, your users' browsers will have Vanilla JS loaded into memory before it even requests your site.*
>
> vanilla-js.com

Once you get past its humorous aspect, Wastl's page actually is worthwhile to browse, especially its intriguing comparisons between vanilla JS and popular libraries — comparisons which show that "pure" JS is actually not a bad choice at all.

1.4 Prerequisites

We assume that you've either worked your way through volume 1 of this series ("*JavaScript — The First Step is the Easiest*") or have equivalent knowledge of basic JavaScript. If you're not familiar with concepts such as variables, control structures (e. g. `if`), recursion and higher-order functions (`map`, `reduce`, etc.), we recommend you first get a handle on the basics (incidentally, volume 1 *JavaScript — The First Step is the Easiest* (Emrich, Marit 2015) is perfect for this :))

> This class consistently uses JavaScript language version ***ES6***, also known as ***EcmaScript 2015***.

You should therefore also be familiar with basic ES6 features such as string templates, `let`, `const` and the *fat arrow* `=>`.

1.5 What This Class Is

This class concentrates on JavaScript in the browser. In particular, we cover browser DOM and present examples of how you can actually use it in real-life projects. In other words, you'll learn browser functions by looking at code samples which are virtually identical to code we'd write in real life. As a result, don't expect this class to be a fireworks display of features that we go through one by one. Instead, we'll show you practical, useful ways of how you can use new functions and concepts in actual environments. Many of our code samples also walk you though a series of improvements whose final code will be something of which you can truly be proud.

1.6 What This Class is NOT

If you use JS at some later point to develop software, you'll probably have to go back and look up some details. However, this class has only limited value as a reference text — it's intended more as a guide to give you a quick start in using browser DOM. We've organized the lessons in this class with the goal of optimizing your learning progress and not with the goal of helping you find a specific topic quickly. We also make no claims to completeness and are in fact consciously limiting ourselves to those topics which, for the moment, are most important to you as a beginner — in other words, the "big picture".

If in the future, as you're doing your day-to-day development work and find you need to reference JS in its entirety including all its gory details, we can recommend the following references:

- www.mozdev.org — Mozilla Developer Network (MDN)
  The MDN much more than just a reference for the Firefox browser. You'll also find detailed information on the core JS language and on browser functionality. The MDN even includes tables describing what browser implements what feature as of what version, and often provides polyfills (code which allows earlier browser versions to support a feature they lack) for worst-case scenarios.
- www.webplatform.org
  A joint documentation project from W3C, Microsoft, Adobe and other organizations. Very good, highly detailed documentation of HTML5 and CSS. Documentation of JS is still in progress but already looks very promising.
- caniuse.com

Can I use a certain feature? This site will give you the answer! Detailed overviews of new features in HTML, CSS and JS, including information on browser versions as well as polyfills.

If you need to Google for JS content, we recommend you add "mdn" to your search. This will land you directly on the right Mozilla Developer Network page — otherwise, you might get web pages in your hit list whose explanations are out-of-date or are simply wrong.

1.7 Why Firefox?

In this course, we use Mozilla Firefox (version 46+, English version) as a browser model. The advantage of Firefox is that it already supports all the ES6 features we'll be using in this course.

If you do want to use a different browser, we generally have no objection — just make sure it already comes with all the ES6 features used in the course. You'll find an overview of browsers and their support for ES6 in the ES6 compatibility table.

1.8 "NerdWorld" Project

In the first volume, we introduced the *NerdWorld* project and its client *Marty*. NerdWorld is a fictitious project intended to teach you a lot about JS — primarily where it's a good idea to use JS, and how you can do this well.

Again, Marty has all sorts of ideas and a bunch of intriguing challenges for you to implement — challenges you'll no doubt meet **with ease**, thanks to JavaScript!

If you already know the project from the first class in the series *JavaScript — The First Step is the Easiest*, you can skip the following section. Otherwise, we'll just reiterate the project description for you.

A Client Meeting

We'll say your first client is *Marty*. Marty owns a store and now wants to make his products available over the Internet as well. The name of his store is "*NerdWorld*" (this example is entirely fictitious — you can spare yourself the Google search :).

NerdWorld is an online shop where nerds, techies and other kindred spirits can buy their daily necessities: Nerf guns, UFOs with a USB connection, Klingon weapons for office use, caffeinated beverages, highly-caffeinated beverages, super-highly-caffeinated beverages, etc. In other words, everything a modern developer needs to survive in the office jungle.

Even though NerdWorld already has a static HTML site up, you can't buy any products through it yet. The new site is to consist primarily of the

shop. In order to make the overall experience a little more interesting, existing features will be enhanced and additional features are also being planned. These include:

- a chat client through which customers can chat with salespeople/customer consultants
- an online newsletter through which the shop can inform customers of new products.
- a newsboard through which customers can learn about the shop's latest offerings.

To make his dream reality, the shop's owner is now turning to an expert — you! You meet with him and he describes the situation:

For years, we've been selling cool stuff geared to nerds and any other people who like that kind of thing. Our assortment of products ranges from items for Star Trek fans and paraphernalia for programmers and extends all the way up to smartphone-controlled mini-drones and joke items like an electroshock pen.

Throughout this course, we'll keep coming back to the *NerdWorld Project* to show you how JS and the DOM make it possible for you to overcome each new challenge it presents.

2 Your First Challenge in DOM Manipulation

2.1 "Hello DOM" Using querySelector and innerHTML

Fig. 3 *Photo: oskay*
License: CC BY 2.0

Any serious class on programming starts with a *Hello World* example, and we certainly don't want to be the ones to break that tradition. In volume 1 of this series, we printed out *Hello World* to the console, but this class focuses on HTML structure. So as our first example, we'll "say" *Hello World*, but on an HTML page instead of the console. To do this, you'll need:

1. a simple HTML page (listing 1)
2. the JavaScript console provided with your browser

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Hello World</title>
</head>

<body>
```

```
10
11    <h1>Hello <em>World</em></h1>
12
13  </body>
14
15  </html>
```

Listing 1 *Simple HTML document*

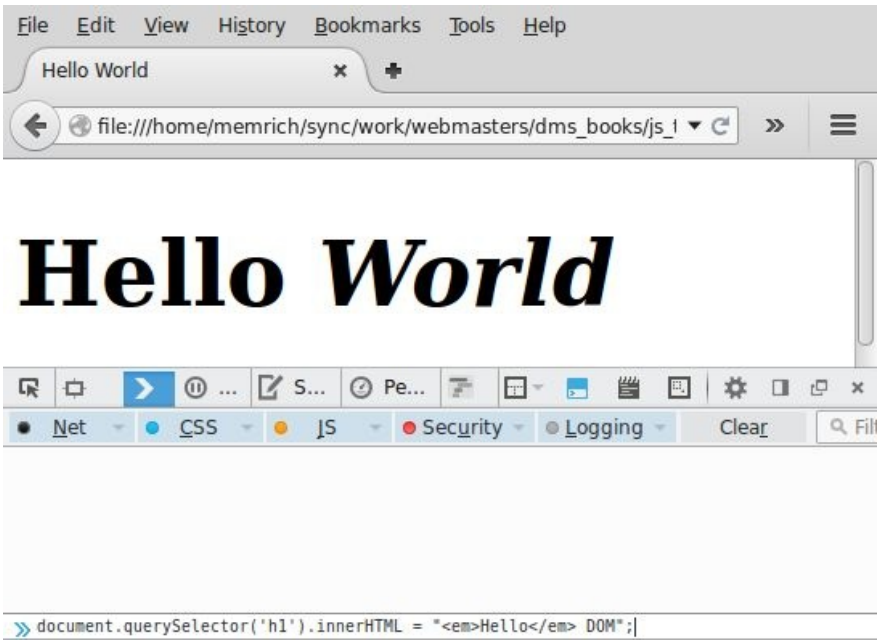


Fig. 4 *Hello World in Firefox*

Walkthrough 1:

Open your HTML document from listing 1 in Firefox. You'll find the file in the accompanying material to this class, in *02/examples/simple.html* Open the Web Console (**`Ctrl-Shift-K`**) and enter the following code on the JavaScript command line (fig. 4):

```
document.querySelector('h1').innerHTML = "<em>Hello</em> DOM";
```

After you hit **`Enter`** to confirm the command, the heading will change from *Hello World* to *Hello DOM*, with the markup code changing accordingly (see fig. 5).

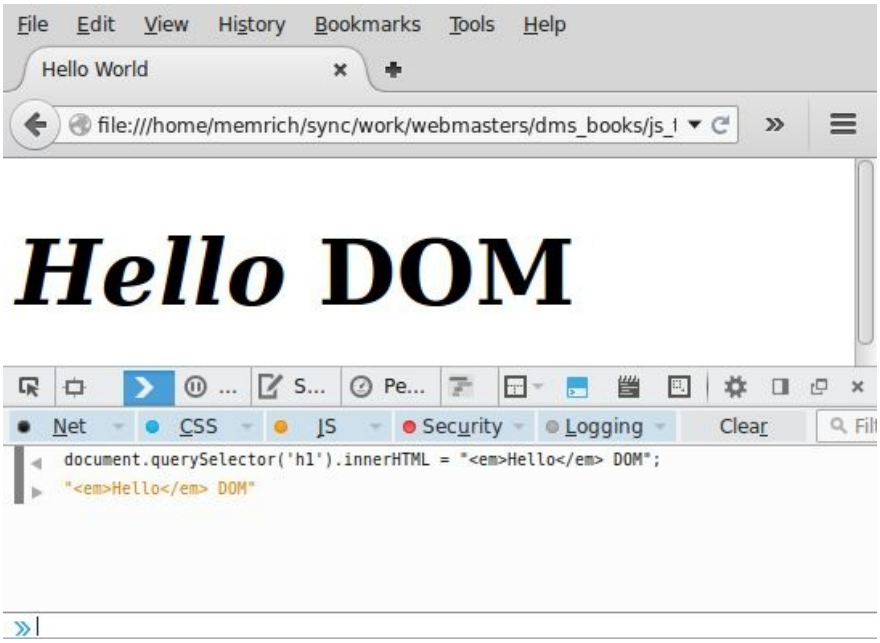


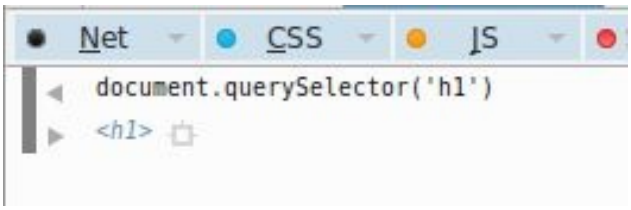
Fig. 5 *Hello DOM in Firefox*

2.2 A Look Behind the Scenes

So exactly how does it work?

Let's take a look at the overall process in slow motion. Okay, this class doesn't really provide a slow motion feature, but it does come with rewind (i. e. a browser reload) and we can step through each of the two parts of our statement.

> So reload the HTML page (**`Ctrl-R`** or **`cmd-R`**) to get back to the original *Hello World* page.
>
> Enter just `document.querySelector('h1')` (without the assignment part of the statement).



The function `document.querySelector` allows you to select HTML elements within an HTML document. It takes any CSS selector as a parameter. So, the command `document.querySelector('h1')` will return the first `h1` element on the page, as a JS object.

(We realize we still haven't told you exactly what an ***object*** is. All you need to know at this point is that objects are values, such as numbers or strings. Objects can also represent more complex things, such as the `h1` element in our current example.) You can also inspect the `h1` element in the console. Hovering your mouse over `h1` in the console will cause the

element to be highlighted in the HTML document. If you then click the symbol which appears next to the element in the console, the Firefox Inspector will show you the element's location in the HTML tree (see fig. 6).

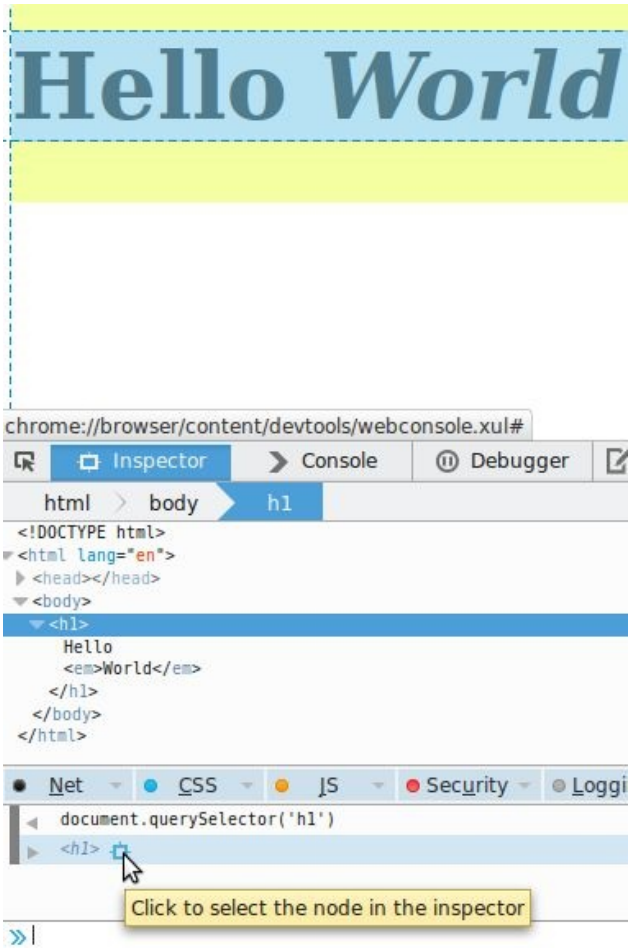


Fig. 6 *h1 in Firefox Inspector*

The value returned by the `document.querySelector` function is therefore a JS object which represents an element in the HTML tree. More precisely, it's a so-called element object. Objects in JS have ***properties***. One fundamental property (especially in element objects) is *innerHTML*. You specify its value using dot notation, e. g. `document.querySelector('h1').innerHTML`.

Now enter the following command into your console:

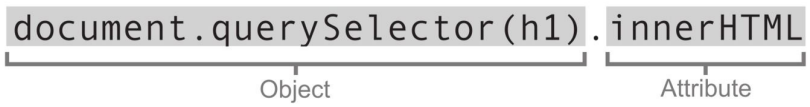
```
document.querySelector('h1').innerHTML
```

You get back the output:

```
Hello <em>World</em>
```

`document.querySelector('h1')` returns the element object whose *innerHTML* property you want to access. The value of `innerHTML` is a string containing the *inner HTML* of the given element — in other words, the HTML source code which exists between the element's opening and closing tags.

Dot notation also works with other objects and properties. Dot notation here is always in the form: `<object>.<property>`



You can use properties like variables. Just like with variables, you can also assign new values to properties — and as soon as you assign the property `innerHTML` a new value, your HTML page will change.

Enter:

```
document.querySelector('h1').innerHTML = "Hello again!";
```

We also said above that objects are really just values, but haven't yet given you proof of that. So here it is — just like any other value, you can

easily bind an object to a variable name.

Enter:

```
let myH1 = document.querySelector('h1');
myH1.innerHTML = "Hello once more!";
```

And if you now enter `myH1` into the console, you'll get back the HTML element object.

Try it for yourself! Just enter:

```
myH1
```

Exercise 1: Almost Famous Quotes

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />

  <title>Famous Quotes</title>

  <link rel="stylesheet" href="../../css/styles.css" type="text/css" media="screen" />
  <link rel="shortcut icon" href="../../img/favicon.ico" type="image/x-icon" />
</head>

<body>

  <header></header>

  <main>
    <h1>Famous Quotes</h1>

    <blockquote></blockquote>
  </main>

</body>

</html>
```

accompanying_files/02/examples/quotes.html

Use JS to change the (`h1`) heading of the HTML page in listing 2 to

Almost Famous Quotes.

The page (listing 2) contains an empty `blockquote` element. Open the HTML page in Firefox. Enter a line of JS code in your console which fills the element with the following content:

```
<p>I have always wished for my computer to be as easy to use as my telephone; my wish has
```

2.3 Using the DOM to Search For and Find Elements

You've seen how you can use `document.querySelector` to extract HTML elements as JS objects. Actually, any HTML document is just a collection of HTML elements arranged in a tree structure. Similarly, the corresponding JS objects are also arranged in a tree — the so-called ***Document Object Model***, or ***DOM*** for short. In the case of the simple HTML page from listing 3, the DOM would look something like fig. 8.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>DOM-Example</title>
  <link rel="stylesheet" href="css/styles.css" type="text/css" media="screen" />
</head>

<body>

  <header>
    <nav>
      <ul>
        <li></li>
        <li></li>
        <li></li>
      </ul>
    </nav>
  </header>

  <main>
    <article>
      <h1></h1>
      <p></p>
      <img alt="" src="#" />
    </article>
  </main>

  <footer>
    <nav></nav>
  </footer>

</body>

</html>
```

Listing 3 *accompanying_files/02/examples/dom_example.html*

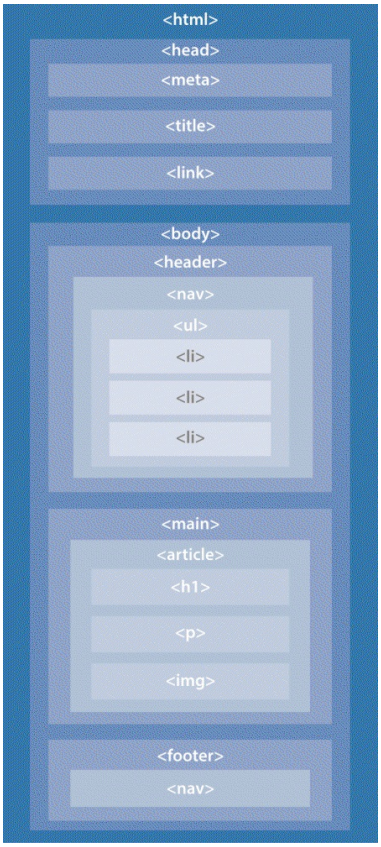


Fig. 7 *Visualization of nested structure of an HTML page*

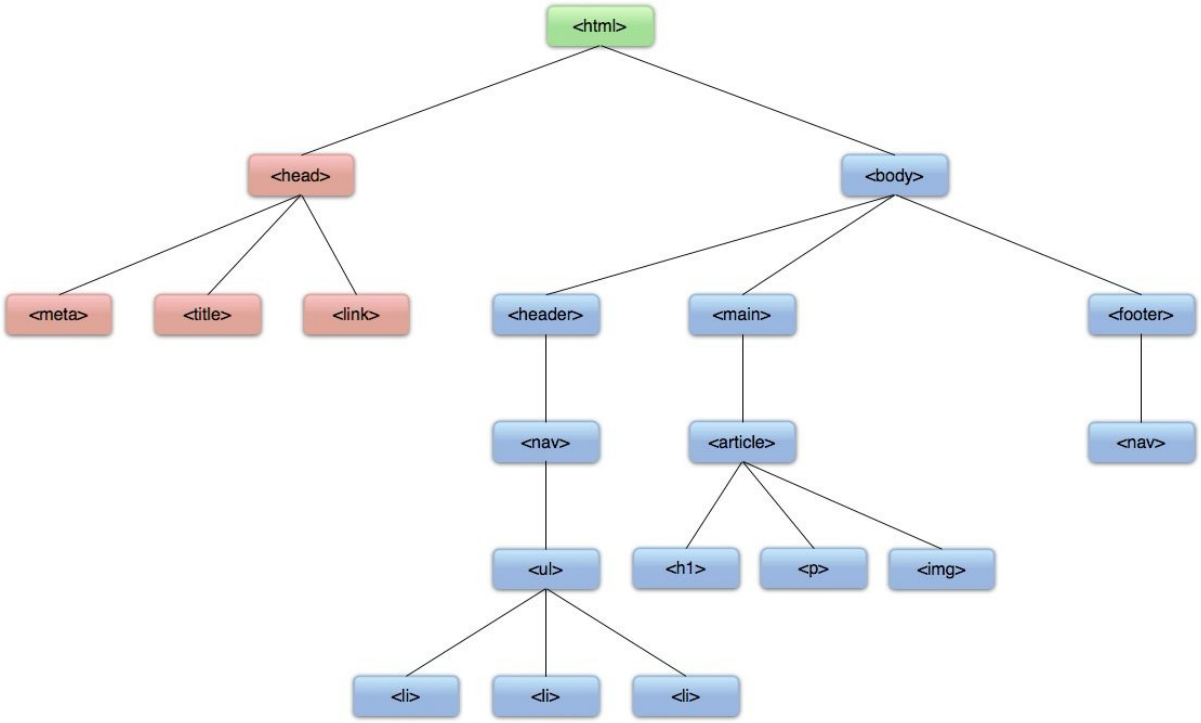


Fig. 8 *Tree structure of HTML page*

The function `document.querySelector` always selects and returns the first element which matches the CSS selector you pass it. This selector can be any valid CSS selector. table 2.1 shows the most essential selectors.

Name	Example	Description
Universal selector	`*`	Selects any element. You shouldn't use this selector without limiting your selection further, since selecting **all** the elements on a web page can take up a lot of computing time.
Type selector	`h1`	Selects all elements of a specified type.
ID selector	`#some-id`	Selects exactly **one** element with the specified `id`.
Class selector	`.some-class`	Selects all elements which were assigned the specified class. Remember that an element can have more than one class.
-	`h1, p, .go`	The comma itself isn't a selector, but makes it possible to combine multiple selectors. This example would select all `h1` elements, all `p` elements and all elements with the class `go`.

Table 2.1 *Basic CSS3 selectors*

Exercise 2: 01001000010011110101010O — Part 1

Fig. 9 *Hot Binary Heat Changing Mugthinkgeek, © 2016 ThinkGeek, Inc. All Rights Reserved.*

Now it's your turn — create some queries of your own. Use the following HTML document as your training grounds.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />

  <title>01001000010011110101010100</title>
  <meta name="description" content="JavaScript. HTML mühelos manipuliert" />

  <link rel="stylesheet" href="../../css/styles.css" type="text/css" media="screen" /
  <link rel="shortcut icon" href="../../img/favicon.ico" type="image/x-icon" />
</head>

<body>

  <header></header>

  <main>
    <h1 class="article">Hot Binary Heat Changing Mug <span class="keyword">"010010000

    <img alt="Hot Binary Heat Changing Mug" src="../../img/thinkgeek_2024_hot_binary_

    <h2>Description</h2>

    <p>Numbers make up <span class="special">everything</span> in our digital world.

    <p>See, the <span class="keyword">Hot Binary Heat Changing Mug</span> looks like

    <p><img alt="01001000010011110101010100" src="../../img/thinkgeek_2024_hot_binary_h

    <h2>Product Specifications</h2>

    <ul id="product_specification">
      <li class="keyword">Hot Binary Heat Changing Mug</li>
      <li>As you add hot liquids, the binary for "HOT" appears (read from left to rig
```

```
      <li>A <span class="keyword">ThinkGeek</span> creation and exclusive!</li>
      <li>Care Instructions: <span class="i b">Hand wash only. Not microwave or dishw
      <li>Materials: Ceramic</li>
      <li>Dimensions: approx. 3.15" diameter x 3.75" tall</li>
    </ul>

    <h3>You wanna buy it?</h3>

    <p class="buy_info_text">If you like to buy this brilliant mug, just do the follo

    <ol class="model" data-model="LDV73C-X3">
      <li>Select how many items do you want.</li>
      <li>Press "buy".</li>
    </ol>

    <form id="buy_form">
      <select>
        <option>1 item</option>
        <option>2 items</option>
        <option>3 items</option>
        <option>4 items</option>
      </select>
      <input type="button" value="buy" />
    </form>
  </main>

  <footer>(C) by ThinkGeek &ndash; Produkttext mit freundlicher Genehmigung von Think

</body>

</html>
```

accompanying_files/02/exercises/010010000100111101010100.html

Enter a query in the console which will:

1. find the `h1` element.
2. find the element with the `id` *buy_form*.
3. find the element with the `id` *product_img*.

2.4 One Function To Find Them All — querySelectorAll

All articles on our news page have their own heading. We need you to add the text `Current:` before each of these headings.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Headlines</title>
  <link rel="stylesheet" href="../../css/styles.css" type="text/css" media="screen" />
</head>

<body>
  <h2>Lorem ipsum dolor sit amet.</h2>
  <h2 class="news">Quod eligendi saepe voluptates eveniet.</h2>
  <h2 class="news">Architecto reiciendis magnam modi inventore.</h2>
  <h2 class="news">Modi ipsum, velit rem ipsam.</h2>
  <h2 class="news">Provident, officia quisquam aliquam deserunt!</h2>
</body>

</html>
```

Listing 5 *accompanying_files/02/examples/news.html*

You often come across situations where you need to select or change multiple elements all at once. The function `querySelectorAll` is provided for just that purpose. In contrast to `querySelector` it returns **all** elements which match the selector you pass it, instead of just the first one it finds.

Open the page *news.html* (listing 5) and enter the following code in the console.

```
document.querySelectorAll('h2')
```

You get back:

```
NodeList [ <h2>, <h2.news>, <h2.news>, <h2.news>, <h2.news> ]
```

A ***NodeList*** is an object which combines multiple elements (nodes) into a list. The term *node* here refers to a node in an HTML tree. A node always has only one parent element, but may have any number of child elements (including none at all).

A *NodeList* behaves like an array, so you can use an index operator to access the elements in it. For example,

```
document.querySelectorAll('h2')[0].innerHTML
```

will return the text of the first `h2` heading, i. e. the text *Lorem ipsum dolor sit amet.* in the example.

If you want to find out how many nodes a selector has found (or whether the selector has found any elements at all), you can use the property `length` (again, just like an array):

```
document.querySelectorAll('h2').length
// => 5
```

```
document.querySelectorAll('img').length === 0
// => true, means no images found
```

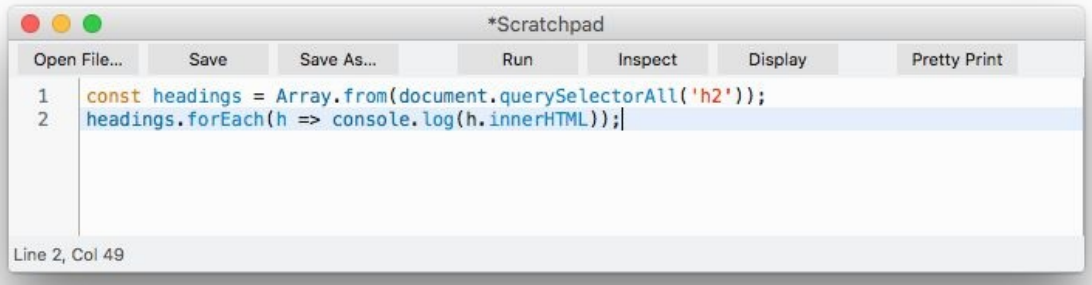
Getting back to Marty's latest requirement, of course it won't be enough for us just to select all the headings — we'll also need to change them (or more specifically, the text within the header tags).

To do this, you'll need to use a *forEach* to loop through the headings in the `NodeList`. However, `forEach` is defined only for arrays — not for

NodeLists. Fortunately, by using `Array.from`, we can convert a NodeList to an array.

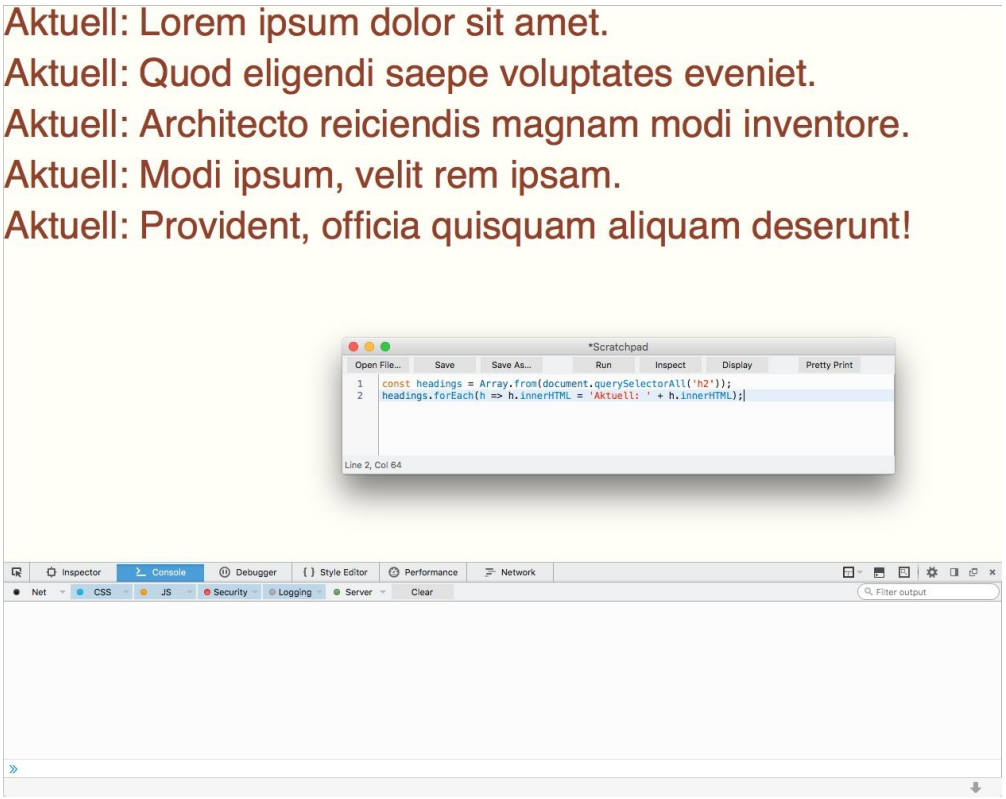
```
const headings = Array.from(document.querySelectorAll('h2'));
headings.forEach(h => console.log(h.innerHTML));
```

FYI, since this code does contain a number of lines, you definitely might want to save it — in fact, it'd probably be better for you to use Scratchpad (**`Shift-f4`**) instead of the console from now on.



So now instead of just printing out headers, you should also change them.

```
const headings = Array.from(document.querySelectorAll('h2'));
headings.forEach(h => h.innerHTML = 'Aktuell: ' + h.innerHTML);
```



Exercise 3: 01001000010011110101010O — Part 2

Continue to build upon the article page, and create some queries of your own. Again, use the HTML document from listing 4 as your training grounds.

Enter a query in the console which will:

1. find all `li` elements.
2. find all `h2` elements.
3. find all elements with the class *special*.
4. find all `li` elements with the class *keyword*.
5. find all `span` elements with the class *special*.
6. find all elements which have the classes *i* AND *b*.

Exercise 4: Agents and Attorneys

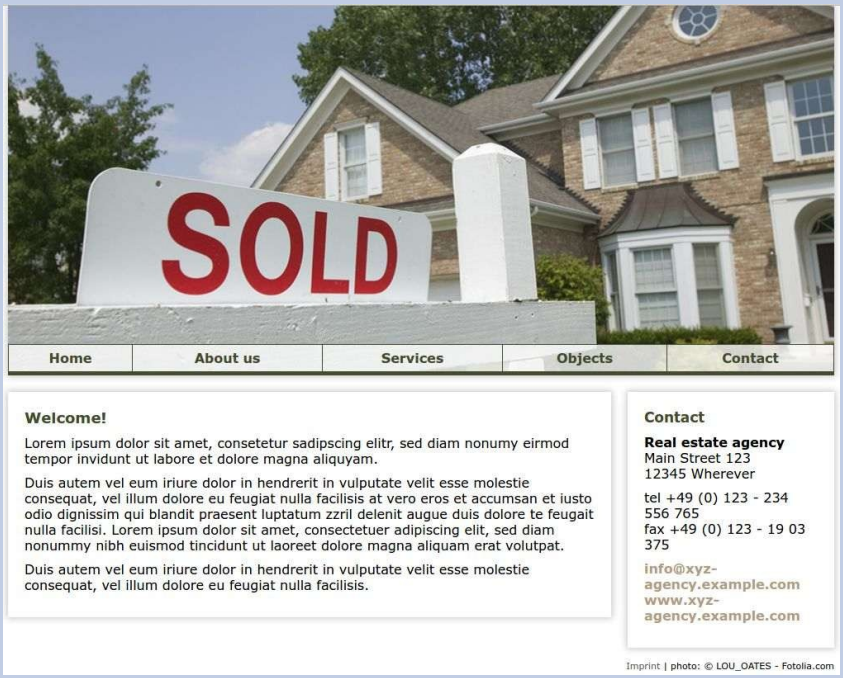


Fig. 10 *additional_files/02/exercises/real_estate.html*

Pretend that you have a fully completed website for a real estate agency tucked away somewhere. Now a new client, a law firm, comes to you, and just by coincidence you find that your agency website would also be perfect for lawyers. Of course, you can't show your new client your "new" design while it still contains the text about the real estate agency. But instead of fiddling around with copy & paste and laboriously replacing the text section by section, you decide to write code that will replace the existing text with **dummy text**. You can use one of the many notorious lorem ipsum generators out there to do this (or even better, the ingenious Bacon Ipsum or Veggie Ipsum generator) — it's all up to you!

Write a function which will replace the content within all `h1` tags.

Write another function which will replace the content within all `p` tags.

Text in navigation elements should also be replaced.

2.4.1 Attribute Selectors

The last exercise was a good warm-up, but there're still a lot more CSS selectors out there — and of course, any valid CSS selector can also be used in JS.

The next CSS selectors you should take a look at are so-called ***attribute selectors***. You can use these to access elements that have specific attributes and values. table 2.2 gives you an overview of common attribute selectors.

Selector	Example based on `<img src="funny_cat.png" ... />`	Description
[*attr*]	`[src]`	Selects elements with the specified attribute. The actual value of the attribute plays no part in the selection (***attribute presence***).
[*attr=val*]	`[src='funny_cat.png']`	Selects elements which contain the specified attribute (*src*) with exactly the specified value (*funny_cat.png*) (***exact attribute match***).
[*attr**=*val*]	`[src*=cat]`	Selects elements in which the value of the specified attribute (*src*) contains the specified value (*cat*) as a substring (***substring attribute match***).
[*attr*^=*val*]	`[src^=funny]`	Selects elements in which the value of the specified attribute (*src*) starts with the specified value (*funny*).
[*attr*$=*val*]	`[src$=png]`	Selects elements in which the value

		of the specified attribute (*src*) ends with the specified value (*png*).

Table 2.2 *CSS attribute selectors. All examples select *

2.4.2 Combinator Selectors

Next, we take a look at ***combinator selectors***. These selectors allow you to find elements based on their relationship to other elements. An ordinary blank space, with which you're no doubt well acquainted, is such a combinator. You use it to select the descendants of a specified element. You'll find other combinator selectors in table 2.3.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Selector Testing</title>
</head>

<body>
  <ul id="ul1">
    <li id="li11"><span>11</span></li>
    <li id="li12">12</li>
    <li id="li13">13</li>
    <li id="li14">14</li>
  </ul>
  <ul id="ul2">
    <li id="li21"><span>21</span></li>
    <li id="li22"><strong><span>22</span></strong></li>
    <li id="li23">23</li>
  </ul>
</body>

</html>
```

Listing 6 *accompanying_files/02/examples/selector_testing.html*

Name	Example	Description
Descendant combinator (blank space)	`li span`	Selects elements (in this case `span`) which are descendants of a different specified element (in this case `li`) — regardless of how many levels separate the specified element and the descendant.

		Result: `[<span>11</span>, <span>21</span>, <span>22</span>]`
Adjacent sibling combinator (+)	`li#li11 + li`	You can use this selector to select the element which **immediately follows** the specified element (in this case `li#li11`) in the HTML source text and is on the same level (sibling element). If the element which follows the specified element in the document doesn't match the element specified after the + symbol, the selection returns nothing. **Result:** `[<li id="li12">12</li>]`
General sibling combinator (~)	`li#li12 ~ li`	You can use this selector to select all elements which follow the specified element (in this case `li#li12`) in the document and are on the same level (sibling element). This selector **doesn't just select the element which immediately follows** the specified element, as does the *next sibling combinator*. **Result:** `[<li id="li13">13</li>, <li id="li14">14</li>]`
Child combinator (>)	`li > span`	Child selectors are one variant of descendant selectors. You can use child selectors to select elements which are **direct** "children" of higher-level parent elements. The child selector in the example only selects `span` elements which appear directly within an `li` element. `<span>22</span>` appears within a `strong` element and is therefore not selected. **Result:** `[<span>11</span>, <span>21</span>]`

Table 2.3 *Combinator selectors — results based on listing 6*

As you see, CSS selectors are very practical and easy to use. Since you've probably already known many of these selectors for some time, it shouldn't be too hard for you to remember a few more.

Exercise 5: 01001000010011110101010O — Part 3

Let's go back to our 01001000010011110101010O mug. Create appropriate selectors to

1. find the first `li` which appears within the `ul` with the `id`*product_specification*.
2. find the first `span` element which appears within the first `h1` element with the CSS class *article*.
3. find all elements with the class *keyword* which appear within `p` elements.
4. find all `li` elements which appear within `ul` elements.

2.4.3 Pseudo Classes

So-called *pseudo classes* allow you to specify your selectors even further.

Name	Example	Description
:first-child	ul li:first-child	This pseudo class allows you to select the first child element in a higher-level container (in this case `ul`), where the child element has the specified tag type (in this case `li`). If the first child element doesn't have the specified tag type, the selection returns

		nothing.
:nth-child(x)	ul li:nth-child(5)	This pseudo class selects the xth child element (in the example, this would be the 5th element which appears within `ul`, if the element has a tag type of `li`).
:last-child	ul li:last-child	Just like `:first-child`, you use this selector to select the last child element, assuming it has the specified tag type.
:only-child	#article p:only-child	This class selects an element only if it's the only child of the specified parent element.
:empty	div:empty	You can use this pseudo class to select only those elements that have no child elements. Note that the pseudo class also considers content in the form of text to be a child element.
:not(selector)	:not(span)	Negation. Selects those elements which don't match the selector specified in parentheses. In the example, everything except `span` elements would be selected.

Table 2.4 *Pseudo Classes*

Of course, the selectors listed above by no means represent all the selectors available to you, but you'll get pretty far with them in practice. If you do need to learn about other selectors, you can check out the official W3C specification or online documentation like MDN or Webplatform.

2.5 Document vs. Element

The methods `querySelector` and `querySelectorAll` can be applied **globally**, over the `document`, and also **locally**, on individual elements. Global means that the selector searches through the entire document. Local means that only the children of the given element node are included in the search.

Example

HTML document:

```
...
<body>
  <ul>
    <li>1</li>
    <li>2</li>
  </ul>
  <ul id="second_list">
    <li>3</li>
    <li>4</li>
  </ul>
</body>
```

JavaScript in the console:

```
document.querySelectorAll('li')
// => <li>1</li><li>2</li><li>3</li><li>4</li>

document.querySelector('#second_list').querySelectorAll('li')
// => <li>3</li><li>4</li>
```

Listing 7 *// => indicates the return value*

2.6 Short and Sweet — $

Many JS libraries and frameworks such as jQuery and Prototypeprovide a function called `$` (or `$$`). This function corresponds more or less to `document.querySelector` and/or `document.querySelectorAll`, with small differences based on the specific implementation. Since you'll be using `document.querySelector` quite frequently in practice, the shorthand notation `$` will come in very handy. Not only is the single character much faster to type, it also improves readability, especially in long lines of code.

You don't even need to import a complete library or large framework into your project to take advantage of this shorthand notation — you can get the same effect simply by renaming those functions. Just bind the functions `document.querySelector` and `document.querySelectorAll` to the two identifiers `$` and `$$`:

```
1 const $  = document.querySelector.bind(document);
2 const $$ = document.querySelectorAll.bind(document);
```

Unfortunately, the `.bind(document)` command is indeed necessary — this is due to a minor imperfection in the language which we won't go into just yet.

After you add those two lines to your code, you'll be able to use the shorthand notation `$` and `$$` anywhere.

FYI, Firefox and Chrome consoles already come complete with the functions `$` and `$$` — although, unfortunately, only for use in the console (but you'll still find that to be very practical in many situations).

Exercise 6: 010010000100111101010100 — Part 4

Let's go back to our 01001000010011110101010O mug. Create appropriate selectors to

1. find all images whose file names end in `jpg`.
2. find all `input` elements of type `button` which appear within forms.
3. find all elements which have the class *model* as well as an attribute *data-model* which contains the value *V7*.
4. find all images which do **not** contain the class `float_left`.
5. find all **second** list items.
6. find all (`ul`) lists which immediately follow a level 2 header (`h2`).

2.7 Summary

Method	Description	Interface/API
querySelector	returns the **first** element which matches the specified selector. Only searches through the elements within the HTML subtree of the root element on which `querySelector` was called.	Element
querySelectorAll	returns **all** elements which match the specified selector. Only searches through the elements within the HTML subtree of the root element on which `querySelectorAll` was called.	Element
document.querySelector *(aka $)*	returns the **first** element which matches the specified selector. Searches through the entire HTML document.	Document
document.querySelectorAll *(aka $$)*	returns **all** elements which match the specified selector. Searches through the entire HTML document.	Document
document.body	returns the `body` element.	Document

Table 2.5 *This table shows the most essential methods for* ***selecting*** *DOM elements. You'll find additional methods for selecting DOM elements in the MDN, in the descriptions of the*

corresponding APIs: Element & Document.

2.8 Exercises

Exercise 7: 01001000010011110101010O — Part 5

For some reason, a client of yours wants some of the text on his website to be displayed in gray.

Write a function that assigns the CSS class `gray` to all `p` elements.

As an exception, your client decides only body text which describes a purchasing transaction should be displayed. So `p` elements with the class `buy_info_text` should be excluded from the operation above.

Now, assign the class `gray` to all list items which still have **no** other class.

Exercise 8: Length Matters???

One of your clients comes to you with the following requirement:

Unfortunately, many of our website visitors are getting a little short with us. Some visitors to our online forum have said they'd like to be given some idea of how long an article is from the header (`h1`) of the article itself. Our designer came up with three styles for headers, and saved these as appropriate CSS classes.

Measure the length of an article in characters and automatically assign the correct class to headers.

- `coffe_break_article` up to 3000 characters
- `normal_length_article` up to 9000 characters
- `lone_weekend_article` 9000 characters and greater

Your code will focus on the article detail pages. Assume that each page has exactly one `h1` header and the entire article (including images and HTML structure) appears in a single element with the id *content*.

To count characters, you should evaluate the entire HTML code in *content*. You only need to provide a rough estimate.

Following is a sample HTML file you can use to test your code. You'll find the sample file in the accompanying material:

```
<!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Article Detail Page</title>
  </head>
  <body>
    <h1>Lorem ipsum</h1>
    <section id="content">
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.</p>

...

      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit
in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
officia deserunt mollit anim id est laborum.</p>
    </section>
  </body>
</html>
```

You can test different text lengths just by duplicating or deleting lorem ipsum paragraphs.

The Insidiousness of Those Apparently Insignificant Errors

Anyone who's searched in desperation for hours to find an a error

only to discover that the problem is just a syntax error, typo or other careless mistake, knows how irritating that can be. My own insight on strange errors is that — they could be ANYWHERE! Well, maybe not anywhere, but certainly in places you might not expect. They'll then lead to, more often than not, the strangest and most illogical problems.

So: Careful, painstaking work always pays off, no matter whether you're writing "just" HTML & CSS, JS or code in another language.

3 And Down into the Depths of HTML

Entering all your commands through the console one by one is slow, if not downright tedious. So now, we'll ask you to save the code from the last lesson in a JavaScript file.

👣 Walkthrough 2:

Open any text editor of your choice, e. g. Scratchpad.

Create a new file if necessary Now copy the code from into this file and save it under the name *highlight_chat_members.js*
Include the JS file in the HTML file *additional_files/03/examples/chat.html*:

```
<head>
  …
  <script
    src="highlight_chat_members.js"
    defer="defer"></script>
</head>
```

As soon as you open the HTML file in your browser and the page finishes loading, your JS code highlights the members matching the search.

Now, change the `searchFor` string from `'ert'` to `'a'`.

Reload your browser (**`Ctrl-R`** or **`cmd-R`**).

The script now highlights different users, those who match the new search and whose names contain `'a'`.

Change the `searchFor` string from 'a' back to its original value `'ert'` and reload your browser again.

3.1 Delays Without Waiting: defer & async

You may have noticed that the `script` element has also been assigned the attribute `defer="defer"`. What does `defer` do?

Try removing the attribute and see what happens. You'll see that our highlighting feature now no longer works. This is due to a timing problem. For our highlighting feature to work, the corresponding DOM elements (here, the current chat members) must already exist by the time the JS code accesses them. However, those elements don't yet exist at the time your browser loads the script (specified in the HTML `head` section). The browser only finds out about them later, further down in the `body` section.

This problem is easily solved — just add the `defer` attribute to the `script` element. `defer` delays execution of the script so that the page runs the script only after it has completely finished building the DOM.

`defer` also provides another advantage — your browser loads the script *at the same time* it parses the rest of the HTML page. This means the HTML construction doesn't need to wait until the JavaScript file has been loaded and executed, and the HTML is displayed faster.

A related attribute is `async`. Just like `defer`, it causes your browser to load the JS file at the same time it's loading other files, so your browser doesn't need to wait for the file to finish loading before continuing to execute the HTML code . Unlike `defer`, however, the browser executes the JS file as soon as it's loaded. This means there's no guarantee the DOM has finished building. In addition, the order in which JS files are loaded and executed doesn't necessarily follow the order of the script tags in the HTML source code (so-called ***source order***). Each script

starts as soon as it's loaded — i. e. smaller scripts generally start up before larger ones.

Life in the Old Days

Browser didn't always "correctly" support the `defer` attribute, and developers used to have to rely on tricks. *Correctly* here means that IE browsers before version 11 did provide support for the attribute, but provided no guarantee of the sequence in which scripts were processed. Chrome and Firefox, on the other hand, have supported the attribute according to standards specifications for quite some time (Firefox 3.5 & Chrome 8).

An example of one of these tricks was to put scripts not in the `head` of the HTML but in its `body`, immediately before the closing body tag.

Each of these tricks came with specific drawbacks and/or other costs.

3.2 In Strict Isolation: Blocks & use strict

Now, you should also add a couple of other things to the JS file. First, it's always a good idea to add a `"use strict";` — this statement will allow you to take advantage of the optional stricter syntax checking provided by the JS engine. Then you should also pack all your code into a single block, i. e. within braces `{}`.

```
"use strict";

{

  // your code here
  …
}
```

A block establishes a scope, or a space in which variables may exist. Any variable (`let`) or constant (`const`) you place in a block only exists within the block and not globally — as a result, they won't conflict with any other identifiers in your code. You never know whether you or someone else is going to eventually add other JS files to your code which might have the same variable names.

Life in the Old Days

We used to use a so-called ***IIFE*** to prevent global identifiers from spreading everywhere. Unlike modern variables (`let`) and constants (`const`), variables we created using `var` didn't have block scope.

Since older variable forms couldn't be restricted to a single block, something else had to be done. Variables created with `var` do have

function scope — so they can be contained within a function.

And that's exactly the trick. We created anonymous functions then executed them immediately — not because we really needed to use a function, but just to restrict the variables' scope of validity:

```
(function() { ... Your code here ... })();
```

IIFE stands for *Immediately Invoked Function Expression* — a function expression which is executed immediately.

3.3 Just a Little Refactoring …

Here again is the complete file for the preceding examples:

```
{
  const searchFor = "ert";

  const liNodes = Array.from($$("#chat_members li"));
  const liNodesFound = liNodes.filter(liNode =>
    liNode.innerHTML.includes(searchFor));

  liNodesFound.forEach(li => li.classList.add("highlighted"));

  const $$ = document.querySelectorAll.bind(document);
}
```

Listing 8 *accompanying_files/04/examples/highlight_chat_members.js*

Since your code is now in a file and you don't have to type it directly into the console anymore, it becomes that much more important for you to make sure your code is readable and maintainable. Although the code from listing 8 is not exactly a prime example of clean code, you can make it so very quickly. FYI, making improvements to one's code without changing its behavior is known as ***refactoring*** (Opdyke, Johnson 1990). Abundant literature exists on this subject, including the famous book *Refactoring* by Martin Fowler (1999).

We need to point out that this class right now doesn't focus on the topic of *refactoring*. As a result, we're not going to stray too far from our main subject at this point to give you a comprehensive explanation of refactoring techniques. On the other hand, since we did say we'd teach you how to write code that you'd be proud of, we do at least want to show you how to make a couple of improvements to this small example, to give you some idea of *refactoring*. Plus, there's no way we could leave the code from listing 8 untouched — otherwise you'd show it to everyone you know, including the professional JS programmer next door, and our reputation would be down the tubes.

So — what can we do to improve this code?

Extract Method

First of all, the code shouldn't be just "out there" — let's pack it into its own function. `highlightChatMembers` might be a good function identifier:

```
"use strict";

{
  const highlightChatMembers = () => {
    const searchFor = "ert";

    const liNodes = Array.from($$("#chat_members li"));
    const liNodesFound = liNodes.filter(liNode =>
      liNode.innerHTML.includes(searchFor));

    liNodesFound.forEach(li => li.classList.add("highlighted"));
  };

  const $$ = document.querySelectorAll.bind(document);

  highlightChatMembers();
}
```

Listing 9 *accompanying_files/04/examples/highlight_chat_members_ref1.js*

Then instead of storing the search string as a constant, let's make it a parameter to our function. The search string will actually be entered later by the user. Given the current context, `partOfMemberName` is a little more accurate and more understandable identifier than `searchFor`.

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    const liNodes = Array.from($$("#chat_members li"));
    const liNodesFound = liNodes.filter(liNode =>
      liNode.innerHTML.includes(partOfMemberName));

    liNodesFound.forEach(li => li.classList.add("highlighted"));
  };

  const $$ = document.querySelectorAll.bind(document);

  highlightChatMembersBy("ert");
}
```

Listing 10 *accompanying_files/04/examples/highlight_chat_members_ref2.js*

In addition, the old function name `highlightChatMembers` is now no longer

appropriate. The name `highlightChatMembersBy` clearly shows that the function takes a parameter (in this case `partOfMemberName`). You could almost express the signature

```
highlightChatMembersBy(partOfMemberName)
```

as an actual sentence:

> *"Highlight chat members by a part of their name"*

One essential principle in programming is the ***single responsibility principle*** (Martin 2002). When applied to functions, it means that each function should always do only one thing. The function `highlightChatMembersBy` is definitely doing too much — it's finding chat members, filtering them, highlighting them, etc. One cure for this is the ***extract method*** (Fowler 1999) — i. e. extracting new methods and functions from existing code.

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    const liNodesFound = chatMembers()
      .filter(liNode =>
        liNode.innerHTML.includes(partOfMemberName));

    liNodesFound.forEach(highlight);
  };

  const chatMembers = () => Array.from($$("#chat_members li"));
  const highlight = el => el.classList.add("highlighted");

  const $$ = document.querySelectorAll.bind(document);

  highlightChatMembersBy("ert");
}
```

Listing 11 *accompanying_files/04/examples/highlight_chat_members_ref3.js*

The function `chatMembers` is now responsible for finding members, while `highlight` takes care of highlighting a specific element. Our code for checking for search matches is another candidate for a separate function:

```
liNode.innerHTML.includes(partOfMemberName)
```

A good identifier name for that function would be `doesMemberMatch`. listing 12 shows the new code with the functions we extracted.

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    const liNodesFound = chatMembers()
      .filter(memberElement =>
        doesMemberMatch(partOfMemberName, memberElement));

    liNodesFound.forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.includes(partOfMemberName);

  const chatMembers = () => Array.from($$("#chat_members li"));
  const highlight = el => el.classList.add("highlighted");

  const $$ = document.querySelectorAll.bind(document);

  highlightChatMembersBy("ert");
}
```

Listing 12 *accompanying_files/04/examples/highlight_chat_members_ref4.js*

You may have noticed that we essentially don't need the constant `liNodesFound` anymore. Also, the name *liNodesFound* doesn't necessarily help to make the code more understandable. So let's just remove those identifiers — this will reduce the code

```
…
const liNodesFound = chatMembers()
  .filter(memberElement =>
    doesMemberMatch(partOfMemberName, memberElement));

liNodesFound.forEach(highlight);
…
```

down to a single line (even though we're breaking it up into multiple lines here to make the code more readable):

```
…
chatMembers()
  .filter(memberElement =>
    doesMemberMatch(partOfMemberName, memberElement))
  .forEach(highlight);
…
```

This also makes the implementation of the function `highlightChatMembersBy` very easy to read:

1. Get the chat members.
2. Filter out those who match `partOfMemberName`.
3. Highlight everything that was found.

Anyone who needs to understand the details of this program can always take a look at the specific functions, but now just `highlightChatMembersBy` alone communicates the general steps of the program, which were previously hidden within its details.

Newspaper Metaphor

Fig. 11 *Photo: Olu Eletu, CC0 1.0*

Next, we can improve our code by putting our newly extracted functions in some kind of order. General high-level functions belong at the top of the file, while detailed low-level auxiliary functions should go below these.

Ordering functions in this way will allow you to read your code just like you're reading a newspaper — the large headlines are at the top, and if you need the details, you can just keep reading until you've had your fill. This is the so-called ***newspaper metaphor*** invented by Robert C. Martin (2008).

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.includes(partOfMemberName);

  const chatMembers = () => Array.from($$("#chat_members li"));
  const highlight = el => el.classList.add("highlighted");

  const $$ = document.querySelectorAll.bind(document);

  highlightChatMembersBy("ert");
}
```

Listing 13 *accompanying_files/04/examples/highlight_chat_members_ref5.js*

3.4 Array.from was So Yesterday — These Days, it's Array Methods for Everything!

Every time you need to call a higher-order function like `map` or `filter` on the result of the `$$` function, you first need to use `Array.from` to convert the result, e. g.

```
1 const chatMembers = () => Array.from($$("#chat_members li"));
```

Don't you find that a little irritating?

You can solve this by using the following line of code (taken from the legendary bling.js-Gist by Paul Irish). You don't need to understand the details, all that's important is that the code essentially ensures that a *NodeList* has all the methods of an `array`.

```
NodeList.prototype.__proto__ = Array.prototype;
```

This avoids the need of having to use `Array.from` to convert a *NodeList* into an `array`, making the function `chatMembers` much simpler.

```
1  "use strict";
2
3  {
4    const highlightChatMembersBy = partOfMemberName => {
5      chatMembers()
6        .filter(member =>
7          doesMemberMatch(partOfMemberName, member))
8        .forEach(highlight);
9    };
10
11   const doesMemberMatch = (partOfMemberName, memberElement) =>
12     memberElement.innerHTML.includes(partOfMemberName);
13
14   const chatMembers = () => $$("#chat_members li");
15   const highlight = el => el.classList.add("highlighted");
16
17   const $$ = document.querySelectorAll.bind(document);
18   NodeList.prototype.__proto__ = Array.prototype;
19
20   highlightChatMembersBy("ert");
```

```
21 }
```

Listing 14 *accompanying_files/04/examples/highlight_chat_members_ref6.js*

Admittedly, it's hardly worth it here, since `Array.from` only occurs once in our code. However, it's generally a good idea just to include the line to make your programming life a little easier. You won't have to stop every time and think about whether you might be using an `array` method on *NodeList* and need to convert it first.

You just need to keep an eye out when you're importing other third-party libraries which might also change *NodeList*. In that case, you'll just have to go back to using `Array.from`.

3.5 One More Tiny Improvement

If you enter `'Bert'` as your search string, your program will only select Berthold, but not Heribert. This is because the string comparison is ***case-sensitive***, in other words it takes into account both uppercase and lowercase letters. One easy way of having your searches ignore the difference between uppercase and lowercase lettering is to first convert both strings in your comparison completely to lowercase, using the string method `toLowerCase`.

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.toLowerCase()
      .includes(partOfMemberName.toLowerCase());

  const chatMembers = () => $$("#chat_members li");
  const highlight = el => el.classList.add("highlighted");

  const $$ = document.querySelectorAll.bind(document);
  NodeList.prototype.__proto__ = Array.prototype;

  highlightChatMembersBy("ert");
}
```

Listing 15 *accompanying_files/04/examples/highlight_chat_members_ref6_lowercase.js*

FYI, our code as it stands still shows a few areas which could use refactoring, for example refactoring using ***functional*** or ***object-oriented programming*** or even ***hexagonal architecture***. But that's a story we'll save for another time...

4 Events (Not the Kind You Celebrate)

If you can finish the new chat feature by tonight, we'll throw an office party to celebrate its release. Nothing fancy, just some drinks, good music and a little something to eat …

Fig. 12 *Photo: Yutacar, CC0 1.0*

The chat code is now stored in a separate JavaScript file and is also a little cleaner. But Marty's not completely happy, there's still something missing...

Of course, visitors to the website shouldn't have to program JS to highlight chat members — highlighting should change automatically based on the input in the search box.

In order for that to happen, the code needs to react to user input. Your browser provides so-called ***events*** for just that purpose (no, we don't mean events like weddings or parties — unfortunately).

4.1 Browser Events

Any exertion of influence by a user, no matter how small, is considered by your browser to be an event. An event can be almost anything, including:

- clicking on a button
- moving your mouse over an image
- releasing an input key
- leaving an input field using your mouse or by tabbing
- finger gestures on a touch screen, e. g. zoom

There're also events which aren't triggered directly by users, and are triggered instead through occurrences like network requests or files which have finished loading. But first, we'll concentrate only on those events directly related to user behavior.

4.2 Dealing With Events as They Come Up: Event Handlers

4.2.1 Preliminary Measures

First, try a small experiment in your console. The event you're interested in is a user releasing a key while typing in an input field. Look in the chat HTML document and you'll find the input field (`input` element) within a `div` element with the `id` *member_search*:

```
…
<div id="member_search"><input type="text" placeholder="...Find a member..."/></div>
…
```

First, let's select the field using `$('#member_search input')`. You'll get back the `input` element as a return value in the console. Move your cursor over the element in the console and you'll see that it's the right field.

Now, we'll bind an event to that field. To do this, we'll use the method `addEventListener`. That method is available on almost every HTML element object.

```
$('#member_search input').addEventListener( … );
```

`addEventListener` requires two arguments. The first argument is the ***event type***, in the form of a string. The event type we should use in this case is `'keyup'`, or releasing a key.

```
$('#member_search input').addEventListener('keyup', …);
```

`addEventListener`'s second argument must be a function — this will be executed by your browser when the specified event occurs. In this case, we'll just use a simple function to make sure the process is working:

```
() => alert(1)
```

FYI, the function you register on an event is called an ***event listener*** or ***event handler***, and it's responsible for processing the event you specify.

And now our statement is complete — let's test it out in the console:

```
$('#member_search input').addEventListener('keyup', () => alert(1));
```

If you go to the member search input box and type in a character, an alert box with the alert `1` will appear as soon as you release the key. Of course, this function isn't very useful yet. And maybe clicking away alert boxes is making you a little irritated — but at least now you know how event handlers work. Mission accomplished!

4.2.2 Putting Everything Together

Now comes the tough part — you need to combine your existing code for highlighting and for event registration into one complete program.

Delete the call `hightlightChatMembers('ert');` from your JS file. Instead, your `keyup` event should now trigger the function call:

```
$('#member_search input')
  .addEventListener('keyup', () =>
    hightlightChatMembers('ert'));
```

And actually, as soon as you type a key in the search box, the highlighting changes. However, instead of permanently highlighting the string `'ert'`, we want to use the actual input. But how can we do that?

Once again, we'll need to use the input element object. We can retrieve it by using the selector `$('#member_search input')`, then query it for the value input by the user. We'll find that value in the property `value`, i. e. `$('#member_search input').value`. You'll learn more about `value` and other properties in lesson 6.

Putting it all together, we get the following statement:

```
$('#member_search input')
  .addEventListener('keyup', () =>
    hightlightChatMembers($('#member_search input').value));
```

Which then gives us the following overall code:

```
"use strict";

{
  const highlightChatMembersBy = partOfMemberName => {
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.toLowerCase()
      .includes(partOfMemberName.toLowerCase());

  const chatMembers = () => $$("#chat_members li");
  const highlight = el => el.classList.add("highlighted");

  const $ = document.querySelector.bind(document);
  const $$ = document.querySelectorAll.bind(document);
  NodeList.prototype.__proto__ = Array.prototype;

  $("#member_search input")
    .addEventListener("keyup", () =>
      highlightChatMembersBy($("#member_search input").value));
}
```

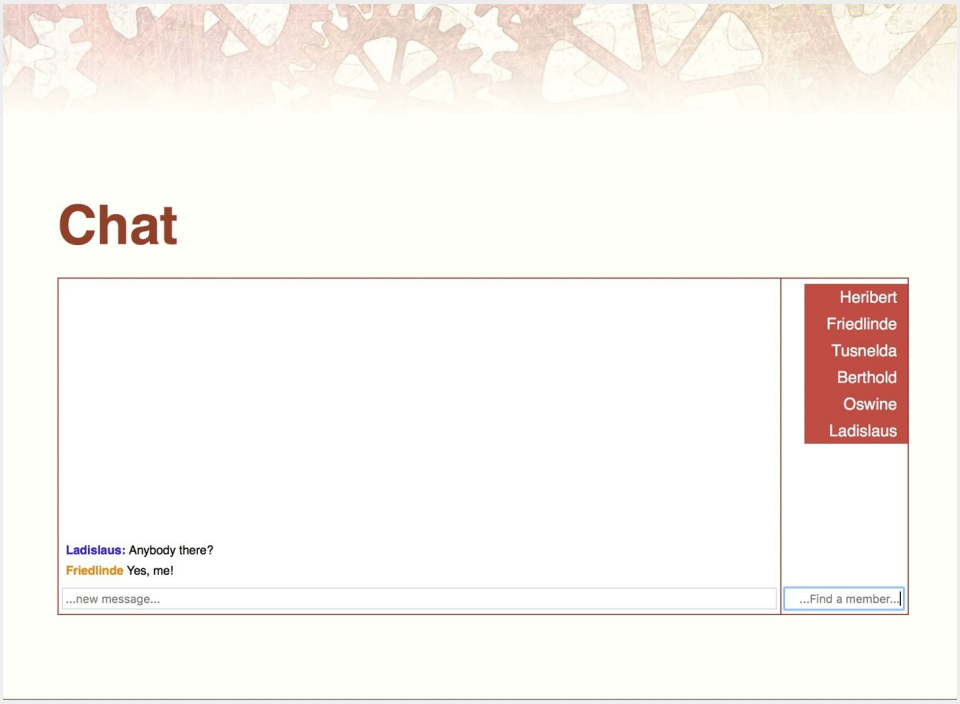
Listing 16
accompanying_files/05/examples/highlight_chat_members_1_event/highlight_chat_members.js

Enter different strings and experiment a little with your current implementation. What do you notice?

4.3 Guard Clauses: Protection for Your Functions

Oops — I just found a bug. If I search for a chat member, suddenly *all* of them are highlighted.

Could you fix that? As it is, we can't possibly put the highlight feature online.



Okay, here's the problem — although the program is highlighting the matching members, it's not removing that highlighting again when the search string is updated.

Instead of going through the trouble to find out what highlighting we need to remove, it'd be a lot simpler just to remove all highlighting, then

highlight only those chat members who match the search. To do this, we'll need some additional code which removes the corresponding CSS class from all members:

```
const removeHighlightsFromAllChatMembers = () =>
  chatMembers().forEach(removeHighlight);

const removeHighlight = el => el.classList.remove('highlighted');
```

FYI, it's not a problem that a few `li` elements with chat members don't have the class *highlighted* at all — in that case, nothing will be removed.

Order the two functions according to their level of detail. In addition, we still need to add a call to the new function `removeHighlightsFromAllChatMembers`. To do this, we recommend you create a higher-level function called `updateHighlightingOfChatMembers`, which first removes all highlights (`removeHighlightsFromAllChatMembers`) then marks the matching chat members (`hightlightChatMembersBy`):

```
const updateHighlightingOfChatMembers = partOfMemberName => {
  removeHighlightsFromAllChatMembers();
  hightlightChatMembersBy(partOfMemberName);
};
```

Now we just need to add the new function `updateHighlightingOfChatMembers` to the event handler in place of the function `hightlightChatMembersBy`, which was only responsible for highlighting. We then get the following code:

```
"use strict";

{
  const updateHighlightingOfChatMembers = partOfMemberName => {
    removeHighlightsFromAllChatMembers();
    highlightChatMembersBy(partOfMemberName);
  };

  const removeHighlightsFromAllChatMembers = () =>
    chatMembers().forEach(removeHighlight);

  const highlightChatMembersBy = partOfMemberName => {
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
```

```
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.toLowerCase()
      .includes(partOfMemberName.toLowerCase());

  const chatMembers = () => $$("#chat_members li");
  const highlight = el => el.classList.add("highlighted");
  const removeHighlight = el => el.classList.remove("highlighted");

  const $ = document.querySelector.bind(document);
  const $$ = document.querySelectorAll.bind(document);
  NodeList.prototype.__proto__ = Array.prototype;

  $("#member_search input")
    .addEventListener("keyup", () =>
      updateHighlightingOfChatMembers($("#member_search input").value));
}
```

Listing 17
accompanying_files/05/examples/highlight_chat_members_2_remove/highlight_chat_members.js

The code from listing 17 works … almost. It works except for the small glitch that all members are selected in case of blank input. But the empty string occurs in every string!

We can suppress this behavior just by adding code at the beginning of the function `hightlightChatMembersBy` to check whether `partOfMemberName` is empty. In case of an empty string, the function just needs to refuse to work. Add a `return` to make the function exit prematurely.

```
if (partOfMemberName === "") return;
```

Code like this, which runs before the main body of a function is actually executed to make sure the values of the arguments passed to the function make sense, are called ***guard clauses***or just ***guards***. They protect functions from invalid input values.

```
const hightlightChatMembersBy = partOfMemberName => {
  if (partOfMemberName === "") return;
  chatMembers()
    .filter(member => doesMemberMatch(partOfMemberName, member))
    .forEach(highlight);
};
```

Listing 18 *The function highlightChatMembersBy with a guard*

The code from listing 18 finally behaves like it should. But before you run

to Marty to bring him the good news, let's take a little time to add a couple of improvements. Your maintenance programmer will thank you for that someday!

4.4 Making a Stunning Entrance Using init

The event registration code is still just kind of out there, with no motivation — it would be better to put it into its own function. This will give us a number of advantages — the code will be reusable and it'll also have its own name, making it easier to identify. In addition, putting the code into a separate function will provide us advantages in terms of things we haven't yet covered in this class — e. g. the code will be easier to test.

A good function name might be *registerEvents*, or even just *init*. If we wanted to, we could get even more specific and name the function *registerEventsForChatMemberHighlighting*. However, given the context (i. e. our entire file is geared for a specific task) the generic *init* is perfectly acceptable.

The name *init* stands for initialization. It's a function which calls all other functions, acting as an entry point into the rest of the code in the program. We could essentially name the function anything we wanted to, but *init* has already been established as the name for such functions. Other programmers can read the function name and understand immediately what we mean by it.

Based on our newspaper metaphor, `init`'s function definition belongs at the beginning of your code. However, the call to `init()` can be made only after all other function have been defined.

```
"use strict";

{
  const init = () => $("#member_search input")
    .addEventListener("keyup", () =>
      updateHighlightingOfChatMembers($("#member_search input").value));

```

```
  const updateHighlightingOfChatMembers = partOfMemberName => {
    removeHighlightsFromAllChatMembers();
    hightlightChatMembersBy(partOfMemberName);
  };

  const removeHighlightsFromAllChatMembers = () =>
    chatMembers().forEach(removeHighlight);

  const hightlightChatMembersBy = partOfMemberName => {
    if (partOfMemberName === "") return;
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.toLowerCase()
      .includes(partOfMemberName.toLowerCase());

  const chatMembers = () => $$("#chat_members li");
  const highlight = el => el.classList.add("highlighted");
  const removeHighlight = el => el.classList.remove("highlighted");

  const $ = document.querySelector.bind(document);
  const $$ = document.querySelectorAll.bind(document);
  NodeList.prototype.__proto__ = Array.prototype;

  init();
}
```

Listing 19
accompanying_files/05/examples/highlight_chat_members_3_init/highlight_chat_members.js

4.5 Take a Look Behind the Facade With the Event Object

The fact that our current code can use some improvements will become clear once we take a closer look at event registration.

```
$('#member_search input')
  .addEventListener('keyup', () =>
    hightlightChatMembers($('#member_search input').value));
```

The function we'll register as our event handler here is:

```
() => hightlightChatMembers($('#member_search input').value);
```

The interesting thing to note here is that the function, when it's called by the event, is automatically passed the event as an argument. So we just need to implement a parameter to capture the event:

```
event => hightlightChatMembers($('#member_search input').value);
```

What can we do with the `event`? One thing would be to print it out to the console to see what it contains:

```
event => {
  console.log(event);
  hightlightChatMembers($('#member_search input').value);
}
```

If you type the letter **a**, the console will show:

```
keyup { target: <input>, key: "a", charCode: 0, keyCode: 65 }
```

If you click on *keyup*, you'll see that `event` is a JS object of "type"[1]`KeyboardEvent`. Keyboard events have a number of very interesting

properties — the table below shows you just a few:

---

1 In reality, JS doesn't have specific object types. It would be more correct to say that *keyup* is a JS object which has the `KeyboardEvent` object in its prototype chain. We're using the term *type* here for purposes of simplification. It's only to show that all objects of the same "type" also have the same properties.

Property	Content
type	keyup
code	KeyA
key	a
timeStamp	[timestamp, i. e. when the event occurred]
target	[element which triggered the event]
ctrlKey	false [was the Ctrl key pressed?]
altKey	false [was the Alt key pressed?]

Events have many other properties, but just the few we've listed above show you that an event object contains all the information which comes into play when an event occurs, e. g.:

- What type of event is it? *keyup*
- What key was released? *a*
- When was the event triggered?
- From where was the event triggered?

Your event handler can then put this information to use. `target` is the property which is of primary interest to us in our current program. `target` contains the input field we intend to use, which we'd otherwise select by using `$('#member_search input')`. Therefore, we could also register our event handler as follows:

```
event => hightlightChatMembers(event.target.value);
```

Doing so gives us a number of concrete advantages. In all cases, it's more efficient, since your browser doesn't have to find the element object again — it's already in `target`. But a much more important advantage is better maintainability. If our selector for finding the input field should change (e. g. because of a change to the HTML structure), we won't have to worry about also changing the event handler. It always refers to the current `target`, no matter how we found the element and registered the event on it.

```
"use strict";

{
  const init = () => $("#member_search input")
    .addEventListener("keyup", event =>
      updateHighlightingOfChatMembers(event.target.value));

  const updateHighlightingOfChatMembers = partOfMemberName => {
    removeHighlightsFromAllChatMembers();
    hightlightChatMembersBy(partOfMemberName);
  };

  const removeHighlightsFromAllChatMembers = () =>
    chatMembers().forEach(removeHighlight);

  const hightlightChatMembersBy = partOfMemberName => {
    if (partOfMemberName === "") return;
    chatMembers()
      .filter(member =>
        doesMemberMatch(partOfMemberName, member))
      .forEach(highlight);
  };

  const doesMemberMatch = (partOfMemberName, memberElement) =>
    memberElement.innerHTML.toLowerCase()
      .includes(partOfMemberName.toLowerCase());

  const chatMembers = () => $$("#chat_members li");
  const highlight = el => el.classList.add("highlighted");
  const removeHighlight = el => el.classList.remove("highlighted");

  const $ = document.querySelector.bind(document);
  const $$ = document.querySelectorAll.bind(document);
  NodeList.prototype.__proto__ = Array.prototype;

  init();
}
```

Listing 20
accompanying_files/05/examples/highlight_chat_members_4_target/highlight_chat_members.js

FYI, we can abbreviate `event` as `e`. Normally abbreviations are considered "bad" and should be avoided, but in this case it's okay. The single letter `e` is a common abbreviation for `event` and is acceptable as long as its scope (range over which a variable exists) is limited to a very small function.

Now we're ready! You can finally show Marty the improved feature, and he's so happy about it he throws another party — here's to hoping it'll be an unforgettable event!

Thanks! That's EXACTLY what I wanted! And hey, we'll see you tonight!

Exercise 9: Please Click Me

When you click on the button below, it opens an alert box with the text "*Hey I like it when you click me!*".

```
<button>Click Me</button>
```

Remove the alert box, and instead have the button's text change when you click on it.

Change the text to "*Cool, you found an Easter egg*" when the user holds down the **`Alt`** key while clicking on the button.

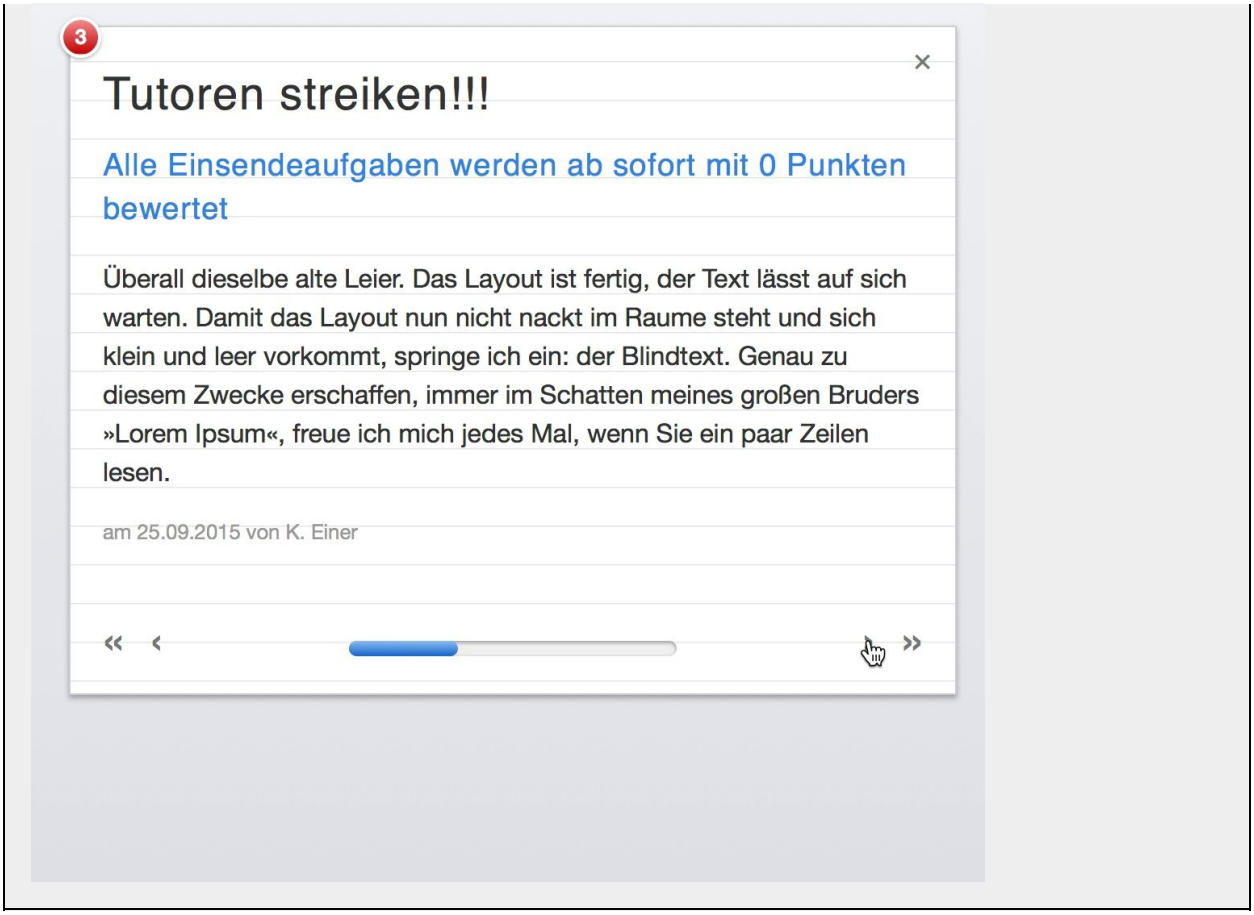
4.6 preventDefault or, Who Always Wants to be Just Standard?

Two days later, after Marty has recuperated from the party, he presents you with your next challenge.

By now, you've certainly seen the small news section on our home page, the one which displays the current news in a box with scroll bars. We need some changes to this.

From now on, the box should always display only one news item. Then there should also be previous and next buttons so a user can go back and forth through different news items.

Our designer has already completed the final layout, so we just need the functionality.



It's often not a good sign when a client says "*just*". But who cares, let's get started — after all, Marty does pay pretty well …

Sometime later, when everything's finished, the actual news will come from the server. But so that you'll have some data to test out your code, we've taken the liberty of providing you dummy text in the form of an array.

```
const messages = [
  `<h1>Tutoren streiken!!!</h1>
   <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
   <p>[&hellip;]</p>
   <p>am 25.09.2015 von K. Einer</p>`,

  `<h1>Wahnsinn!</h1>
   <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
   <p>[&hellip;]</p>
   <p>am 13.08.2015 von Dr. B. Lödmann</p>`,

  `<h1>Prokrastination?</h1>
   <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab's noch nicht erledigt; ich ha
   <p>[&hellip;]</p>
```

```
    <p>am 02.06.2015 von A. Meisenbär</p>`
];
```

Listing 21 *Excerpt from additional_files/05/examples/newsboard.js*

This is the original HTML:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />

  <title>Newsboard</title>

  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link rel="stylesheet" type="text/css" href="newsboard.css" media="screen" />
  <script src="newsboard.js" defer="defer"></script>
</head>

<body>

  <div class="newsboard_wrapper">
    <div class="newsboard">
      <span class="message_number">3</span>

      <a href="#" class="close_button" title="Delete message">&times;</a>

      <div class="newsboard_content"></div>

      <div class="paging_bar">
        <span class="float_left">
          <a href="#" title="first">«</a>
          <a href="http://example.com" title="prev">‹</a>
        </span>

        <span class="float_right">
          <a href="http://example.com" title="next">›</a>
          <a href="#" title="last">»</a>
        </span>

        <span class="progressbar">
          <progress max="3" value="1" id="messages_progress"></progress>
        </span>
      </div>
    </div>
  </div>

</body>

</html>
```

Listing 22 *accompanying_files/05/examples/newsboard.html*

The news should appear in the `div` element with the class `newsboard_content`. The first news item should already be displayed by the time the page finishes loading. The first part of the code isn't anything special:

```
"use strict";

{
  const init = () => {
    showFirstMessage();
  };

  const showFirstMessage = () => showMessageByNumber(1);

  const showMessageByNumber = messageNumber =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const messages = […];

  const $ = document.querySelector.bind(document);
  const $$ = document.querySelectorAll.bind(document);

  init();
}
```

Upon initialization, you call the function `showFirstMessage`, which displays the first news item by calling `showMessageByNumber(1)`. This latter function does the actual work — it extracts the news item from the array and uses `innerHTML` to move it to the correct location in the document.

We use a little trick in line 11. We're using the parameter `messageNumber` as an index for the `messages` array, but first we subtract 1 — we need to do this, since the array begins at index `0`. But we want to talk about the first news item, not the 0th.

Now it gets a little more interesting. Bind the `click` event to the previous and next characters `‹` and `›`. You can access the elements which contain these characters through the corresponding `title` attribute `prev` or `next`.

```
…
const init = () => {
  showFirstMessage();
  $('[title=next]').addEventListener('click', nextMessage);
  $('[title=prev]').addEventListener('click', prevMessage);
};

…

const nextMessage = e => console.log('next');
const prevMessage = e => console.log('prev');
…
```

Unfortunately, it's not working just yet. This is because the designer used

links (`a` elements) for the next and previous buttons. If they were `button` elements, there wouldn't be a problem, but links already have a default behavior when you click on them — they open the URL specified in the `href` attribute!

So what do we do? Of course, we could just change the element, but it's possible the designer already has something planned for them. For example, the links might be there so the newsboard will work even without JS, and just with conventional HTML.

There's a better solution — we can just prevent the links from triggering their default behavior.

The Event object provides you the method `preventDefault` for just that purpose.

```
const nextMessage = e => {
  console.log('next');
  e.preventDefault();
};

const prevMessage = e => {
  console.log('prev');
  e.preventDefault();
};
```

After we solve this problem, we'll be able to look into actually flipping through news items instead of just printing out messages to the console!

The fundamental problem here is that we need somewhere to remember the number of the news item which was just displayed. In other words, our newsboard application has a so-called ***state***. What actually happens when a user clicks on the next button depends upon this state. To determine the next news item, our program needs to know what news item was just displayed.

The process of maintaining state is always a little problematic, and makes it harder to make a program understandable. Although we

generally want to avoid a ***mutable state*** when programming, it's often a necessary evil — especially when it comes to graphic interfaces.

Right now, we don't have a good solution to get around the problem of maintaining state — we'll need to come back to this later. As a result, we'll first use a solution which is less than ideal, then come back to it later and apply a little refactoring.

Store the number of the current news item in a variable which will shared by several functions:

```
let currentMessageNumber = 1;
```

This way we have an easy way of changing the value of the current item as the user goes back and forth through news items.

```
{
  ...

  const nextMessage = e => {
    showMessageByNumber(currentMessageNumber += 1);
    e.preventDefault();
  };
  const prevMessage = e => {
    showMessageByNumber(currentMessageNumber -= 1);
    e.preventDefault();
  };

  ...
  let currentMessageNumber = 1;
  ...
}
```

Fortunately, the variable `currentMessageNumber` doesn't have ***global scope*** since it only exists in the outer block. This will be tolerable as long as that block doesn't become too large. But once our application grows, we'll need a better solution. We'll look at such a solution in lesson 6.

Hooray, it's working! Here's the code once more in its entirety:

```
"use strict";

{
```

```
  const init = () => {
    showFirstMessage();
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
  };

  const showFirstMessage = () => showMessageByNumber(1);

  const nextMessage = e => {
    showMessageByNumber(currentMessageNumber += 1);
    e.preventDefault();
  };
  const prevMessage = e => {
    showMessageByNumber(currentMessageNumber -= 1);
    e.preventDefault();
  };

  const showMessageByNumber = messageNumber =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const $ = document.querySelector.bind(document);

  const messages = [
    `<h1>Tutors are on a strike!!!</h1>
    <h2>All assignmnent are automatically graded with 0 points</h2>
    <p>Duis pretium ornare odio nec cursus. Nulla quis dolor vitae nulla condimentum ma
    <p class="newsboard_footer">9/25/2015 by N. O'body</p>`,

    `<h1>Madness!</h1>
    <h2>How to earn a fortune, with a complete stupid idea</h2>
    <p>Ut molestie elementum risus, eget rutrum dui tristique id. Duis ac elit a mi con
    <p class="newsboard_footer">08/13/20156 by Dr. Ken Hurt</p>`,

    `<h1>I did something, and you will never guess what happened next...</h1>
    <h2>Donec tristique, leo at suscipit pellentesque, mauris neque congue leo!</h2>
    <p>Aenean egestas mauris at neque egestas hendrerit id ut erat. Donec iaculis ornar
    <p>Integer non venenatis tellus. Phasellus tellus leo, suscipit ac vulputate non, v
    <p class="newsboard_footer">2015/06/02 by Chris P. Bacon</p>`
  ];

  let currentMessageNumber = 1;

  init();
}
```

Listing 23 *accompanying_files/05/examples/newsboard.js*

A couple of items do need some improvement — but you can take care of those in a minute, in the exercises.

4.7 Brief and To The Point — on

Just like `document.querySelectorAll`, `.addEventHandler` is often cumbersome to use. Many libraries (e. g. jQuery) have provided the method `on` to be used in its place. For example, `$("#my_button").on('click', …)` means something like "when clicking (*on click*) on the button with the id *my_button*". Since of course we always want to take advantage of simplified notation whenever possible, we'll re-use some code which originally comes from bling.js:

```
Node.prototype.on = function (name, fn) {
  this.addEventListener(name, fn);
  return this;
};

NodeList.prototype.on = NodeList.prototype.addEventListener = function (name, fn) {
  this.forEach(elem  => elem.on(name, fn));
  return this;
};
```

This code makes `on` available on both `Node` as well as `NodeList`, meaning it's available on everything you can select using $ and $$. Below is an example for using `on` on `NodeList`:

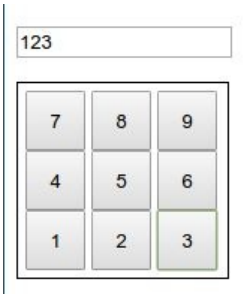


Fig. 13 *Screenshot of the Calculator*

We need to create a keypad for an *online pocket calculator*, with each number entered by the user shown on the display. Here's the HTML for the calculator:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <title>Calculator</title>

  <script src="calculator.js" defer="defer"></script>

  <style type="text/css">
    #display {
      width: 140px;
    }
    #keyfield {
      width: 130px;
      padding: 5px;
      border: 1px solid black;
    }
    #keyfield > button {
      width: 40px;
      height: 40px;
    }
  </style>
</head>

<body>
  <p><input type="text" id="display"/></p>
  <div id="keyfield">
    <button>7</button>
    <button>8</button>
    <button>9</button>
    <button>4</button>
    <button>5</button>
    <button>6</button>
    <button>1</button>
    <button>2</button>
    <button>3</button>
  </div>
</body>

</html>
```

Listing 24 *accompanying_files/05/examples/calculator/calculator.html*

We only need to add a single line of code in JS (in addition to our auxiliary functions) to implement the required functionality:

```
$$("#keyfield > button")
  .on("click", e => $("#display").value += e.target.innerHTML);
```

The function passed to the event

```
e => $("#display").value += e.target.innerHTML
```

writes the given number out to the `display`. Instead of registering this

function separately on every button, we only need one `on` to provide all buttons with the event handler.

We can even chain together multiple `.on` calls if we want to:

```
$$("#keyfield > button")
  .on("click", e => $("#display").value += e.target.innerHTML)
  .on("mousenter", e => console.log(e.target.innerHTML));
```

4.8 Summary

Event name	is triggered when…	e. g. available on
blur	An element loses focus.	input, textarea
change	A user makes a change to an element, e. g. by selecting an option within a `select` box or by clicking a checkbox. Unlike an `input` event, this event (in the case of an `input` element or a `textarea`) only "fires" after it again loses focus (i. e. the input is completed).	input, select, textarea
click	A user clicks on an element (and releases the button). The `click` event represents the complete process, i. e. pressing and releasing. In cases where you want to react only to a user pressing the mouse key down, use `mousedown`; if you want to react just to a user releasing the mouse key, use `mouseup`.	button, img, body, p, div
dblclick	A user double-clicks on an element.	button, img, body
focus	An element gets focus.	input, textarea
hashchange	The fragment identifier of a URL changes (the fragment identifier is the part of the URL which follows the # symbol and includes the symbol itself).	window

input	The content of an `input` or `textarea` element changes. Unlike `change`, this event fires immediately each time a user inputs text. You'll find a good comparison of `input` and `change` at http://jsfiddle.net/AtvtZ/.	input, textarea
keydown	The user presses a key.	input, textarea
keypress	The user holds a key down.	input, textarea
keyup	The user releases a key.	input, textarea
mouseenter	The user moves the mouse cursor over an element (more precisely, the mouse cursor enters the area of the element).	button, img, body, p, div
mouseleave	The user moves the mouse cursor out of an element.	button, img, body, p, div
mousemove	The mouse cursor moves over an element.	button, img, body, p, div
reset	A form is reset.	form
resize	The user changes the size of the browser window.	window
scroll	The user scrolls within an element (or the browser window).	window, div
select	The user selects text.	input, textarea, p
submit	A form is submitted.	form

transitionend	A CSS transition completes.	img, div, p
visibilitychange	The content of an element becomes visible or becomes hidden.	img, div, p
wheel	The user turns a wheel. Normally this is the mouse wheel, but can also be a trackball or touchpad.	window, div

Table 4.2 *A selection of browser events required in practice. You'll find a complete list of these events in the MDN Event reference.*

Attribute	Meaning
altKey	Returns `true` if the **`Alt`** key is being held down.
button	Number of the button pressed (0 to 4)
clientX	X-position of the mouse cursor in the DOM
clientY	Y-position of the mouse cursor in the DOM
ctrlKey	Returns `true` if the **`Ctrl`** key is being held down.
metaKey	Returns `true` if the **`Meta`** key (Windows key or Macintosh command key) is being held down.
movementX	Relative movement of the mouse cursor along the X-coordinate from the last `mousemove`
movementY	Relative movement of the mouse cursor along the Y-coordinate from the last `mousemove`
screenX	X-position of the mouse cursor on the screen
screenY	Y-position of the mouse cursor on the screen
shiftKey	Returns `true` if the **`Shift`** key is being held down.

Table 4.3 MouseEvent*: essential attributes*

Attribute	Meaning
altKey	Returns `true` if the **`Alt`** key is being held down.

code	Code value of the key that was pressed, e. g. `KeyA` (regardless of keyboard layout)
ctrlKey	Returns `true` if the **`Ctrl`** key is being held down.
key	The key that was pressed, e. g. `a` or `ArrowDown` (regardless of keyboard layout)
metaKey	Returns `true` if the **`Meta`** key (Windows key or Macintosh command key) is being held down.
shiftKey	Returns `true` if the **`Shift`** key is being held down.

***Table 4.4 KeyboardEvent**: essential attributes*

Attribute	Meaning
deltaX	Scroll movement along X-axis
deltaY	Scroll movement along Y-axis

***Table 4.5 WheelEvent**: essential attributes*

Attribute	Meaning
currentTarget	Element on which the current event handler is registered (often identical to `target`)
target	Element which triggered the event (often identical to `currentTarget`)
timestamp	Time at which the event was triggered

***Table 4.6 Event**: essential attributes (are inherited by all special event types, e. g. `MouseEvent`)*

4.9 Exercises

Exercise 10: Newsboard Still Needs Work

The newsboard is still waiting for its functions to be fully completed.

Add functionality to the buttons » and «. The button » should allow a user to jump to the first news item, while the button « should allow a user to jump to the last item.

The red circle in the upper left corner of the newsboard, which up to this point seemed a little meaningless, should now display the total number of news items. Write a function which retrieves this number and enters it in the `<span>` element with the class `message_number`.

Make it possible for users to be able from now on to navigate through the news using the keyboard. The left and right arrow keys should provide the same functionality as the buttons < and >.

It should also be possible for a user to jump to the first and last news items using just the keyboard. The key combination for this should be **`Alt`** and the left arrow key (for the first news item) or the right arrow key (for the last news item).

5 Using Multiple JS Files

Hi! I just got out of a meeting with Carl Biersee (Carl's our consultant). We're paying him a lot of money for him to keep telling us how great our new platform is and that we're moving in the right direction.

He took a look at our current codebase today. I think Carl's a nice guy and he didn't want to say anything bad, but he did make it pretty clear to us that our code has a lot of room for improvement.

In any case, there is one thing you should improve — divide the code up over different files. Carl was saying something about DRY and wheels …or something like that?

Although opinion is definitely divided on whether Carl is really worth the money they're paying him, he is right about one thing — once your code gets to be a certain size, it's a good idea to split it over different files. This gives you a number of advantages:

- better overview
- simpler reuse
- easier teamwork

Better Overview

The larger a file is, the more difficult it is to find the code you're looking for. Splitting your code up into different files based on theme and giving each file a meaningful name will help you tremendously in maintaining

and enhancing your code.

Just pretend you're a repair technician and you're looking for a specific tool. In one possible scenario, you have a desk whose drawers are all labeled. In another scenario, you have a big box into which the tool was tossed by the last person who used it. Where would you rather look for your tool?

Simpler Reuse

If you have functions which you'll be using in many different places in your application, naturally it'd be a little clumsy to keep reinventing the wheel. *Copying and pasting* functions isn't a very good solution either. Of course, we've all resorted to doing that in the past. But now, let's improve the way we work.

At the very least, the problem with copying and pasting will appear when you're revising your code or weeding out bugs. When that happens, you'll also need to change all copies of the code fragment you're working on. If you don't, the bug will end up haunting you or lead to inconsistencies and unexpected behavior.

A much better approach is to save your common code out to one file, then include that file wherever you need to. This will let you avoid inconsistencies and the need to chase down clones.

> **The DRY Principle**
>
> We don't mean DRY literally — in this case, it stands for ***D****on't* ***R****epeat* ***Y****ourself*! It means that developers should always try to avoid redundancies in their code.
>
> This concept involves more than just simply avoiding duplicate code.

Every piece of knowledge and every concept you use should have exactly **one** authoritative source and should be represented in your code just **once** (Venners 2003). Then if you do change a concept, ideally you'll have only **one** portion of code you'll need to change. Sometimes it's debatable whether code is redundant or just happens to be the same by chance.

Functions created through copy & paste usually violate the DRY principle. **Avoid copy & paste!**

Easier Teamwork

If you also have a few other people working with you on the same project, it can be very helpful to use a large number of small files instead of just a few big ones. The more effectively a project is divided up, the less you'll get in each other's way Good versioning systems (e. g. Git) are a great help in managing changes made by different developers working on the same file. But it's always easiest when that isn't necessary at all.

FYI, the recommended file size is about **50 to 60 lines of code**. This is roughly the amount of code you see on a screen page without scrolling. Of course this is just a rule of thumb, and you'll always find (well-justified) exceptions to this.

5.1 Happy File Hacking

So what's the best way to do this? It's not a difficult process overall, but you do need to keep your eye out for a couple of stumbling blocks. The order in which JS files are included in a HTML document is often relevant.

The `defer` attribute causes scripts to be loaded asynchronously, so scripts arrive in (more or less) random order. Fortunately, the `defer` attribute also guarantees that after scripts are loaded, they're executed in the order specified in the HTML document. This is different from the `async` attribute, which causes scripts to be executed immediately after they're loaded, completely regardless of the order of `script` tags in the HTML.

```
<script src="code1.js" defer="defer"></script>
<script src="code2.js" defer="defer"></script>
<script src="code3.js" defer="defer"></script>
```

Still, for reasons of maintenance, it's a little inelegant when a change in that order results in the program no longer working. With just a little discipline, you can avoid this.

Just follow the rules below:

- Only make function calls in the last file you include. Ideally, your code should only have one function call, e. g. `init()` or `run()`.
- All other files should just define variables (`let`) or, even better, constants (`const`).

Stick to these two rules and you'll already be on the safe side. All your functionality will be loaded by the time you need to make a function call. In addition, coding guidelines have been established to help you organize

your code sensibly.

Coding Guideline

- Order your JS files from *generic* to *specific*.

The last file you include should contain your most specialized code, i. e. the code which concerns your current application. Files included before that point can contain more general functions which you use in other applications.

- Always load external libraries and third-party code before you load your own code.

If you do include multiple, independent collections of JS functionality in the same HTML page, you should treat each of these as a separate group of scripts. The rules above then apply to each group.

Example

```
<script src="gruppe1_allgemeine_bibliothek1.js" defer="defer"></script>
<script src="gruppe1_allgemeine_bibliothek2.js" defer="defer"></script>
<script src="gruppe1_eigene_funktionen.js" defer="defer"></script>
<script src="gruppe1_eigene_datei_mit_init_aufruf.js" defer="defer"></script>

<script src="gruppe2_allgemeine_bibliothek1.js" defer="defer"></script>
<script src="gruppe2_allgemeine_bibliothek2.js" defer="defer"></script>
<script src="gruppe2_eigene_funktionen.js" defer="defer"></script>
<script src="gruppe2_eigene_datei_mit_init_aufruf.js" defer="defer"></script>
```

To shut Carl up (he gets paid to find "problems", you know), you divide your newsboard code appropriately into three files.

```
"use strict";

const $ = document.querySelector.bind(document);
```

Listing 25 *accompanying_files/06/examples/newsboard/dom_helper.js*

Right now, the file *dom_helper.js* just contains the $ function; you can

add other general functions to that file as necessary.

```
"use strict";

const messages = [
  `<h1>Tutors are on a strike!!!</h1>
  <h2>All Assignmnent are automatically graded with 0 points</h2>
  <p>Duis pretium ornare odio nec cursus. Nulla quis dolor vitae nulla condimentum maxi
  <p class="newsboard_footer">9/25/2015 by N. O'body</p>`,

  `<h1>Madness!</h1>
  <h2>How to earn a fortune, with complete stupid idea</h2>
  <p>Ut molestie elementum risus, eget rutrum dui tristique id. Duis ac elit a mi conva
  <p class="newsboard_footer">08/13/20156 by Dr. Ken Hurt</p>`,

  `<h1>I did something, and you will never guess what happened next...</h1>
  <h2>Donec tristique, leo at suscipit pellentesque, mauris neque congue leo!</h2>
  <p>Aenean egestas mauris at neque egestas hendrerit id ut erat. Donec iaculis ornare
  <p>Integer non venenatis tellus. Phasellus tellus leo, suscipit ac vulputate non, var
  <p class="newsboard_footer">2015/06/02 by Chris P. Bacon</p>`
];
```

Listing 26 *accompanying_files/06/examples/newsboard/messages.js*

The file *message.js* contains the news items themselves. Anyone who needs to add additional news items will just need to change this one file. This reduces the likelihood that errors will sneak into your actual application code at a later time.

```
"use strict";

{
  const init = () => {
    showFirstMessage();
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
  };

  const showFirstMessage = () => showMessageByNumber(1);

  const nextMessage = e => {
    showMessageByNumber(currentMessageNumber += 1);
    e.preventDefault();
  };
  const prevMessage = e => {
    showMessageByNumber(currentMessageNumber -= 1);
    e.preventDefault();
  };

  const showMessageByNumber = messageNumber =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  let currentMessageNumber = 1;

  init();
}
```

Listing 27 *accompanying_files/06/examples/newsboard/newsboard.js*

The following code includes the three files in the HTML:

```
<head>
  ...
  <script src="dom_helper.js" defer="defer"></script>
  <script src="messages.js" defer="defer"></script>
  <script src="newsboard.js" defer="defer"></script>
  ...
</head>
...
```

Listing 28 *newsboard.html*

5.2 The Terror of Global Pollution

Unfortunately, we still need to deal with another obstacle. Variables and constants which exist in one file and need to be accessed by code in other files can't be placed in a block, and using an external block specifically ensures that the names of variables (and constants) don't exist outside of the block and go beyond the scope of the file.

However, not containing variables and constants in a block inevitably leads to problems which a block should guard against: *global namespace pollution*.

When you include a number of files in your program, it can sometimes happen that two or more use the same variable or constant names. This can result in name conflicts and in turn errors, and it may quickly turn out that you lose control, especially when you include a bunch of third-party scripts (e.g. Youtube, Google-Analytics) or external libraries. You'll then be faced with a time-consuming error search (aka ***bug hunt***).

As first, this might sound like a rather unlikely situation, but we can assure you that it's a very real problem, and happens a lot more often than you might imagine.

Therefore, try to keep the number of global variables and constants you use to a minimum.

A Small Experiment

Find any experienced JS developer in your neighborhood (if you're unsuccessful, expand your search radius ;) Then ask the developer to tell you his or her favorite story concerning *name conflicts* — no doubt

your developer friend will have some hair-raising stories to tell.

Fight Against Global Pollution

Solutions to the dilemma of *global namespace pollution* actually do exist. However, the related principles are a little beyond the scope of this class. If you're interested, here're a couple of references you can check out for more information:

- Module pattern & revealing module pattern (Osmani)
- ES6 imports with webpack and Babel (Rauschmayer 2016)
- ES6 imports with ES6 module loader

Exercise 11:

Break up the code from listing 20 following the principles given in this lesson.

5.3 Not Just Another Dusty Old Library

The term ***library*** (***lib*** for short) when used in the context of programming languages refers to a collection of code (e.g. provided as a set of functions) which you can then include in your own programs. Library code can be specialized in a specific area or be kept general. Normally, however, a library is more generic and can be used in many different programs.

You've essentially already put together a library in this lesson — maybe not one which shelves weighty tomes and old volumes, but one which exists in the form of a file. That's right — the file *dom_helper.js* is already a small library.

```
"use strict";

const $ = document.querySelector.bind(document);
```

Listing 29 *accompanying_files/06/examples/newsboard/dom_helper.js*

Admittedly, it does contain just one function right now (`$`), and you'd hardly call a library with only one book a library. And in practice, it would also be a little excessive to include a library for just one function. But we actually do already have a few functions and extensions that we need on a regular basis.

Save those functions in the file *dom_helper.js*, and in the future you won't have to constantly copy & paste them.

```
"use strict";

const $ = document.querySelector.bind(document);
const $$ = document.querySelectorAll.bind(document);

NodeList.prototype.__proto__ = Array.prototype;
HTMLCollection.prototype.__proto__ = Array.prototype;

Node.prototype.on = function(name, fn) {
```

```
  this.addEventListener(name, fn);
  return this;
};

NodeList.prototype.on = NodeList.prototype.addEventListener = function(name, fn) {
  this.forEach(elem => elem.on(name, fn));
  return this;
};
```

Listing 30 *accompanying_files/lib/dom_helper.js*

You'll find we've already created this file for you, in the `/lib` directory in the accompanying material.

Another plus is that if you find an error, you'll only need to fix it once and not in all the copies you created through copy and paste.

Sharing is Caring: Including External Libraries

The good thing is that many JS developers have already developed libraries which focus on a variety of topics and have published them under an open source license. You can include these libraries in your own code whenever you need to, and that way you won't have to write everything yourself.

Package Managers

The easiest way to include and reuse third-party libraries is to use a ***package manager***. Although the use of such managers is beyond the scope of this class, we can provide you with the URLs of a couple of well-known package managers if you'd like to learn more:

- https://www.npmjs.com
- https://yarnpkg.com
- http://jspm.io

6 Attributes: Retrieving, Changing and Manipulating Expressly Permitted

We like another improvement added to the news section (it's really just a small thing).

To give users a better overview of the news items available, we're picturing having a progress bar show how many items a user has already read and how many there are still left. You can definitely get that done by today...right?

Well, we'll see... But before you throw yourself wholeheartedly into Marty's newest task, we'd first like to show you how you can control a progress bar using JS. You can add a progress bar to your HTML using

the HTML5 element `progress`:

```
<progress value="3" max="10"></progress>
```

Listing 31 *HTML progress element*

The attribute `max` specifies the maximum length of the bar, i. e. its length when it reaches the end. `value` is the current value of the bar. So if you want to just display *news item 3 of 10*, you would set `max` to `10` and `value` to `3`, as in listing 31.

Of course, now you also need the ability to change the value of the `value` attribute through JavaScript.

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <meta charset="utf-8" />
6    <title>Progressbar</title>
7  </head>
8
9  <body>
10   <progress id="messages_progress" value="3" max="10"></progress>
11 </body>
12
13 </html>
```

Listing 32 *accompanying_files/07/examples/progressbar.html*

Open the file for listing 32 in your browser and access the progress bar element object using its id `messages_progress`:

```
$('#messages_progress')
```

All HTML element objects provide the methods `getAttribute` and `setAttribute` to retrieve and change attribute values.

The command

```
$('#messages_progress').getAttribute('value')
```

returns the current value of the `value` attribute (`3`), and the command

```
$('#messages_progress').setAttribute('value', 8)
```

allows you to change that value to `8`.

Element objects also have a property called `attributes` which returns a list of all available attributes:

```
$('#messages_progress').attributes // => NamedNodeMap [ max="20", value="7", id="messages
```

This return value is a so-called `NamedNodeMap` — an object which contains all element object attributes and their associated values.

6.1 IDL Attribute vs. Content Attribute

In general, you can use the two methods `getAttribute` and `setAttribute` to retrieve and change virtually any attribute of an HTML element. However, in many cases the DOM also makes those attributes available directly, as JS properties of the HTML element object. For example, we could also retrieve the value of the progress bar directly by using the command

```
$('#messages_progress').value
```

or change its value to `8` by typing

```
$('#messages_progress').value = 8
```

Attributes which you retrieve through access methods are called ***content attributes***, while those you retrieve directly via properties are called ***IDL attributes***. In general, it makes no difference what attribute type you use — IDL attributes themselves access the underlying content attributes.

However, there is a small difference in the data type used by each attribute type. Content attributes are always of data type `string`.

```
typeof $('#messages_progress').getAttribute('value') // => string
```

On the other hand, IDL attributes can have different data types. For example, the `value` of the `progress` element is of type `number`.

```
typeof $('#messages_progress').value // => number
```

6.2 Long Live Progress (Bars)!

Now that you know how everything works, we just need to connect up the progress bar. The nice thing here is that we can now get rid of that unlovable variable `currentMessageNumber` and store that value directly in the `value` attribute of `progress` instead of in a separate variable.

In order to make that value easier to access, let's first define a function `progressbar` which returns the `progressbar` element.

```
const progressbar = () => $("#messages_progress");
```

We can now change the current value of the progress bar (`progressbar().value`) just by using `+=1` and `-=1`.

```
const nextMessage = e => {
  showMessageByNumber(progressbar().value += 1);
  e.preventDefault();
};
const prevMessage = e => {
  showMessageByNumber(progressbar().value -= 1);
  e.preventDefault();
};
```

A Little Refactoring

To improve readability, let's move the code which changes the `value` of the progress bar out into a separate function:

```
const nextMessage = e => {
  showMessageByNumber(incCurrentMessageNumber());
  e.preventDefault();
};
const prevMessage = e => {
  showMessageByNumber(decCurrentMessageNumber());
  e.preventDefault();
};

const incCurrentMessageNumber = () => progressbar().value += 1;
const decCurrentMessageNumber = () => progressbar().value -= 1;
```

Now try it out. Hooray! Our progress bar's a success!

Initialization in Progress

So that we'll be able to respond flexibly to `messages` arrays of any size, we should now initialize the progress bar with the proper values.

```
const initProgressbar = () => {
  progressbar().max = messages.length;
  progressbar().value = 1;
};
```

We can then call `initProgressbar` in our main `init` function. Add a couple of new news items to the array to try out the code — you'll see that the progress bar adjusts immediately and dynamically to arrays of any size.

6.3 Attention! Boolean Attributes at Work

You need to be a little careful when you're working with boolean attributes. For example, let's take the property `disabled`. This property is available on `button`, `input`, `option`, `select` and `textarea` elements, among others.

```
<button disabled="disabled">Not today, sorry</button> <button>Click Me - I'm enabled</bu
```

We can deactivate the button above by adding `disabled="disabled"`. It's now grayed out, and a user can't click on it.

Fig. 14 *Inactive button and active button*

The IDL attribute is of type `boolean`:

```
typeof $('button').disabled // => boolean
```

We can switch between button states by using the statements

```
$('button').disabled = true
```

and

```
$('button').disabled = false
```

Use listing 33 and the console to experiment.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
```

```
  <title>Buttons</title>
</head>

<body>
  <button disabled="disabled">Not today, sorry</button>
  <button>Click Me - I'm enabled</button>
</body>

</html>
```

Listing 33 *accompanying_files/07/examples/button.html*

On the other hand, if we're working with content attributes, the correct notation for graying out buttons is `$('button').setAttribute('disabled', 'disabled')`. And it becomes even crazier when we need to reactivate a button. Naively entering `$('button').setAttribute('disabled', '')` won't work.

Actually, that statement would again set the value `disabled` to `true`, since the only thing that really matters in boolean attributes is whether or not the attribute exists — the actual string value doesn't matter. So if we want to get rid of the content attribute again, we'll need to use `removeAttribute`:

```
$('button').removeAttribute('disabled');
```

6.4 It's All in How You Ask the Question — The value Attribute

Take a look at listing 34. The `value` attribute of the input element has been assigned the default text *hello*.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Input</title>
</head>

<body>
  <input type="text" value="hello" />
</body>

</html>
```

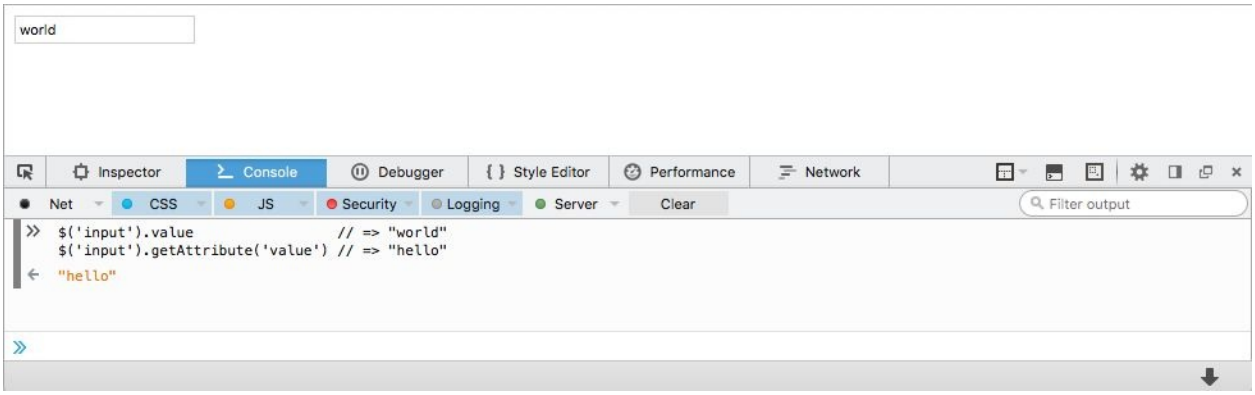
Listing 34 *accompanying_files/07/examples/input.html*

As long as this text doesn't change, the IDL and content attributes will be identical.

```
$('input').value                  // => "hello"
$('input').getAttribute('value') // => "hello"
```

Now enter some new text in the field (e. g. *world*) and again query the value using JS:

```
$('input').value                  // => "world"
$('input').getAttribute('value') // => "hello"
```



The IDL attribute reflects the change and thus returns the current input (in this case `"world"`). On the other hand, the content attribute still refers to the value in the HTML source code (in this case `"hello"`). In this case, this value can also be called a *default value*, since it was previously prescribed for the user in the input element.

My Recommendation

Always use IDL attributes — except when you expressly need an original value from the HTML code!

You should give preference to using IDL attributes for the following reasons:

- You get back a more precise data type, not just a string.
- You get back current values which may have been modified (e. g. through user input).
- They behave more consistently; you have fewer special cases you need to take into consideration.

6.5 Summary

Attribute name	Elements, for which the attribute is often used	Description
alt	`area, img, input`	Alternative text, e. g. if an image can't be displayed.
checked	`input`	Indicates whether a radio button has been selected or a checkbox was ticked.
cols	textarea	Number of columns
disabled	`button, input, option, select, textarea`	Indicates whether an element is deactivated.
href	`a, link`	URL of a linked resource
max	`progress, input`	Maximum value — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
maxlength	`input, textarea`	Maximum number of characters allowed in an element
min	`input`	Minimum value — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
multiple	`input, select`	When `true`, allows multiple input values for `input type="email"` or `input type="file"`. In case of `select`, multiple options can be selected using **`Ctrl/cmd`**.
name	`button,`	Name of the element. This attribute is

	`form`, `fieldset`, `input`, `select`, `textarea`	mainly used by server-side applications for purposes of identification.
placeholder	`input`, `textarea`	Preset text which indicates to the user what is expected in terms of input content.
readonly	`input`, `textarea`	Indicates whether an element can be modified.
rel	`a`, `link`	Defines the relationship of the target object to the link object. Can also be used for internal relationship types, e. g. internal links for a JS image gallery.
required	`input`, `select`, `textarea`	Indicates whether an element is a mandatory field.
rows	`textarea`	Number of rows
selected	`option`	Indicates whether an `option` has been selected.
size	`select`	Number of options visible in a `select` box
src	`img`	URL of the image file
start	`ol`	Defines the first number of a list if the list should start with a value other than 1.
step	`input`	Increment — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
tabindex	Available on all elements.	Overwrites the conventional order of (form) elements when controlled using the Tab key.
title	Available on all elements.	Small text box (tooltip) displayed when hovering the mouse over the element.
type	`button`, `input`	Type of the element

value	`button`, `option`, `input`, `progress`	Value of the element

Table 6.1 *A selection of attributes which are usually retrieved with JS in practice. You'll find a complete list of all attributes in the MDN attribute reference.*

6.6 Exercises

Exercise 12: It's a Buyout!

Users are often impatient and have a tendency to click on something a few times if it's not immediately obvious that something's happening. It's been proven to be a good idea to disable buttons as soon as they're clicked so that a user doesn't accidentally submit multiple orders for that new home theater system or sailboat .

Write an appropriate function to do just that for the button which appears in the accompany material (*buy_button.html*).

Exercise 13: Newsboard — One Final Feature, Promise!

The new newsboard features are almost finished. But there's still a small problem — the next and previous buttons are still active even if there's no next or previous news item. This results in a very ugly error.

Make the links (a elements) which are used to control the newsboard actual buttons (button elements) instead. This way, they can be disabled when necessary.

Change your code so that the next and previous buttons are disabled as soon as the user gets to the end or beginning of the news list respectively.

The buttons which bring the user to the first and last news items should likewise be disabled as soon as the user gets to the first or last news item respectively.

Exercise 14: Return of the Running Lights: Part 1 - Chasing Lights

Fig. 15 *Photo: Axel Kuhlmann*
License: CC-BY-2.0

Chains of lights were once all the rage (and often still are at Christmas time). Many of these even feature running lights.

You can program just such a chain of lights. You'll find the following basic HTML template in the exercise material:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
    <title>Beispiel</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <!-- <script src="running_light.js" defer="defer"></script> -->
    <script src="../../../lib/dom_helper.js"></script>
</head>

<body>

<img src="light_on.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />
<img src="light_off.png" alt="lightbulb" />

</body>
</html>
```

accompanying_files/07/exercises/running_light/running_light.html

Your first version will be *chasing lights*. Here, the user "chases after" the running light using the mouse. Any time the mouse cursor is over a light bulb, switch on the next bulb and turn the current one off. If the user tries to touch the lit bulb with the mouse cursor, the light should skip to the next. This makes it look as if the user is "chasing" the running light.

Then after the user reaches the last light, the game should begin again from the start.

Exercise 15: Return of the Running Lights: Part 2 - Triggered Classic

Program another running light game. This time, the running light should jump forward whenever the user clicks on any light.

The same thing applies here — i. e. after the last bulb is reached, the light should again begin to run from the start.

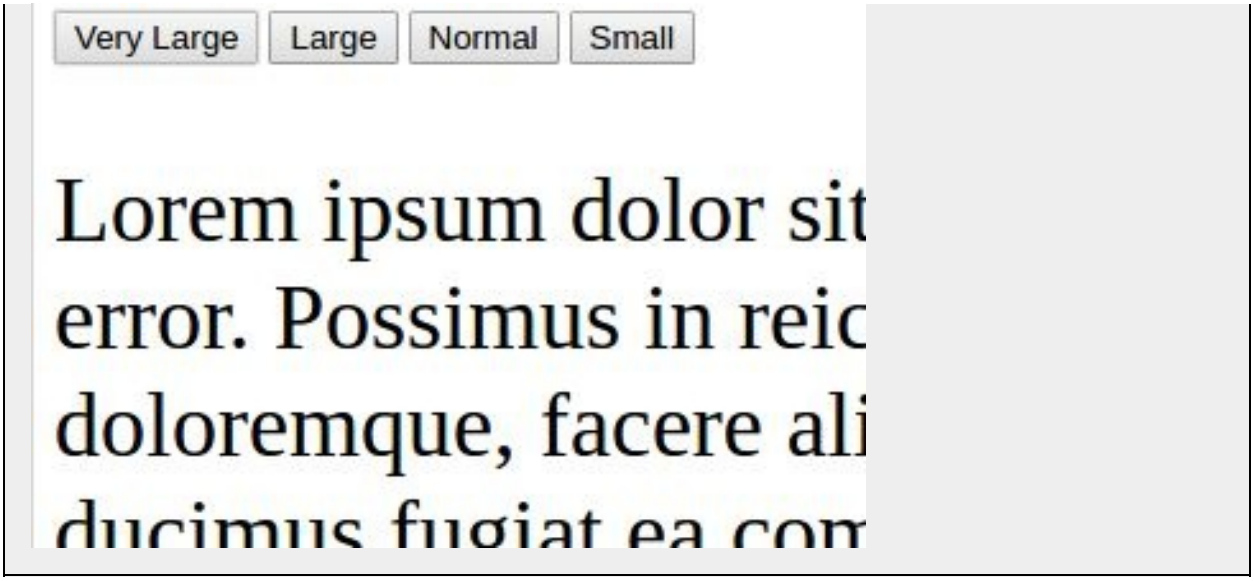
7 A Question of Style: The Style Object

Hey, we just launched a hip new business concept — nerd products for seniors! I'm sure they'll be a hit!

The only thing is, my grandmother and a few other older folks I talked to have complained that the font size on our page is too small, so of course we're going to have to increase it overall. But unfortunately, the needs of our seniors and other users are pretty varied. So I think it would be good if you could put a few buttons up on the page so that our users could select the font size themselves:

- small (14px)
- normal (16px)
- large (24px)
- very large (36px)
- blind as a bat (72px)
- ultra enormously extraordinarily extremely large (200px)

Well, maybe you can ignore those last two for now…



We could also create additional CSS classes for the different font sizes, but in this case that might be excessive. A much easier solution would be just to manipulate the font size directly.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Font Size</title>
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="font_size_refactored.js" defer="defer"></script>
</head>

<body>

  <button id="very_big">Very Large</button>
  <button id="big">Large</button>
  <button id="normal">Normal</button>
  <button id="small">Small</button>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Nam, error. Possimus in

</body>

</html>
```

Listing 36 *accompanying_files/08/examples/font_size_big_small/font_size.html*

The DOM provides the property `style` for HTML element objects for just that purpose. Open listing 36 in your browser. Enter `$('p').style` in the console — you'll get back a `CSSStyleDeclaration` object. This object makes it possible for you to access all available CSS properties as JS

properties, such as the font color `color`. Enter:

```
$('p').style.color = 'red'
```

The text of the first paragraph (`p` tag) is now red. Similarly, you can also change the font size of the paragraph. You just need to be careful of a little nitpicking when specifying properties, namely...

Presto Chango — You're Now a Camel!

In CSS, many properties have a hyphen in their names, such as `font-size`. However, JS identifiers may not contain hyphens. We therefore need to convert such property names into *camelCase* notation instead — for example, `font-size` will become `fontSize`. Now, to increase font size, we could enter:

```
$('p').style.fontSize = '36px'
```

Don't forget to include the unit of measurement (in this case `px`), otherwise your browser will ignore your specification!

Now all we need to do is bind our statements to the `click` event for each button. For example, the code for the big button would be:

```
$("#big").on("click", () => $("p").style.fontSize = "36px");
```

listing 37 shows the complete code for all buttons.

```
"use strict";

{
  const init = () => {
    $("#very_big").on("click", () => $("p").style.fontSize = "36px");
    $("#big").on("click", () => $("p").style.fontSize = "24px");
    $("#normal").on("click", () => $("p").style.fontSize = "16px");
    $("#small").on("click", () => $("p").style.fontSize = "14px");
  };
```

```
10
11   init();
12 }
```

Listing 37 *accompanying_files/08/examples/font_size_big_small/font_size.js*

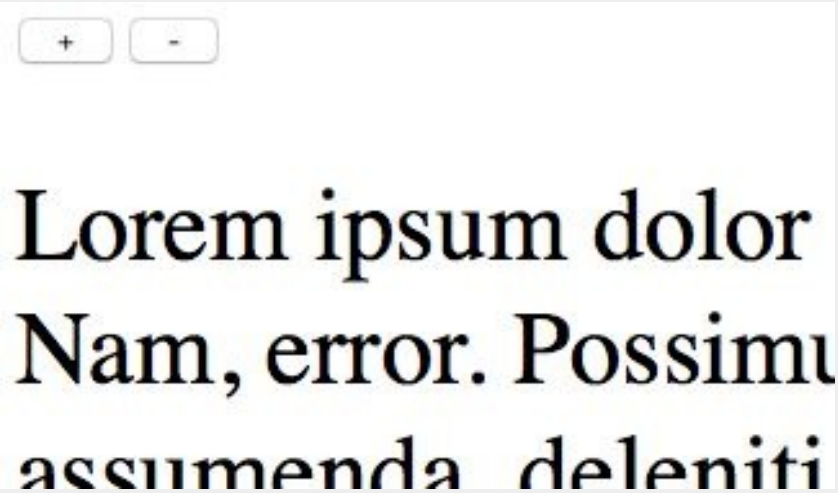
The usual refactoring (in this case, getting rid of redundancies and magic numbers) now results in the following code.

```
 1 "use strict";
 2
 3 {
 4   const SMALL = 14;
 5   const NORMAL = 16;
 6   const BIG = 24;
 7   const VERY_BIG = 36;
 8
 9   const init = () => {
10     $("#very_big").on("click", () => setFontSizeTo(VERY_BIG));
11     $("#big")     .on("click", () => setFontSizeTo(BIG));
12     $("#normal")  .on("click", () => setFontSizeTo(NORMAL));
13     $("#small")   .on("click", () => setFontSizeTo(SMALL));
14   };
15
16   const setFontSizeTo = size =>  $("p").style.fontSize = size + "px";
17
18   init();
19 }
```

Listing 38 *accompanying_files/08/examples/font_size_big_small/font_size_refactored.js*

7.1 Completely Calculating and Manipulative: getComputedStyle

Oh, right...err...you definitely did an awesome job with the font sizes, but...how can I put this...we need to change things again. We surveyed our customers and found out that they'd much rather click on "+" and "-" buttons to change font size.



No problem! Here's the new HTML document:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Font Size</title>
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="font_size.js" defer="defer"></script>
</head>

<body>

  <button id="inc">+</button>
  <button id="dec">-</button>
  <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Nam, error. Possimus in

```

```
</body>

</html>
```

Listing 39 *accompanying_files/08/examples/font_size_inc_dec_non_functional/font_size.html*

First, let's again register the click events on the buttons. It'd also be a good idea to name the functions for increasing and decreasing font size appropriately, say `incFontSize` and `decFontSize`:

```
const init = () => {
  $("#inc").on("click", incFontSize);
  $("#dec").on("click", decFontSize);
};
```

Now, let's create the two functions. We'll start with `incFontSize`.

The idea is quite simple. First, retrieve the current font size. We're pretending as if a function called `currentFontSize` already exists and returns the current font size as a `number`. Add `5` to that value. The result is our new `fontSize`, which you assign back to the style object.

```
const incFontSize = () => setFontSizeTo(currentFontSize() + 5);
```

Use the same process to decrease font size, but instead subtract `5` from `currentFontSize`:

```
const decFontSize = () => setFontSizeTo(currentFontSize() - 5);
```

Now all you need to do is implement the function `currentFontSize` and everything should work. Here's an initial attempt at writing that function:

```
const currentFontSize = () => parseInt($("p").style.fontSize);
```

The idea here is to retrieve the current `fontSize` and use `parseInt` to extract just the number portion, without the `"px"` at the end.

parseInt, or the Secret of the Disappearing Pixel

It's probably a good idea to try out `parseInt` directly in the console first. That function is extremely useful for our current program since it makes it possible for us to get rid of the `"px"` at the end of `fontSize`. For example, try entering `"40px" - 5` in your console. You get back `NaN` (Not a Number) as a return value. The string `"40px"` just isn't a number, so unfortunately it can't be used in a calculation.

But we can use `parseInt` to extract the number. That function takes in a string and ***parses*** it, as long as it finds valid characters (i. e. digits) in the string. Enter the following in your console:

```
parseInt("40px")
```

You get back `40` as a *number* — and now we can do the calculation.

Our first problem is already solved. Now take another look at the implementation of `currentFontSize`:

```
1 const currentFontSize = () => parseInt($("p").style.fontSize);
```

Unfortunately, the code still doesn't work! Why is this?

Really a Question of Style?

Let's try a little experiment — try to retrieve a style which already exists. Refresh the page and open the JS console. Now, print out the style object by entering the following line in the console:

```
console.log($('p').style)
```

You see the style object is of type `CSSStyleDeclaration`. Expand it and display it.

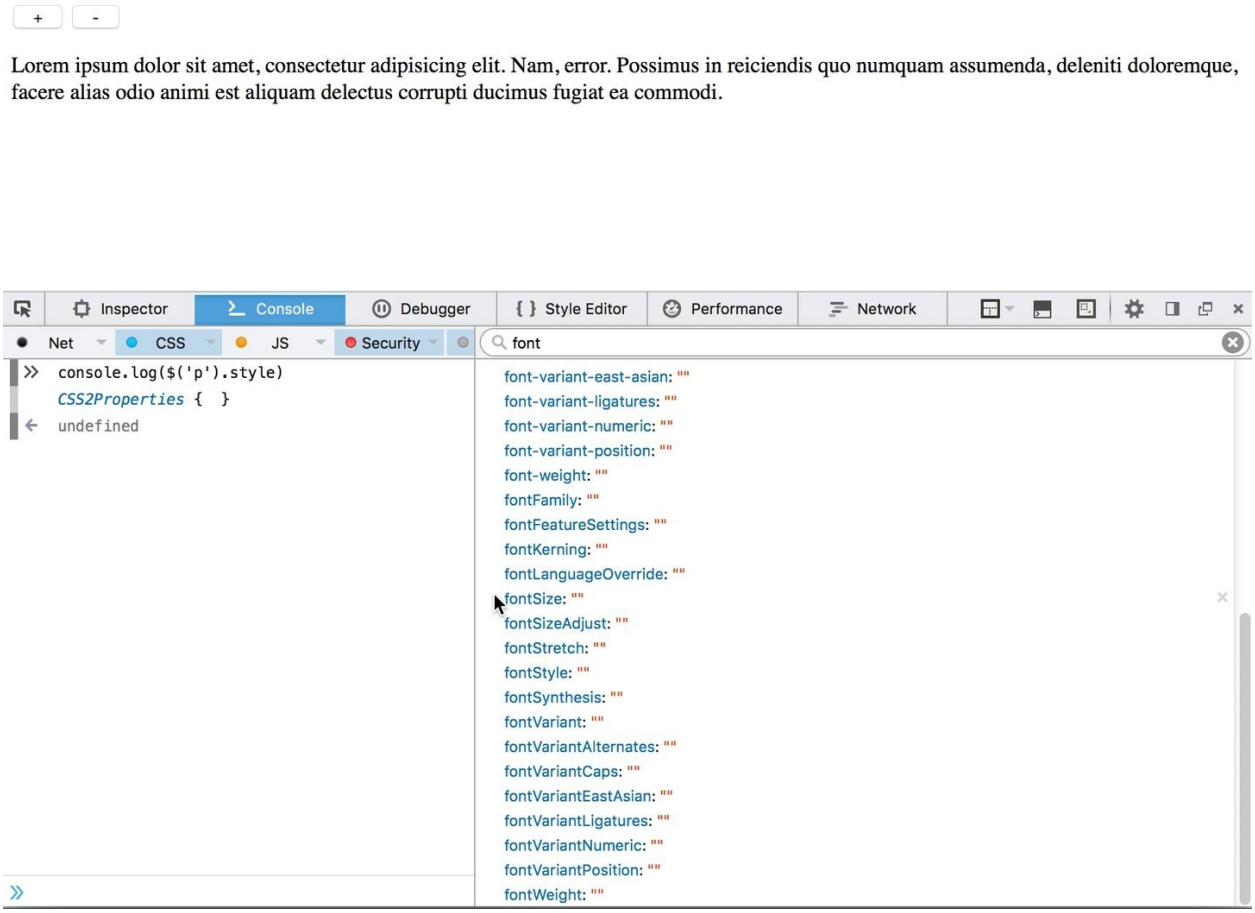


Fig. 16 *`fontSize` property in `style` object with empty string*

You see it has a bunch of properties, including `fontSize`. But what's surprising is that instead of an actual font size like `"16px"`, `fontSize` just contains an empty string `""`.

Try assigning `fontSize` the value "40px" (again in the console).

```
$("p").style.fontSize = "40px";
```

And print it out again:

```
console.log($('p').style);
```

This time, you find that `"40px"` is in fact the value of `fontSize` in the style object.

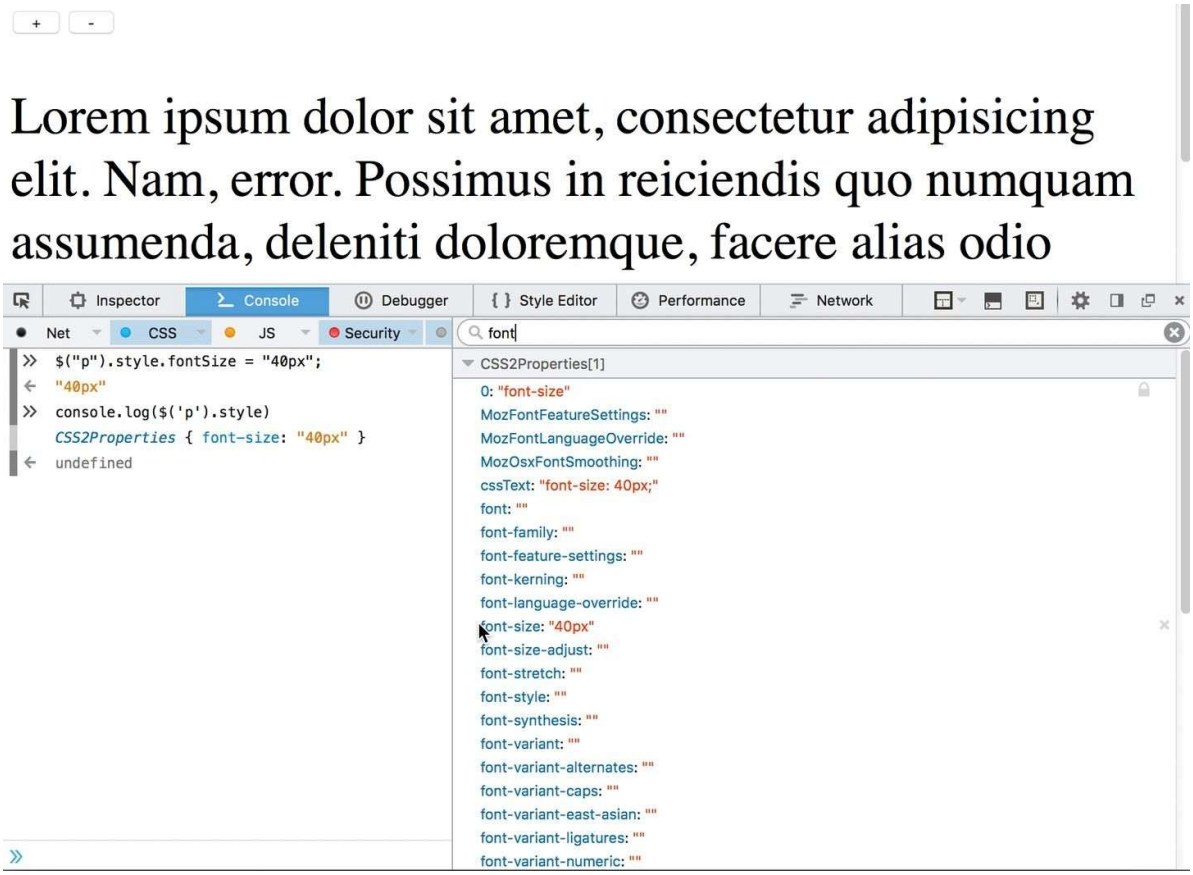


Fig. 17 *`fontSize` property in `style` object with existing px value*

And that's exactly the problem — **you can only retrieve a value from the style object if you've first set that value via JS**. A so-called inline style (style attribute) would set the value in the object as well (however, we recommend that you don't work with inline styles in practice). Unfortunately, the style object doesn't expose default values which the browser sets or values you specify in a CSS style sheet.

But consistently using JS instead of CSS to specify styling values also wouldn't be a good solution — Nerdworld designer Denise wouldn't like that at all …

Fundamental Questions with getComputedStyle

In fact, there is a good solution available: the function `getComputedStyle`.

This function is a method of the `window` object and is therefore available globally — i. e. instead of `window.getComputedStyle`, you can just write `getComputedStyle`. The function is available in all modern browsers and can be used in IE versions 9 and higher.

When you pass `getComputedStyle` an HTML element object (such as our `p` tag), you get back a style object (of type `CSSStyleDeclaration`, just as usual). But what's special about this object is that is contains the actual, calculated values of the HTML element object. So the function also evaluates default browser values and CSS properties, including those in external style sheets. Try this — reload the page and enter:

```
console.log(getComputedStyle($("p")).fontSize)
```

You get back `16px`, the default value for your browser.

If you print out the entire object, you'll see that practically all properties now have values.

Perfect! Marty's definitely going to be happy about that. We can now use `getComputedStyle` for our `currentFontSize` function:

```
const currentFontSize = () =>
  parseInt(getComputedStyle($("p")).fontSize);
```

And here's the complete code:

```
"use strict";

{
  const init = () => {
    $("#inc").on("click", incFontSize);
    $("#dec").on("click", decFontSize);
  };

  const incFontSize = () => setFontSizeTo(currentFontSize() + 5);
  const decFontSize = () => setFontSizeTo(currentFontSize() - 5);

  const currentFontSize = () =>
    parseInt(getComputedStyle($("p")).fontSize);
```

```
14
15    const setFontSizeTo = size =>
16      $("p").style.fontSize = size + "px";
17
18    init();
19  }
```

Listing 40 *accompanying_files/08/examples/font_size_inc_dec/font_size.js*

getComputedStyle Peculiarities and Drawbacks

Yep, that `getComputedStyle` definitely does sound pretty good — **but** unfortunately we're going to have to rain on your parade and clarify a couple of things before you start using it.

Read-Only

The `CSSStyleDeclaration` object that's returned differs from a normal `.style` object in one very significant way: Its properties can't be changed — they're ***read-only***! This really isn't a problem — after all, you can use the `.style` object when you do need to change property values, but you have to keep that fact in mind.

For example, enter

```
getComputedStyle($("p")).fontSize = "19px";
```

into the console. Your browser will bombard you with messages like:

```
VM36020:1 Uncaught DOMException: Failed to set the 'font-size' property on 'CSSStyleDecl
```

Performance

`getComputedStyle` isn't exactly known for being a racehorse — it's much, much slower than `.style`. Therefore it's sometimes a good trick to use `getComputedStyle` to get initial values (e. g. in your `init`

function), then to work with the much faster `.style` after that point.

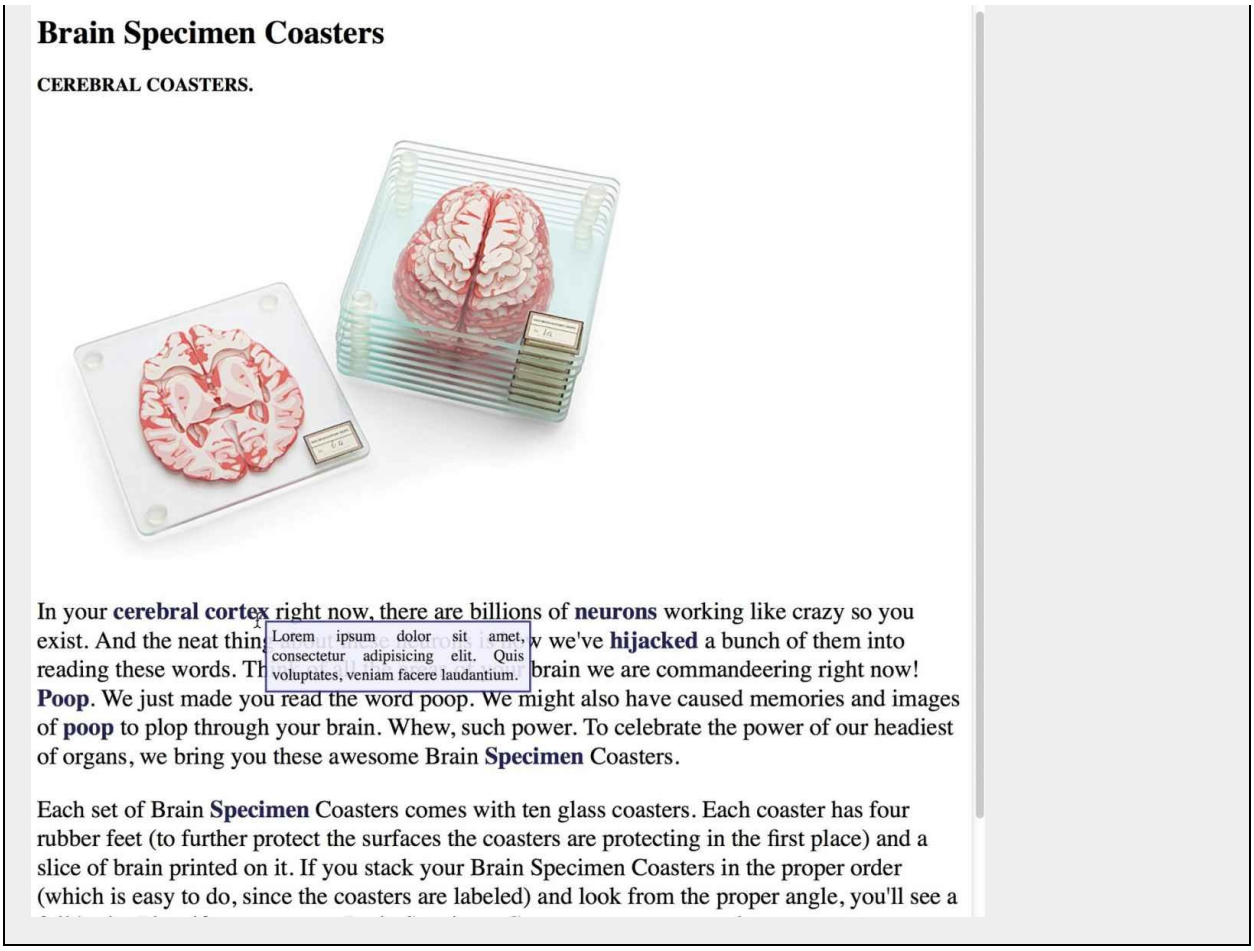
7.2 Even More Stylish with Tooltips

Wow — I showed the new prototype to my grandmother last night! She was thrilled! Thanks to your `"+"` button, she could even read the text without using her glasses.

But she did have a little trouble understanding some of the terms on the website, and wanted to know what *cerebral cortex* and *neuron* mean. Could we maybe add some small tooltips which appear when a user moves the mouse over an uncommon word, to explain what the word means?

Oh, and I don't mean those mini tooltips which you get from the `title` attribute — we need some large, nice-looking tooltip boxes which Denise our designer can add styles to.

Our plan is to have the explanation text for each term to be read in later from an external file, so that we can maintain the terms ourselves. But we don't need that immediately — for now, we'll email you the text and you can add it right in the code.



And we're off again! After just a little reflection, it quickly becomes clear that this is another case in which we'll need to use the style object. Take a look at the following HTML template which represents a simplified version of the actual page[2]:

---

2 Many thanks to Think Geek Inc. for their kind permission! Taken fromhttp://www.thinkgeek.com/product/huir/

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <title>Tooltips</title>

  <script src="../../../../lib/dom_helper.js" defer="defer"></script>
  <script src="tooltips.js" defer="defer"></script>

  <link rel="stylesheet" href="tooltips.css" />
</head>

```

```
<body>
  <h1>Brain Specimen Coasters</h1>
  <h3>CEREBRAL COASTERS.</h3>

  <img src="../images/brain_specimen_coasters.jpg" alt="brain_specimen_coasters" />

  <p>In your <span class="keyword">cerebral cortex</span> right now, there are billions

  <p>Each set of Brain <span class="keyword">Specimen</span> Coasters comes with ten gl
</body>

</html>
```

Listing 41 *accompanying_files/08/examples/tooltips/1/tooltips.html*

Marty has already marked some words with the class `keyword` to indicate they'll need an explanatory tooltip. The first thing we'll need to do is select those words — try running the following statement in your JS console:

```
$$(".keyword").forEach(n => console.log(n.innerHTML));
```

You get back:

```
cerebral cortex
neurons
hijacked
Poop
poop
2 Specimen
poop
```

So far, so good! Now, add that line of code to an actual program, and register a function called `showTooltip` on the `mouseenter` event. Your browser triggers that event when a user's mouse cursor moves over one of the corresponding words.

First, let's just check to make sure we registered the function correctly, so we'll just use `console.log` to print out what we already have.

```
"use strict";

{
  const init = () => $$(".keyword").on("mouseenter", showTooltip);
```

```
5
6   const showTooltip = e => console.log(e.target.innerHTML);
7
8   init();
9 }
```

Listing 42 *accompanying_files/08/examples/tooltips/1/tooltips.js*

Next, we'll need the actual tooltip. Fortunately, Nerdworld's designer Denise has already created one and has styled it using CSS. The tooltip itself is just a `div` tag which contains the explanatory text, and since Marty hasn't yet provided any text, we'll just use *lorem ipsum* text as a placeholder.

```
1 <div id="tooltip">Lorem ipsum dolor sit amet, consectetur adipisicing elit. Quis volupta
```

We'll just insert the tag at the end of the HTML document, right before the closing `body` tag.

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>...</head>
5
6  <body>
7    <p>...</p>
8
9    <div id="tooltip">Lorem ipsum dolor sit amet, consectetur adipisicing elit. Quis volu
10 </body>
11
12 </html>
```

The tooltip already looks pretty good, thanks to the following CSS rule in *tooltips.css*:

```
1  #tooltip {
2    width: 250px;
3    border: 2px solid #339;
4    padding: 5px;
5    background-color: #eef;
6    opacity: 0.95;
7
8    font-size: 1em;
9    text-align: justify;
10 }
```

And Denise plans to make it look even better.

Add the style `display: none;` to the rule so the tooltip is initially hidden

when the page loads. The tooltip should be displayed only if a visitor to the website moves his/her mouse cursor over an appropriate word. To do this, we just need to use JavaScript to set the tooltip's `display` value to `block`. This is because the tooltip is a `div` element. If it were a `span` element, we'd have to use the value `inline` instead.

```
$("#tooltip").style.display = "block";
```

Setting display style is the job of the function `showTooltip`, so let's modify it accordingly.

```
const showTooltip = () => $("#tooltip").style.display = "block";
```

…And Gone Again

The tooltip should disappear again after the user moves the mouse cursor away from the element. To do this, we'll need an appropriate function called `hideTooltip`, which is best registered with the `mouseleave` event. Here's the complete code:

```
"use strict";

{
  const init = () => $$(".keyword")
    .on("mouseenter", showTooltip)
    .on("mouseleave", hideTooltip);

  const showTooltip = () => $("#tooltip").style.display = "block";
  const hideTooltip = () => $("#tooltip").style.display = "none";

  init();
}
```

Listing 43 *accompanying_files/08/examples/tooltips/2/tooltips.js*

For the tooltip to actually be a tooltip, it shouldn't appear under the text, but should always be displayed near the mouse cursor. Assign the tooltip the property `position: absolute` in the CSS so it can be positioned

anywhere we like:

```
p {
  font-size: 1.5em;
}

.keyword {
  font-weight: bolder;
  color: #225;
}

#tooltip {
  width: 250px;
  border: 2px solid #339;
  padding: 5px;
  background-color: #eef;
  opacity: 0.9;

  font-size: 1em;
  text-align: justify;

  display: none;
  position: absolute;
}
```

Listing 44 *accompanying_files/08/examples/tooltips/3/tooltips.css*

In addition, we should change the event from `mouseenter` to `mousemove` so that the position of the tooltip is constantly updated:
`$$(".keyword").on("mousemove", showTooltip) …`

Mouse in Focus

Next, we need to find out exactly where the mouse cursor is positioned. Conveniently enough, the event object takes care of just that for mouse events, and registers the current position of the mouse cursor in the properties `clientX` and `clientY`.

Let's add code to the function `showTooltip` so the tooltip jumps to the position of the mouse cursor:

```
const showTooltip = e => {
  $("#tooltip").style.display = "block";
  $("#tooltip").style.top = e.clientY + "px";
  $("#tooltip").style.left = e.clientX + "px";
};
```

By using the style properties `top` and `left`, we can position the tooltip with pixel precision (even that might be a little too fine). Now, the tooltip is always "stuck" right to the cursor. Adding a small interval, say `10px`, will solve that problem.

```
$("#tooltip").style.top = e.clientY + 10 + "px";
$("#tooltip").style.left = e.clientX + 10 + "px";
```

Ideally that value should be stored in a constant to avoid the use of ***magic numbers***[3]. And here's the complete code:

---

3 You'll find a good explanation of magic numbers in *JavaScript volume 1 — The First Step is the Easiest*

```
"use strict";

{
  const MOUSE_TO_TOOLTIP = 10;

  const init = () => {
    $$(".keyword").on("mousemove", showTooltip).on("mouseleave", hideTooltip);
  };

  const showTooltip = e => {
    $("#tooltip").style.display = "block";
    $("#tooltip").style.top = e.clientY + MOUSE_TO_TOOLTIP + "px";
    $("#tooltip").style.left = e.clientX + MOUSE_TO_TOOLTIP + "px";
  };

  const hideTooltip = () => $("#tooltip").style.display = "none";

  init();
}
```

Listing 45 *accompanying_files/08/examples/tooltips/3/tooltips.js*

Hooray! The tooltips are working perfectly. Granny was thrilled — she finally knows what a *cerebral cortex* is. But maybe we'll take out the explanation of *poop*…

With Style or With Class?

You often come across situations in which you need to make a decision — you can either use `classList.add`/`classList.remove` and change CSS properties (see), or you can access the style object and change a property directly. So when should you change classes, and when should you change style properties?

It's better to work with CSS classes in most cases. The main advantage of doing this is that you keep the design aspects of your program (CSS) separate from its behavioral aspects (JS). CSS properties are encapsulated within a CSS class. JS code just decides whether to add and remove classes, but doesn't make any decisions concerning the actual properties affected by this.

This means, for example, that a designer can make later improvements to CSS without having to touch the JS code. This makes work flow tremendously easier, even if the designer and developer are the same person. If the JS and CSS code are mixed together, a change to the visual element of a program (CSS) always brings the risk of introducing undesired changes in behavior (i. e. bugs).

Nevertheless, there are situations in which it's better to work with the style object directly, such as in our previous example with fonts. If we didn't make use of the style object in that example, we would have had to create appropriate classes for a wide range of font sizes, since the *plus/minus* buttons were using an increment of 5px.

7.3 Exercises

Exercise 16: 01001000010011110101010O — Part 6

We're now asking you to color the text `blue` (for the time being, it's better if you don't ask why ;) But this time, don't use CSS classes.

Use the style object to assign the color `blue` to all `p` elements. You can work directly in the JavaScript console.

Exercise 17: TABula Rasa

One of your clients is a training company. The director of the academy wants the description pages for the workshops offered by the academy to be laid out more clearly. The overview pages for workshops should now appear on separate tabs so that visitors to the website can easily jump back and forth between the different sections which describe a workshop, which include:

- *overview*,
- *subjects*,
- *learning objectives* and
- *schedule*

The director already had a design agency create an appropriate HTML template (including the CSS). Your job now is to extend the JS code appropriately.

```
1 <!DOCTYPE html>
```

```
<html lang="en">

<head>
  <meta charset="utf-8" />

  <title>SEO Seminar: 2-tägige Schulung Suchmaschinenoptimierung</title>

  <link href="tabs.css" media="all" rel="stylesheet" type="text/css" />
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="tabs.js" defer="defer"></script>
</head>

<body>

  <div class="tabs">
    <nav>
      <ul>
        <li>Overview</li>
        <li>Topics</li>
        <li>Learning Objectives</li>
        <li>Date</li>
      </ul>
    </nav>

    <article>
      <h1>2-day SEO intensive seminar: with search engine optimization (SEO) to succe
      <p>&hellip;</p>
    </article>

    <article>
      <h2>Topics</h2>
      <p>&hellip;</p>
    </article>

    <article>
      <h2>Learning Objectives</h2>
      <p>&hellip;</p>
    </article>

    <article>
      <h2>Date</h2>
      <p>&hellip;</p>
    </article>
  </div>

</body>

</html>
```

accompanying_files/08/exercises/tabbed_navigation/tabs.html

You'll need to implement the following features for the website:

- After a workshop page loads, all `article` elements except the first should be hidden.
- Clicking on an `li` element should cause the associated `article` element to become active (i. e. visible) and all other `article`

elements to become inactive.

- In addition, so that you can check which tab is currently active, only the `li` element of the tab which is currently visible should have the CSS class `active`.

Exercise 18: Playing Tag With Colors

Create an HTML page with three buttons (`button` tag) labeled "Red", "Green" and "Blue". When a user clicks on a button, the background of the page should change to the corresponding color.

Exercise 19: Playing Tag With Colors — Part 2

Create an HTML page with three control elements (`input type="range"`), each of which is responsible for controlling one color proportion of the RGB color model. Every time a user moves a controller, the background of the page should change accordingly. The three controllers each regulate the proportion of their respective color and thus determine the background color of the page.

8 Undermining HTML Using Data Attributes

Like I said, those tooltips are awesome! But now we need an easy way to maintain the text for the tooltips. Right now, I think the best way would be if I could store the text directly in the HTML. Can you do that?

Yes, you can! You even have your choice of solutions. The easiest would be to use so-called `data attributes`. These make it possible for you to store any content in your HTML virtually "invisibly"!

HTML lets you define your own attributes, provided they start with the prefix `data-`. For example, you could define an attribute called `data-tooltip` and store corresponding text there. `tooltip` in this case is a name you choose yourself, and not something dictated by HTML. You can use virtually anything you like as an attribute name, with just a few restrictions (e. g. uppercase letters aren't permitted). Even tooltip names like `data-foo`, `data-my-client-is-an-idiot` or `data-whatever-you-want` are fine.

```
1 <p>In your <span class="keyword" data-tooltip="The brain's outer layer of neural tissue
```

Experiment with the attribute and try to retrieve it from the console. You'll first need the element: `$(".keyword")`. Then to access a `data-` attribute, use the property `dataset`:

```
$(".keyword").dataset
```

This returns a *DOMStringMap*, which you process just like any other map. Every data attribute in this map, including `tooltip`, is represented by its own key. This is the return value from the last statement:

```
1 DOMStringMap {tooltip: "The brain's outer layer of neural tissue in humans and other mam
```

Now, we just need to modify our JS code so that it displays the correct text from the `data` attribute instead of the static lorem ipsum text. Let's add the following line to our `showTooltip` function:

```
$("#tooltip").textContent = e.target.dataset.tooltip;
```

The attribute `textContent` makes it possible to assign any text content to a tag (such as the `tooltip` div here, which contains the lorem ipsum text). In that way, it's very similar to the attribute `innerHTML`. We could actually use `innerHTML` here instead with no problem. However, using `textContent` is preferred when we're dealing with just text, as we are here. `textContent` doesn't interpret a string assigned to it as HTML — this means we could also use characters like `&` and `<` directly in the text.

It's for this reason that `innerHTML` is actually a potential security risk in situations where you don't trust the source of the content. `textContent` always escapes characters, so there's no danger that "malicious" tags will creep into your HTML code.

> Always use `textContent` instead of `innerHTML` when you need to insert text!

8.1 Summary

Attribute	Example	Description
dataset	$("#bar").dataset	Contains a *DOMStringMap* with the data of `data` attributes.
textContent	$("#bar").textContent	Returns the text (without HTML tags) which appears in an HTML element (in addition, the value of textContent can be modified).

Table 8.1 *Attributes in this lesson*

8.2 Exercise

Exercise 20: awesome tours

Fig. 18 *awesome tours prototype*

The travel company *awesome tours* offers tours of various tourist attractions in Europe. The company is in the process of rebuilding its website and at the same time wants to modernize the display of its tourist attractions. An agency has already completed the design portion of the rebuild, including the HTML and CSS code.

```
...
<body>
  <section id="pois">
    <img src="eiffelturm.jpg" id="fr" alt="Eiffel Tower" data-description="The Eiffel

    <img src="tower_bridge.jpg" id="uk" alt="Tower Bride" data-description="Tower Bri

    <img src="colosseum.jpg" id="it" alt="Colosseum" data-description="The Colosseum
  </section>
```

```
10
11   <section id="info"><p>Move your mouse pointer over the images to read the descripti
12 </body>
13 ...
```

The agency also took care of storing the information on tourist attractions (name, description, country, etc.) in the HTML code. Your job now is to make it possible for this information to be displayed as shown in fig. 18 when a mouseenter event occurs.

Your display should include:

- the name of the attraction
- its description
- the corresponding country's flag, as an image.

For example, the HTML for displaying information about the Colosseum might look like:

```
1 <section id="info">
2   <h3>
3      <img src="it.png" alt="Italian Flag" title="Italian Flag">
4      Colosseum
5   </h3>
6   <p>
7     The Colosseum or Coliseum, also known as the Flavian Amphitheatre or Colosseo, is
8   </p>
9 </section>
```

9 Of New Elements and Appended Children: createElement & appendChild

The guys and gals in our business development unit finally finished the server code for our new product manager tool. However, we're still a little unsure of what the interface to it should look like, and we're in desperate need of a prototype. It doesn't have to support a connection to the server yet — it should essentially demonstrate how product maintenance over the web might work. Naturally I thought of you …

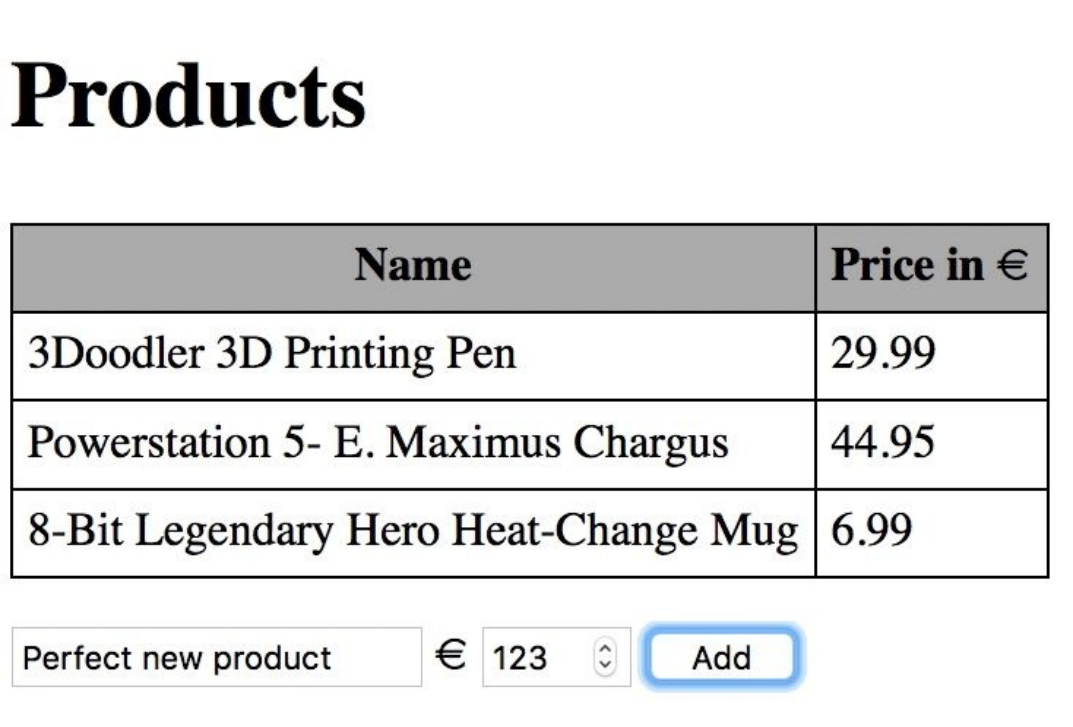


Fig. 19 *Initial product manager prototype (with add feature)*

After you meet with Marty and get the detailed specifications for his new requirement, you're able to provide him with an HTML framework for the product interface, which includes a few sample products:

```
<h1>Products</h1>
<table id="products">
  <thead>
    <tr>
      <th>Name</th>
      <th>Price in €</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>3Doodler 3D Printing Pen</td>
      <td>29.99</td>
    </tr>
    <tr>
      <td>Powerstation 5- E. Maximus Chargus</td>
      <td>44.95</td>
    </tr>
    <tr>
      <td>8-Bit Legendary Hero Heat-Change Mug</td>
      <td>6.99</td>
    </tr>
  </tbody>
</table>

<p id="new_product">
  <input type="text" placeholder="name" class="name"/>
  € <input type="number" max="999" placeholder="price" class="price"/>
  <button id="add_product">Add</button>
</p>
```

Listing 47 *additional_files/10/examples/product_manager_1/product_manager.html*

Our first step is to get the *add* button to work. As usual, we register a function on the click event:

```
"use strict";
{
  const init = () => {
    $("#add_product").on("click", addProduct);
  };

  const addProduct = () => { ... };
  init();
}
```

9.1 Generating HTML Using document.createElement

Exactly what should `addProduct` do? One possible approach, which you already know how to implement, would be to use `innerHTML` to rewrite the table and append the latest product.

However, this approach brings the serious disadvantage of having to rewrite the entire table in HTML every time we add a product, even though we're just adding one line. Another disadvantage is that we're working with "malleable" strings instead of a DOM subtree. Strings are susceptible to syntax errors, plus we can't use node object methods as long as the string hasn't yet been "appended" to the page using `innerHTML`.

A better approach would be to construct complete HTML ***subtrees***, then to append these to the main tree as necessary.

That's very easily done!

You'll just need to learn a couple of new methods. We can use `document.createElement` to create an *HTMLElement* object. So first, let's create a `td`.

```
const addProduct = () => {
  const valueTd = document.createElement("td");
  ...
};
```

We can then use `textContent` to assign this new element content in the form of text (in this case, the new product name from the input field).

```
const addProduct = () => {
  const valueTd = document.createElement("td");
```

```
  valueTd.textContent = $("#new_product .name").value;
  ...
};
```

Take a second to print out the object we created by adding `console.log(valueTd);`:

```
const addProduct = () => {
  const valueTd = document.createElement("td");
  valueTd.textContent = $("#new_product .name").value;
  console.log(valueTd);
};
```

Try out the code, giving your product a name like *My New Product* and a price of *$100*. You get back:

```
<td>My New Product</td>
```

`valueTd` is not the string `<td>My New Product</td>` but an actual `HTMLTableCellElement` — a subclass of `HTMLElement` and of `Node`. In a nutshell, this means you can manipulate its `classList` or attach an event handler to it, which you couldn't do with a string.

9.2 Finally Attached: appendChild

To complete the table row, we now need to construct the `td` with the product price, then append the two elements to a new `tr`. We'll use the node object's `appendChild` method to append the elements:

```
const addProduct = () => {
  const valueTd = document.createElement("td");
  valueTd.textContent = newProductName();

  const priceTd = document.createElement("td");
  priceTd.textContent = newProductPrice();

  const productTr = document.createElement("tr");
  productTr.appendChild(valueTd);
  productTr.appendChild(priceTd);

  console.log("productTr", productTr);
}
```

The `console.log` we included at the end of the function now shows you the complete HTML fragment for the table row:

```
<tr>
  <td>My New Product</td>
  <td>100</td>
</tr>
```

The new row will be displayed as soon as we append that fragment to the correct place (the `tbody` of the table) in the main HTML tree.

```
...
$("#products > tbody").appendChild(productTr);
```

And here's the complete code:

```
"use strict";

{
  const init = () => {
    $("#add_product").on("click", addProduct);
  };
```

```
  const addProduct = () => {
    const valueTd = document.createElement("td");
    valueTd.textContent =  $("#new_product .name").value;

    const priceTd = document.createElement("td");
    priceTd.textContent = $("#new_product .price").value;

    const productTr = document.createElement("tr");
    productTr.appendChild(valueTd);
    productTr.appendChild(priceTd);

    $("#products > tbody").appendChild(productTr);
  };

  init();
}
```

Listing 48 *accompanying_files/10/examples/product_manager_1/product_manager.js*

Of course, the code isn't exactly a work of art. But even though we weren't expecting it would be, we should at least make it a little more readable. And since you don't want any more grief from Carl the consultant, a little refactoring might also be in order.

9.3 Once Again, a Little Refactoring

The basic problem is that the function `addProduct` has way too many responsibilities. It

- takes in the product data from the form
- essentially adds the product
- takes care of low-level details such as creating `tr` and `td` elements.

We should separate precisely those responsibilities (***separation of concerns***), otherwise our code will result in undesired redundancies. It's possible that our code makes more accesses to `createElement`, `appendChild` and `textContent` than necessary. This can be an initial indication of the places where we should pull functionality out and store it in a separate function.

Let's start with the low-level details. A pattern emerges from the following code:

```
const valueTd = document.createElement("td");
valueTd.textContent = $("#new_product .name").value;

const priceTd = document.createElement("td");
priceTd.textContent = $("#new_product .price").value;
```

The code creates a `td` twice, and assigns the `td` text content each time. How about creating a method called (surprise, surprise) `td` that does exactly that?

```
const td = text => {
  const tdNode = document.createElement("td");
  tdNode.textContent = text;
  return tdNode;
};
```

So all that remains of `addProduct` is:

```
const addProduct = () => {
  const productTr = document.createElement("tr");
  productTr.appendChild(td($("#new_product .name").value));
  productTr.appendChild(td($("#new_product .price").value));
  $("#products > tbody").appendChild(productTr);
};
```

The obvious thing to do would be to do the same thing with the `tr` element. The difference in this case is that instead of a text string, the `tr` can take in multiple `td` elements (as an array).

```
const tr = tds => { const trNode = document.createElement("tr"); tds.forEach(td => trNode
```

`addProduct` now looks like:

```
const addProduct = () =>
  $("#products > tbody").appendChild(
    tr([
      td($("#new_product .name").value),
      td($("#new_product .price").value)
    )];
```

Finally, we should separate the code for retrieving product data from the code for adding the product. It may well be that eventually we'll want to add products whose data doesn't come from form fields (spoiler alert):

```
const init = () => {
  $("#add_product").on("click", addProductFromInput);
};

const addProductFromInput = () =>
  addProduct(
    $("#new_product .name").value,
    $("#new_product .price").value
  );

const addProduct = (name, price) =>
  $("#products > tbody").appendChild(
    tr([
      td(name), td(price)
    ])
  );
```

…so after about 2 minutes and 33½ seconds of refactoring, we get the final result below:

```
"use strict";

{
  const init = () => {
    $("#add_product").on("click", addProductFromInput);
  };

  const addProductFromInput = () =>
    addProduct(
      $("#new_product .name").value,
      $("#new_product .price").value
    );

  const addProduct = (name, price) =>
    $("#products > tbody").appendChild(
      tr([
        td(name), td(price)
      ])
    );

  const td = text => {
    const tdNode = document.createElement("td");
    tdNode.textContent = text;
    return tdNode;
  };

  const tr = tds => {
    const trNode = document.createElement("tr");
    tds.forEach(td => trNode.appendChild(td));
    return trNode;
  };

  init();
}
```

Listing 49 *accompanying_files/10/examples/product_manager_2_refact/product_manager.js*

Exercise 21: 01001000010011110101010O — Part 7

Use JavaScript to change the product page for the binary mug. You can again use the JavaScript console for this exercise.

Use JavaScript to add the following list item to the `product specifications` list: *Capacity: 11 oz.*

Use JavaScript to add the following entry to the select box next to the Buy button: *5 items*.

9.4 And Now a Little JSON

Could you remove the three products from the HTML? I'd like it better if we could maintain them in an external file. I know, I know … it's just a prototype. But in order to test it meaningfully, it'd also be a good idea if we could quickly change products without having to change the HTML.

We do have the ability to export data from our database in the form of JSON — so if we need to, we can just run a test with a couple of actual products. Can we store products in the external file in JSON (***JavaScript Object Notation***) form?

Of course! The file for the three products looks like this:

```
"use strict";

const PRODUCTS = [
  {name: "3Doodler 3D Printing Pen", price: 29.99},
  {name: "Powerstation 5- E. Maximus Chargus", price: 44.95},
  {name: "8-Bit Legendary Hero Heat-Change Mug", price: 6.99}
];
```

Listing 50 *accompanying_files/10/examples/product_list.js*

The three products are stored as an array in the constant `PRODUCTS`. Each array element is an object which contains the attributes `name` and `price` and the corresponding values.

We should first refactor the product manager code a little more so that it's easier to make use of the data in the new format: Instead of passing the `addProduct` function `name` and `price` separately, it'd be a good idea to combine these attributes into one object, just as they appear in the

`PRODUCTS` array — e.g. `{name: "3Doodler 3D Printing Pen", price: 29.99}`.

To do this, let's first modify the function `addProduct`. Within the function, we can access the original `name` and `price` values as the attributes `product.name` and `product.price` of the `product` object passed to the function.

```
const addProduct = product =>
  $("#products > tbody").appendChild(
    tr([
      td(product.name), td(product.price)
    ])
  );
...
```

Now we need to modify the call to `addProduct` which appears in the function `addProductFromInput` — we now need to pass `addProduct` an object so that our code will continue to work correctly. We'll create this object immediately when the function is called, using the form values for `name` and `price`.

```
...
const addProductFromInput = () =>
  addProduct({
    name: $("#new_product .name").value,
    price: $("#new_product .price").value
  });
```

`addProduct` now works with entire objects instead of just individual values. This makes it easy to add the objects from the `PRODUCTS` constant as the file is being loaded. To do this, let's write a new function called `addExistingProducts` which uses a `forEach` to add all products.

```
const addExistingProducts = () => PRODUCTS.forEach(addProduct);
```

Call that function from `init`.

```
const init = () => {
  addExistingProducts(PRODUCTS);
```

```
  $("#add_product").on("click", addProductFromInput);
};
```

Hurray! — we've imported the products. As usual, here's the code in its entirety:

```
"use strict";

{
  const init = () => {
    addExistingProducts(PRODUCTS);
    $("#add_product").on("click", addProductFromInput);
  };

  const addExistingProducts = () => PRODUCTS.forEach(addProduct);

  const addProductFromInput = () =>
    addProduct({
      name: $("#new_product .name").value,
      price: $("#new_product .price").value
    });

  const addProduct = product =>
    $("#products > tbody").appendChild(
      tr([
        td(product.name), td(product.price)
      ])
    );

  const td = text => {
    const tdNode = document.createElement("td");
    tdNode.textContent = text;
    return tdNode;
  };

  const tr = tds => {
    const trNode = document.createElement("tr");
    tds.forEach(td => trNode.appendChild(td));
    return trNode;
  };

  init();
}
```

Listing 51 *accompanying_files/10/examples/product_manager_3_json/product_manager.js*

Ha — I just tested your code with a huuuuge product file in JSON format. Works perfectly!

9.5 Comparison of Various DOM Creation Methods

In this course, you learned a number of ways to create DOM elements. table 9.1 summarizes and compares these different methods. For purposes of illustration, each method includes an example based on the following simple HTML structure:

```
... <body> <p>World</p> </body> ...
```

JavaScript Statement	HTML Structure After Execution

`document.body`
`  .innerHTML = "<p>Hello</p>";`
	`<body>`
	`  <p>Hello</p>`
	`<body>`

`document.body` `  .textContent = "Hello";`	`<body>Hello</body>`

`document.body`
`  .appendChild($(p).cloneNode());`
	`<body>`
	`  <p>World</p>`
	`  <p>World</p>`
	`<body>`

`document.body` `  .appendChild(` `  document.createElement("br")` `  );`	`<body>` `  <p>World</p>` `   ` `<body>`

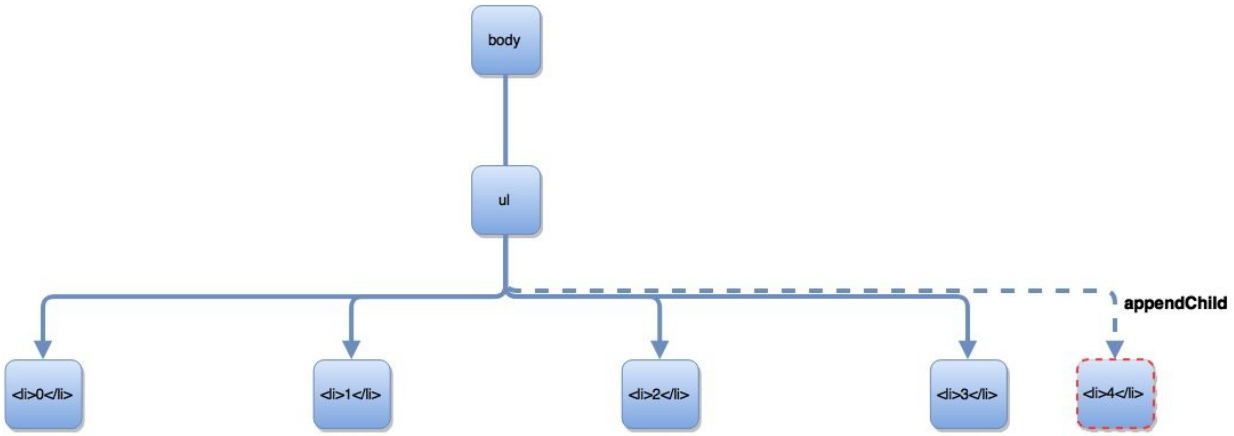
Table 9.1 *Comparison of DOM creation methods*

9.6 Summary

Method	Description
appendChild	Appends the node passed to it (e. g. an element) to the parent node on which the method was called. The node is appended as the last child of the parent node. If the appended node is already part of the HTML document, it's also removed from its original position.
document.createElement	Creates an element of the specified type.
cloneNode	Copies a node and returns the copy. `cloneNode(true)` also copies all child nodes.

Table 9.2 *Methods in this Lesson*

Example for appendChild with createElement



HTML structure (before):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

JavaScript in the console:

```
const li = document.createElement("li");
li.textContent = "4";
$("ul").appendChild(li);
```

HTML structure (after):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
  </ul>
</body>
```

Example for appendChild *with* cloneNode

HTML structure (before):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

JavaScript in the console:

```
const li = $("ul li:first-child").cloneNode(true);
$("ul").appendChild(li);
```

HTML structure (after):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>0</li>
  </ul>
</body>
```

Example for appendChild *without* cloneNode

HTML structure (before):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

JavaScript in the console:

```
const li = $("ul li:first-child");
$("ul").appendChild(li);
```

HTML structure (after):

```
...
<body>
  <ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>0</li>
  </ul>
</body>
```

10 Tossing Elements in the Trash with remove

I tested our website and used the form to add a bunch of products. But now I'd like to delete some of them. Could you add a remove button?

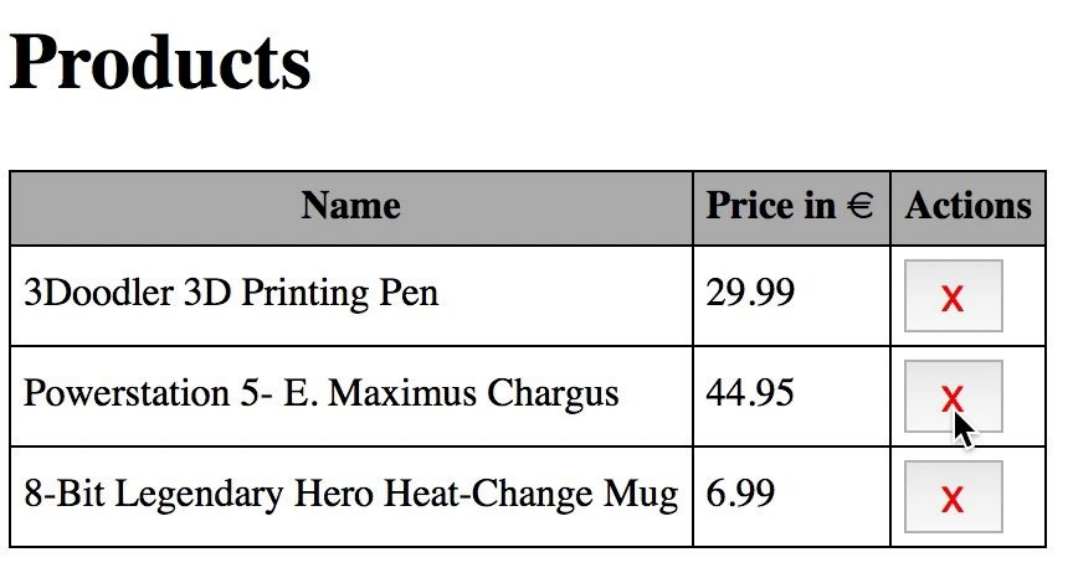


Fig. 20 *Product manager: remove feature*

First, it'd be a good idea to add another column to the table in the HTML so that each product has its own remove button. We'll give this column a general name of *Actions* just in case Marty eventually wants other actions in addition to the remove button.

```
<table id="products">
    <tr>
      <th>Name</th>
      <th>Price in €</th>
      <th>Actions</th>
    </tr>
</table>
```

Now let's extend the `addProduct` function — each product row now needs an additional table data tag (`td`) for the remove button.

```
const addProduct = product =>
  $("#products > tbody").appendChild(
    tr([
      td(product.name), td(product.price), tdWithRemoveButton()
    ])
  );
```

We're still missing a function `tdWithRemoveButton` to create the corresponding HTML fragment.

```
const tdWithRemoveButton = () => {
  const td = document.createElement("td");
  td.appendChild(removeButton());
  return td;
};

const removeButton = () => {
  const button = document.createElement("button");
  button.innerText = "x";
  button.classList.add("remove_product");
  return button;
};
```

Dividing the code up into two functions improves readability. Our remove button displays an `x` as a remove symbol, and is assigned an appropriate CSS class `remove_product`. Now, let's add some functionality: Let's register a function called `removeProduct` on the `click` event.

```
...
const removeButton = () => {
    button.on("click", removeProduct);
    return button;
};
```

`removeProduct` needs to remove the entire table row (`tr`) which contains the product. In order to do that, we first need to find the right row. If we start from the button (here, `event.target`) and move up two levels in the tree structure, we'll have the row. We can use the attribute `parentNode` to perform such moves to a higher level — we're looking for the appropriate

parent node.

```
event.target.parentNode.parentNode
```

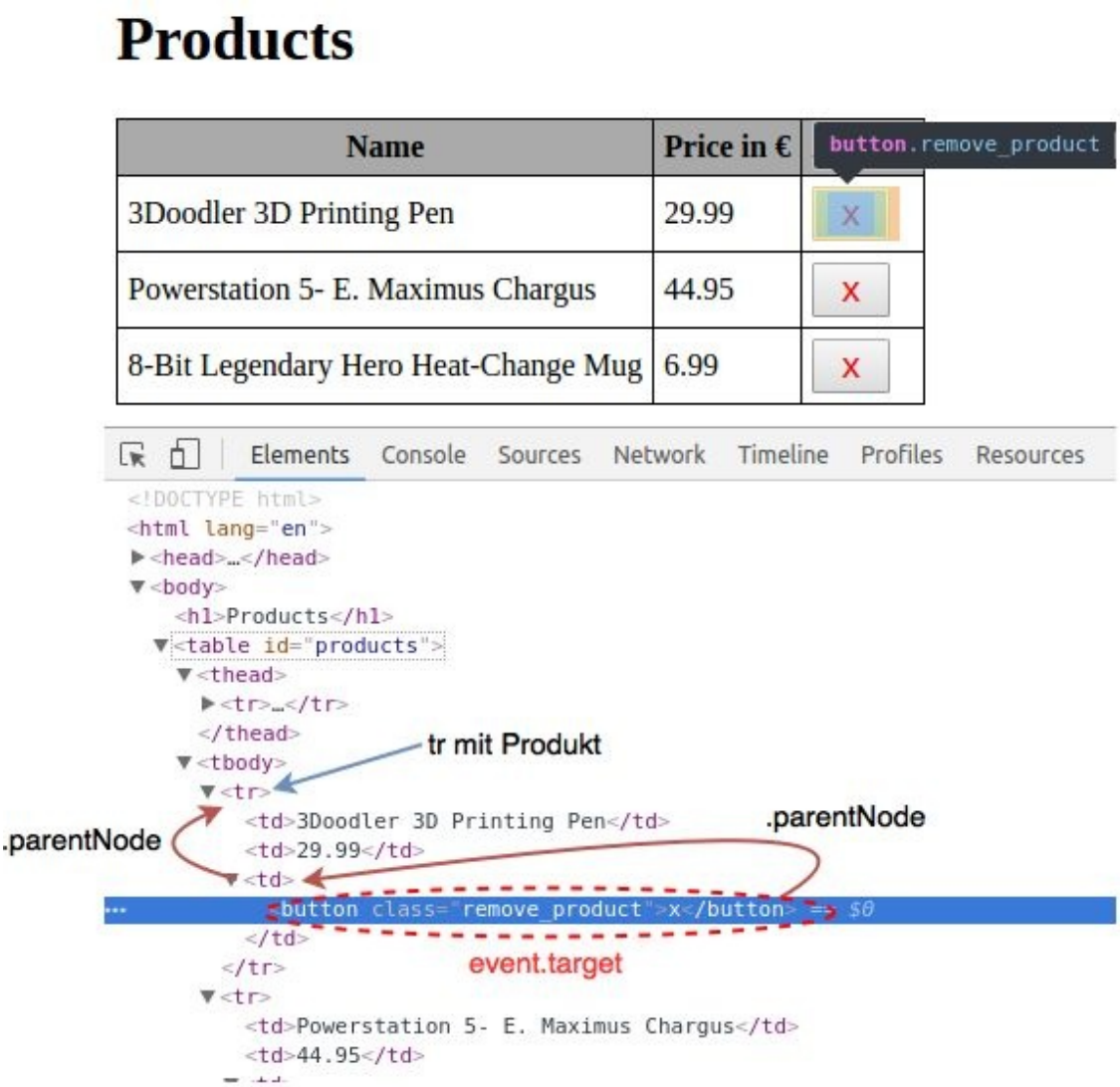


Fig. 21 *Product manager: from button to product row*

The attribute `parentNode` is one of a group of attributes which allow you to move from one node to another within the DOM tree. The category into which such attributes are classified is called ***DOM traversal*** (we'll take a closer look at that in just a little while).

Remove the row we found (`tr`) by using the `remove` method of the node

object associated with that row — and it's gone!

```
const removeProduct = event =>
  event.target.parentNode.parentNode.remove();
```

After some extract refactoring, we get the following, more readable, code:

```
const removeProduct = event =>
  productRowForAction(event.target).remove();

const productRowForAction = button => button.parentNode.parentNode;
```

And here's the code once more in its entirety:

```
"use strict";

{
  const init = () => {
    addExistingProducts(PRODUCTS);
    $("#add_product").on("click", addProductFromInput);
  };

  const addExistingProducts = () => PRODUCTS.forEach(addProduct);

  const addProductFromInput = () =>
    addProduct({
      name: $("#new_product .name").value,
      price: $("#new_product .price").value
    });

  const addProduct = product =>
    $("#products > tbody").appendChild(
      tr([
        td(product.name), td(product.price), tdWithRemoveButton()
      ])
    );

  const tdWithRemoveButton = () => {
    const td = document.createElement("td");
    td.appendChild(removeButton());
    return td;
  };

  const removeButton = () => {
    const button = document.createElement("button");
    button.textContent = "x";
    button.classList.add("remove_product");
    button.on("click", removeProduct);
    return button;
  };

  const removeProduct = event =>
    productRowForAction(event.target).remove();

  const productRowForAction = button =>
```

```
    button.parentNode.parentNode;

  const td = text => {
    const tdNode = document.createElement("td");
    tdNode.textContent = text;
    return tdNode;
  };

  const tr = tds => {
    const trNode = document.createElement("tr");
    tds.forEach(td => trNode.appendChild(td));
    return trNode;
  };

  init();
}
```

Listing 52 *accompanying_files/11/examples/product_manager_4_remove/product_manager.js*

Life in the Old Days — remove vs. removeChild

The `remove` method of the element object is a relatively new development. In fact, it's not available in Internet Explorer (including IE11). Only modern browsers like Chrome, Firefox and Edge come with it.

However, that's not the end of the world. Older browsers already provide a similar method called `removeChild`. As a result, it's very easy to create the following polyfill:

```
if (!('remove' in Element.prototype)) {
    Element.prototype.remove = function() {
        if (this.parentNode) {
            this.parentNode.removeChild(this);
        }
    };
}
```

Listing 53 *Source: https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove*

Just add the polyfill to your *dom_helper* library if you want to provide support for older browsers.

10.1 Classification

FYI, you'll probably find it quite helpful to classify DOM properties by their purpose. Classifying them will make it easier for you to get some overview over the multitude of properties which exist. You'll also find such classification very helpful when you're looking for something specific.

DOM attributes and methods can typically be divided into these three primary categories:

- DOM selection
- DOM manipulation
- DOM traversal

The category of ***DOM selection*** primarily includes methods like `document.querySelectorAll`, which make it possible for you to isolate DOM elements. You can then modify these elements by using DOM manipulation methods like `classList.add` or DOM manipulation properties like `.innerHTML`.

FYI, we also use this classification in the Reference appendix to make it easier for you to look up information.

10.2 Summary

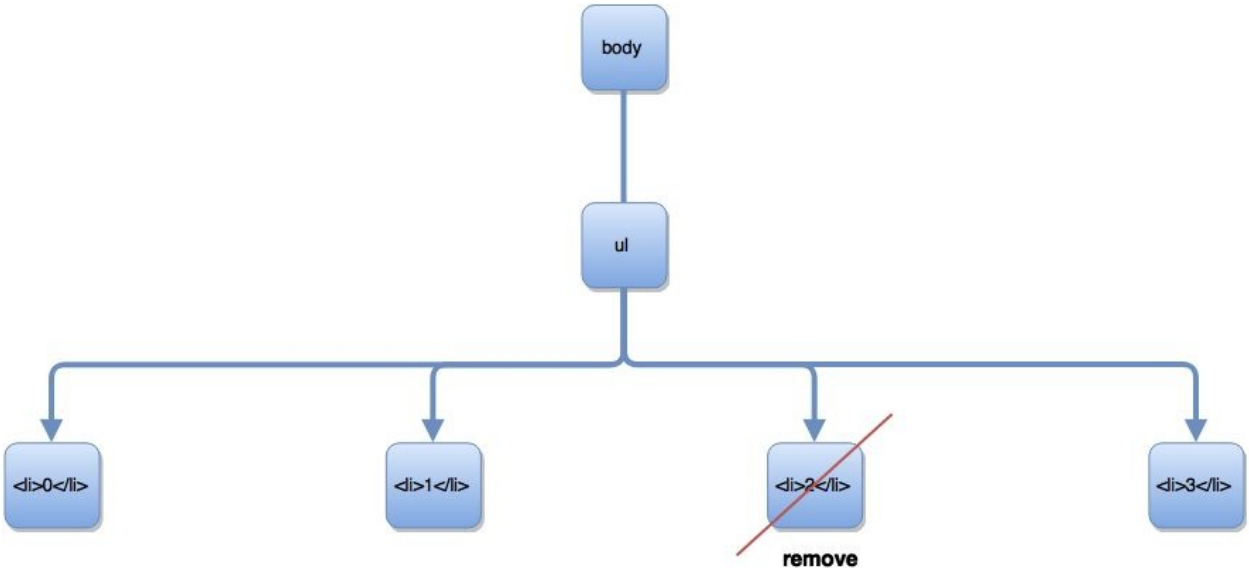
Method	Description
remove	Removes a node from its HTML tree. The node is returned and can be reused for further processing.

Table 10.1 *Methods in this Lesson*

Attribute	Description
parentNode	Contains the parent node of the current element.

Table 10.2 *Attributes in this lesson*

Example for `remove`



HTML structure (before):

```
...
<body>
  <ul>
```

```
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

JavaScript in the console

```
$$("li")[2].remove();
```

HTML structure (after):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>3</li>
  </ul>
</body>
```

10.3 Exercises

Exercise 22: 01001000010011110101010O — Part 8

Use JavaScript to make some more changes to the binary mug product page. You can again use the JavaScript console for this exercise.

Use the `remove` method to remove the `Description` header.

Use the `remove` method to remove the text `"01001000010011110101010O"` from the main header.

11 Going On a Family Trip: siblings & insertBefore

Finally, it'd be fantastic if we had two arrow buttons "*up*" and "*down*" in addition to the remove button, so users could move products within the list. We could then use the prototype to experiment first and decide whether we actually want that functionality in the final product.

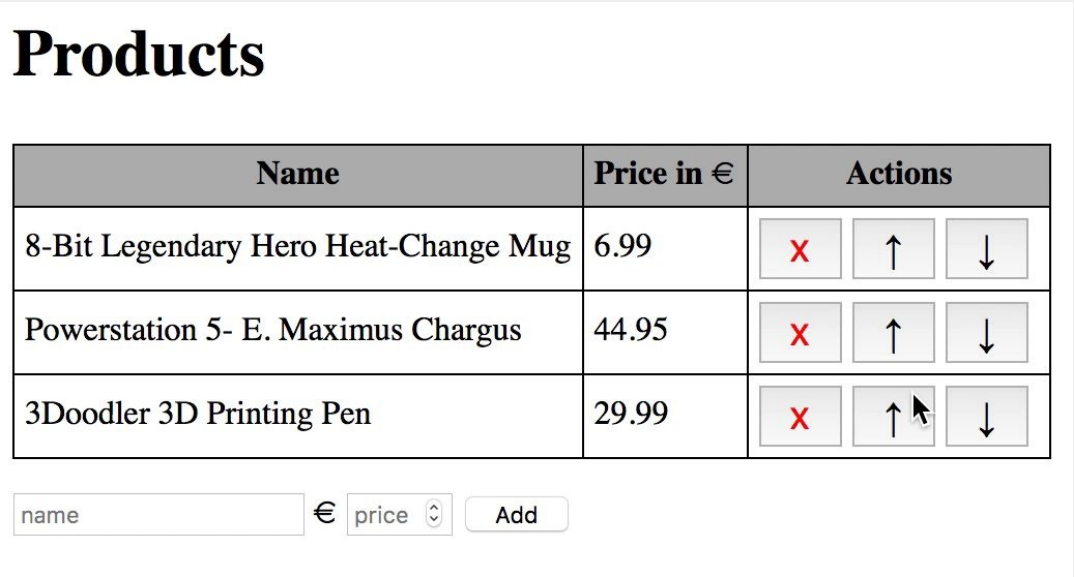


Fig. 22 *Product manager: up & down feature*

First, we'll extend the function `tdWithRemoveButton` so that it'll also add the two new buttons. Then we should probably also rename it something like `tdWithActionButtons`, since now it's creating more than just the remove button.

```
1 const tdWithActionButtons = () => {
2     const td = document.createElement("td");
3     td.appendChild(removeButton());
4     td.appendChild(moveUpButton());
```

```
    td.appendChild(moveDownButton());
    return td;
};
```

We'll create the buttons following the usual process:

```
const moveUpButton = () => {
  const button = document.createElement("button");
  button.innerText = "↑";
  button.classList.add("move_product_up");
  button.on("click", moveProductUp);
  return button;
};

const moveDownButton = () => {
  const button = document.createElement("button");
  button.innerText = "↓";
  button.classList.add("move_product_down");
  button.on("click", moveProductDown);
  return button;
};
```

Okay, we already know what you're going to say — too much redundancy caused by that evil copy & paste. And of course, you're right. But we'll deal with that a little later, because we first need to implement the functions `moveProductUp` and `moveProductDown`.

How do the actual movement functions `moveProductUp` and `moveProductDown` look now? Here again, the first thing we need to do is to grab the product row, using the existing function `productRowForAction(event.target)` which again just moves up two parent nodes. Let's just store it in a variable:

```
const currentProductRow = productRowForAction(event.target);
```

Now we can add this product row to some other spot in the HTML code. To do that, we'll need the method `insertBefore`, which inserts one element before another. Calls to that method are always made from the parent element:

```
currentProductRow.parentNode.insertBefore(/* ... */)
```

Exactly where should that element go?

In the case where we're moving a product up, we'll need to insert the element at the same level, one element before the current product row. Elements on the same level are called ***siblings***. The sibling element before the current element is then the `previousElementSibling`. In the code, this means:

```
currentProductRow.parentNode.insertBefore(
  currentProductRow,
  currentProductRow.previousElementSibling);
```

Although this looks a little complicated, it actually isn't. Just read the statement as:

> *Starting from the parent (*`currentProductRow.parentNode` *— in this case the* `tbody`*), insert the product row you want to move (*`currentProductRow`*)* ***before*** *the previous sibling element (*`currentProductRow.previousElementSibling`*).*

As a result, we wind up with the following function:

```
const moveProductUp = event => {
  const currentProductRow = productRowForAction(event.target);
  currentProductRow.parentNode.insertBefore(
    currentProductRow,
    currentProductRow.previousElementSibling);
};
```

Now we can move the product up. Hooray! Just try it out for yourself.

The function `moveProductDown` likewise is no great feat. We'll need the attribute `nextElementSibling`. Unfortunately, there's a slight problem here — the next sibling `tr` of the *3Doodler* `tr` is *Powerstation*. If we use `insertBefore` to insert 3Doodle before *Powerstation* (*before next* in fig. 23), we haven't changed anything!

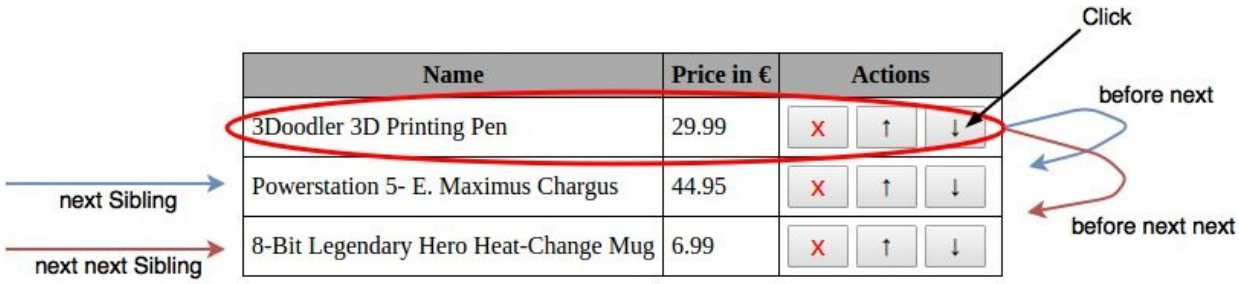


Fig. 23 *moveProductDown using* `insertBefore` *on* `.nextElementSibling.nextElementSibling`

Instead, we need to insert *3Doodler* before the `tr` after the next (*before next next* in fig. 23), i. e. *8.Bit…Mug*:

```
const moveProductDown = event => {
  const currentProductRow = productRowForAction(event.target);
  currentProductRow.parentNode.insertBefore(
    currentProductRow,
    currentProductRow.nextElementSibling.nextElementSibling);
};
```

And here's the code again in its entirety:

```
"use strict";

{
  const init = () => {
    addExistingProducts(PRODUCTS);
    $("#add_product").on("click", addProductFromInput);
  };

  const addExistingProducts = () => PRODUCTS.forEach(addProduct);

  const addProductFromInput = () =>
    addProduct({
      name: $("#new_product .name").value,
      price: $("#new_product .price").value
    });

  const addProduct = product =>
    $("#products > tbody").appendChild(
      tr([
        td(product.name), td(product.price), tdWithActionButtons()
      ])
    );

  const tdWithActionButtons = () => {
    const td = document.createElement("td");
    td.appendChild(removeButton());
    td.appendChild(moveUpButton());
    td.appendChild(moveDownButton());
    return td;
  };

  const removeButton = () => {
```

```
    const button = document.createElement("button");
    button.textContent = "x";
    button.classList.add("remove_product");
    button.on("click", removeProduct);
    return button;
  };

  const moveUpButton = () => {
    const button = document.createElement("button");
    button.textContent = "↑";
    button.classList.add("move_product_up");
    button.on("click", moveProductUp);
    return button;
  };

  const moveDownButton = () => {
    const button = document.createElement("button");
    button.textContent = "↓";
    button.classList.add("move_product_down");
    button.on("click", moveProductDown);
    return button;
  };

  const removeProduct = event =>
    productRowForAction(event.target).remove();

  const moveProductUp = event => {
    const currentProductRow = productRowForAction(event.target);
    currentProductRow.parentNode.insertBefore(
        currentProductRow,
        currentProductRow.previousElementSibling);
  };

  const moveProductDown = event => {
    const currentProductRow = productRowForAction(event.target);
    currentProductRow.parentNode.insertBefore(
        currentProductRow,
        currentProductRow.nextElementSibling.nextElementSibling);
  };

  const productRowForAction = button =>
    button.parentNode.parentNode;

  const td = text => {
    const tdNode = document.createElement("td");
    tdNode.textContent = text;
    return tdNode;
  };

  const tr = tds => {
    const trNode = document.createElement("tr");
    tds.forEach(td => trNode.appendChild(td));
    return trNode;
  };

  init();
}
```

Listing 54 *accompanying_files/12/examples/product_manager_5_move/product_manager.js*

Pretty long, no?

11.1 Back to Your Old Friend Refactoring

We should at least remove the redundancies All three button functions have been set up according to the same pattern:

```
const removeButton = () => {
  const button = document.createElement("button");
  button.textContent = "x";
  button.classList.add("remove_product");
  button.on("click", removeProduct);
  return button;
};

const moveUpButton = () => {
  const button = document.createElement("button");
  button.textContent = "↑";
  button.classList.add("move_product_up");
  button.on("click", moveProductUp);
  return button;
};

const moveDownButton = () => {
  const button = document.createElement("button");
  button.textContent = "↓";
  button.classList.add("move_product_down");
  button.on("click", moveProductDown);
  return button;
};
```

Let's define a function called `buildButton` which brings together the different functions:

```
const buildButton = (symbol, cssClass, action) => {
  const button = document.createElement("button");
  button.textContent = symbol;
  button.classList.add(cssClass);
  button.on("click", action);
  return button;
};
```

We can now use the new function to create all three buttons:

```
const removeButton = () =>
  buildButton("x", "remove_product", removeProduct);
const moveUpButton = () =>
  buildButton("↑", "move_product_up", moveProductUp);
```

```
const moveDownButton = () =>
  buildButton("↓", "move_product_down", moveProductDown);
```

Not only is our code shorter, you can also tell at a glance how the three buttons actually differ from one another.

Like our buttons, the two `move` functions aren't very different from one another. The only difference is in how they access sibling elements in order to determine direction. Let's pull out the differences and put them in a single function:

```
const moveProductUp = event => moveProduct(event, up);
const moveProductDown = event => moveProduct(event, down);

const moveProduct = (event, direction) => {
  const currentProductRow = productRowForAction(event.target);
  currentProductRow.parentNode.insertBefore(
      currentProductRow, direction(currentProductRow));
};

const down = el => el.nextElementSibling.nextElementSibling;
const up = el => el.previousElementSibling;
```

And here's an overview of our improved code:

```
"use strict";

{
  const init = () => {
    addExistingProducts(PRODUCTS);
    $("#add_product").on("click", addProductFromInput);
  };

  const addExistingProducts = () => PRODUCTS.forEach(addProduct);

  const addProductFromInput = () =>
    addProduct({
      name: $("#new_product .name").value,
      price: $("#new_product .price").value
    });

  const addProduct = product =>
    $("#products > tbody").appendChild(
      tr([
        td(product.name), td(product.price), tdWithActionButtons()
      ])
    );

  const tdWithActionButtons = () => {
    const td = document.createElement("td");
    td.appendChild(removeButton());
    td.appendChild(moveUpButton());
```

```
    td.appendChild(moveDownButton());
    return td;
  };

  const removeButton = () =>
    buildButton("x", "remove_product", removeProduct);
  const moveUpButton = () =>
    buildButton("↑", "move_product_up", moveProductUp);
  const moveDownButton = () =>
    buildButton("↓", "move_product_down", moveProductDown);

  const removeProduct = event =>
      productRowForAction(event.target).remove();
  const moveProductUp = event => moveProduct(event, up);
  const moveProductDown = event => moveProduct(event, down);

  const moveProduct = (event, direction) => {
    const currentProductRow = productRowForAction(event.target);
    currentProductRow.parentNode.insertBefore(
        currentProductRow, direction(currentProductRow));
  };

  const down = el => el.nextElementSibling.nextElementSibling;
  const up = el => el.previousElementSibling;

  const productRowForAction = button =>
    button.parentNode.parentNode;

  const buildButton = (symbol, cssClass, action) => {
    const button = document.createElement("button");
    button.textContent = symbol;
    button.classList.add(cssClass);
    button.on("click", action);
    return button;
  };

  const td = text => {
    const tdNode = document.createElement("td");
    tdNode.textContent = text;
    return tdNode;
  };

  const tr = tds => {
    const trNode = document.createElement("tr");
    tds.forEach(td => trNode.appendChild(td));
    return trNode;
  };

  init();
}
```

Listing 55

accompanying_files/12/examples/product_manager_6_move_refact/product_manager.js

11.2 Oops! We Still Need to Do a Little Better

I just noticed a small glitch. I can click on the "up" button for the first product, and I get the following result:

3Doodler 3D Printing Pen	29.99	x ↑ ↓
Name	**Price in €**	**Actions**
Powerstation 5- E. Maximus Chargus	44.95	x ↑ ↓
8-Bit Legendary Hero Heat-Change Mug	6.99	x ↑ ↓

And it also makes just as little sense for a user to be able to click on a "down" button for the last product. Do you think you could just disable those buttons?

That's true — in fact, those buttons shouldn't be active at all. A small function will solve this problem.

```
1 const disableNonFunctionalButtons = () =>
2   $$("#products > tbody > tr").forEach(tr => {
3     tr.querySelector(".move_product_up").disabled = !tr.previousElementSibling;
4     tr.querySelector(".move_product_down").disabled = !tr.nextElementSibling;
5   });
```

This really isn't anything new. The `forEach` loops through all table rows (`tr`) and puts the up/down buttons in the correct state. The only trick here is to determine what that correct state is. If a row *has no predecessor* (`!tr.previousElementSibling`), it must be the top row. Accordingly, the

button for "Up" (`tr.querySelector(".move_product_up")`) in that row must be disabled (`.disabled = true`). On the other hand, if the row does have a predecessor, `!tr.previousElementSibling` will return a value of `false`, so the `disabled` attribute will be set to `false`. The button remains (or becomes) active.

The "Down" button can be disabled or activated in exactly the same way.

Now, the function just needs to be called once. So when's the best time to call it?

Ideally, the best time to call it is always after the user has clicked a button.

Call in table

In case of the buttons `↑`, `↓` and `x`, we just add the function `disableNonFunctionalButtons` to the click event in the corresponding `buildButton` function. The second `on` in line 5 registers `disableNonFunctionalButtons` as the second event handler on the `click` event (***chaining***). Both functions are executed when the event occurs.

```
1 const buildButton = (symbol, cssClass, action) => {
2   const button = document.createElement("button");
3   button.innerText = symbol;
4   button.classList.add(cssClass);
5   button.on("click", action).on("click", disableNonFunctionalButtons);
6   return button;
7 };
```

Every click on a button which was created using `buildButton` causes `disableNonFunctionalButtons` to be called. This ensures that all buttons will be moved into the correct state whenever a user clicks on any button for any product.

Call after a new product is added

So that we can also ensure our buttons will have the correct state after a new product is added, we should also add the function at the end of `addProduct`:

```
const addProduct = product => {
  $("#products > tbody").appendChild(
    tr([
      td(product.name), td(product.price), tdWithActionButtons()
    ])
  );
  disableNonFunctionalButtons();
};
```

As always, here's the code in its entirety:

```
"use strict";

{
  const init = () => {
    addExistingProducts(PRODUCTS);
    $("#add_product").on("click", addProductFromInput);
  };

  const addExistingProducts = () => PRODUCTS.forEach(addProduct);

  const addProductFromInput = () =>
    addProduct({
      name: $("#new_product .name").value,
      price: $("#new_product .price").value
    });

  const addProduct = product => {
    $("#products > tbody").appendChild(
      tr([
        td(product.name), td(product.price), tdWithActionButtons()
      ])
    );
    disableNonFunctionalButtons();
  };

  const tdWithActionButtons = () => {
    const td = document.createElement("td");
    td.appendChild(removeButton());
    td.appendChild(moveUpButton());
    td.appendChild(moveDownButton());
    return td;
  };

  const removeButton = () =>
    buildButton("x", "remove_product", removeProduct);
  const moveUpButton = () =>
    buildButton("↑", "move_product_up", moveProductUp);
  const moveDownButton = () =>
    buildButton("↓", "move_product_down", moveProductDown);

```

```
41    const removeProduct = event =>
42      productRowForAction(event.target).remove();
43    const moveProductUp = event => moveProduct(event, up);
44    const moveProductDown = event => moveProduct(event, down);
45
46    const moveProduct = (event, direction) => {
47      const currentProductRow = productRowForAction(event.target);
48      currentProductRow.parentNode.insertBefore(
49          currentProductRow, direction(currentProductRow));
50    };
51
52    const down = el => el.nextElementSibling.nextElementSibling;
53    const up = el => el.previousElementSibling;
54
55    const productRowForAction = button =>
56      button.parentNode.parentNode;
57
58    const buildButton = (symbol, cssClass, action) => {
59      const button = document.createElement("button");
60      button.textContent = symbol;
61      button.classList.add(cssClass);
62      button.on("click", action).on("click", disableNonFunctionalButtons);
63      return button;
64    };
65
66    const disableNonFunctionalButtons = () =>
67      $$("#products > tbody > tr").forEach(tr => {
68        tr.querySelector(".move_product_up").disabled = !tr.previousElementSibling;
69        tr.querySelector(".move_product_down").disabled = !tr.nextElementSibling;
70      });
71
72    const td = text => {
73      const tdNode = document.createElement("td");
74      tdNode.textContent = text;
75      return tdNode;
76    };
77
78    const tr = tds => {
79      const trNode = document.createElement("tr");
80      tds.forEach(td => trNode.appendChild(td));
81      return trNode;
82    };
83
84    init();
85  }
```

Listing 56

accompanying_files/12/examples/product_manager_7_move_disabled_buttons/product_manager.

11.3 Summary

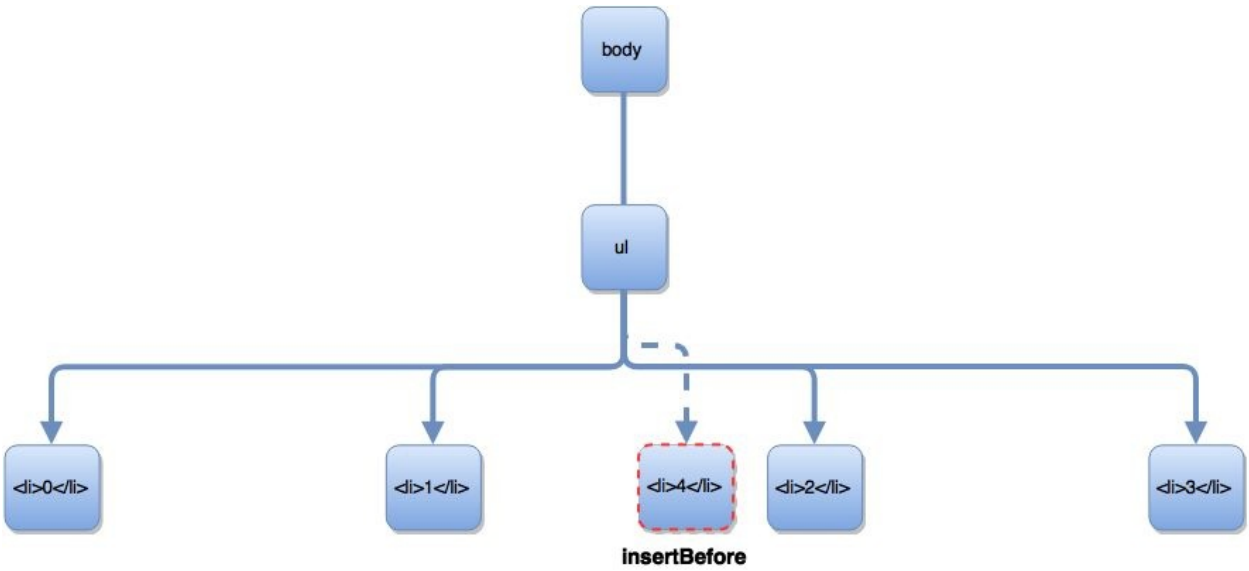
Method	Description
insertBefore	Adds the node passed to it as the first argument before the node passed to it as the second argument. The insertion is based on the parent node on which the method was called.

Table 11.1 *Methods in this Lesson*

Attribute	Description
previousElementSibling	Contains the previous sibling element
nextElementSibling	Contains the following sibling element

Table 11.2 *Attributes in this lesson*

11.3.1 Example for `insertBefore`



HTML structure (before):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

JavaScript in the console

```
const li = document.createElement("li");
li.textContent = "4";
$("ul").insertBefore($$("ul > li")[2], li);
```

HTML structure (after):

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>4</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

11.3.2 DOM traversal

A number of different attributes are required in order to move from one element to another within the DOM (***traversal***). This overview shows the most essential attributes, based on listing 57.

All attributes listed in table 11.3 and fig. 24 are *read-only*. Although you can use these attributes for DOM traversal, you can't use them to make changes to the tree structure.

Attribute	Description

parentNode	The parent node
children	All child elements
childElementCount	Number of child elements
firstElementChild	The first child element
lastElementChild	The last child element
previousElementSibling	The previous sibling element
nextElementSibling	The next sibling element

Table 11.3 *This table lists the most essential methods for DOM* ***traversal*** *(moving within the DOM)*

Examples

```
...
<body>
  <ul>
    <li>0</li>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>
</body>
```

Listing 57 *HTML example of a simple unordered list*

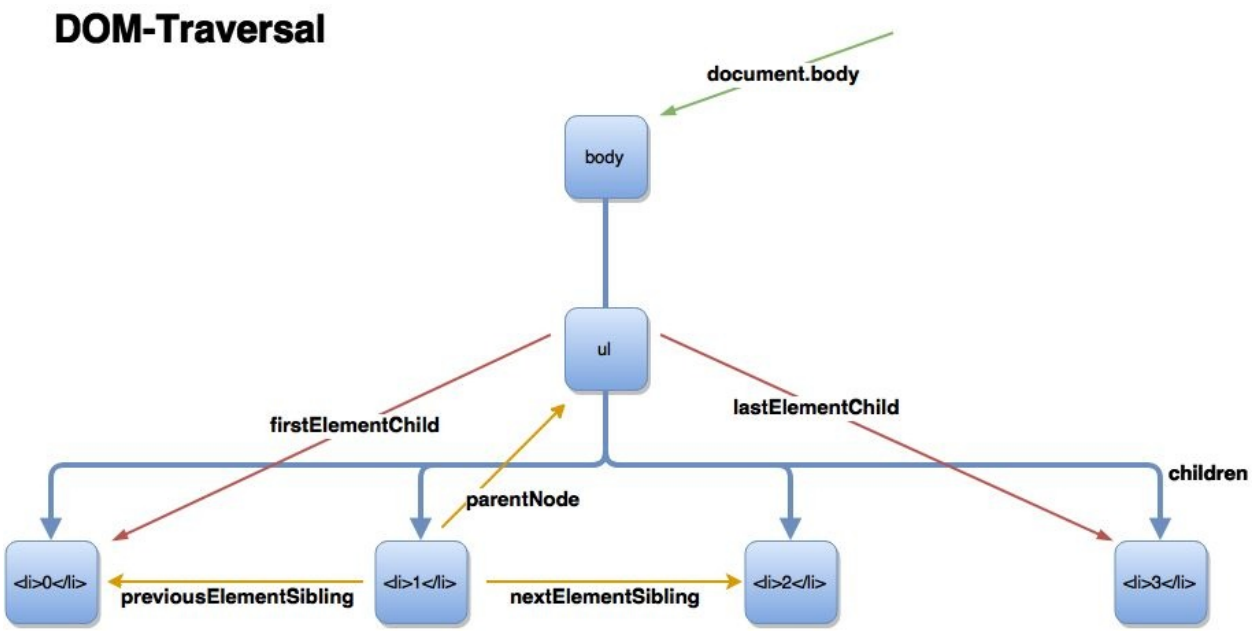


Fig. 24 *DOM traversal attributes for listing 57*

```
document.body                   // => <body>...</body>
document.body.children          // => [<ul>...</ul>]
document.body.childElementCount // => 0

$('ul')                   // => <ul>...</ul>
$('ul').parentNode        // => <body>...</body>
$('ul').firstElementChild // => <li>0</li>
$('ul').lastElementChild  // => <li>3</li>
$('ul').childElementCount // => 4
$('ul').children // => [<li>0</li>, ..., <li>3</li>]

$$('li')[1]                         // => <li>1</li>
$$('li')[1].parentNode              // => <ul>...</ul>
$$('li')[1].previousElementSibling // => <li>0</li>
$$('li')[1].nextElementSibling     // => <li>2</li>
```

Reference

The techniques for manipulating elements in the DOM described in the last few lessons became known primarily through the book *DOM Scripting* by *Jeremy Keith* (Keith 2005). The book appeared in 2005 and was thus able to ride the crest of the wave of Ajax hype. It was a very impressive demonstration of how modern applications could be developed using clean, standards-compliant code — which

was a novelty at that time.

11.4 Exercises

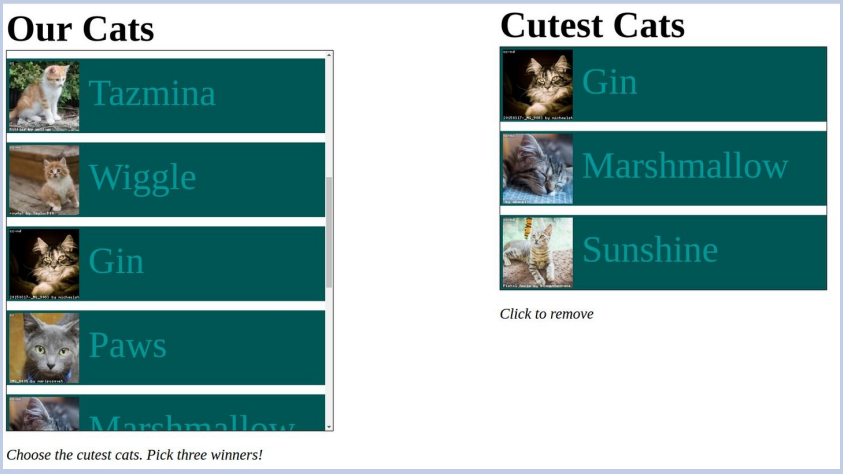
Exercise 23: 01001000010011110101010O — Part 9

Use JavaScript to make some more changes to the binary mug product page.

Add one more list item to the `product specifications` list: *Capacity: 11 oz*.

This time however, don't add the list item to the end of the list, but before the entry *Materials: Ceramic*.

Exercise 24: Cuteycat



The feline friends association *Cuteycat* has a job for you. The association wants to run a contest on its website. The contest page shows a number of cats, and visitors to the page are supposed to select the three cutest.

Use the HTML and CSS provided in the accompanying material. As soon as a visitor clicks on a cat, it should also appear in the box at the right of the page (*Cutest Cats*). However, it mustn't also disappear from the overall list of cats (*Our Cats*)!

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <title>Cutycat</title>
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="cuteycat.js" defer="defer"></script>
  <link rel="stylesheet"  type="text/css" href="cuteycat.css" />
</head>

<body>

  <section id="candidates_section">
    <h1>Our Cats</h1>
    <ul id="candidates" class="kittenlist">
      <li><img src="../../kittens/1.jpg" alt="cat 1" /><span>Chance</span></li>
      <li><img src="../../kittens/2.jpg" alt="cat 2" /><span>MeowMix</span></li>
      <li><img src="../../kittens/3.jpg" alt="cat 3" /><span>Stripes</span></li>
      <li><img src="../../kittens/4.jpg" alt="cat 4" /><span>Schrodinger</span></li>
      <li><img src="../../kittens/5.jpg" alt="cat 5" /><span>Waffles</span></li>
      <li><img src="../../kittens/6.jpg" alt="cat 6" /><span>Tazmina</span></li>
      <li><img src="../../kittens/7.jpg" alt="cat 7" /><span>Wiggle</span></li>
      <li><img src="../../kittens/8.jpg" alt="cat 8" /><span>Gin</span></li>
      <li><img src="../../kittens/9.jpg" alt="cat 9" /><span>Paws</span></li>
      <li><img src="../../kittens/10.jpg" alt="cat 10" /><span>Marshmallow</span></li
      <li><img src="../../kittens/11.jpg" alt="cat 11" /><span>Caesar</span></li>
      <li><img src="../../kittens/12.jpg" alt="cat 12" /><span>Trogdor</span></li>
      <li><img src="../../kittens/13.jpg" alt="cat 13" /><span>Sourpuss</span></li>
      <li><img src="../../kittens/14.jpg" alt="cat 14" /><span>Stitch</span></li>
      <li><img src="../../kittens/15.jpg" alt="cat 15" /><span>Sunshine</span></li>
    </ul>
    <p>Choose the cutest cats. Pick three winners!</p>
  </section>

  <section id="cutest_section">
    <h1>Cutest Cats</h1>
    <ul id="cutest" class="kittenlist">
    </ul>
    <p>Click to remove</p>
  </section>

</body>
</html>
```

accompanying_files/12/solutions/cute_kittens/cuteycat.html

```
h1 {
  font-size: 5em;
  margin: 0;
  padding: 0;
}

p {
```

```
  font-size: 2em;
  font-style: italic;
}

.kittenlist {
  border: 1px solid black;
  width: 700px;
  margin: 0;
  padding: 0;
  height: 80vh;
  overflow: auto;
}

.kittenlist li {
  list-style: none;
  font-size: 5em;
  background-color: #055;
  padding: 5px;
  margin-bottom: 20px;
  height: 150px;
}

.kittenlist li:hover {
  background-color: #755;
  outline: 5px solid #700;
}

.kittenlist li img {
  margin-right: 20px
}

.kittenlist li span {
  display: inline;
  top: -0.7em;
  position: relative;
  color: #099;
}

#cutest {
  height: 520px;
  overflow: hidden;
}

#cutest_section {
  position: absolute;
  top: 0;
  left: 50vw;
}
```

accompanying_files/12/solutions/cute_kittens/cuteycat.css

Make sure users can't select the same cat twice.

There may be no more than 3 cutest cats.

User can click on a cat already in the winner's box (*Cutest Cats*) in order to remove it.

Notes

- Try to solve the exercise without using `.innerHTML`!
- You can access the URL http://loremflickr.com/150/150/cute,kitten/all to generate random cat images.
- In addition, the page http://fantasynamegenerators.com/cat-names.php produces random cat names.
- Use `.clone(true)` on `li` elements to prevent cats from being removed from *Our Cats*.
- Register the event handler on the `li` elements. However, if a user doesn't click on the `li` directly but on an image or text (name of the cat) instead, `event.target` will indicate the element that was clicked on. However, you can always access the `li` by using `event.currentTarget`. `currentTarget` always contains the element on which the handler was actually registered.

The End

PS: If you liked the course, we would be very happy about a book review at Amazon. Five stars would be great - if you didn't like it, you don't have to write anything ;) No seriously, write your honest opinion. The reviews help us to continue this work and to develop new, ever better courses.

12 Appendix A: Reference

The reference compiles together all the browser APIs mentioned in this course. We've intentionally left out properties and methods which are rarely used or which are too complex.

Use this reference if you need to look anything up while you're doing the exercises. We also hope this reference will come in handy for you later in your day-to-day development work. However, as soon as you have a fairly good idea of the names of the basic methods, attributes and interfaces, it would probably make more sense to switch to an online reference such as developer.mozilla.org or www.webplatform.org. If you want to know the browser versions which support some given functionality, we recommend you use caniuse.com.

My Tip

Don't just use this reference when you need to look something up. Leaf through the reference whenever you get a chance — like a Sunday paper or today's Twitter feed.

You shouldn't always read it focused and intent on each word, but also when you're just relaxing. Try to imagine the exciting things you'll program using the attributes and methods you come across. And when you get the chance, read through the reference once again.

This way you'll commit to memory the most important items in this class virtually "in passing", and you'll already have seen everything once. Then if at some later time you need say, a specific method, it'll occur to you immediately that something which does exactly what you

need actually does exist.

Reading through references really doesn't have to be something that's "dry". When, at the age of 12, I read my first programming book (the Basic manual for the Commodore C64), I virtually swallowed up the reference section. With every new function I came across, I dreamed of the fantastic things I'd program with it.

12.1 Elements

Note: The methods available in the DOM come from different interfaces/APIs. The APIs from which methods originate are listed along with the method in the reference tables below. This is purely for reasons of documentation. You can look up the corresponding API to see exactly what methods it has available.

12.1.1 Element vs. Node

The DOM differentiates between nodes in general and specific node types such as element nodes, text nodes, comment nodes, etc. Normally, only element nodes are really relevant in practice. As a result, our course concentrates on those nodes. If you're interested in other node types which come up in use less often, you'll find a listing along with explanations under nodeType in the MDN.

Since an element node (*Element*) is a subclass of a node (*Node*), an element node will provide you, in addition to element properties (e. g. `children`), all node properties (e. g. `parentNode`) as well. Normally, you can just ignore the difference between those two node types.

12.1.2 Document Node

The `document` is also a special case of a node. It represents a complete HTML document and provides a few additional methods such as `createElement`.

The methods `querySelector` and `querySelectorAll` can be applied

globally, over the `document`, and also **locally**, on individual elements. Global means that the selector searches through the entire document. Local means that only the descendants (i. e. children, grandchildren, etc.) of the given element node are included in the search.

12.1.3 DOM Selection

Method	Description	Interface/API
querySelector	Returns the **first** element which matches the specified selector. Only searches through the elements within the HTML subtree of the root element on which `querySelector` was called.	Element
querySelectorAll	Returns **all** elements which match the specified selector. Only searches through the elements within the HTML subtree of the root element on which `querySelectorAll` was called.	Element
document.querySelector *(aka $)*	Returns the **first** element which matches the specified selector. Searches through the entire HTML document.	Document
document.querySelectorAll *(aka $$)*	Returns **all** elements which match the specified selector. Searches through the entire HTML document.	Document

Table 12.1 *This table shows the most essential methods for* ***selecting*** *DOM elements. You'll find additional DOM selection methods in the MDN in the descriptions of the corresponding APIs: Element& Document.*

Attribute	Description	Interface/API
document.head	Returns the `head` element.	Document
document.body	Returns the `body` element.	Document

Table 12.2 *This table shows the most essential attributes for* ***selecting*** *DOM elements. You'll find additional attributes in the MDN in the descriptions for the Document API.*

CSS Selectors

Name	Example	Description
Universal selector	`*`	Selects any element. You shouldn't use this selector without limiting your selection further, since selecting **all** the elements on a web page can take up a lot of computing time.
Type selector	`h1`	Selects all elements of a specified type.
ID selector	`#some-id`	Selects exactly **one** element with the specified `id`.
Class selector	`.some-class`	Selects all elements which were assigned the specified class. Remember that an element can have more than one class.
-	`h1, p, .go`	The comma itself isn't a selector, but makes it possible to combine multiple selectors. This example would select all `h1` elements, all `p` elements and all elements with the class `go`.

Table 12.3 *Basic CSS3 selectors*

Selector	Example based on `<img`	Description

	src="funny_cat.png" ... />	
[*attr*]	`[src]`	Selects elements with the specified attribute. The actual value of the attribute plays no part in the selection (***attribute presence***).
[*attr=val*]	`[src='funny_cat.png']`	Selects elements which contain the specified attribute (*src*) with exactly the specified value (*funny_cat.png*) (***exact attribute match***).
[*attr*=val*]	`[src*=cat]`	Selects elements in which the value of the specified attribute (*src*) contains the specified value (*cat*) as a substring (***substring attribute match***).
[*attr^=val*]	`[src^=funny]`	Selects elements in which the value of the specified attribute (*src*) starts with the specified value (*funny*).
[*attr*$=val*]	`[src$=png]`	Selects elements in which the value of the specified attribute (*src*) ends with the specified value (*png*).

Table 12.4 *CSS attribute selectors. All examples select *

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <title>Selector Testing</title>
</head>

<body>
  <ul id="ul1">
    <li id="li11"><span>11</span></li>
    <li id="li12">12</li>
    <li id="li13">13</li>
    <li id="li14">14</li>
  </ul>
  <ul id="ul2">
    <li id="li21"><span>21</span></li>
    <li id="li22"><strong><span>22</span></strong></li>
    <li id="li23">23</li>
  </ul>
```

```
</body>

</html>
```

Listing 60 *accompanying_files/02/examples/selector_testing.html*

Name	Example	Description
Descendant combinator (blank space)	`li span`	Selects elements (in this case `span`) which are descendants of a different specified element (in this case `li`) — regardless of how many levels separate the specified element and the descendant. **Result:** `[<span>11</span>, <span>21</span>, <span>22</span>]`
Adjacent sibling combinator (+)	`li#li11 + li`	You can use this selector to select the element which **immediately follows** the specified element (in this case `li#li11`) in the HTML source text and is on the same level (sibling element). If the element which follows the specified element in the document doesn't match the element specified after the + symbol, the selection returns nothing. **Result:** `[<li id="li12">12</li>]`
General sibling combinator (~)	`li#li12 ~ li`	You can use this selector to select all elements which follow the specified element (in this case `li#li12`) in the document and are on the same level (sibling element). This selector **doesn't just select the element which immediately follows** the specified element, as does the *next sibling combinator*. **Result:** `[<li id="li13">13</li>, <li id="li14">14</li>]`
Child combinator	`li > span`	Child selectors are one variant of descendant selectors. You can use child selectors to select

(>)		elements which are **direct** "children" of higher-level parent elements. The child selector in the example only selects `span` elements which appear directly within an `li` element. `<span>22</span>` appears within a `strong` element and is therefore not selected. **Result:** `[<span>11</span>, <span>21</span>]`

Table 12.5 *Combinator selectors — results based on listing 60*

Name	Example	Description
:first-child	ul li:first-child	This pseudo class allows you to select the first child element in a higher-level container (in this case `ul`), where the child element has the specified tag type (in this case `li`). If the first child element doesn't have the specified tag type, the selection returns nothing
:nth-child(x)	ul li:nth-child(5)	This pseudo class selects the xth child element (in the example, this would be the fifth element which appears within `ul`, if the element has a tag type of `li`.
:last-child	ul li:last-child	Just like `:first-child`, you use this selector to select the last child element, assuming it has the specified tag type.
:only-child	#article p:only-child	This class selects an element only if it's the only child of the specified parent element.
:empty	div:empty	You can use this pseudo class to select only those elements that have no child elements. Note that the pseudo class also considers content in the form of text to be a child element.

:not(selector)	:not(span)	Negation. Selects those elements which don't match the selector specified in parentheses. In the example, everything except `span` elements would be selected.

Table 12.6 *Pseudo Classes*

12.1.4 DOM Manipulation

Method	Description	API
appendChild	Appends the node passed to it (e. g. an element) to the parent node on which the method was called. The node is appended as the last child of the parent node. If the appended node is already part of the HTML document, it's also removed from its original position.	Node
insertBefore	Adds the node passed to it as the first argument before the node passed to it as the second argument. The insertion is based on the parent node on which the method was called.	Node
remove	Removes a node from its HTML tree. The node is returned and can be reused for further processing.	ChildNode

Table 12.7 *This table shows the most essential methods for* ***manipulating*** *DOM elements. You'll find additional DOM manipulation methods in the MDN in the descriptions of the corresponding APIs: Node & ChildNode.*

12.1.5 DOM Traversal

A variety of attributes are required in order to move from one element to another within the DOM (***traversal***); this section of the reference shows the most essential.

All attributes listed in table 12.8 and fig. 25 are *read-only*. Although you can use these attributes for DOM traversal, you can't use them to make changes to the tree structure.

Attribute	Description	API
parentNode	The parent node	Node
children	All child elements	parentNode
childElementCount	Number of child elements	parentNode
firstElementChild	The first child element	parentNode
lastElementChild	The last child element	parentNode
previousElementSibling	The previous sibling element	NonDocumentTypeChildNode
nextElementSibling	The next sibling element	NonDocumentTypeChildNode

Table 12.8 *This table lists the most essential attributes for DOM* ***traversal*** *(moving within the DOM) You'll find additional DOM traversal attributes in the MDN in the descriptions of the corresponding APIs:*

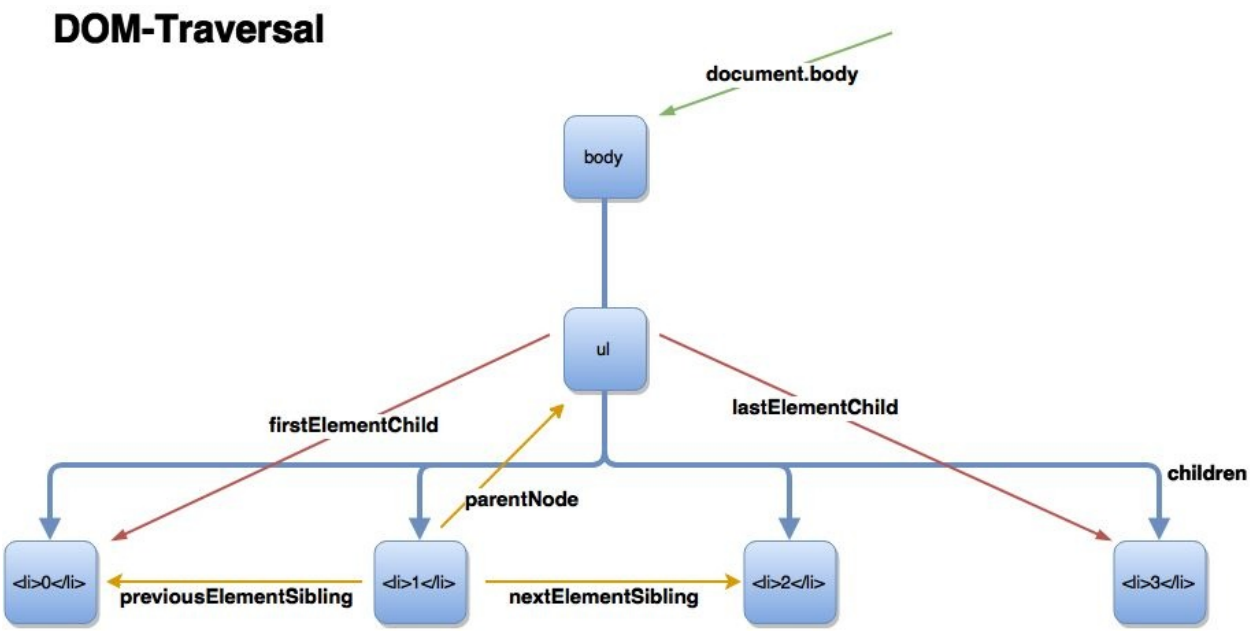


Fig. 25 *DOM traversal attributes*

12.1.6 DOM Creation

Method	Description	API
cloneNode	Copies a node and returns the copy. `cloneNode(true)` also copies all child nodes.	Node
document.createElement	Creates an element of the specified type.	Document

Table 12.9 *This table shows the most essential methods for* ***creating*** *DOM elements. You'll find additional DOM creation methods in the MDN in the descriptions of the corresponding APIs:*

Attribute	Description	API
innerHTML	Returns the HTML inside of an element — that value can also be modified.	Element
textContent	Returns the text (without HTML tags) which appears in an HTML element (in addition, the value of textContent can be modified).	Node

Table 12.10 *This table shows the most essential attributes for* ***creating*** *DOM elements. You'll find additional DOM creation attributes in the MDN in the descriptions of the corresponding APIs.*

12.2 Attributes

Attribute name	Elements, for which the attribute is often used	Description
alt	`area, img, input`	Alternative text, e. g. if an image can't be displayed.
checked	`input`	Indicates whether a radio button has been selected or a checkbox was ticked.
cols	textarea	Number of columns
disabled	`button, input, option, select, textarea`	Indicates whether an element is deactivated.
href	`a, link`	URL of a linked resource
max	`progress, input`	Maximum value — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
maxlength	`input, textarea`	Maximum number of characters allowed in an element
min	`input`	Minimum value — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
multiple	`input, select`	When `true`, allows multiple input values for `input type="email"` or `input type="file"`. In case of `select`, multiple options can be selected using **`Ctrl/cmd`**.
name	`button,`	Name of the element. This attribute is

	`form`, `fieldset`, `input`, `select`, `textarea`	mainly used by server-side applications for purposes of identification.
placeholder	`input`, `textarea`	Preset text which indicates to the user what is expected in terms of input content.
readonly	`input`, `textarea`	Indicates whether an element can be modified.
rel	`a`, `link`	Defines the relationship of the target object to the link object. Can also be used for internal relationship types, e. g. internal links for a JS image gallery.
required	`input`, `select`, `textarea`	Indicates whether an element is a mandatory field.
rows	`textarea`	Number of rows
selected	`option`	Indicates whether an `option` has been selected.
size	`select`	Number of options visible in a `select` box
src	`img`	URL of the image file
start	`ol`	Defines the first number of a list if the list should start with a value other than 1.
step	`input`	Increment — in case of `input` elements, this attribute only makes sense for `type="number"` and `type="range"`.
tabindex	Available on all elements.	Overwrites the conventional order of (form) elements when controlled using the Tab key.
title	Available on all elements.	Small text box (tooltip) displayed when hovering the mouse over the element.
type	`button`, `input`	Type of the element

value	`button`, `option`, `input`, `progress`	Value of the element

Table 12.11 *A selection of attributes which in practice are usually retrieved with JS. You'll find a complete list of all attributes in the MDN attribute reference.*

12.3 CSS

Method	Example	Description
add	$("#bar").classList.add("foo")	Adds a class. In the example, the element with the `id`*bar* gets the class *foo*.
remove	$("#bar").classList.remove("foo")	Removes a class. In the example, the class *foo* is removed from the element with the `id`*bar*.
item	$("#bar").classList.item(2)	Returns a class from the classList according to the index argument passed to it. In the example, the method returns the third class name of the element with the `id`*bar*.
toggle	$("#bar").classList.toggle("foo")	Removes or adds a class. In the example, the method adds the class *foo* to the element with the `id`*bar* if the element doesn't yet have this class. If the element does have this class, the method removes it.
contains	$("#bar").classList.contains("foo")	Returns `true` if the specified class appears in classList.

Table 12.12 *classList interface methods. You'll find a detailed reference of these methods in theMDN*

Attribute	Example	Description
style	$("#bar").style	Contains the CSS styles of the element as a *CSSStyleDeclaration* (can be modified)
classList	$("#bar").classList	Contains the list of CSS classes as a *DOMTokenList* (can be modified).

Table 12.13 *HTMLElement interface attributes for accessing CSS. You'll find a detailed reference of these methods in the MDN*

12.4 Events

Event name	is triggered when…	e. g. available on
blur	An element loses focus.	input, textarea
change	A user makes a change to an element, e. g. by selecting an option within a `select` box or by clicking a checkbox. Unlike an `input` event, this event (in the case of an `input` element or a `textarea`) only "fires" after it again loses focus (i. e. the input is completed).	input, select, textarea
click	A user clicks on an element (and releases the button). The `click` event represents the complete process, i.e. pressing and releasing. In cases where you want to react only to a user pressing the mouse key down, use `mousedown`; if you want to react just to a user releasing the mouse key, use `mouseup`.	button, img, body, p, div
dblclick	A user double-clicks on an element.	button, img, body
focus	An element gets focus.	input, textarea
hashchange	The fragment identifier of a URL changes (the fragment identifier is the part of the URL which follows the # symbol and includes the symbol itself).	window

input	The content of an `input` or `textarea` element changes. Unlike `change`, this event fires immediately each time a user inputs text. You'll find a good comparison of input and change at http://jsfiddle.net/AtvtZ/.	input, textarea
keydown	The user presses a key.	input, textarea
keypress	The user holds a key down.	input, textarea
keyup	The user releases a key.	input, textarea
mouseenter	The user moves the mouse cursor over an element (more precisely, the mouse cursor enters the area of the element).	button, img, body, p, div
mouseleave	The user moves the mouse cursor out of an element.	button, img, body, p, div
mousemove	The mouse cursor moves over an element.	button, img, body, p, div
reset	A form is reset.	form
resize	The user changes the size of the browser window.	window
scroll	The user scrolls within an element (or the browser window).	window, div
select	The user selects text.	input, textarea, p
submit	A form is submitted.	form

transitionend	A CSS transition completes.	img, div, p
visibilitychange	The content of an element becomes visible or becomes hidden.	img, div, p
wheel	The user turns a wheel. Normally this is the mouse wheel, but can also be a trackball or touchpad.	window, div

Table 12.14 *A selection of browser events required in practice. You'll find a complete list of these events in the MDN Event reference.*

13 Appendix B: Sources & References

13.1 APA Style

We use so-called ***APA Style*** to list references, sources and related literature. APA Style is one system, among others, used to indicate references. The system was developed by the *American Psychological Association* (APA).

You'll find more information on APA Style in Wikipedia or in the Online Writing Lab. An APA Style reference usually consists of last name and year. Here're two examples:

> *Another indication of the meaningfulness of an identifier is its length. Variables which consist of only a single character are usually problematic (Kellerwessel 2002).*
>
> *Douglas Crockford (2008) even calls == the "evil twin".*

You can then look up the references listed in the following list of sources.

13.2 Sources

Crockford D. (2001). JavaScript: The World's Most Misunderstood Programming Language. Douglas Crockford's Wrrrld Wide Web (private Website). Seen on March 19, 2015 athttp://javascript.crockford.com/javascript.html

Crockford D. (May 2008). JavaScript: The Good Parts: Working with the Shallow Grain of JavaScript. O'Reilly **Ecma International (June 2015).** ECMAScript 2015 Language Specification. Standard ECMA-262, 6th Edition / June 2015

Ecma International (June 2011). ECMAScript Language Specification. Standard ECMA-262, 5.1 Edition / June 2011

Fowler M., Beck K., Brant J., Opdyke W. et al. (1999). Refactoring: Improving the Design of Existing Code, Amsterdam: Addison-Wesley Longman **Keith, J. (2005)**. DOM Scripting: Web Design with JavaScript and the Document Object Model, 1st Edition, Friends of Ed **Martin R. C. (2002).** Agile Software Development. Principles, Patterns, and Practices. Prentice Hall **Martin R. C. (2008).** Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall **Nicholus R. (2015).** Beyond jQuery. Beta Book 2015-09-25. Lean Pub **Opdyke W. F., Johnson R. E. (September 1990).** Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA). ACM

Osmani A. (2015). Learning JavaScript Design Patterns. O'Reilly **Rauschmayer A. (April 2016).** Setting up ES6. Leanpub **Venners B. (March 2003).** Orthogonality and the DRY Principle - A Conversation with

Andy Hunt and Dave Thomas, Part II. Interview in a web article. Seen on April 12, 2016 atwww.artima.com/intv/dry.html

Solutions (Exercises)

Exercise 1: Almost Famous Quotes

1. ```
   document.querySelector('h1').innerHTML = 'Almost Famous Quotes';
   ```

2. ```
   document.querySelector('blockquote').innerHTML = '"<p>I have always wished for my c
   ```

Exercise 2: 01001000010011110101010O0 — Part 1

1. document.querySelector('h1')
2. document.querySelector('#buy_form')
3. document.querySelector('#product_img')

Exercise 3: 01001000010011110101010O0 — Part 2

1. document.querySelectorAll('li')
2. document.querySelectorAll('h2')
3. document.querySelectorAll('.special')
4. document.querySelectorAll('li.keyword')
5. document.querySelectorAll('span.special')
6. document.querySelectorAll('.i.b')

Exercise 4: Agents and Attorneys

1. ```
   1 const headings = Array.from(document.querySelectorAll('h1'));
   2 headings.forEach(h => h.innerHTML = 'Lorem ipsum');
   ```
   ```

2. ```
   const text = Array.from(document.querySelectorAll('p'));
   text.forEach(p => p.innerHTML = 'Lorem ipsum dolor sit amet, consectetuer adipiscir
   ```

3. ```
   const menuItems = Array.from(document.querySelectorAll('#main_menu a'));
   menuItems.forEach(a => a.innerHTML = 'Lorem');
   ```

Exercise 5: 01001000010011110101010100 — Part 3

1. document.querySelector('ul#product_specification li')
2. document.querySelector('h1.article span')
3. document.querySelectorAll('p .keyword')
4. document.querySelectorAll('ul li')

Exercise 6: 01001000010011110101010100 — Part 4

1. $$('img[src$=jpg]')
2. $$('form input[type=button]')
3. $$('.model[data-model*=V7]')
4. $$('img:not(.float_left)')
5. $$('li:nth-child(2)') or $$('li:nth-type-of-child(2)')
6. $$('h2 + ul')

Exercise 7: 01001000010011110101010100 — Part 5

1.

```
Array.from($$('p')).forEach(p => p.classList.add('gray'));
```

2.

```
Array.from($$('p:not(.buy_info_text)')).forEach(p => p.classList.add('gray'));
```

3.

```
Array.from($$('li:not([class])')).forEach(li =>li.classList.add('gray'));
```

Exercise 8: Length Matters???

```
"use strict";

const articleSizeCssClass = (contentLength) => {
  if (contentLength <= 3000) return "coffe_break_article";
  if (contentLength <= 9000) return "normal_length_article";
  return "lone_weekend_article";
};

const contentLength = () => $("#content").innerHTML.length;

const $ = document.querySelector.bind(document);

$("h1").classList.add(articleSizeCssClass(contentLength()));
```

accompanying_files/03/solutions/article_length/article_length.js

Exercise 9: Please Click Me

```
"use strict";

{
  const init = () =>
    $("#click_me").addEventListener("click", handleButtonClick);

  const handleButtonClick = event =>
    changeButtonTextTo(event.target,
        event.altKey
        ? "Cool, you found an easteregg!"
        : "Hey I like it when you click me!");

  const changeButtonTextTo = (button, text) => button.innerHTML = text;

  const $ = document.querySelector.bind(document);

  init();
}
```

accompanying_files/05/solutions/click_me.js

Exercise 10: Newsboard Still Needs Work

1.

```
"use strict";

{
  const init = () => {
    showMessageByNumber(currentMessageNumber);
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
    $("[title=first]").addEventListener("click", firstMessage);
    $("[title=last]").addEventListener("click", lastMessage);
  };

  const nextMessage = e => {
    showMessageByNumber(currentMessageNumber += 1);
    e.preventDefault();
  };
  const prevMessage = e => {
    showMessageByNumber(currentMessageNumber -= 1);
    e.preventDefault();
  };

  const firstMessage = e => {
    showMessageByNumber(currentMessageNumber = 1);
    e.preventDefault();
  };

  const lastMessage = e => {
    showMessageByNumber(currentMessageNumber = messages.length);
    e.preventDefault();
  };

  const showMessageByNumber = messageNumber =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const $ = document.querySelector.bind(document);

  const messages = [
    `<h1>Tutoren streiken!!!</h1>
    <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
    <p>Überall dieselbe alte Leier. Das Layout ist fertig, der Text lässt auf sich wart
    <p class="newsboard_footer">am 25.09.2015 von K. Einer</p>`,

    `<h1>Wahnsinn!</h1>
    <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
    <p>Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark. "Fix, Schwyz!
    <p class="newsboard_footer">am 13.08.2015 von Dr. B. Lödmann</p>`,

    `<h1>Prokrastination?</h1>
    <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab’s noch nicht erledigt; ich ha
    <p>Denn esse est percipi - Sein ist wahrgenommen werden. Und weil Sie nun schon die
    <p>Sehen Sie, Webstandards sind das Regelwerk, auf dem Webseiten aufbauen. So gibt
    <p class="newsboard_footer">am 02.06.2015 von A. Meisenbär</p>`
  ];

  let currentMessageNumber = 1;

  init();
}
```

accompanying_files/05/solutions/newsboard/newsboard_1a.js

1. (refactored)

```
"use strict";

```

```
{
  const init = () => {
    showMessageByNumber(currentMessageNumber);
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
    $("[title=first]").addEventListener("click", firstMessage);
    $("[title=last]").addEventListener("click", lastMessage);
  };

  const nextMessage = e => showMessageForEvent(e, currentMessageNumber += 1);
  const prevMessage = e => showMessageForEvent(e, currentMessageNumber -= 1);
  const firstMessage = e => showMessageForEvent(e, currentMessageNumber = 1);
  const lastMessage = e => showMessageForEvent(e, currentMessageNumber = messages.lengt

  const showMessageForEvent = (e, targetMessageNumber) => {
    showMessageByNumber(targetMessageNumber);
    e.preventDefault();
  };

  const showMessageByNumber = (messageNumber) =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const $ = document.querySelector.bind(document);

  const messages = [
    `<h1>Tutoren streiken!!!</h1>
    <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
    <p>Überall dieselbe alte Leier. Das Layout ist fertig, der Text lässt auf sich wart
    <p class="newsboard_footer">am 25.09.2015 von K. Einer</p>`,

    `<h1>Wahnsinn!</h1>
    <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
    <p>Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark. "Fix, Schwyz!
    <p class="newsboard_footer">am 13.08.2015 von Dr. B. Lödmann</p>`,

    `<h1>Prokrastination?</h1>
    <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab’s noch nicht erledigt; ich h
    <p>Denn esse est percipi - Sein ist wahrgenommen werden. Und weil Sie nun schon die
    <p>Sehen Sie, Webstandards sind das Regelwerk, auf dem Webseiten aufbauen. So gibt
    <p class="newsboard_footer">am 02.06.2015 von A. Meisenbär</p>`
  ];

  let currentMessageNumber = 1;

  init();
}
```

accompanying_files/05/solutions/newsboard/newsboard_1b.js

2.

```
"use strict";

{
  const init = () => {
    showNumberOfAvailableMessages();
    showMessageByNumber(currentMessageNumber);
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
    $("[title=first]").addEventListener("click", firstMessage);
    $("[title=last]").addEventListener("click", lastMessage);
  };

  const showNumberOfAvailableMessages = () =>
```

```
    $(".message_number").innerHTML = messages.length;

  const nextMessage = e => showMessageForEvent(e, currentMessageNumber += 1);
  const prevMessage = e => showMessageForEvent(e, currentMessageNumber -= 1);
  const firstMessage = e => showMessageForEvent(e, currentMessageNumber = 1);
  const lastMessage = e => showMessageForEvent(e, currentMessageNumber = messages.lengt

  const showMessageForEvent = (e, targetMessageNumber) => {
    showMessageByNumber(targetMessageNumber);
    e.preventDefault();
  };

  const showMessageByNumber = (messageNumber) =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const $ = document.querySelector.bind(document);

  const messages = [
    `<h1>Tutoren streiken!!!</h1>
    <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
    <p>Überall dieselbe alte Leier. Das Layout ist fertig, der Text lässt auf sich wart
    <p class="newsboard_footer">am 25.09.2015 von K. Einer</p>`,

    `<h1>Wahnsinn!</h1>
    <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
    <p>Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark. "Fix, Schwyz!
    <p class="newsboard_footer">am 13.08.2015 von Dr. B. Lödmann</p>`,

    `<h1>Prokrastination?</h1>
    <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab’s noch nicht erledigt; ich h
    <p>Denn esse est percipi - Sein ist wahrgenommen werden. Und weil Sie nun schon die
    <p>Sehen Sie, Webstandards sind das Regelwerk, auf dem Webseiten aufbauen. So gibt
    <p class="newsboard_footer">am 02.06.2015 von A. Meisenbär</p>`
  ];

  let currentMessageNumber = 1;

  init();
}
```

accompanying_files/05/solutions/newsboard/newsboard_2.js

3.

```
"use strict";

{
  const init = () => {
    showNumberOfAvailableMessages();
    showMessageByNumber(currentMessageNumber);
    $("body").addEventListener("keyup", handleKeyPress);
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
    $("[title=first]").addEventListener("click", firstMessage);
    $("[title=last]").addEventListener("click", lastMessage);
  };

  const handleKeyPress = e => {
    if (e.key === "ArrowRight") nextMessage(e);
    if (e.key === "ArrowLeft") prevMessage(e);
  };

  const showNumberOfAvailableMessages = () =>
    $(".message_number").innerHTML = messages.length;
```

```
21
22    const nextMessage = e => showMessageForEvent(e, currentMessageNumber += 1);
23    const prevMessage = e => showMessageForEvent(e, currentMessageNumber -= 1);
24    const firstMessage = e => showMessageForEvent(e, currentMessageNumber = 1);
25    const lastMessage = e => showMessageForEvent(e, currentMessageNumber = messages.lengt
26
27    const showMessageForEvent = (e, targetMessageNumber) => {
28      showMessageByNumber(targetMessageNumber);
29      e.preventDefault();
30    };
31
32    const showMessageByNumber = messageNumber =>
33      $(".newsboard_content").innerHTML = messages[messageNumber - 1];
34
35    const $ = document.querySelector.bind(document);
36
37    const messages = [
38      `<h1>Tutoren streiken!!!</h1>
39      <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
40      <p>Überall dieselbe alte Leier. Das Layout ist fertig, der Text lässt auf sich wart
41      <p class="newsboard_footer">am 25.09.2015 von K. Einer</p>`,
42
43      `<h1>Wahnsinn!</h1>
44      <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
45      <p>Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark. "Fix, Schwyz!
46      <p class="newsboard_footer">am 13.08.2015 von Dr. B. Lödmann</p>`,
47
48      `<h1>Prokrastination?</h1>
49      <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab’s noch nicht erledigt; ich h
50      <p>Denn esse est percipi - Sein ist wahrgenommen werden. Und weil Sie nun schon die
51      <p>Sehen Sie, Webstandards sind das Regelwerk, auf dem Webseiten aufbauen. So gibt
52      <p class="newsboard_footer">am 02.06.2015 von A. Meisenbär</p>`
53    ];
54
55    let currentMessageNumber = 1;
56
57    init();
58  }
```

accompanying_files/05/solutions/newsboard/newsboard_3.js

4.

```
 1  "use strict";
 2
 3  {
 4    const init = () => {
 5      showNumberOfAvailableMessages();
 6      showMessageByNumber(currentMessageNumber);
 7      $("body").addEventListener("keyup", handleKeyPress);
 8      $("[title=next]").addEventListener("click", nextMessage);
 9      $("[title=prev]").addEventListener("click", prevMessage);
10      $("[title=first]").addEventListener("click", firstMessage);
11      $("[title=last]").addEventListener("click", lastMessage);
12    };
13
14    const handleKeyPress = e => {
15      if (e.altKey) return handleKeyCodeWithAlt(e);
16      handleKeyCodeWithoutSpecialKeys(e);
17    };
18
19    const handleKeyCodeWithoutSpecialKeys = e => {
20      if (e.key === "ArrowRight") nextMessage(e);
21      if (e.key === "ArrowLeft") prevMessage(e);
```

```
  };

  const handleKeyCodeWithAlt = e => {
    if (e.key === "ArrowRight") lastMessage(e);
    if (e.key === "ArrowLeft") firstMessage(e);
  };

  const showNumberOfAvailableMessages = () =>
    $(".message_number").innerHTML = messages.length;

  const nextMessage = e => showMessageForEvent(e, currentMessageNumber += 1);
  const prevMessage = e => showMessageForEvent(e, currentMessageNumber -= 1);
  const firstMessage = e => showMessageForEvent(e, currentMessageNumber = 1);
  const lastMessage = e => showMessageForEvent(e, currentMessageNumber = messages.lengt

  const showMessageForEvent = (e, targetMessageNumber) => {
    showMessageByNumber(targetMessageNumber);
    e.preventDefault();
  };

  const showMessageByNumber = (messageNumber) =>
    $(".newsboard_content").innerHTML = messages[messageNumber - 1];

  const $ = document.querySelector.bind(document);

  const messages = [
    `<h1>Tutoren streiken!!!</h1>
    <h2>Alle Einsendeaufgaben werden ab sofort mit 0 Punkten bewertet</h2>
    <p>Überall dieselbe alte Leier. Das Layout ist fertig, der Text lässt auf sich wart
    <p class="newsboard_footer">am 25.09.2015 von K. Einer</p>`,

    `<h1>Wahnsinn!</h1>
    <h2>Wie ich mit einer dämlichen Idee ein Vermögen machte</h2>
    <p>Polyfon zwitschernd aßen Mäxchens Vögel Rüben, Joghurt und Quark. "Fix, Schwyz!
    <p class="newsboard_footer">am 13.08.2015 von Dr. B. Lödmann</p>`,

    `<h1>Prokrastination?</h1>
    <h2>Wie oft hörst du dich selbst sagen: „Nein, ich hab’s noch nicht erledigt; ich h
    <p>Denn esse est percipi - Sein ist wahrgenommen werden. Und weil Sie nun schon die
    <p>Sehen Sie, Webstandards sind das Regelwerk, auf dem Webseiten aufbauen. So gibt
    <p class="newsboard_footer">am 02.06.2015 von A. Meisenbär</p>`
  ];

  let currentMessageNumber = 1;

  init();
}
```

accompanying_files/05/solutions/newsboard/newsboard_4.js

Exercise 12: It's a Buyout!

```
"use strict";

{
  const init = () =>
    $("#buy").addEventListener("click", disableBuyButton);

  const disableBuyButton = e => e.target.disabled = true;

  init();
```

```
}
```

accompanying_files/07/solutions/buy_button/buy_button.js

Exercise 13: Newsboard — One Final Feature, Promise!

```
"use strict";

{
  const init = () => {
    initProgressbar();
    firstMessage();
    $("[title=next]").addEventListener("click", nextMessage);
    $("[title=prev]").addEventListener("click", prevMessage);
    $("[title=first]").addEventListener("click", firstMessage);
    $("[title=last]").addEventListener("click", lastMessage);
  };

  const initProgressbar = () => {
    progressbar().max = messages.length;
    progressbar().value = 1;
  };

  const nextMessage = () => {
    incCurrentMessageNumber();
    update();
  };
  const prevMessage = () => {
    decCurrentMessageNumber();
    update();
  };
  const firstMessage = () => {
    setCurrentMessageNumberToFirstMessage();
    update();
  };
  const lastMessage = () => {
    setCurrentMessageNumberToLastMessage();
    update();
  };

  const incCurrentMessageNumber = () => progressbar().value += 1;
  const decCurrentMessageNumber = () => progressbar().value -= 1;
  const setCurrentMessageNumberToFirstMessage = () => progressbar().value = 1;
  const setCurrentMessageNumberToLastMessage = () => progressbar().value = progressbar(

  const update = () => {
    updateMessage();
    updateButtonsState();
  };

  const updateMessage = () => showMessageByNumber(currentMessageNumber());

  const updateButtonsState = () => {
    buttonNext().disabled = lastMessageReached();
    buttonPrev().disabled = firstMessageReached();
    buttonLast().disabled = lastMessageReached();
    buttonFirst().disabled = firstMessageReached();
  };

  const lastMessageReached = () => currentMessageNumber() === messages.length;
  const firstMessageReached = () => currentMessageNumber() === 1;
```

```
56
57    const showMessageByNumber = messageNumber =>
58      $(".newsboard_content").innerHTML = messages[messageNumber - 1];
59
60    const buttonNext = () => $("[title=next]");
61    const buttonPrev = () => $("[title=prev]");
62    const buttonFirst = () => $("[title=first]");
63    const buttonLast = () => $("[title=last]");
64
65    const currentMessageNumber = () => progressbar().value;
66    const progressbar = () => $("#messages_progress");
67
68    init();
69  }
```

accompanying_files/07/solutions/newsboard/newsboard.js

Exercise 14: Return of the Running Lights: Part 1 - Chasing Lights

```
 1  "use strict";
 2
 3  {
 4    const init = () => lights().on("mouseenter", e => {
 5      turnAllOff();
 6      turnOnNext(e.target);
 7    });
 8
 9    const turnOnNext = light => turnOn(lights()[nextLightNr(light)]);
10    const nextLightNr = light => isLast(light) ? 0 : lightNr(light) + 1;
11    const isLast = light => lightNr(light) === lights().length - 1;
12    const lightNr = light => lights().indexOf(light);
13    const turnOn = light => light.src = "light_on.png";
14    const turnOff = light => light.src = "light_off.png";
15    const turnAllOff = () => lights().forEach(turnOff);
16    const lights = () => $$("img[alt=lightbulb]");
17
18    init();
19  }
```

accompanying_files/07/solutions/running_light/running_light.js

Exercise 15: Return of the Running Lights: Part 2 - Triggered Classic

```
 1  "use strict";
 2
 3  {
 4    const IMAGE_FILE_ON = "light_on.png";
 5    const IMAGE_FILE_OFF = "light_off.png";
 6
 7    const init = () => document.on("click", () => {
 8      const currentLightNr = findFirstTurnedOnLightNr();
 9      turnOffByNr(currentLightNr);
10      turnOnByNr(nextLightNr(currentLightNr));
11    });
12
13    const findFirstTurnedOnLightNr = () => lights().findIndex(isOn);
14    const isOn = light => light.src.endsWith(IMAGE_FILE_ON);
```

```
  const nextLightNr = nr => isLast(nr) ? 0 : nr + 1;
  const isLast = nr => nr === lights().length - 1;
  const turnOnByNr = nr => turnOn(lights()[nr]);
  const turnOffByNr = nr => turnOff(lights()[nr]);
  const turnOn = light => light.src = IMAGE_FILE_ON;
  const turnOff = light => light.src = IMAGE_FILE_OFF;
  const lights = () => $$("img[alt=lightbulb]");

  init();
}
```

accompanying_files/07/solutions/running_light/running_light2.js

Exercise 16: 01001000010011110101010O0 — Part 6

```
$$('p').forEach(p => p.style.color = "blue");
```

Exercise 17: TABula Rasa

```
"use strict";

{
  const init = () => {
    registerTabEvents();
    activateTabByIndex(0);
  };

  const registerTabEvents = () =>
    navItems().forEach((elem, i) =>
      elem.on("click", () => activateTabByIndex(i)));

  const activateTabByIndex = index => {
    removeHighlightFromActiveNavItem();
    hightlightNavItemByIndex(index);
    hideAllArticles();
    showArticleByIndex(index);
  };

  const removeHighlightFromActiveNavItem = () =>
    activeNavItems().forEach(elem =>
      elem.classList.remove("active"));

  const hightlightNavItemByIndex = index =>
    navItems()[index].classList.add("active");

  const hideAllArticles = () => articles().forEach(hide);

  const showArticleByIndex = index => show(articles()[index]);

  const articles = () => $$(".tabs article");
  const navItems = () => $$(".tabs > nav li");
  const activeNavItems = () => $$(".tabs > nav li.active");

  const show = elem => elem.style.display = "block";
  const hide = elem => elem.style.display = "none";

  init();
```

```
}
```

accompanying_files/08/solutions/tabbed_navigation/tabs.js

Exercise 18: Playing Tag With Colors

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="color_change.js" defer="defer"></script>
</head>

<body>
  <button>Red</button>
  <button>Green</button>
  <button>Blue</button>
</body>

</html>
```

accompanying_files/08/solutions/color_change1/color_change.html

```
"use strict";

$$("button").on("click", e => $("body").style.backgroundColor = e.target.innerHTML);
```

accompanying_files/08/solutions/color_change1/color_change.js

Exercise 19: Playing Tag With Colors — Part 2

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <title>Document</title>
  <script src="../../../lib/dom_helper.js" defer="defer"></script>
  <script src="color_change.js" defer="defer"></script>
</head>

<body>
  Red <input type="range" max="255" id="red"/>
  Green <input type="range" max="255" id="green" />
  Blue <input type="range" max="255" id="blue" />
</body>

</html>
```

accompanying_files/08/solutions/color_change2/color_change.html

```
"use strict";

const r = () => $("#red").value;
const g = () => $("#green").value;
const b = () => $("#blue").value;
```

```
6
7 $$("input[type='range']").on("input", () =>
8   $("body").style.backgroundColor = `rgb(${r()}, ${g()}, ${b()})`);
```

accompanying_files/08/solutions/color_change2/color_change.js

Exercise 20: awesome tours

```
 1 "use strict";
 2
 3 const showInfo = e => $("#info").innerHTML = `
 4   <h3>
 5      <img src="${country(e)}.png" alt="${flagName(e)}" title="${flagName(e)}" />
 6      ${name(e)}
 7   </h3>
 8   <p>${desc(e)}</p>`;
 9
10 const name = e => e.target.alt;
11 const desc = e => e.target.dataset.description;
12 const country = e => e.target.dataset.countryCode;
13 const flagName = e => e.target.dataset.flagName;
14
15 $$("img[data-description]").on("mouseenter", showInfo);
```

accompanying_files/09/solutions/awesome_tours/awesome_tours.js

Exercise 21: 01001000010011110101010O — Part 7

1. ```
   1 const li = document.createElement("li");
   2 li.textContent = "Capacity: 11 oz.";
   3 $("#product_specification").appendChild(li);
   ```

2. ```
   1 const option = document.createElement("option");
   2 option.textContent = "5 items";
   3 $("#buy_form select").appendChild(option);
   ```

Exercise 22: 01001000010011110101010O — Part 8

1. ```
   $("h2").remove()
   ```

2. ```
   $("h1 .keyword").remove()
   ```

Exercise 23: 0100100001001111010100 — Part 9

```
const li = document.createElement("li");
li.textContent = "Capacity: 11 oz.";
$("#product_specification").insertBefore(li, $$("#product_specification li")[4])
```

Exercise 24: Cuteycat

```
"use strict";

const NUMBER_OF_WINNERS = 3;

const mayAddCutie = cutie =>
   $$("#cutest li").length < NUMBER_OF_WINNERS
   && !isInCutest(cutie);

const isInCutest = cutie =>
  $$("#cutest li span")
    .filter(span => span.textContent === cutie.textContent)
    .length > 0;

$$("#candidates li").on("click", e => {
  const cutie = e.currentTarget;
  if (!mayAddCutie(cutie)) return;

  $("#cutest").appendChild(
    cutie.cloneNode(true).on("click", e => e.currentTarget.remove())
  );
});
```

accompanying_files/12/solutions/cute_kittens/cuteycat.js

We Want Your Feedback!

We would love to know what you think about this book. What did you like best, what wasn't so good? Have you missed any content or should we have shortened certain topics? How did you cope with knowledge questions and exercises?

Give us your opinion!

Send an email to the OWL Team at *feedback@owl.institute*