JavaScript: the language of browser interactions

👉 Overview of all Lecture 3 materials

This is the densest web lecture of this course. Learning how to code takes time. Take a look at the exercises that are relevant for this lecture.

At times we use ☝️ and 👇 to make it clear whether an explanation belongs to the code snippet above or below the text. The ‼️ sign is added to code examples you should run yourself. When you see a 🐛, we offer advice on how to debug your code with the browser's and VSC's tooling - these hints are solely to help you with your programming project and not exam material! Lastly, paragraphs with a 🚩 are just for your information and also not exam material.

Table of Contents

Learning goals

- Employ JavaScript objects.
- Employ the principle of callbacks.
- Write interactive web applications based on click, mouse and keystroke events.
- Explain and use `jQuery` (the library introduced in the course book).

Take-aways of book chapter 4

Haing read chapter 4 of the course book in preparation for this lecture, you should know:

- the basics of JavaScript;
- how to include JavaScript in your web application;
- what the strict mode is;
- the DOM;
- the basics of `jQuery` - a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. Its popularity has steadily declined over the past few years as modern browsers implement more and more functionalities that `jQuery` was designed to make easy. It is still though very much in use as many large-scale projects still have it as a dependency.

In this lecture we built upon book chapter 4 and cover a number of important JavaScript design patterns.

Examples throughout the lectures

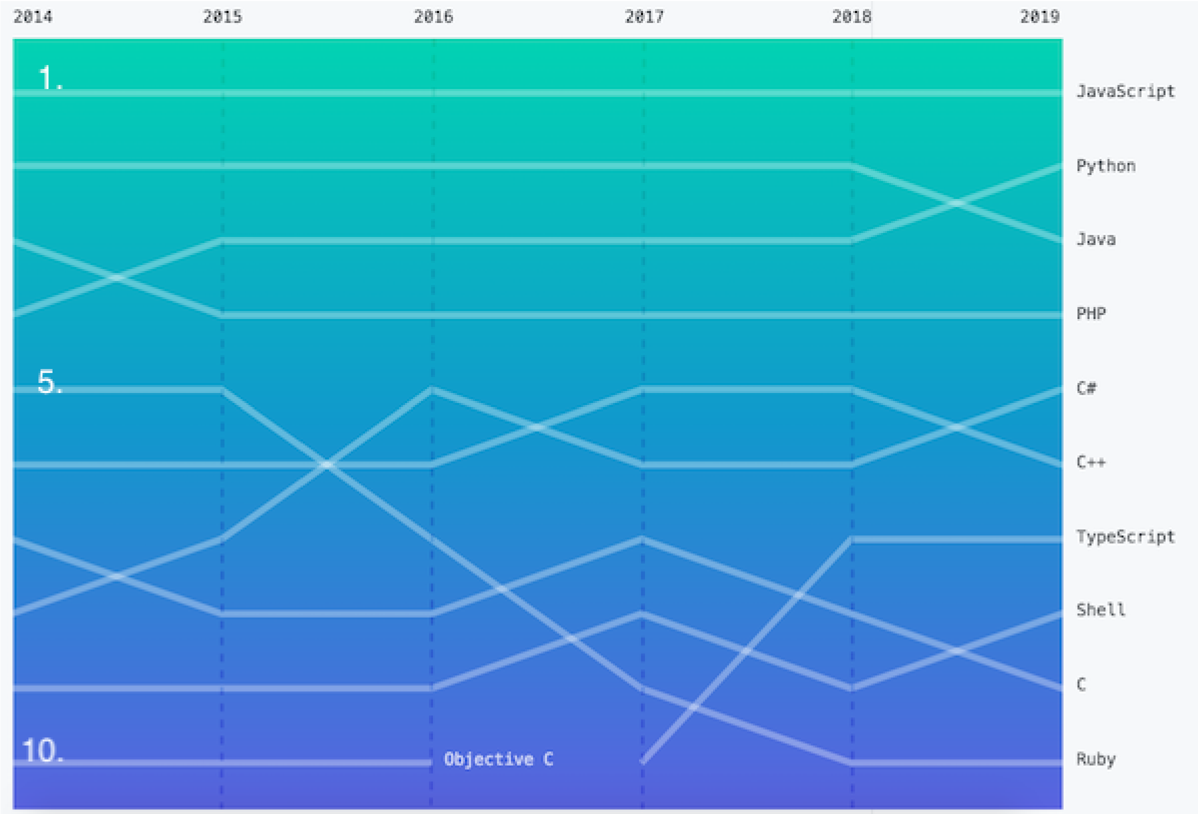
The code examples throughout these course materials tend to based on three different example applications:

- A todo application as introduced in the web course book.
- A habit tracker application as students had to implement in the 2017/18 edition of this course.
- A board game application as you need to implement this year.

As the course material has been developed over time, you will get a glimpse of each of those applications.

JavaScript's reputation

In the early years of JavaScript, it was considered more of a toy language. Today though, it is the most important language of the modern web stack. On GitHub, one of the most popular social coding platforms world-wide, JavaScript has taken the number 1 language spot in the past few years, with TypeScript, a language developed by Microsoft which compiles into JavaScript, claiming rank #7:



Top languages over time (as measured by number of contributors) on GitHub. Image source.

Vital to JavaScript's rise from toy language to serious contender is the availability of tooling, frameworks and libraries such as browsers' built-in dev tools, build tools, testing frameworks, UI frameworks, and so on. Another reason that Javascript became so popular is that it enables development in multiple programming paradigms (read this interview with Kyle Simpson, author of one of the most popular JavaScript book series if you want to know more).

Today's **JavaScript runtime environments** are highly efficient and a number of them co-exist peacefully:

- V8 is Google's JavaScript engine (used in Chrome and other browsers);
- SpiderMonkey is Mozilla's engine (used in Firefox);
- Chakra is Microsoft's JavaScript runtime engine (used in the Edge browser). In December 2018, Microsoft announced that they will adopt Chromium (Google's open-source browser project) and thus Edge will switch to the V8 JavaScript engine eventually.

While the browser is the most obvious usage scenario for JavaScript runtime environments, they are also used elsewhere (such as microcontrollers). Most importantly for us: the Node.js platform we cover in the next lecture is built on top of V8.

Javascript is an *interpreted* language, a browser's JavaScript engine reads and interprets JavaScript code line-by-line (advantage: quick upstart, disadvantage: the program overall *runs* slower than one written in a language requiring compilation). Today's JavaScript engines both interpret *and* compile by employing so-called **just-in-time (JIT) compilation**. This means that JavaScript code that is run repeatedly such as often-called functions (the JavaScript engine monitors the frequency of code usage) is eventually compiled and no longer interpreted. This article by Lin Clark explains this in more detail for those that want to know more.

JavaScript tracks ECMAScript, the scripting-language specification standardized by Ecma International. While JavaScript is the most popular implementation of the standard, other implementations or dialects exist as well (e.g. ActionScript).

JavaScript is a language in flux.

One of the confusing aspects about JavaScript today are the naming conventions, you may come across terms such as **ES6**, **ES7**, **ES2015**, **ECMAScript 2017**, and so on. These names refer to different version of ECMAScript (ES for short) which is in continuous development. Most often, you are likely to encounter **ES6** (also referred to as **ES2015**) which added a host of new features to the standard (a good overview is provided at http://es6-features.org/) and required a long-standing effort: *the completion of the sixth edition is the culmination of a fifteen year effort* (source). Starting with **ES2016** (also known as **ES7**), ECMAScript is updated in a yearly cycle.

Similar to HTML5, after a number of years with hardly any development, we are currently in a phase of continuous updates and changes.

In this course we include very few **ES6** features, as we only have one lecture to introduce JavaScript (*this* lecture ...). The course book has been released before the release of ES6 and thus does not incorporate any ES6 features; **this is a useful limitation**. If you want to go beyond the coverage of JavaScript in this course, take a look at the very comprehensive You Don't Know JS series.

In this course we cover *plain JavaScript*, but it is also worthwhile to know that many languages compile into JavaScript. The three most well-known of such languages are CoffeeScript, TypeScript and Dart, all three fill one or more gaps of the original JavaScript language. Once you work on complex projects in collaboration, these higher-level languages can make a difference, especially when it comes to debugging.

Here is one example of what TypeScript offers: JavaScript is a **dynamic language**, this means that you have no way of enforcing a certain **type** on a variable. Instead, a variable can hold any type, a String, a Number, an Array ... but of course often you *know* what you want the type to be (for instance function parameters). It is useful to provide this knowledge upfront. TypeScript allows you to do that, by **enabling static type checking**.

Scripting overview

Server-side vs. client-side scripting

Server-side scripting refers to scripts that run on the **web server** (in contrast to the client). Executing the scripts on the server means they are **private** and only the result of the script execution is returned to the client - often an HTML document. The client thus has to trust the server's computations (there is no possibility to validate the code that ran on the server). Server-side scripts can access **additional resources** (most often databases) and they can use **non-standard language features** (when you run a server you know which type of software is installed on it and what type of language features it supports). At the same time, as all computations are conducted on the server, with many clients sending HTTP requests, this can quickly **increase the server's load**. As clients often only receive an HTML document as result of the computation, the app developer does not have to worry about clients' device capabilities - any modern browser can render HTML.

Client-side scripting on the other hand does not return the result of a computation to the client, but instead sends the script (and if necessary the data) to the client which enables the user to dig through the code. A famous example of the uproar such code digging can cause is the *NYTimes election needle jitter*: a jitter was introduced to an election needle visualization in order to convey the uncertainty around election forecasting. This jitter though was not based on data as readers were expecting, but instead hard-coded as a random component into the client-side script. This was quickly spotted by a Twitter user:

> Looking for trends in @nytimes's presidential forecast needle? Don't look too hard - the bounce is random jitter from your PC, not live data pic.twitter.com/pwcV6epee7
>
> — Alp Toker (@atoker) November 9, 2016

and a lot of criticism followed (1, 2).

A clear advantage of client-side coding is **reduced server load**, as clients execute the scripts, though all data necessary for the scripts (which could be megabytes of data) need to be downloaded and processed by the client.

Modern browsers implement the IndexedDB API which provides a standard for an in-browser database that is transaction-based and stores key-value pairs persistently. While it cannot be queried with SQL directly, libraries such as JSstore exist that act as wrapper around IndexedDB to enable SQL-like querying. The storage limits are browser and device-dependent; in principle it is possible to store Gigabytes of data within the browser's database. This can be very useful for instance for games (game objects are stored in the database) as well as data processing pipelines that are designed to be easy-to-use for non-experts such as our recently open-sourced ELAT tool (the processed data is stored in the database). //TODO: URL ELAT

The `<script>` tag

The placement of the `<script>` tag is an often discussed issue (1000+ upvotes for this question on Stack Overflow alone). In this lecture, we follow the course book argument (page 98):

*"We place the `<script>` tags in the `<body>` element: the browser displays the page in a **top-down fashion, creating the DOM elements as it comes across them**. By placing the `<script>` tags at the end, the JavaScript files will be one of the last things the page loads. Because JavaScript files often take time to load, we do that last so that the user will get visual feedback from the other elements as fast as possible."*

In other words, interactivity based on the DOM should only start **after** the DOM has been fully loaded; if you decide to place your script's elsewhere, `jQuery`'s `document.ready` function is a useful utility.

‼️ Activity

Based on chapter 4 of the course book, you should be able to answer the following two questions.

Executing the JavaScript code snippet 👇 yields what output?

```
function giveMe(x){
    return function(y){
      return x*y;
    }
}
var giveMe5 = giveMe(5);

console.log( giveMe5(10) );
```

Executing the JavaScript code snippet 👇 yields what output?

```
function toPrint(x){
    console.log(x);
}

function my_func(x,y){
    y(x);
}

my_func(5, toPrint);
```

To assess your answers, run the code snippets' in the browser's Web Console.

Scoping, hoisting and this

We now cover three JavaScript principles that are often confusing for JavaScript novices.

Scoping

Scoping is the **context in which values and expressions are visible**. In contrast to other languages, JavaScript has very few scopes:

- local;
- global;
- block (introduced in **ES6**).

A *block* is used to group a number of statements together with a pair of curly brackets `{...}`.

The scopes of values and expressions depend on ***where*** and ***how*** they are declared:

- `var` declared within a function: **local** scope;
- `var` declared outside of a function: **global** scope;
- no `var`: **global scope** (no matter where declared);
- `let` was introduced in **ES6**: **block** scope;
- `const` was introduced in **ES6**: **block** scope, no reassignment or redeclaration (but the originally assigned element can change).

Before **ES6** there was no **block scope**, we only had two scopes available: local and global. Having only two scopes available resulted in code behaviour that is not always intuitive. Let's look at one popular example: imagine we want to print out the numbers 1 to 10. This is easy to achieve in JavaScript 👇 :

```
for (var i = 1; i <= 10; i++) {
    console.log(i);
}
```

Let's now imagine that the print outs should happen each after a delay of one second. Once you know that `setTimeout(fn, delay)` initiates a timer that calls the specified function `fn` (below: an **anonymous**

function) after a `delay` (specified in milliseconds) you might expect the following piece of code 👇 to print out the numbers 1 to 10 with each number appearing after roughly a second (***roughly***, as JavaScript timers are not overly precise due to JavaScript's single-thread nature):

```
for (var i = 1; i <= 10; i++) {
    setTimeout(function() {
        console.log(i);
    }, 1000);
}
```

☝ When you run the code you will actually find it to behave very differently: after around one second delay, you will see ten print outs of the number `11`. Make sure to try this out for yourself! Here is why: `setTimeout` is executed ten times without delay. Defined within `setTimeout` is a **callback**, i.e. the function to execute when the condition (the delay) is met. After the tenth time, the `for` loop executes `i++` and then breaks as the `i<=10` condition is no longer fulfilled. This means `i` is `11` at the end of the `for` loop. As `i` has **global scope** (recall: `var i` is declared outside a function), every single callback refers to the same variable. After a bit more time passes (reaching ~1 second), each of the function calls within `setTimeout` is now being executed. Every single function just prints out `i`. Since `i` is `11`, we will end up with ten print outs of `11`.

Let's fix the two issues (printing 11s instead of 1...10 and waiting a second ***between print outs*** one by one). In the code above, `var i` has **global** scope, but we actually need it to be of **local scope** such that every function has its own local copy of it. In addition, we increment the delay with each increment of `i`. Before **ES6** the following code snippet 👇 was the established solution:

```
function fn(i) {
    setTimeout(function() {
        console.log(i);
    }, 1000 * i);
}

for (var i = 1; i <= 10; i++)
    fn(i);
```

☝ You will find this construct in all kinds of code bases. We first define a function `fn` with one parameter and then use `setTimeout` within `fn`. JavaScript passes the value of a variable in a function; if the variable refers to an array or object, the value is the **reference** to the object. Here, `i` is a `number` and thus every call to `fn` has its own local copy of `i`.

With the introduction of **ES6** and `let`, we no longer need this additional function construct as `let` has block scope and thus every `i` referred to within `setTimeout` is a different variable. This now works as we would expect 👇 :

```
for (let i = 1; i <= 10; i++)
    setTimeout( function() {
        console.log(i)
    }, 1000 * i)
```

Scoping is also important when it comes to larger programming projects: imagine that you are working on a large project which makes use of a dozen or more JavaScript libraries. If all of these libraries would fill up the global namespace, inevitably at some point your code would stop working due to collisions in the global namespace. Here is a toy `jQuery` example to showcase this issue 👇 :

```
<!DOCTYPE html>
<html>
    <head>
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js">
</script>
        <script>
        $(document).ready(function(){
            //$ = "overwriting";
            $("#b").click(function(){
                $("#b").hide();
            });
        });
        </script>
    </head>
    <body>
        <h1>Hide this button</h1>
        <button id="b">Hide me forever</button>
    </body>
</html>
```

☝️ This code does exactly what we expect (hiding a button once we click it). Try it for yourself (save the code in a `.html` file and open it with the browser). You should also be familiar with the `jQuery` syntax and know that `$(..)` is an alias for the function `jQuery(..)`. But what happens if we overwrite `$`? Find out by uncommenting the `$ = "overwriting";` line of code. Result: the code is broken and we end up with `TypeError: $ is not a function`.

`jQuery` and other libraries have very few variables ending up in global scope in order to **reduce potential conflicts** with other JavaScript libraries. In addition, the **public API is minimized** in order to avoid unintentional side-effects (incorrect usage of the library by end users) as much as possible. We will later see how to achieve this with the module design pattern.

Hoisting

Hoisting is best explained with a concrete example. Consider this JavaScript code snippet 👇. What kind of console output do you expect after executing this snippet?

```
var x = six();

//function declaration
function six(){
    return 6;
```

```
}

var y = seven();

//function expression
var seven = function(){
    return 7;
}

console.log(x+" - "+y);
```

In both cases we seem to be executing a function (`six()` and `seven()` respectively) before they are defined. You may either believe that the JavaScript runtime does not care about when something is declared and the output will be `6 - 7` or you may believe that the JavaScript runtime does indeed care and the output will be a `TypeError: six is not a function`. Neither of these two options are correct however (verify for yourself in the browser by copying the entire snippet at once into the web console!), the output will be `TypeError: seven is not a function`. This means that while `var x = six();` works (i.e., we can call `six()` before declaring it), `var y = seven();` does not.

The difference lies in how we went about defining our `six` and `seven` functions:

- `var seven = function(){...}` is a **function expression** and is only defined when that line of code is reached.
- `function six(){...}` on the other hand is a **function declaration** and is defined as soon as its surrounding fucntion or script is executed due to the **hoisting principle**: declarations are processed before any code is executed.

In our example, the JavaScript runtime *hoists* the declaration of `six`; it is processed before the remaining code is executed.

Once more:

- Declarations are hoisted to the top.
- Expressions are not hoisted.

This is not only the case for functions, also variable declarations are hoisted. Consider this example 👇:

```
function f(){
    x = 5;
    y = 3;
};
f();
console.log(x);
console.log(y);
```

☝️ Variables `x` and `y` have global scope as they are not prefixed by `var` or `let` or `const`. And so the console output will be `5` and `3`.

But what happens in this slightly changed piece of code? 👇

```
function f(){
    a = 5;
    b = 3;
    var a, b;
};
f();
console.log(a);
console.log(b);
```

Now we will end up with a `ReferenceError: a is not defined` as the `var a` declaration at the end of function `f` is **hoisted** to the top of the function. The same applies to `b`. Both variables `a` and `b` thus have local scope and are not accessible to the `console.log` calls.

The keyword `this`

As you know, in Java, `this` refers to the current object.

In JavaScript what `this` refers to is dependent on *how* the function containing `this` was called.

We also have the option to set the value of a function's `this` independent of how the function was called, using the `bind` function.

Let's walk through this concrete code example to showcase the behaviour of `this` 👇 :

```
//We assume execution in the browser's Web Console, we thus
//know the global window object exists (it is provided by the browser).

//h is now a property of the global `window` variable;
//it can also be accessed as window.h
var h = "Sports";

function habit(s){
    this.h = s;
    this.printHabit = function(){
        console.log(this.h);
    }
}

//CASE 1
//Creating a new object and calling the object's printHabit() function
var habitObj = new habit("Reading");
habitObj.printHabit(); // this.h = "Reading"

//CASE 2
//Copying the printHabit function;
//printHabit is now a property of the global window object
var printHabit = habitObj.printHabit;
printHabit(); // this.h = "Sports"
```

```
//CASE 3
//Fixing 'this' of the printHabit function
var boundPrintHabit = printHabit.bind({h: "Music"});
boundPrintHabit(); // this.h = "Music"
```

☝ If you execute this code in the browser's Web Console, you will observe the output of the `printHabit` function, originally defined inside the `habit` function 👇:

```
function(){
    console.log(this.h);
}
```

to be different each time, as each time, `this` refers to a different object. We called the function in three different ways:

- CASE 1: as a method of an object;
- CASE 2: as a property of the global `window` object;
- CASE 3: as a bound function.

We will come across a number of other examples in this and the following lectures that will give you an intuition of what `this` is about. While a detailed discussion of `this` is outside the scope of this lecture, you should realize that it is a complex concept. MDN has a whole page dedicated to `this`, while the popular You Don't Know JavaScript book series covers the concept in about half a book.

🚩 In ES6 so-called arrow functions were introduced. Instead of writing `let sum = function(a,b){return a+b}` we can shorten it to `let sum = (a,b) => {return a+b}`. This may look initially just like a more compact way of writing a function expression, but there is more to it - in particular the way `this` behaves in this context is different to that of regular functions! Be aware of this if you are looking into the use of arrow functions!

🚩 A canonical use case for arrow functions are the `.map()`, `.reduce()` and `.filter()` functions introduced in ES6 for arrays. They are not difficult to understand, we here explain them on an example. Consider the array `let ar=['garden','town','carriage','mice','wizzard','hat']`. If we want to convert all array elements to uppercase, we apply a function to every array element: `let arUpperCase = ar.map(x => x.toUpperCase())`. If we want to only keep those array elements with strings of more than five characters we use `let arLong = ar.filter(x => x.length > 5)`. The `.reduce()` function is maybe the hardest to wrap one's head around: it works on every element of the array and produces a single output value. For instance, all characters of the array can be computed as follows: `let reducer = (accumulator, currentValue) => accumulator + currentValue.length; let totalChars = ar.reduce(reducer,0));`.

JavaScript design patterns

"*Design patterns are reusable solutions to commonly occurring problems in software design.*" This quote is on the first page of Addy Osmani's popular book on JavaScript design patterns. A basic example of a reusable solution is one for object creation. Instead of everyone trying to figure out how to create objects, we use well-known **recipes** (a.k.a. design patterns) that were developed over time and apply them. There are many different design patterns, some are known to work across languages and some are specific to just a small subset of programming languages. What we cover in this lecture is mostly specific to JavaScript. Note that besides **design patterns**, there also exist **anti-patterns**, that are programming recipes which are popular but ineffective at tackling a recurring problem.

JavaScript objects

In JavaScript, **functions are first-class citizens** of the language. This means that **functions can be passed as parameters**, they can be **returned from functions** and they can be **assigned to a variable**. This is quite a difference to Java for example, where functions cannot be passed around.

The object-oriented programming paradigm is based on a set of cooperating objects (each one able to send/receive "messages" and process data) instead of a collections of functions or a set of commands. The goal of object-oriented design is to assign every object a distinct role, in order to improve code maintainability.

In JavaScript, **functions are also objects**. Apart from functions, JavaScript also comes with a number of other built-in objects: Strings, arrays and objects specific to the fact that JavaScript was developed to add interaction to HTML. One example is the `document` object, which only makes sense in the context of an HTML page. Note, that the `document` object is not part of core JavaScript (the language is defined independently of the browser context), however when we discuss client-side scripting we do mean JavaScript in the browser. The browser is the host application in this case and provides the `document` object.

JavaScript objects can be created in different ways. This is very much unlike Java where there is essentially only one: you have a class, write a constructor and then use the `new` keyword to create an object. We will not consider all the manners of creating JavaScript objects here, you should remember though that there are different ways (especially when you look at existing code bases).

Object creation with `new`

Let's start with the creation of objects. Here you see one way of creating objects in JavaScript 👇:

```javascript
var game = new Object();
game["id"] = 1;
game["player1"] = "Alice";
game.player2 = "Bob";
console.log( game["player2"] ); //prints out "Bob"
console.log( game.player1 ); //prints out "Alice"

game["won lost"] = "1 12";

game.printID = function(){
    console.log( this.id );
}
game["printID"](); // prints out "1"
game.printID(); //prints out "1"
```

☝️ We first create an empty object with `new Object()` that we can then assign name/value pairs. Here, `id`, `player1`, etc. are the object's **properties** and their name must be a valid JavaScript identifier (basically a string that does not start with a number). Note, that `printID` is also an object property, although it is often also referred to as a method because we define a function as part of an object. As seen here, JavaScript makes it easy to add methods, by assigning a function to the property of an object.

We have two ways to set and get an object's properties: either through the bracket notation (`[name]`) or the dot notation (`.name`). It usually does not matter which notation to use, the exception here being property names with whitespaces. Property names that contain whitespaces must be set and accessed through the bracket notation (as in the example above for `game["won lost"]`, the alternative `game.won lost` or `game."won lost"` will lead to a `SyntaxError`).

Object literals

There is a second way to create objects and that is via **object literals**. An object literal is a list of zero or more pairs of property names and associated values of an object, enclosed in curly braces 👇:

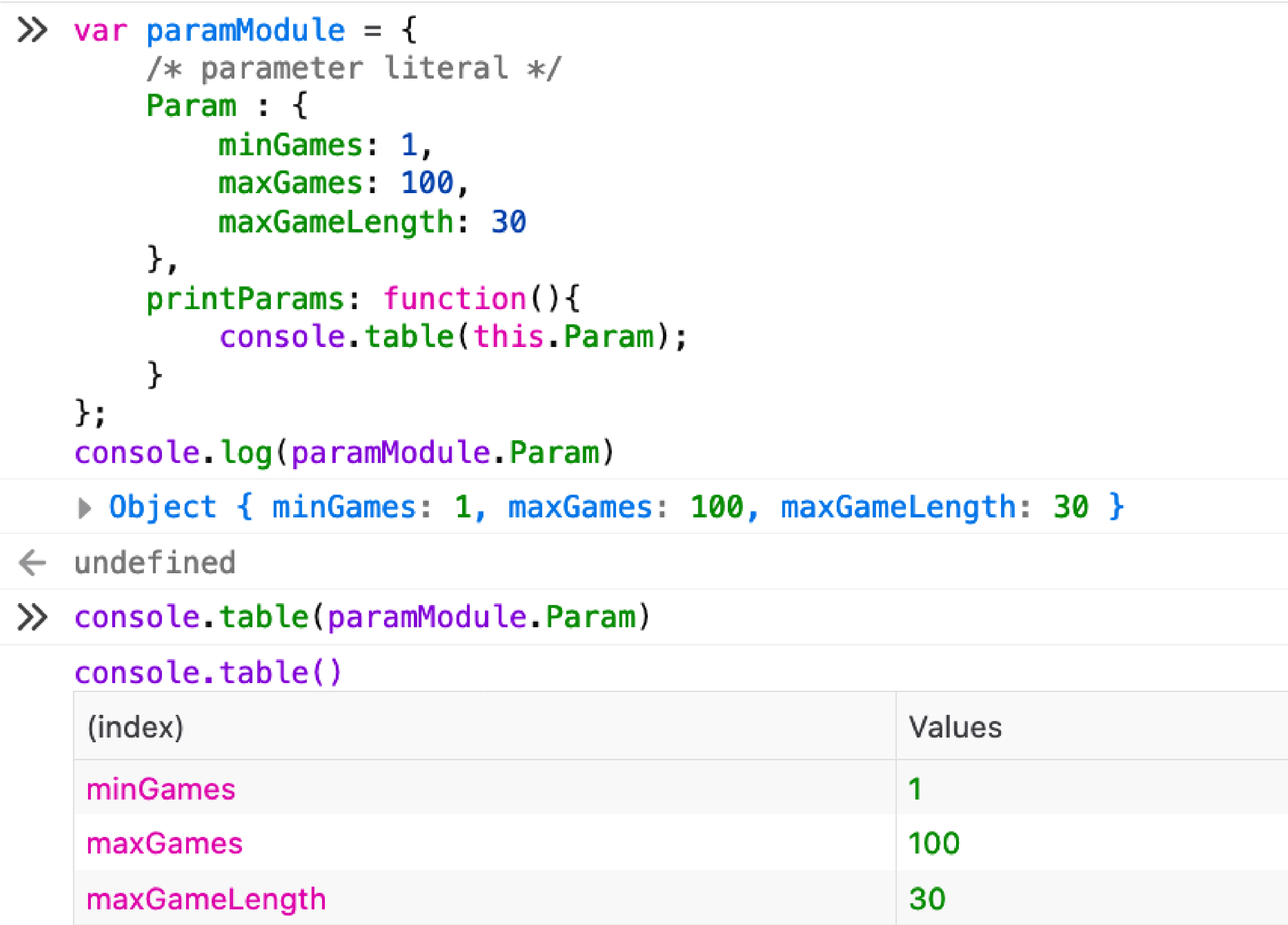
```javascript
var game = {
    id: 1,
    player1: "Alice",
    player2: "Bob",
    "won lost": "1 12",
    printID: function(){
        console.log(this.id);
    }
};
```

This time, `"won lost"` is a valid property name, but only if enclosed in quotation marks. *Instead of remembering when whitespaces are allowed, it is best to avoid them at all when assigning property names.*

Object literals can be complex, they can contain objects themselves 👇:

```javascript
var paramModule = {
    /* parameter literal */
    Param : {
        minGames: 1,
        maxGames: 100,
        maxGameLength: 30
    },
    printParams: function(){
        console.table(this.Param);
    }
};
```

🐛 For debugging purposes, the function `console.table` is a good alternative to `console.log`, especially for objects and arrays, as it displays tabular data as a table:



Screenshot of Firefox's Web Console.

Continuing on the debugging theme, another worthwhile function to know about is `console.assert` which prints an error if an assertion is false. If you have for instance a function that should always be called with a single positive integer, there is nothing you can do to enforce this - JavaScript is a dynamic language. However, if you know that any valid function call must have a single integer argument, you can use assertions to - at least at runtime - observe the assertion failure in case the function is used in an unintended manner:

```
>> var paramModule = {
       /* parameter literal */
       Param : {
           minGames: 1,
           maxGames: 100,
           maxGameLength: 30
       },
       printParams: function(){
           console.table(this.Param);
       },
       setMaxGames: function(n){
           console.assert(Number.isInteger(n), "Param needs to be an integer");
           console.assert(n > 0, "Param needs to be greater than 0");
           this.Param.maxGames = n;
       }
   };
← undefined
>> paramModule.setMaxGames(200);
← undefined
>> paramModule.setMaxGames("A string");
! Assertion failed: Param needs to be an integer            debugger eval code:12:11
! Assertion failed: Param needs to be greater than 0        debugger eval code:13:11
← undefined
```

Screenshot of Firefox's Web Console.

Let's go back to object literals: what happens if we need 1000 objects of the same kind? What happens if a method needs to be added to all objects? We can hardly copy and paste a method to all objects.

One idea could be to simply ***copy*** an object over and over again, however that turns out to be quite complicated. JavaScript passes everything by reference, which can cause issues when objects are complex (i.e. many of their properties are objects themselves). This guide explains in detail how to create deep copies of objects in JavaScript. This is for your information only, we do not cover deep copies in class.

Let's look at three design patterns to simplify this work for us! The first one should be the most familiar to you, as it looks similar to the object creation pattnern we use in Java.

Design pattern I: Basic constructor

First, let's quickly recap what classes in Java offer us:

- we can encapsulate private members, i.e. members of the class that are not accessible externally;
- we define constructors that define how to initialize a new object;
- we define methods (public, protected, private).

Here is a **Java** example 👇:

```
public class Game {
    private int id; /* encapsulate private members */

    /* constructor: a special method to initialize a new object */
    public Game(int id){
        this.id = id; /* this: reference to the current object */
```

```
    }

    public int getID(){
        return this.id;
    }

    public void setID(int id){
        this.id = id;
    }
}
```

And here is how we do the same in JavaScript 👇:

```
function Game(id){
    this.id = id;
    this.totalPoints = 0;
    this.winner = null;
    this.difficulty = "easy";

    this.getID = function(){ return this.id; };
    this.setID = function(id){ this.id = id; };
}
```

We use functions as constructors and rely on `this`. We rely on the keyword `new` to initialize a new object similar to what you have already seen before 👇:

```
var g1 = new Game(1);
g1.getID();
g1.setID(2);
var g2 = new Game(3);

//🚩 ES6: object destructuring allows us to extract several object
properties at once instead of one-by-one
var {totalPoints,winner,difficulty} = g1;
//🚩 ES6: template literals to make string concatenations more readable
console.log(`This game reached ${totalPoints} points, was won by ${winner}
and had difficulty ${diff}.`);
```

In JavaScript, an object constructor is just a normal function. When the `new` keyword appears, the JavaScript runtime executes two steps:

1. A new anonymous empty object is created and `this` refers to it.
2. The new object is **returned** at the end of the function (though no `return` statement exists).

A common error is to forget the `new` keyword. The JavaScript runtime will not alert you to this mistake, in fact, the JavaScript runtime will simply execute the function as-is. Let's take a look at what happens when you copy and paste the following code into your browser's Web Console 👇:

```
function Game(id){
    this.id = id;
    this.getID = function(){ return this.id; };
    this.setID = function(id){ this.id = id; };
}

var g1 = new Game("ONE"); //remember: dynamic language, we cannot enforce a parameter type
var id = g1.getID();
console.log(id); //prints out "ONE"
g1.setID(2);

var g2 = Game("TWO"); //what does "this" refer to now?
```

☝ In this code snippet we created a new object assigned to variable `g1`, but for `g2` we forgot the keyword `new` and thus no object was created or assigned to `g2`. If you check what was assigned to `g2` you will find it to be `undefined` (the variable was declared but not defined). So, what happened to the line `this.id = id`? What did `this` refer to in this case? It turns out that without an object, in the browser context, `this` refers to the global `window` object (which represents the window in which the script is running). If you type `window.id` you will find the property to exist and hold the value of `TWO`. Of course, this is not desired as you may accidentally overwrite important properties of the `window` object.

Lesson here: be sure to know when to use `new` and what `this` refers to when.

Another interesting feature of JavaScript is the possibility to add new properties and methods **on the fly**, after object creation. In Java, once we have written our class and instantiated objects from the class, we cannot rewrite the class blueprint to affect the already created objects. JavaScript is a **prototype-based language** and here we can actually change our objects on the fly 👇:

```
function Game(id){
    this.id = id;
    this.getID = function(){ return this.id; };
    this.setID = function(id){ this.id = id; };
}

var g1 = new Game("1");
g1.player1 = "Alice";

var g2 = new Game("2");
g2.player1 = "Bob";

g1.printPlayer = function(){ console.log(this.player1); } //we add a method on the fly!
g1.printPlayer(); //prints out "Alice"

g2.printPlayer(); //TypeError: g2.printPlayer is not a function (method was added to g1 alone!)

g1.hasOwnProperty("printPlayer"); //true
```

```
g2.hasOwnProperty("printPlayer"); //false

g1.toString(); //"[object Object]" (we never defined it, but it is there)
```

Here is a quick summary of the basic constructor:

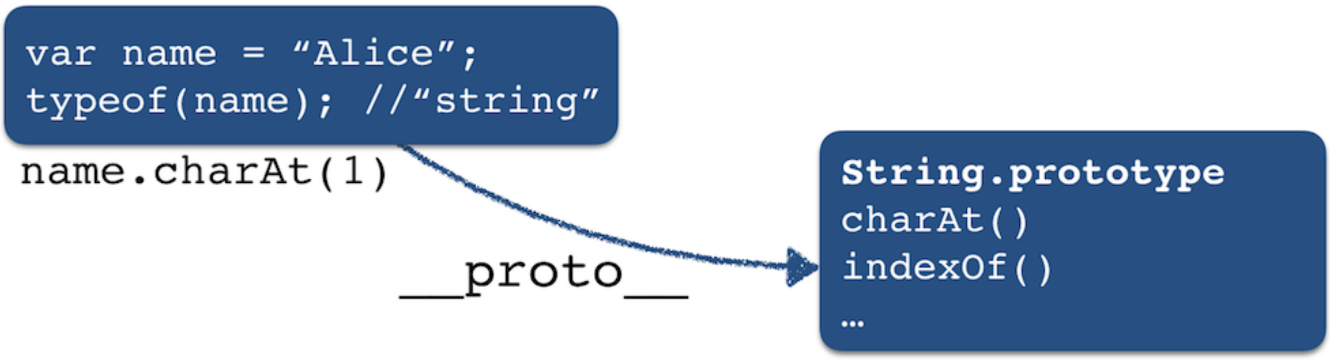
- Advantage: very easy to use
- Issues:
    1. Not obvious how to use **inheritance**;
    2. Objects **do not share** functions (`g2` did not have a `printPlayer` method, but `g1` had);
    3. All members are **public** and **any piece of code can be accessed/changed/deleted** (which makes for less than great code maintainability).

We tackle issues 1. and 2. with the next design pattern.

We have already touched upon the drawback of the third issue: imagine you are using a particular JavaScript library; if you are not aware of the library' internals, you may inadvertently "overwrite" important parts of the code without ever being informed about it, because that is not how the JavaScript runtime works.

Design pattern 2: Prototype-based constructor

The last line of the code snippet above ☝ shows that objects come with **default methods**, and so the natural question should be, where do these methods come from? The answer is **prototype chaining**. Objects have a **secret pointer** to another object - the object's prototype. And thus, when creating for instance an object with a basic constructor as just seen, **the properties of the constructor's prototype are also accessible in the new object**. If a property is not defined in the object, the **prototype chain** is followed:



Here, `name.__proto__` points to the object that is next in the lookup chain to resolve property names. As always though, things are not quite as simple and over time JavaScript runtimes have evolved in their implementation of **proto**. Normally, it is not necessary to manually *walk up* the prototype chain, instead the JavaScript runtime does the work for you.

So, why is this important and how can you make use of this knowledge? Recall, that one of the issues in the basic constructor is that **objects do not share functions**. Often we do want objects to share functions and if a function changes that change should be reflected in all objects that have this property/method.

This is exactly what the prototype-based constructor provides. Let's look at an example 👇:

```
function Game(id){
    this.id = id;
}

/* new member functions are defined once in the prototype */
Game.prototype.getID = function(){ return this.id; };
Game.prototype.setID = function(id){ this.id = id; };

//using it
var g1 = new Game("1");
g1.setID("2"); //that works!

var g2 = new Game("2");
g2.setID(3); //that works too!

//g1 and g2 now point to different setID properties
//g2 follows the prototype chain
//g1 has  property setID
g1["setID"] = function(id){
    this.id = "ID"+id;
}
g1.setID(4);

console.log(g1.getID()); //ID4
console.log(g2.getID()); //3
```

☝ All we have to do to make properties available to all objects is to use the `.prototype` property to walk up the prototype chain and assign a property to `Game.prototype`. When the two game objects are created and `setID()` is called, the JavaScript runtime walks up the prototype chain and "finds" the **first** property that matches the desired property name.

This explanation should also answer the following question: what happens if a property is defined as property of the object **as well as** as property of the prototype? The JavaScript runtime stops as soon as the property is found in the chain and this means that `g1.setID` and `g2.setID` now refer to different pieces of code.

Changes made to the prototype are also reflected in existing objects 👇:

```
function Game(id){
    this.id = id;
}

/* new member functions are defined once in the prototype */
Game.prototype.getID = function(){ return this.id; };
Game.prototype.setID = function(id){ this.id = id; };

//using it
var g1 = new Game("1");
g1.setID("2"); //works
console.log( g1.getID() ); //prints out "2"
```

```
Game.prototype.setID = function(id){
    console.assert(typeof(id)=="number", "Expecting a number");
    this.id = id;
}

g1.setID("3");//leads to "Assertion failed: Expecting a number"
```

The prototype chaining allows us to set up **inheritance through prototyping**. This requires two steps:

1. Create a new constructor.
2. Redirect the prototype.

Let's assume we want to inherit from `Game` to create a more specialized two-player game variant 👇:

```
function Game(id){
    this.id = id;
}

/* new member functions are defined once in the prototype */
Game.prototype.getID = function(){ return this.id; };
Game.prototype.setID = function(id){ this.id = id; };

/* constructor */
function TwoPlayerGame(id, p1, p2){
    /*
     * call(...) calls a function with a given `this` value and arguments.
     */
    Game.call(this, id);
    this.p1 = p1;
    this.p2 = p2;
}

/* redirect prototype */
TwoPlayerGame.prototype = Object.create(Game.prototype);
TwoPlayerGame.prototype.constructor = TwoPlayerGame;

/* use it */
var TPGame = new TwoPlayerGame(1, "Alice", "Bob");
console.log( TPGame.getID() ); //prints out "1"
console.log( TPGame.p1 ); //prints out "Alice"
```

☝ Why do we need to redirect the prototype? Recall the prototype chain: when we make the call to `TPGame.getID()` the JavaScript runtime finds `getID()` to not be a property of `TPGame`. So it attempts to walk up the prototype chain and in order to make `Game` part of the `TPGame` prototype chain we have to manually set it.

Why do we have to also set the `constructor` property? You will see if you run this piece of code and remove the line 👇:

```
TwoPlayerGame.prototype.constructor = TwoPlayerGame;
```

the code still works as expected. Why do we even add this line? If we do not add this line, then the `constructor` of `TwoPlayerGame.prototype` will be `Game` (check it out for yourself). With this extra line of code we "hand-wire" the correct constructor (which for `TwoPlayerGame.prototype` should be `TwoPlayerGame`). You can think of this as making sure the wiring is correct, even if your code does not rely on this wiring at the moment.

Here is one example where it does indeed matter whether whether this wiring is correct 👇:

```
function Game() {}
function TwoPlayerGame() {}

TwoPlayerGame.prototype = Object.create(Game.prototype);

TwoPlayerGame.prototype.create = function create() {
  return new this.constructor();
}

var o = new TwoPlayerGame().create();
console.log( o instanceof TwoPlayerGame ); //prints out "false" as long as
the constructor is not set to TwoPlayerGame
```

As a rule of thumb: when using prototypical inheritance, always set up both the `prototype` and `prototype.constructor`; in this manner the wiring is correct, no matter how you will use the inheritance chain later on.

To finish off, here is a summary of the prototype-based constructor:

- Advantages:
  - **Inheritance is easy** to achieve;
  - **Objects share functions**;
- Issue:
  - All members are **public** and **any piece of code can be accessed/changed/deleted** (which makes for less than great code maintainability).

We now tackle the remaining issue in the next design pattern.

Design pattern 3: Module

The module pattern has the following goals:

- **Do not declare any global variables** or functions unless required.
- Emulate **private/public** membership.
- Expose only the **necessary** members to the public.

We start with a concrete example of the **module pattern** 👇:

```javascript
/* creating a module */
var gameStatModule = ( function() {

    /* private members */
    var gamesStarted = 0;
    var gamesCompleted = 0;
    var gamesAbolished = 0;

    /* public members: return accessible object */
    return {
        incrGamesStarted : function(){
            gamesStarted++;
        },
        getNumGamesStarted : function(){
            return gamesStarted;
        }
    }
})();

/* using the module */
gameStatModule.incrGamesStarted();
console.log( gameStatModule.getNumGamesStarted() ); //prints out "1"
console.log( gameStatModule.gamesStarted ); //prints out "undefined"
```

In this code snippet ☝, we are defining a variable `gameStatModule` which is assigned a `function` expression that is immediately invoked. This is known as an *Immediately Invoked Function Expression* (or IIFE).

An IIFE itself is also a design pattern, it looks as follows 👇:

```javascript
(function () {
    //statements
})();
```

☝ The function is **anonymous** (it does not have a name and it does not need a name, as it is immediately invoked) and the final pair of brackets `()` leads to its immediate execution. The brackets surrounding the function are not strictly necessary, but they are commonly used.

Going back to our `gameStatModule` ☝ ☝, we immediately execute the function. The function contains a number of variables with function scope and a return statement. **The return statement contains the result of the function invocation**. In this case, an *object literal* is returned and this object literal has two methods: `incrGamesStarted()` and `getNumGamesStarted()`. Outside of this module, we cannot directly access `gamesStarted` or any of the other emulated *private* variables, all we will get is an `undefined` as the returned object does not contain those properties. The returned object though **has access** to them through JavaScript's concept of closures). *A closure is the combination of a function and the lexical environment within which that function was declared* (as defined by MDN); in our case the lexical environment includes the emulated private variables. Once again, things are not as easy as they seem, in the You Don't Know JS series, half a book is dedicated to closures.

A common error in the module pattern is to forget to add the final bracket pair `()` when defining the IIFE. Those issues will be caught at runtime when the code does not work as expected. In our game module example, the line `gameStatModule.incrGamesStarted();` will lead to the error `TypeError: gameStatModule.incrGamesStarted is not a function` if we remove the final IIFE bracket pair (try it out!). VSC offers a simple way to catch those errors already when coding. We simply add the line:

```
//@ts-check
```

at the top of any JavaScript file we want to have type-checked. The error of the missing bracket pair is now caught:

We thus borrow the type checker of TypeScript to make sure to catch - at least some - coding mistakes we make early on. To avoid copying this line everywhere, we can also set up VSC to perform type checking automatically for all JavaScript files.

Finally we note that in the module pattern, the encapsulating function can also contain parameters (here: arguments `1, 1, 1`) :

```
/* creating a module */
var gameStatModule =  function(s, c, a) {

    /* private members */
    var gamesStarted = s;
    var gamesCompleted = c;
    var gamesAbolished = a;

    /* public members: return accessible object */
```

```
    return {
        incrGamesStarted : function(){
            gamesStarted++;
        },
        getNumGamesStarted : function(){
            return gamesStarted;
        }
    }
}(1, 1, 1);

/* using the module */
gameStatModule.incrGamesStarted();
console.log( gameStatModule.getNumGamesStarted() ); //prints out 2

//can be defined on the fly, but ...
gameStatModule.decrGamesStarted = function(){
    gamesStarted--;
}

/*
 * once this method is called, it leads to an error:
 * ReferenceError: gamesStarted is not defined;
 * methods added on-the-fly cannot access 'private' variables
 */
gameStatModule.decrGamesStarted();
```

Summarizing the module pattern:

- Advantages:
  - **Encapsulation is achieved**;
  - Object members are either public or private;
- Issues:
  - Changing the type of membership (public/private) takes effort (unlike in Java).
  - Methods added on the fly later cannot access emulated private members (as seen in the last code snippet).

Events and the DOM

Having read chapter 4 of the web course book, you should be familiar with code such as this 👇 (the strict mode applies to functions or entire scripts and is a manner of cutting down on potential silent errors):

```
var main = function () {
  "use strict";
  $(".comment-input button").on("click", function (event) {
    var $new_comment = $("<p>"),
    comment_text = $(".comment-input input").val();
    $new_comment.text(comment_text);
    $(".comments").append($new_comment);
  });
```

```
};
$(document).ready(main);
```

The course book makes extensive use of `jQuery`. With `jQuery` it does not matter if you are after a `class` or `id`, the access pattern is always the same: `$()`. This is in contrast to plain JavaScript where we deal with `document` (the web page object and our entry point to the DOM) which comes with a number of manners to select groups or single DOM elements:

- `document.getElementById`
- `document.getElementsByClassName`
- `document.getElementsByTagName`
- `document.querySelector`: returns the first element within the DOM tree according to a **depth-first pre-order traversal** that matches the selector
- `document.querySelectorAll`: returns all elements that match the selector

The last two ways of selecting DOM elements allow complex selector rules to be specified!

The code snippet ☝ also shows off the **callback principle**, which we come across in all of JavaScript: we define **what happens *when* an event fires**. In the example above, the event is the click on a button.

The course book walks you through several examples of making a responsive UI control. Here is a step-by-step guide:

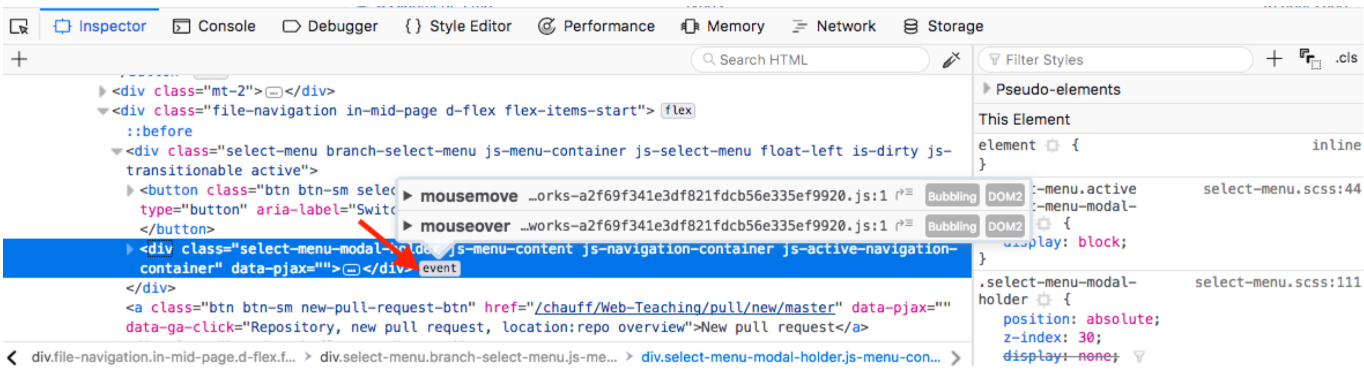
1️⃣ Pick a control, e.g. a `button`.

2️⃣ Pick an event, e.g. a `click`.

3️⃣ Write a JavaScript function: what should happen when the event occurs, e.g. an `alert` message may appear.

4️⃣ Attach the function to the event **on** the control.

🐛 If you want to examine how web applications make use of events, the browser developer tools will help you once more. On Firefox, the **HTML panel** allows you to explore **which events are attached to which controls** and with a click on the event button itself, you can dig into the callback function as seen here:



Screenshot of Firefox's Web Console.

🚩 If you are already familiar with modern JavaScript you may wonder why we do not cover concepts such as promises and async/await (which were designed to solve the major issue that arise with callbacks) in this class. The reason is simply a lack of time. For those interested in how to make asynchronous programming in JavaScript less painful (and once you will have used callbacks in the programming project of this class you will understand what the phrase ***callback hell*** refers to), take a look at this [very detailed video]() on the topic.

Document Object Model

The DOM is our entry point to interactive web applications. It allows use to:

- **Extract an element's state**
  - Is the checkbox checked?
  - Is the button disabled?
  - Is a <h1> appearing on the page?
- **Change an element's state**
  - Check a checkbox
  - Disable a button
  - Create an <h1> element on a page if none exists
- **Change an element's style** (material for a later lecture)
  - Change the color of a button
  - Change the size of a paragraph
  - Change the background color of a web application

We will now walk through a number of examples that add an interactive element to a web application. These examples are small and self-contained. This means that all necessary code is contained within a single code snippet.

They strongly overlap with what is discussed in the required reading (chapter 4 of the course book) of this lecture. Take it as a reminder of what is covered in the book chapter.

‼️ Example 1: document.getElementById / document.querySelector

Here 👇 we have a page with two elements: a button and a text box. A click on the button will show `Hello World!` in the text box. As you can see there are different ways (we have listed four here) of pinpointing a DOM element:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 1</title>
    </head>
    <body>
        <button onclick="
            //var tb = document.querySelector('#out');
            //var tb = document.querySelector('input')
            //var tb = document.querySelector('input[type=text]')
            var tb = document.getElementById('out');
            tb.value = 'Hello World!'
        ">Say Hello World</button>
```

```
        <input id="out" type="text">
    </body>
</html>
```

This code ☝️ is of course not ideal as we are writting JavaScript code in the middle of HTML elements, so let us refactor to achieve a better code separation 👇:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 1</title>
        <script>
            /* we define the function */
            function sayHello() {
              var tb = document.getElementById("out");
              tb.value = "Hello World";
            }

            /* we attach a function to a button's click event
             * after the DOM finished loading
             */
            window.onload = function() {
                document.getElementById("b").onclick = sayHello;
            };
        </script>
    </head>

    <body>
        <button id="b">Say Hello World</button>
        <input id="out" type="text">
    </body>
</html>
```

Although all code is still in a single file, we have now moved all JavaScript code within `<script>` tags. Try the code out yourself! Be sure to check out what happens if the snippet 👇:

```
window.onload = function() {
    document.getElementById("b").onclick = sayHello;
};
```

is replaced by 👇:

```
document.getElementById("b").onclick = sayHello;
```

Explanation: the HTML page is parsed sequentially from top to bottom; without the `window.onload` event handler the browser engine will try to acess the button element before it has been defined, leading to an error: `TypeError: document.getElementById(...) is null.`

‼️ Example 2: creating new nodes

Note: as all our examples are simple, we will stick to `document.getElementById` to select DOM elements. In more realistic coding scenarios, `document.querySelector(All)` will most often be used.

HTML tags and content can be added dynamically in two steps:

1️⃣ Create a DOM node.

2️⃣ Add the new node to the document as a **child of an existing node**.

To achieve step 2️⃣, a number of methods are available to every DOM element:

Name	Description
`appendChild(new)`	places the new node at the end of this node's child list
`insertBefore(new, old)`	places the new node in this node's child list just before the old child
`removeChild(node)`	removes the given node from this node's child list
`replaceChild(new, old)`	replaces the old node with the new one

Let's look at how this works in practice 👇 :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 2</title>
        <script>
            window.onload = function() {
              document.getElementById("b").onclick = addElement;
            };

            function addElement() {
              var ul = document.getElementById('u');
              var li = document.createElement('li');
              var count = ul.childElementCount+1;

              //li.innerHTML = 'List element ' + (ul.childElementCount+1)
+' '; //before ES6: + to concatenate strings
              li.innerHTML = `List element ${ul.childElementCount+1}`; //
🚩 ES6: template literals
              ul.appendChild(li);
            }
        </script>
    </head>
```

```
    <body>
        <button id="b">Add List Element</button>
        <ul id="u"></ul>
    </body>
</html>
```

☝️ The HTML initally contains an **empty `<ul>` element**. Instead of directly adding `<li>` elements, we could have also added a single child `<ul>` to the `<body>` node and then started adding children to it. The code example also shows off template literals which were introduced in ES6: they allow us to plug variables (here: `ul.childElementCount+1`) into strings (demarked with backticks) in a more readable manner.

We can of course also remove elements 👇:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 2 (Removal)</title>
        <script>
          window.onload = function() {
            document.getElementById("bRemoveF").onclick =
removeFirstChild;
            document.getElementById("bRemoveL").onclick = removeLastChild;
          };

          function removeLastChild() {
            var ul = document.getElementById('u');
            if(ul.childElementCount>0)
              ul.removeChild(ul.lastElementChild);
          }

          function removeFirstChild() {
            var ul = document.getElementById('u');
            if(ul.childElementCount>0)
              ul.removeChild(ul.firstElementChild);
          }
        </script>
    </head>

    <body>
        <button id="bRemoveF">Remove first child</button>
        <button id="bRemoveL">Remove last child</button>
        <ul id="u">
          <li>Item 1</li>
          <li>Item 2</li>
          <li>Item 3</li>
          <li>Item 4</li>
          <li>Item 5</li>
        </ul>
    </body>
</html>
```

Important to note here is that there are often methods available for DOM elements which look similar, but are leading to quite different behaviors. Case in point: in the example ☝️ we used `ul.firstElementChild` and `ul.lastElementChild`. Instead, we could have also used `ul.firstChild` and `ul.lastChild`. And this will work to - ***at least with every second click***, as those methods also keep track of a node's children that are comments or text nodes, instead of just `li` nodes as we intend with our code.

‼️ Example 3: `this`

Event handlers are bound to the attached element's objects and the handler function "knows" which element it is listening to (the element pointed to by `this`). This simplifies programming as a function can serve different objects.

Imagine you want to create a multiplication app that has one text input box and three buttons, each with an arbitrary number on it. A click on a button multiplies the number found in the input with the button's number.

We could write three different functions and then separately attach each of them to the correct button 👇:

```
document.getElementById("button10").onclick = computeTimes10;
document.getElementById("button23").onclick = computeTimes23;
document.getElementById("button76").onclick = computeTimes76;
```

This is tedious, error prone and not maintainable (what if you need a hundred buttons). We could also be tempted to use the following construct 👇:

```
document.getElementById("button10").onclick = computeTimes(10);
document.getElementById("button23").onclick = computeTimes(23);
document.getElementById("button76").onclick = computeTimes(76);
```

but this will not work either, as in this case ☝️ the JavaScript runtime will parse each line will immediately execute the `computeTimes` function instead of attaching it to the click event.

We can avoid code duplication with the use of `this` 👇:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 3</title>
        <script>
          window.onload = function() {
            document.getElementById("button10").onclick = computeTimes;
            document.getElementById("button23").onclick = computeTimes;
            document.getElementById("button76").onclick = computeTimes;
          };
```

```
            function computeTimes() {
              /*
               * this.innerHTML returns to us "N times",
               * parseInt() then strips out the " times" suffix
               * as it stops parsing at an invalid number character
               */
              var times = parseInt(this.innerHTML);
              var input =
parseFloat(document.getElementById("input").value);
              var res = times * input;
              alert("The result is " + res);
            }
        </script>
    </head>

    <body>
        <input type="text" id="input">
        <button id="button10">10 times</button>
        <button id="button23">23 times</button>
        <button id="button76">76 times</button>
    </body>
</html>
```

☝ Depending on which button is clicked, `this` refers to the corresponding DOM tree element and `.innerHTML` allows us to examine the label text. The `parseInt` function is here used to strip out the " times" string suffix, forcing a conversion to type `number`.

‼ Example 4: mouse events

A number of different mouse events exist (`mouseup`, `mousedown`, `mousemove`, ...) and some are defined as a series of simpler mouse events, e.g.

- A click of the mouse button in-place consists of:
    1. `mousedown`
    2. `mouseup`
    3. `click`
- A click of the mouse button while moving the mouse ("dragging") consists of:
    1. `mousedown`
    2. `mousemove`
    3. ...
    4. ..
    5. `mousemove`
    6. `mouseup`

Let's look at an example 👇 of `mouseover` and `mouseout`. A timer starts and remains active as long as the mouse pointer hovers over the button element and it resets when the mouse leaves the element. Each of the three buttons has a different timer speed:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 4</title>
        <script>
          window.onload = function() {
            document.getElementById("b1").onmouseover = mouseover;
            document.getElementById("b1").onmouseout = mouseout;

            document.getElementById("b10").onmouseover = mouseover;
            document.getElementById("b10").onmouseout = mouseout;

            document.getElementById("b100").onmouseover = mouseover;
            document.getElementById("b100").onmouseout = mouseout;
          };

          var intervals = {};

          function updateNum(button){
            var num = parseInt(button.innerHTML);
            num = num + 1;
            button.innerHTML = num;
          }

          function mouseover() {
            var incr = parseInt(this.id.substr(1));
            intervals[this.id] = setInterval(updateNum, 1000/incr, this);
          }

          function mouseout()
          {
            clearInterval(intervals[this.id]);
            this.innerHTML = "0";
          }
        </script>
    </head>

    <body>
      <button style="width:500px" id="b1">0</button>
      <br>
      <button style="width:500px" id="b10">0</button>
      <br>
      <button style="width:500px" id="b100">0</button>
    </body>
</html>
```

Mouse events can be tricky, the more complex ones are not consistently implemented across browsers.

‼️ **Example 5: a crowdsourcing interface**

Here is another event that can be useful, especially for text-heavy interfaces: `onselect`. Here 👇, we have an interface with a read-only text that the user can select passages in. If enough passages have been selected, the user can submit the selected passages:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 5</title>
        <script>
          window.onload = function() {
            document.getElementById("ta").onselect = updateNuggets;
          };

          function updateNuggets() {
            var n1 = document.getElementById('n1').value;
            var n2 = document.getElementById('n2').value;
            var n3 = document.getElementById('n3').value;

            var selected = null;
            var myTextArea = document.getElementById('ta');
            if (myTextArea.selectionStart != undefined)
            {
              var p1 = myTextArea.selectionStart;
              var p2 = myTextArea.selectionEnd;
              selected = myTextArea.value.substring(p1, p2);
            }

            //if the selected phrase is already in a nugget, remove it
            if(selected==n1) {document.getElementById('n1').value = "";}
            else if(selected==n2){document.getElementById('n2').value =
"";}
            else if(selected==n3){document.getElementById('n3').value =
"";}
            //if the first nugget is empty, add it
            else if(n1.length==0){document.getElementById('n1').value =
selected;}
            //if the second nugget is empty, add it
            else if(n2.length==0){document.getElementById('n2').value =
selected;}
            //third nugget is treated differently, as now the button
becomes unhidden
            else if(n3.length==0)
            {
              document.getElementById('n3').value = selected;
              document.getElementById('b').hidden = false;
            }
            else {
              alert('You have selected three information nuggets. Either
unselect one or manually empty text box.');
            }
          }
        </script>
```

```
    </head>

    <body>
      <form>
          <p><em>Task: Write down / mark the 3 most important information nuggets</em>
            <br>

            <textarea id="ta" cols="50" rows="5" readonly>
                "Hobey Baker (1892-1918) was an American amateur athlete of the early
                twentieth century, widely regarded by his contemporaries as one of the
                best athletes of his time."
            </textarea>
            <br>

            <label>Nugget 1: <input type="text" id="n1" autocomplete="off"></label>
            <br>
            <label>Nugget 2: <input type="text" id="n2" autocomplete="off"></label><br>
            <label>Nugget 3: <input type="text" id="n3" autocomplete="off"></label><br>
            <button hidden="hidden" id="b">Submit Answers</button><br>
        </form>
    </body>
</html>
```

‼️ Example 6: a typing game

The last example is a typing game 👇. Given a piece of text, type it correctly as fast as possible. The interface records how many seconds it took to type and alerts the user to mistyping. In this example we make use of the `keypress` event type. We start the timer with `setInterval` (incrementing once per second), which returns a handle that we can later pass to `clearInterval` to stop the associated callback from executing (thus stopping the clock).

In this example we do do make slight use of CSS (to flash a red background and alter the color of the timer in the end), you can recognize those line on the `.style` properties.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example 6</title>
        <script>
          window.onload = function() {
            document.getElementById("typed").onkeypress = checkTextAtKeyPress;
          };
```

```
            var currentPos = 0;
            var givenText = "given";
            var typedText = "typed";
            var timerLog = "timer";
            var intervals = {};

            //e refers to the event (we need it to extradt the char typed)
            function checkTextAtKeyPress(e) {

              var textToType = document.getElementById(givenText).value;

              //we reached the end, do nothing
              if(currentPos >= textToType.length) {return;}

              var nextChar = textToType.charAt(currentPos);

              var keyPressed = String.fromCharCode(e.which);
              console.log("Key pressed: "+keyPressed+", charCode:
"+e.which);

              //correct key was pressed
              if(nextChar==keyPressed) {
                //CSS is used here to "style" the text box

document.getElementById(typedText).style.backgroundColor="rgb(255,255,255)
";
                document.getElementById(typedText).value =
textToType.substring(0,currentPos+1);

                currentPos++;

                //first time key was pressed, start counter
                if(currentPos==1) {
                  intervals[this.id]=setInterval(function(){
                      var t =
parseInt(document.getElementById(timerLog).innerHTML);
                      t = t + 1;
                      document.getElementById(timerLog).innerHTML = t +"
seconds";
                  }, 1000);
                }

                //we reached the end
                if(currentPos==textToType.length) {
                  clearInterval(intervals[this.id]);
                  //CSS is used here to "style" the text box
                  document.getElementById(timerLog).style.color="orange";
                }
              }
              //incorrect key
              else {
                //CSS is used here to "style" the text box

document.getElementById(typedText).style.backgroundColor="rgb(255,100,100)
```

```
";
                    }
                }
            </script>
        </head>

        <body>
          <form id="f">
              <p>
                <em>Task: Type out the following text correctly:</em>
                <br>

                <textarea id="given" cols="50" rows="5" readonly=""
autocomplete="off">
                    H. Baker was an American amateur athlete of the 20th
century.</textarea>
                <br>

                <textarea id="typed" cols="50" rows="5" readonly=""
autocomplete="off"></textarea>
                <br>

                <span id="timer">0 seconds</span>
              </p>
          </form>
        </body>
</html>
```

To conclude this DOM section, here is an overview of important keyboard and text events:

Event	Description
`blur`	element loses keyboard focus
`focus`	element gains keyboard focus
`keydown`	user presses key while element has keyboard focus
`keypress`	user presses and releases key while element has keyboard focus (a problematic event)
`keyup`	user releases key while element has keyboard focus
`select`	user selects text in an element

Self-check

Here are a few questions you should be able to answer after having followed the lecture and having worked through the required readings:

1. After executing the JavaScript code snippet below in the browser, what is the output on the console?

```
var f = ( function myfunc1(a){
    var c = 2 * a;
    return function myfunc2(b){
        return 3 * c;
    }
})(5);

console.log(f(4));
```

2. After executing the JavaScript code snippet below in the browser, how many of `a`, `b`, `c` and `d` have global scope (they become a property of the `window` object)?

```
var a = 5;
b = 10;

function outer(a){
    c = 0;
    d = 1;

    function inner(d){
        c = 12;
    }
    inner(5);
    var c, d;
}
outer(6);
```

3. After executing the JavaScript code snippet below in the browser, what is the output on the console?

```
function Habit(habit){
    this.habit = habit;
}

function WeeklyHabit(habit, timesPerWeek){
    Habit.call(this, habit);
    this.timesPerWeek = timesPerWeek;
}

WeeklyHabit.prototype = Object.create( Habit.prototype );
WeeklyHabit.prototype.constructor = WeeklyHabit;

Habit.prototype.updateFreq = function(f){ this.freq = f; }
WeeklyHabit.prototype.updateFreq = function(f){ this.freq = f + " times";
}

var h1 = new Habit("Go swimming");
var h2 = new WeeklyHabit("Eat healthily", "5");
```

```
h1.updateFreq(1);
h2.updateFreq(2);
console.log(h1.freq);
console.log(h2.freq);
```

4. What is the output on the web console when running the following piece of JavaScript in the browser?

```
function A(x){
    var y = x * 2;
    return function(y){
        var z = y * 3;
        return function(z){
            return x + y + z;
        }
    }
}
console.log( A(3)(4)(5) );
```

5. Which of the following statements about the basic constructor in JavaScript is **wrong**?

- Objects share functions.
- All members are public.
- An object constructor looks like a normal function.
- Prexing a call to a function with the keyword `new` indicates to the JavaScript runtime that the function should behave like a constructor.

6. What is the output on the web console when running the following piece of JavaScript in the browser?

```
var todoModule = ( function() {
    var numTodos = 0;

    return {
        incrNumTodos: function(){
            numTodos++;
        },
        decrNumTodos: function(){
            if(numTodos>0){
                numTodos--;
            }
        },
        printTodos: function(){
            console.log(numTodos);
        }
    };
});

for(let i=0; i<5; i++){
    todoModule.incrNumTodos();
```

```
}
todoModule.printTodos();
```

7. After executing the JavaScript code snippet below in the browser what will be the console output?

```
var message = "Toy kitchen";
var price = "89.90";

var deal1 = {
    message: "Peppa Pig",
    details: {
        price: "29.95",
        getPrice: function(){
            console.log(this.price);
        }
    }
}

var a = deal1.details.getPrice;
a();
```

8. After executing the JavaScript code snippet below in the browser what will be the console output?

```
var message = "Toy kitchen";
var price = "89.90";

var deal1 = {
    message: "Peppa Pig",
    details: {
        price: "29.95",
        getPrice: function(){
            console.log(this.price);
        }
    }
}

deal1.details.getPrice();
```

9. In a prototype version of one of our applications, we want to implement functionality in JavaScript that retrieves details of local deals from the server when a user clicks on a deal. The code below contains a first implementation of this functionality. What is the main issue of this code?

- Reloading the web page *n* times will lead to *n* listeners being attached to the same list item.
- No event listeners will be attached to click events on list items.
- The JavaScript code will be executed before teh DOM tree is loaded, the event listeners will be attached to the `window` object instead of the list items.
- The method `document.getElementsByTagName()` does not exist leading to an error in the script.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Local Deals Finder</title>
        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>
        <script>
            var main = function(){
                var list = document.getElementsByTagName("li");
                for(var i = 0; i<list.length; i++){
                    document.getElementsByTagName("li")[i].onclick =
showDetails();
                }
            }
            $(document).ready(main);

            function showDetails(){
                /* implementation of show details functionality */
                console.log("Showing some details ...");
            }
        </script>
    </head>
    <body>
        <div id="deals">
            <ul>
                <li id="i1" data-id="122" data-latitude="12.43" data-
longitude="44.31" data-price="29.90">Toy car</li>
                <li id="i2" data-id="342" data-latitude="13.01" data-
longitude="43.21" data-price="14.55">Perfume</li>
            </ul>
        </div>
    </body>
</html>
```