howest
hogeschool

Web Technology: Introduction

Dieter Mourisse

Who am I

Dieter Mourisse

- Master in Mathematical Computer Science
- Background in Algorithms
- Sixth year in Howest where I started to focus on Web Development

howest
hogeschool

Course content

Purpose of the course: making more advanced and easy to use web applications.

Web Technology (**Mr Mourisse** / Mr Tack / Mr Casier)

- Advanced Javascript
- Promises
- Async / Await
- TypeScript
- Advanced CSS
- Responsive Design
- HTML Canvas
- SVG
- Web Sockets
- HTML5 APIs
- Graphs and Maps
- Express server

You will be able to use topics discussed in this course in the Analysis and Development project.

Web Technology

- Advanced Javascript
- Advanced CSS
- **TypeScript**
- webpack
- Responsive Design
- HTML Canvas
- **Web Sockets**
- **HTML5 APIs**
- **Graphs and Maps**

User Experience

- User Experience Design
- User Interface Design

Teaching method

- Blended
  - Advanced Javascript / CSS topics
    - Regular input sessions + exercises (Friday)
  - Flipped classroom
    - Independently process tutorials

Shedule

		Les	vrijdag	tutorial
26 sep	1	1	Advanced JS	Graphs and Maps
3 okt	2	2	Closures/OOP	Web API
10 okt	3	3	Advanced CSS	Canvas + SVG
17 okt	4			
24 okt	5	4	Promises	Websockets
14 nov	8	5	Async/Await	
21 nov	9	6	Responsive Design	Express Server
28 nov	10	7	Typescript	
5 dec	11	8	Animations	
12 dec	12			

Input sessions

Exercises

Tutorials

Werken aan project

Changes can happen (I will let you know on Leho)

howest hogeschool

What you need to know

- HTML
- CSS
- JavaScript
- C# (for websockets)

What software will we use

- A code editor (I use Visual Studio code)
- A browser (I use Firefox)
- Node.js
- A C# IDE (I use Rider)

None of these are mandatory, if you are more comfortable with something else, you are free to use that. I however will probably not be able to answer questions about it.

howest hogeschool

howest
hogeschool

Evaluation

Evaluatie(s) voor de eerste examenkans

Moment	Vorm	%	Opmerking
examenperiode 1 (1e sem) binnen examenrooster	examen: andere vorm of combinatie van vormen	60,00	Examen geavanceerde Javascript & CSS. Voor je eindscore wordt het gewogen harmonisch gemiddelde genomen van de 2 onderdelen.
examenperiode 1 buiten examenrooster	opdracht: andere vorm of combinatie van vormen	40,00	Werkstuk web. Voor je eindscore wordt het gewogen harmonisch gemiddelde genomen van de 2 onderdelen.

Project tutorials

weighted harmonic mean

https://en.wikipedia.org/wiki/Harmonic_mean

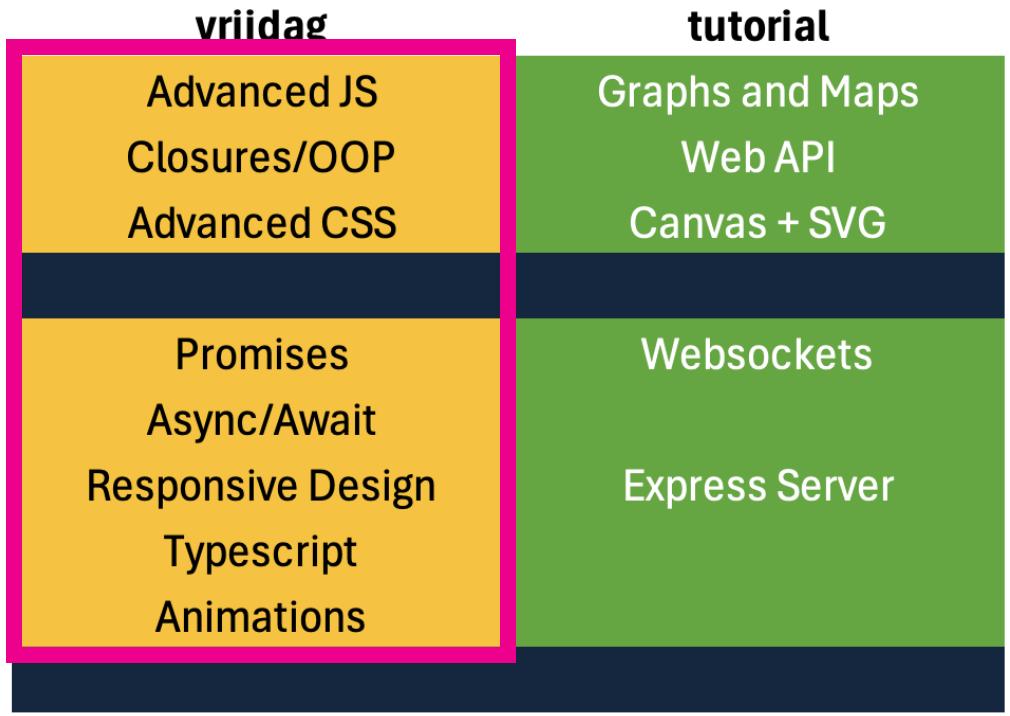
$$\frac{1}{\frac{0.4}{s_1} + \frac{0.6}{s_2}}$$

project

examen

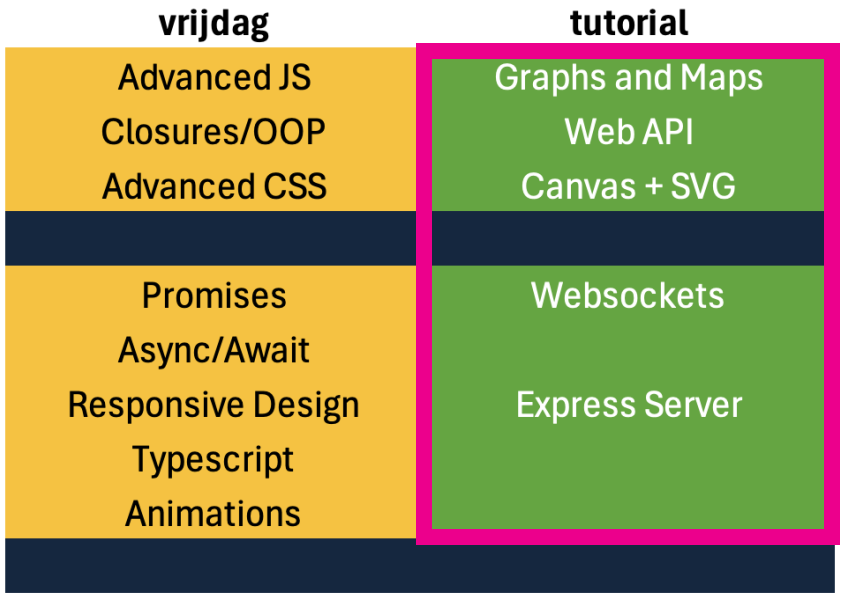
howest hogeschool

Exam (60%)


vrijdag
tutorial
Advanced JS
Closures/OOP
Advanced CSS
Graphs and Maps
Web API
Canvas + SVG
Promises
Async/Await
Responsive Design
Typescript
Animations
Websockets
Express Server

Project tutorials (40%)

- There are 5 tutorials to be made (10%)
- Knowledge from tutorials need to be applied in project (30%)



Making tutorials (10%)

- Tutorials can be found here:
  - https://gitlab.ti.howest.be/ti/2024-2025/s3/web-technology/tutorials
- Clone them and create your own repositories in your own group
  - https://gitlab.ti.howest.be/ti/2024-2025/s3/web-technology/students
- **Deadline: December 14th**

Tutorials project (30%)

- Knowledge from tutorials need to be applied in an **individual** project
- Project can be start before making all tutorials
- Project assignment on Leho
- **Deadline: January 4th**

howest
hogeschool

The <script> tag

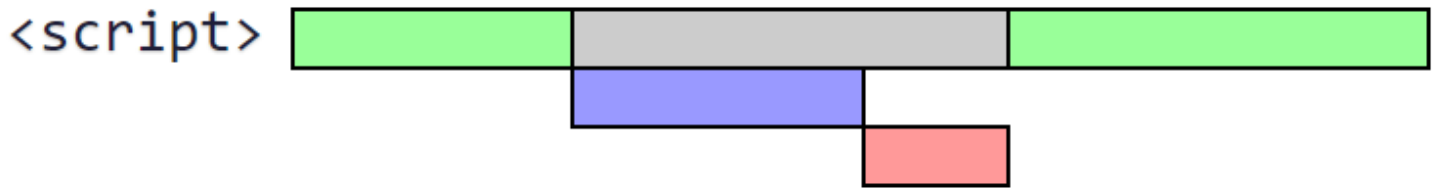
Do these ring a bell?

```
document.addEventListener("DOMContentLoaded", init);
```

```
<script src="assets/js/index.js" type="module"></script>
```


howest hogeschool

Loading a regular script (no module)



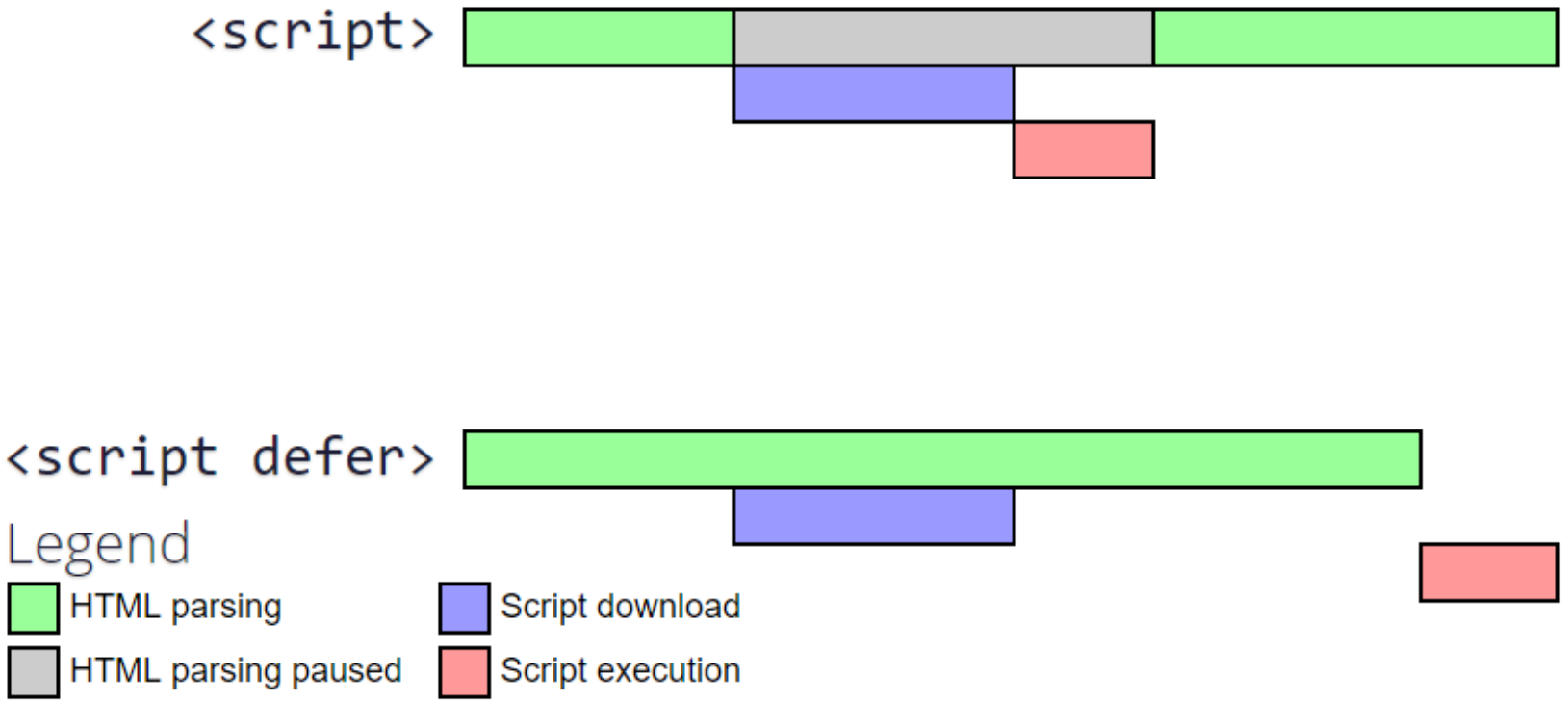
The defer attribute

```
<!DOCTYPE html>
<html lang="en">
<head>
   <title>Document</title>
   <script src="assets/js/regular-script.js" defer></script>
</head>
<body>
   <div id="content"></div>
</body>
</html>
```

The script will wait to run until the full html page has been parsed. So there is no need for the *DOMContentLoaded* event in *regular-script.js*.

DOMContentLoaded will wait firing until all defer scripts have been executed, normally you do not want this.

Async vs defer



The defer attribute

No longer use *DOMContentLoaded*, use the *defer* attribute instead.

Scripts with *type="module"* are deferred by default.

https://www.youtube.com/watch?v=_iq1fPjeqMQ

howest hogeschool

The async attribute

```
<script src="assets/js/regular-script.js" async></script>
```

`async`

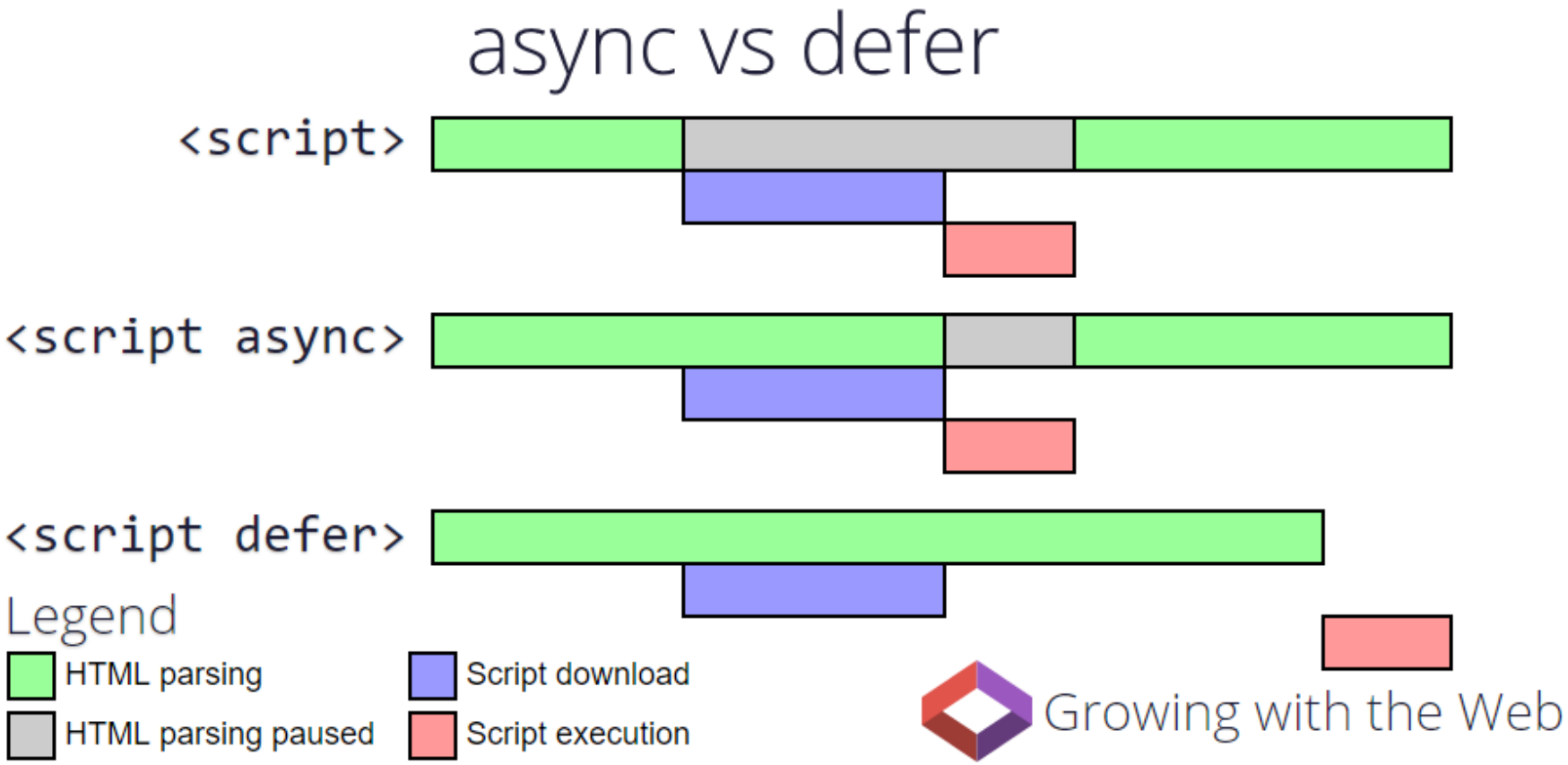
For classic scripts, if the `async` attribute is present, then the classic script will be fetched in parallel to parsing and evaluated as soon as it is available.

For module scripts, if the `async` attribute is present then the scripts and all their dependencies will be executed in the defer queue, therefore they will get fetched in parallel to parsing and evaluated as soon as they are available.

This attribute allows the elimination of **parser-blocking JavaScript** where the browser would have to load and evaluate scripts before continuing to parse. `defer` has a similar effect in this case.

https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script#attr-async

Async vs defer



howest
hogeschool

Node.js

What is Node.js

outside the browser

Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

https://nodejs.org/en

howest hogeschool

basic-script.mjs

```
// Define a simple function that adds two numbers
function addNumbers(a, b) {
    return a + b;
}

// Use the function and log the result
const num1 = 5;
const num2 = 10;
const result = addNumbers(num1, num2);
console.log(`The sum of ${num1} and ${num2} is: ${result}`);
```

Terminal

```
bash-3.2$ node basic-script.js
The sum of 5 and 10 is: 15
```

What is npm

Node Package Manager

npm allows developers to install, update, and manage third-party libraries or packages that are published to the npm registry. These packages can range from small utility libraries to full-fledged frameworks.

Incorporate existing code (written by someone else) into your project to avoid reinventing the wheel.

https://www.npmjs.com/

Package	Description
Debug	Simple logging utility for debugging Node.js applications.
Async	Utility for managing asynchronous JavaScript functions.
Lodash	Utility library for working with arrays, objects, and strings.
ESLint	Linting tool to identify and fix JavaScript code issues.
Express	Minimal web framework for building Node.js applications.
Moment	Library for parsing and formatting dates and times.
Chai	Assertion library for testing with Mocha.
Mocha	Testing framework for running JavaScript tests.
Nodemon	Tool that restarts Node.js automatically when files change.
Socket.IO	Enables real-time, bidirectional communication between client and server.

https://www.capitalnumbers.com/blog/top-nodejs-packages/

Turn a directory into an npm project

Terminal

```
bash-3.2$ npm init -y
```

This creates a package.json file that contains the project settings and all its dependencies.

package.json

```
{
  "name": "temp",
  "version": "1.0.0",
  "main": "basic-script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Hint: the *–y* flag automatically generates a file with default settings bypassing the interactive prompts

Install npm packages

Terminal

```
bash-3.2$ npm install fs
```

Installs the fs module that contains file io functionality

package.json

```
{
  "name": "temp",
  "version": "1.0.0",
  "main": "basic-script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "fs": "^0.0.1-security"
  }
}
```

Dependencies are stored in the node_modules directory, which is usually listed in .gitignore to prevent it from being pushed to Git.

howest hogeschool

Install all dependencies

Terminal

```
bash-3.2$ npm install
```

Installs all dependencies from package.json in the node_modules directory

package.json

```
{
  "name": "temp",
  "version": "1.0.0",
  "main": "basic-script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "fs": "^0.0.1-security"
  }
}
```

Typically done after a fresh clone of a gitlab repository

howest hogeschool

Use a dependency

basic-script.js

```
// Import the built-in 'fs' module for file operations
import fs from 'fs';

// Define a simple function that adds two numbers
function addNumbers(a, b) {
    return a + b;
}

// Use the function and log the result
const num1 = 5;
const num2 = 10;
const result = addNumbers(num1, num2);
console.log(`The sum of ${num1} and ${num2} is: ${result}`);

// Example of writing the result to a file
fs.writeFile('result.txt', `The sum of ${num1} and ${num2} is: ${result}`, (err) => {
    if (err) throw err;
    console.log('The result has been saved to result.txt');
});
```


howest hogeschool

Terminal

```
bash-3.2$ node basic-script.js
The sum of 5 and 10 is: 15
(node:58037) [MODULE_TYPELESS_PACKAGE_JSON] Warning:
file:///Users/dieter/Desktop/temp/basic-script.js parsed as an ES module because module syntax
was detected; to avoid the performance penalty of syntax detection, add "type": "module" to
/Users/dieter/Desktop/temp/package.json
(Use `node --trace-warnings ...` to show where the warning was created)
The result has been saved to result.txt
```

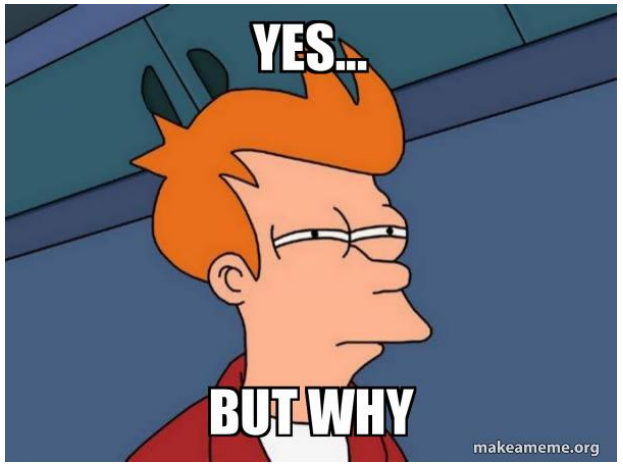
howest hogeschool

Terminal

```
bash-3.2$ node basic-script.js
The sum of 5 and 10 is: 15
(node:58037) [MODULE_TYPELESS_PACKAGE_JSON] Warning:
file:///Users/dieter/Desktop/temp/basic-script.js parsed as an ES module because module syntax
was detected; to avoid the performance penalty of syntax detection, add "type": "module" to
/Users/dieter/Desktop/temp/package.json
(Use `node --trace-warnings ...` to show where the warning was created)
The result has been saved to result.txt
```

package.json

```
{
  "name": "temp",
  "version": "1.0.0",
  "main": "basic-script.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "fs": "^0.0.1-security"
  },
  "type": "module"
}
```



howest hogeschool

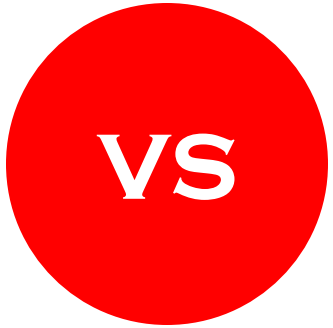
package.json

```
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "fs": "^0.0.1-security"
  },
  "type": "commonjs"
}
```

package.json

```
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "fs": "^0.0.1-security"
  },
  "type": "module"
}
```

CommonJS



ES Modules

```
const fs = require('fs');

module.exports = {
   addNumbers
}
```

```
import { format } from 'date-fns';

export { addNumbers }
```

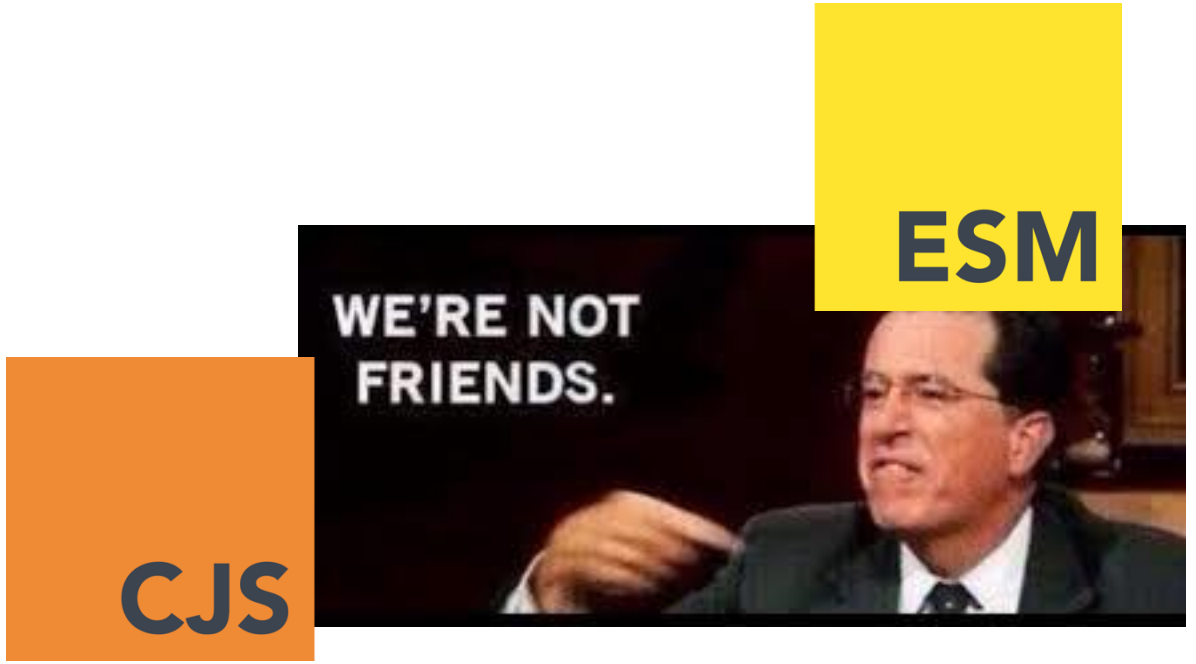

howest hogeschool

ESM and CJS are completely different animals. Superficially, ESM looks very similar to CJS, but their implementations couldn't be more different. One of them is a honey bee, and the other is a murder hornet.

A hornet and a honey bee. One of these is like ESM and the other is like CJS, but I can never remember which one is which. Credit: wikimedia, wikimedia

howest hogeschool

JavaScript has two separate module systems and they don’t work that well together



The good news

we can

The bads news

others don’t

howest hogeschool

- It’s a lot of work to switch your code base over
- When you switch, all packages that have you as a dependency must also switch
- Don’t fix it if it ain’t broken
- Some people have strong opinions

My rules

Always use ES Modules

- When creating a new project, set *type* to *module* in *package.json*
- Ideally your node scripts have the extension of *.mjs*
- **Never** use *require*, only use *import*!

You will however find a lot of documentation using *require*

howest hogeschool

Changing a require to an import

CommonJS modules consist of a `module.exports` object which can be of any type.

When importing a CommonJS module, it can be reliably imported using the ES module default import or its corresponding sugar syntax:

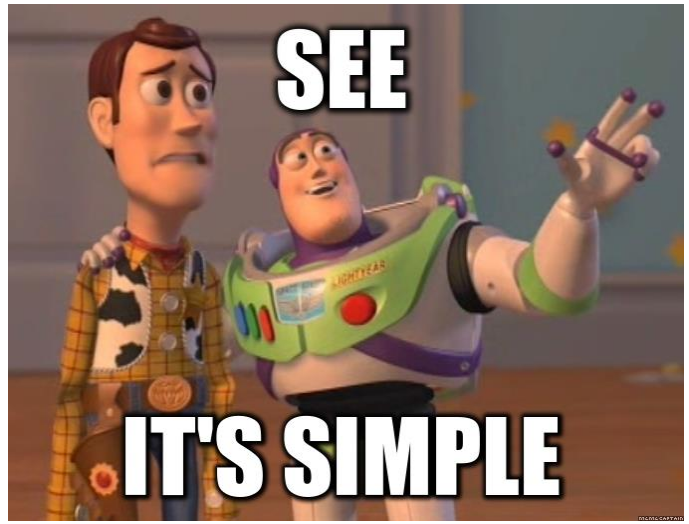
```
import { default as cjs } from 'cjs';

// The following import statement is "syntax sugar" (equivalent but sweeter)
// for `{ default as cjsSugar }` in the above import statement:
import cjsSugar from 'cjs';

console.log(cjs);
console.log(cjs === cjsSugar);
// Prints:
//   <module.exports>
//   true
```

https://nodejs.org/dist/latest/docs/api/esm.html#esm_commonjs_namespaces

howest hogeschool

```
const fs = require('fs');
```

```
import fs from 'fs';
```



howest
hogeschool

Web Technology: Declaring variables

Dieter Mourisse

Declaring variables in JavaScript

keyword	
var	depecrated, don’t use this
let	variables who’s content will change
const	variables who’s content will not change

howest hogeschool

howest
hogeschool

Scope

Scope

Wikipedia

> *In computer programming, the scope of a name binding an association of a name to an entity, such as a variable is the region of a computer program where the binding is valid: where the name can be used to refer to the entity. Such a region is referred to as a scope block. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).*

Which basically means

> *The places in the code where a declared variable can be accessed.*

Types of scope

- Global scope
- Local scope
- Block scope

var

The scope of a variable declared with var is its current *execution context and closures thereof*, which is either the enclosing function and functions declared within it, or, for variables declared outside any function, global.

Duplicate variable declarations using var will not trigger an error, even in strict mode, and the variable will not lose its value, unless another assignment is performed.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

let

Variables declared by **let** have their scope in the block for which they are declared, as well as in any contained sub-blocks. In this way, **let** works very much like **var**. The main difference is that the scope of a **var** variable is the entire enclosing function

Redeclaring the same variable within the same function or block scope raises a SyntaxError.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let

Global scope

```
var name1 = "Alice";
let name2 = "Bob";

function sayHello() {
    console.log(`Hello ${name1}`)
    console.log(`Hello ${name2}`)
    console.log(`Hello ${window.name1}`)
    console.log(`Hello ${window.name2}`)
}
```

```
Hello Alice
Hello Bob
Hello Alice
Hello undefined
```

Both **name1** and **name2** are in the global scope (accessible everywhere in your script), but only *name1* is added to the global object (*window* in the browser). This is because it was declared using var.

howest hogeschool

Local scope

A variable in the local scope is only accessible inside the function that is running. A new local scope is created when a function call happens.

```
function sayHello(name) {
    let age = 24;
    console.log(`Hello ${name}`)
}

sayHello("Alice");
sayHello("Bob");
sayHello("Carol");
```

A new local scope is created, it will immediately contain the variable name with value *“Alice”*.

Block scope

A block scoped variable is a variable only accessible in a block (between {}). Did not exist prior to let.

```
function count() {
    for (var i = 0; i < 10; i++) {
        console.log(i);
    }
    console.log("Still know what i is about", i
);
}
count();
```

```
0
1
2
3
4
5
6
7
8
9
Still know what i is about 10
```


howest hogeschool

Block scope

A block scoped variable is a variable only accessible in a block (between { en }. Did not exist prior to let.

```
function count() {
    for (let i = 0; i < 10; i++) {
        console.log(i);
    }
    console.log("Still know what i is about", i);
}
count();
```

```
0
1
2
3
4
5
6
7
8
9
▶ Uncaught ReferenceError: i is not defined
    at count (main.js:7)
    at main.js:9
```


howest hogeschool

howest
hogeschool

Hoisting

Declaration hoisting

MDN: Conceptually, for example, a strict definition of hoisting suggests that variable and function **declarations** are physically moved to the top of your code, but this is not in fact what happens. Instead, the variable and function declarations are put into memory during the *compile* phase, but stay exactly where you typed them in your code.

Hoisting of functions

```
catName("Tiger");

function catName(name) {
    console.log("My cat's name is " + name);
}
```

For functions, we are used to this.

```
function catName(name) {
    console.log("My cat's name is " + name);
}

catName("Tiger");
```

Hoisting of variables

```
a = 5;
console.log(a);

var a;
```

```
var a;

a = 5;
console.log(a);
```

For variables this is a bit weird

howest hogeschool

Hoisting of variables: only declaration

```
console.log(a);
var a = 5;
```

→ undefined

is actually

```
var a;
console.log(a);
a = 5;
```

This is just confusing and should not be possible.

howest hogeschool

Variables are only hoisted when they are declarated with *var*, not when they are declarated with *const* or *let*.

Always use **const** or **let**. Const for variables that won't change after they have been initialized, let for those that will.

howest
hogeschool

Summary

Declaring variables in JavaScript – summary

Keyword	Scope	Hoisting	Can be reassigned?	Can be redeclared?
var	function scope	yes	yes	yes
let	block scope	no	yes	no
const	block scope	no	no	no

- Commonly accepted practice:
  - Use **const** as much as possible
  - Use **let** in case of loops and reassignment
  - Avoid **var**, except for legacy code

howest
hogeschool

Web Technology: JavaScript is weird

Dieter Mourisse

Guess the output

```
const a = {} + {};
const b = {} + [];
const c = [] + {};
const d = [] + [];

const e = Array(16).join('wat' + 1);
const f = Array(16).join('wat' - 1);
const g = parseInt(0.5);
const h = parseInt(0.0000005);
```

Type coercion

```
> {} + {}
'[object Object][object Object]'
> {} + []
0
> [] + {}
'[object Object]'
> [] + []
''
> Array(16).join('wat'+1)
'wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1'
> Array(16).join('wat'-1)
'NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN'
```

Type coercion is the process of converting a value from one type to another.

Javascript sometimes does this automatically, based on the operator used.

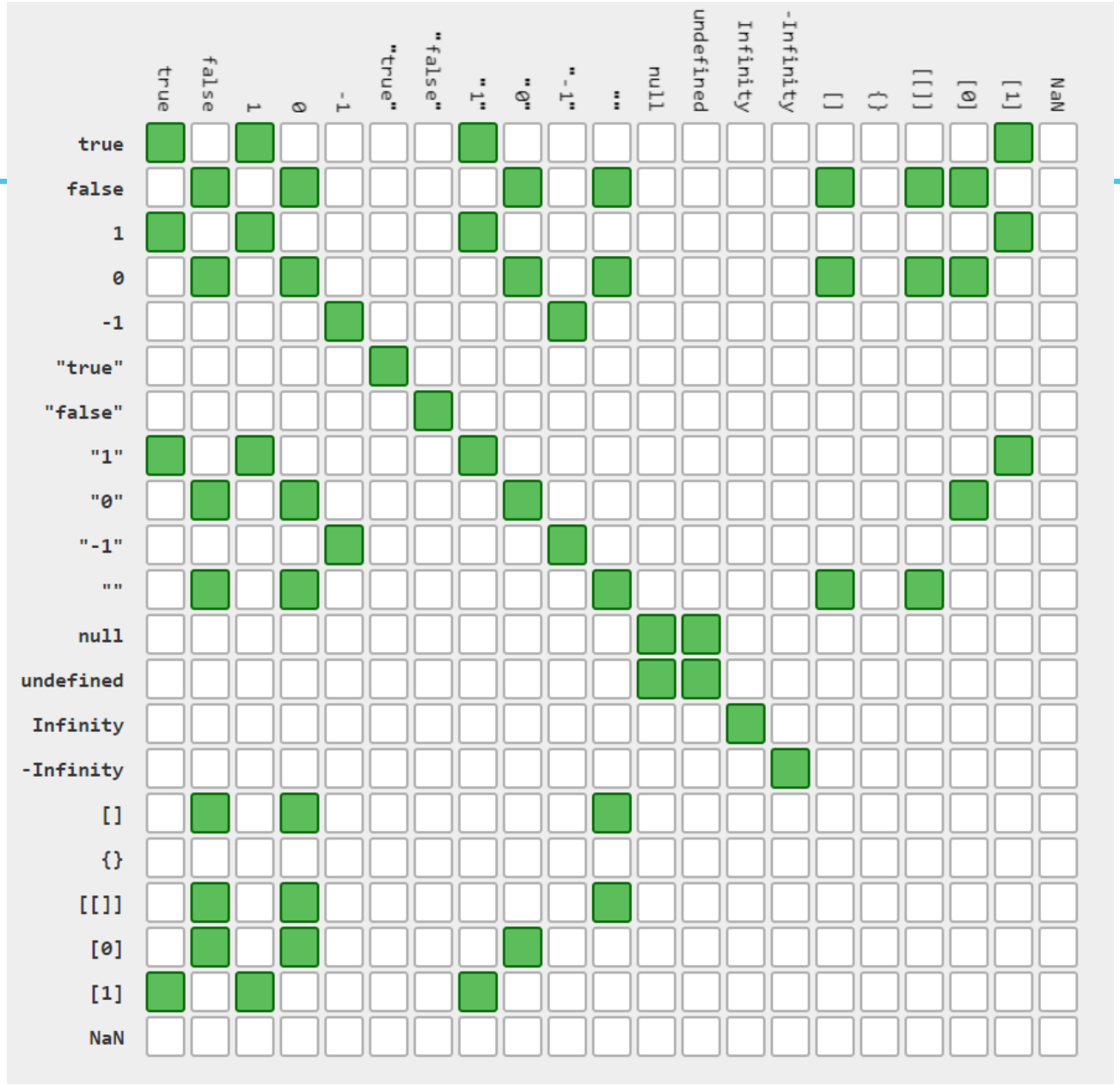
The '+'-operator is defined for two numbers and two strings. So here it will convert arguments to strings and numbers.

[] is converted to 0 or the empty string
{} is converted to 0 or the string '[object Object]'

Equality testing

This is why comparsions should **always** happen with === instead of ==. Two equal signs will do type conversion.

```
2 === '2'
false
2 == '2'
true
```



howest hogeschool

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```


howest hogeschool

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```

```
['1', '7', '11'].map(parseInt)
▶ (3) [1, NaN, 3]
```


howest hogeschool

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```

```
['1', '7', '11'].map(parseInt)
▶ (3) [1, NaN, 3]
```

- The map function actually expects a function with 3 parameters
- **When you call a function with extra parameters, they just get *ignored***

```
function myFunction(parameter) {
    console.log(parameter)
}
```

```
myFunction('a', 'b')
```

```
a
```


howest hogeschool

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```

```
['1', '7', '11'].map(parseInt)
▶ (3) [1, NaN, 3]
```

- The map function actually expect a function with 3 parameters
- When you call a function with extra parameters, they just get ignored
- **The parseInt function actually has two parameters. The number to convert and the radix the number is in.**
  - When you call a function with less parameters, the missing ones will be undefined

```
parseInt('100', 2)
4
```

```
function myFunction(parameter1, parameter2) {
    console.log(parameter1, parameter2)
}

myFunction('a')
a undefined
```


howest hogeschool

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```

```
['1', '7', '11'].map(parseInt)
▶ (3) [1, NaN, 3]
```

- The map function actually expect a function with 3 parameters
- When you call a function with extra parameters, they just get ignored
- The parseInt function actually has two parameters. The number to convert and the radix (base) the number is in.
  - When you call a function with less parameters, the missing ones will be undefined
- So the parseInt function actually is called like this during the map

Guess the output

```
const g = ['1', '7', '11'].map(parseInt);
```

```
['1', '7', '11'].map(parseInt)
▶ (3) [1, NaN, 3]
```

```
parseInt('1', 0, ['1', '7', '11'])
parseInt('1', 0) // 0 is not a valid radix, so the default value of 10 is used
1

parseInt('7', 1, ['1', '7', '11'])
parseInt('7', 1) //7 is not a valid number in base 1
NaN

parseInt('11', 2, ['1', '7', '11'])
parseInt('11', 2) // 11 means 3 in binary (base 2)
3
```

Guess the output

```
date = new Date(2023, 7, 31);
console.log(date);

date.setMonth(1);
console.log(date);
```

Guess the output

```
date = new Date(2023, 7, 31);
console.log(date);          Date Thu Aug 31 2023 00:00:00 GMT+0200 (Central European Summer Time)

date.setMonth(1);
console.log(date);
```

- JavaScript starts counting months at 0. So January is month 0

Guess the output

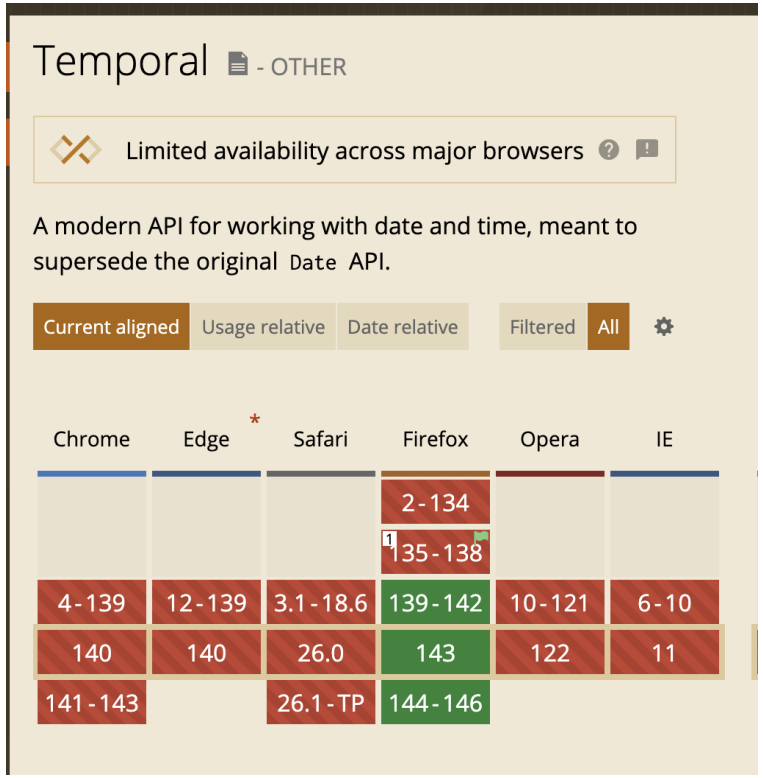
```
date = new Date(2023, 7, 31);
console.log(date);          Date Thu Aug 31 2023 00:00:00 GMT+0200 (Central European Summer Time)

date.setMonth(1);
console.log(date);          ▸ Date Fri Mar 03 2023 00:00:00 GMT+0100 (Central European Standard Time)
```

- JavaScript starts counting months at 0. So January is month 0
- Since February 31 does not exist. Javascript adds three days to February 29, which gives March 3.

Temporal API

The temporal API will help fix this

The **`Temporal`** object enables date and time management in various scenarios, including built-in time zone and calendar representation, wall-clock time conversions, arithmetics, formatting, and more. It is designed as a full replacement for the `Date` object.



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Temporal

howest
hogeschool

Web Technology
The JavaScript Language

Dieter Mourisse

howest
hogeschool

Arguments

Function arguments

```
function func1(a, b, c) {
        console.log(a);
        console.log(b);
        console.log(c);
}
```

function *func1* seems to have three parameters

```
func1(4, 2, 3)
```

```
4
2
3
```


howest hogeschool

Function arguments

```
function func1(a, b, c) {
        console.log(a);
        console.log(b);
        console.log(c);
}
```

When the number of arguments you pass to a function is less than the number of parameters of the function, the remaining arguments will be undefined.

```
func1(4, 2)
```

```
4
2
undefined
```


howest hogeschool

Default parameters

```
function multiply(a, b = 2) {
      console.log(a * b);
}
```

Default function parameters allow named parameters to be initialized with default values if no value or `undefined` is passed.

Recommended to put default parameters last

```
multiply(4)
```

8

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

howest hogeschool

Function arguments

```
function func1(a, b, c) {
        console.log(a);
        console.log(b);
        console.log(c);
}
```

When the number of arguments you pass to a function is more than the number of parameters of the function, the remaining arguments will be thrown away?

```
func1(4, 2, 3, 1)
```

```
4
2
3
```


howest hogeschool

The arguments object

```
function func1(a, b, c) {
        for (let argument of arguments) {
                console.log(argument);
        }
}
```

`arguments` is an `Array`-like object accessible inside functions that contains the values of the arguments passed to that function.

```
func1(4, 2, 3, 1, 9)
```

```
4
2
3
1
9
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments

The arguments object

```
function sum() {
        let sum = 0;
        for (let argument of arguments) {
                sum += argument;
        }
        return sum;
}

sum(4, 2, 3, 1, 9)
```

```
function max() {
        let maximum = 0;
        for (let argument of arguments) {
                if (argument > maximum)
                        maximum = argument;
        }
        return maximum;
}

max(3, 4, 8, 5);
```

The arguments object

> **Note:** If you're writing ES6 compatible code, then rest parameters should be preferred.

Rest parameter

```
function sumWithRestParameter(...numbers) {
        let sum = 0;
        for (let number of numbers) {
                sum += number;
        }
        return sum;
}

sumWithRestParameter(2, 3);
```

The **rest parameter** syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent variadic functions in JavaScript.

Advantages of rest parameter

- You explicitly say a function takes a variable number of arguments
- You can choose your own name for the variable (it mustn't be *arguments*)
- You can combine ordinary parameters and rest parameters Rest parameter should be last
- The rest parameters are combined in a **real** array

Advantages of rest parameter

- You explicitly say a function takes a variable number of arguments
- You can choose your own name for the variable (it mustn't be *arguments*)
- You can combine ordinary parameters and rest parameters Rest parameter should be last
- The rest parameters are combined in a **real** array

```
function showName(firstname, lastname, ...titles) {
        let fullName = `${titles.join(' ')} ${firstname} ${lastname}`
        return fullName;
}

const result = showName('Mattias', "De Wael", "dr.", "ir.");
```

dr. ir. Mattias De Wael

howest
hogeschool

Spread syntax

Spread syntax

Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Spread syntax

Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

For function calls

```
function minimum(a, b) {
      return a < b ? a : b;
}

const numbers = [3, 4];
const min = minimum(...numbers);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Spread syntax

Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

For array literals

```
const firstarray = [3, 4];
const secondarray = [1, 8, 9];

const bigarray = [...firstarray, ...secondarray, 5];
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Spread syntax

Spread syntax (...) allows an iterable such as an array expression or string to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

For object literals

```
const firstobject = { name: "Dieter", age: 33 };
const secondobject = { courses: ["Web Technology", "Programming Fundamentals"] };

const combined = { ...firstobject, ...secondobject };
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

howest
hogeschool

Optional chaining

Optional chaining

The **optional chaining** operator (**?.**) enables you to read the value of a property located deep within a chain of connected objects without having to check that each reference in the chain is valid.

The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining

Optional chaining

The **optional chaining** operator (**?.**) enables you to read the value of a property located deep within a chain of connected objects without having to check that each reference in the chain is valid.

The ?. operator is like the . chaining operator, except that instead of causing an error if a reference is nullish (null or undefined), the expression short-circuits with a return value of undefined. When used with function calls, it returns undefined if the given function does not exist.

```
function peopleWithCatNamed(name) {
   return people.filter(person => person.cat?.name === name)
}
```

Stop and return undefined when previous result is undefined. Otherwise keep evaluating.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining

Optional chaining with event listeners

```
document.querySelector("#my-button").addEventListener("click", e => {
   /* some code */
});
```

This code crashes when there is no element with id *my-button*

```
document.querySelector("#my-button")?.addEventListener("click", e => {
   /* some code */
});
```

This code does not and simply doesn't add the event listener.
The optional chaining operator simply halts when *document.querySelector("#my-button")* is *undefined*.

howest
hogeschool

Copying objects

Value vs reference

- There are two kinds of data types
  - value types (primitive types)
    - Boolean, null, undefined, String, Number
    - When passed to a function it's value is passed
  - reference types
    - Array, Function, Object
    - When passed to a function, a reference to where the variable is stored in memory is passed

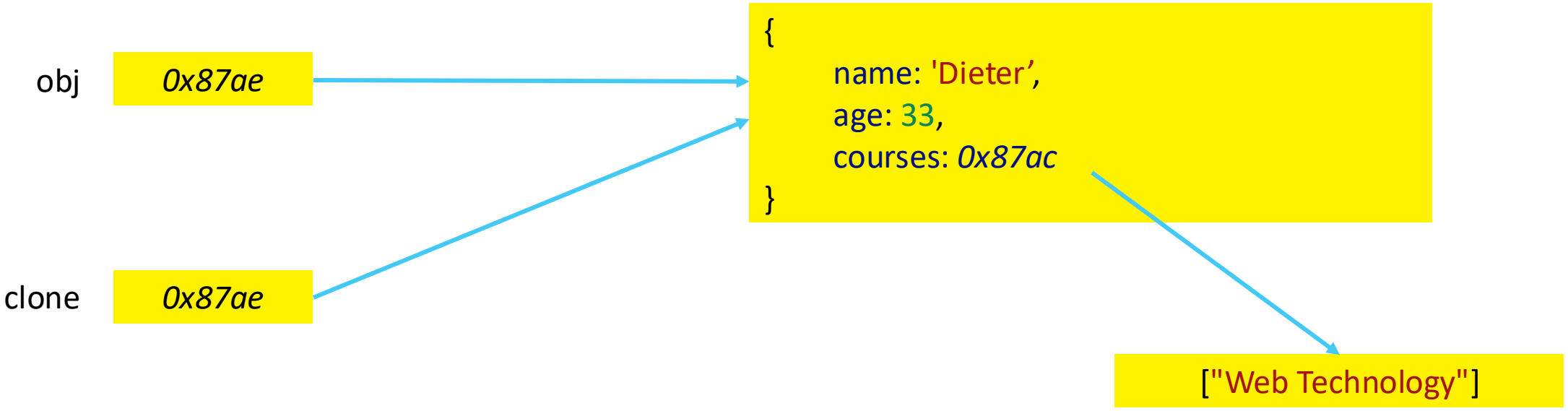
```
function myFunc(array) {
        array.push(8);
}

const arr = [2, 3];
myFunc(arr);
```

This means the function can modify it's value and it will also be modified when the function has exitted

Copy reference

```
const obj = { name: 'Dieter', age: 33, courses: ["Web Technology"] };
const clone = obj;
```

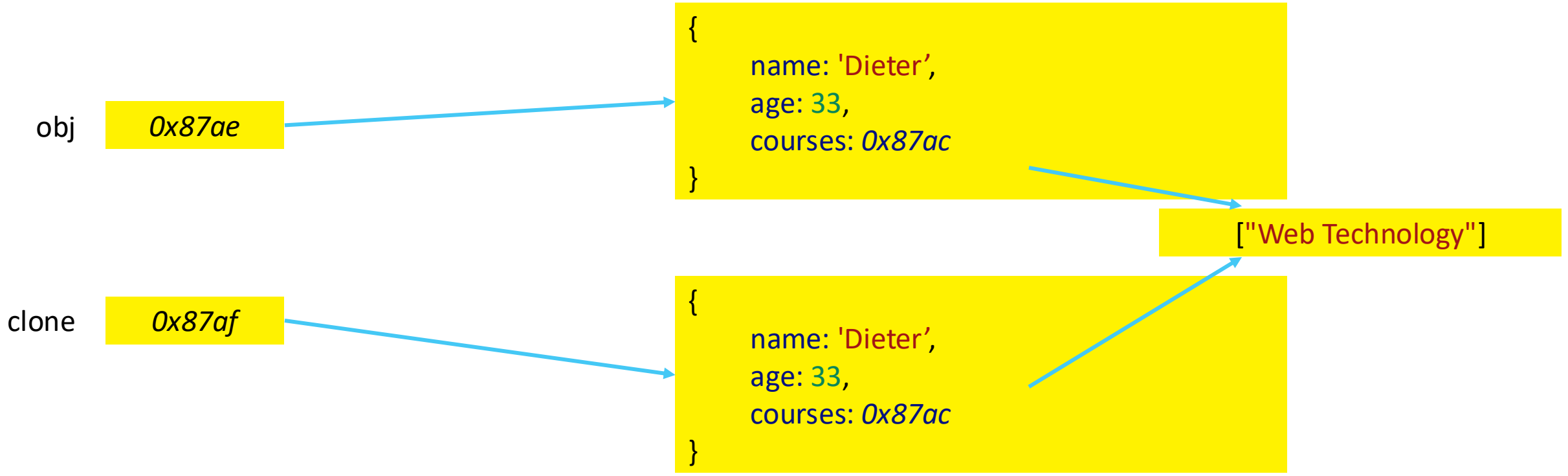
obj en clone point to the same place in memory, modifying clone will modify obj and vice versa



Shallow copy

```
const obj = { name: 'Dieter', age: 33, courses: ["Web Technology"] };
const clone = { ...obj };
```

The spread operator will create a **shallow** copy of obj. All properties will be duplicated in the new object.



Deep copy with JSon

```
const obj = { name: 'Dieter', age: 33, courses: ["Web Technology”]};
const clone = JSON.parse(JSON.stringify(obj));
```

```
const obj = { name: 'Dieter’,
                        birthday: new Date(1990, 0, 1),
                        courses: ["Web Technology"] };
const clone = JSON.parse(JSON.stringify(obj));
```

Date object will be gone, it will be a String

howest hogeschool

StructuredClone

The global **structuredClone()** method creates a deep clone of a given value using the structured clone algorithm.

```
const obj = { name: 'Dieter’,
                        birthday: new Date(1990, 0, 1),
                        courses: ["Web Technology"] };
const clone = structuredClone(obj);
```

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

howest
hogeschool

Destructuring

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

For arrays

```
const values = ["one", "two", "three", "four"];
const [a, b, c, d] = values;

console.log(a); // one
console.log(b); // two
console.log(c); // three
console.log(d); // four
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

For arrays

```
function minmax(numbers) {
    return [Math.min(...numbers), Math.max(...numbers)];
}

numbers = [3, 8, 4, 5];
let [minimum, maximum] = minmax(numbers);
```

```
// swap variables
let first = 3;
let second = 5;

[second, first] = [first, second];
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

For objects

```
const person = { name: "Dieter", age: 33 };
const { name, age } = person;
```

Variable names same as property names

```
const person = { name: "Dieter", age: 33 };
const { name: teacherName, age: teacherAge } = person;
```

Variable names different from property names

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

howest
hogeschool

Functions on Objects

Functions on Objects

Object.keys()

The **Object.keys()** method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

Object.values()

The **Object.values()** method returns an array of a given object's own enumerable property values, in the same order as that provided by a for...in loop. (The only difference is that a for...in loop enumerates properties in the prototype chain as well.)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values

Functions on Objects

Object.entries()

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs. This is the same as iterating with a for...in loop, except that a for...in loop enumerates properties in the prototype chain as well.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries

Object.fromEntries()

The **Object.fromEntries()** method transforms a list of key-value pairs into an object.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/fromEntries

Functions on objects

```
const data = { name: "Dieter", age: 33, courses: ["Web Technology", "Progressive Web Apps"] }

console.log(Object.keys(data));
['name', 'age', 'courses' ]

console.log(Object.values(data));
['Dieter', 33, ['Web Technology', 'Progressive Web Apps' ] ]

console.log(Object.entries(data));
[ ['name', 'Dieter'], ['age', 33], ['courses', ['Web Technology', 'Progressive Web Apps']]]

const arr = [["Michiel", 15], ["Justine", 12], ["Bo", 11]];
const scores = Object.fromEntries(arr);
console.log(scores);
{'Michiel': 15, 'Justine': 12, 'Bo': 11}
```

howest
hogeschool

Object properties

Properties

Every property in JavaScript objects can be classified by three factors:

- Enumerable or non-enumerable;
- String or symbol;
- Own property or inherited property from the prototype chain.

Enumerable properties are those properties whose internal enumerable flag is set to true, which is the default for properties created via simple assignment or via a property initializer. Properties defined via Object.defineProperty and such are not enumerable by default. Most iteration means (such as for...in loops and Object.keys) only visit enumerable keys.

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

Querying object properties

	Enumerable, own	Enumerable, inherited	Non-enumerable, own	Non-enumerable, inherited
`propertyIsEnumerable()`	`true` ✅	`false` ❌	`false` ❌	`false` ❌
`hasOwnProperty()`	`true` ✅	`false` ❌	`true` ✅	`false` ❌
`Object.hasOwn()`	`true` ✅	`false` ❌	`true` ✅	`false` ❌
`in`	`true` ✅	`true` ✅	`true` ✅	`true` ✅

Object.hasOwn

The **Object.hasOwn()** static method returns true if the specified object has the indicated property as its *own* property. If the property is inherited, or does not exist, the method returns false.

```
const obj = {}

console.log('toString' in obj); //true
console.log(Object.hasOwn(obj, 'toString')); //false
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn

Traversing object properties

	Enumerable, own	Enumerable, inherited	Non-enumerable, own	Non-enumerable, inherited

`Object.keys`
`Object.values`
`Object.entries`
	✅
	(strings)
		❌	❌	❌

`Object.getOwnPropertyNames`	✅ (strings)	❌	✅ (strings)	❌
`Object.getOwnPropertySymbols`	✅ (symbols)	❌	✅ (symbols)	❌
`Object.getOwnPropertyDescriptors`	✅	❌	✅	❌
`Reflect.ownKeys`	✅	❌	✅	❌
`for...in`	✅ (strings)	✅ (strings)	❌	❌
`Object.assign` (After the first parameter)	✅	❌	❌	❌
Object spread	✅	❌	❌	❌

For...in

The **for...in statement** iterates over all enumerable string properties of an object (ignoring properties keyed by symbols), including inherited enumerable properties.

```
for (let key in obj) {
    if (Object.hasOwn(obj, key)) {
        console.log(key);
    }
}
```

```
for (let key of Object.keys(obj)) {
    console.log(key);
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in

howest
hogeschool

Loops

Ways to iterate

for statement	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for_statement
for … in	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in
for … of	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of
foreach	https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

Regular for statement

A for loop repeats until a specified condition evaluates to false.
The JavaScript for loop is similar to the Java and C for loop.

```
const arr = [4, 5, 8, 9];

for (let i = 0; i < arr.length; i++) {
        console.log(arr[i]);
}
```

for … of

The for...of statement creates a loop Iterating over **iterable objects** (including Array, Map, Set, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
const arr = [4, 5, 8, 9];
for (let element of arr) {
        console.log(element);
}
```

for … of

The for...of statement creates a loop Iterating over **iterable objects** (including Array, Map, Set, arguments object and so on), invoking a custom iteration hook with statements to be executed for the value of each distinct property.

```
const obj = { name: "Dieter", age: 31 };;
for (let [key, value] of obj.entries()) {
        console.log(`key: ${key}, value: ${value}`);
}
```

The **Object.entries()** method returns an array of a given object's own enumerable string-keyed property [key, value] pairs.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries

for ... in

The for...in statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements.

```
const obj = { name: "Dieter", age: 31 };
for (let prop in obj) {
	console.log(`property: ${prop}, value: ${obj[prop]}`);
}
```

for … in

The for...in statement iterates a specified variable over all the enumerable properties of an object. For each distinct property, JavaScript executes the specified statements.

All enumerable also means inherited properties. Normally you don't want this behaviour which is why the use of ***for in*** is discouraged.

```
const obj = { name: "Dieter", age: 31 };
for (let prop in obj) {
        console.log(`property: ${prop}, value: ${obj[prop]}`);
}
```

```
for (let prop of Object.keys(obj)) {
   console.log(`property: ${prop}, value: ${obj[prop]}`);
}
```

forEach

The **forEach()** method executes a provided function once for each **array** element.

```
function doSomethingWithElement(element) {
        console.log(element);
}

const arr = [4, 5, 8, 9];
arr.forEach(doSomethingWithElement);
```

This means the function can modify it’s value and it will also be modified when exiting the function

forEach

The **forEach()** method executes a provided function once for each array element.

```
const arr = [4, 5, 8, 9];
arr.forEach(element => console.log(element));
```