Web Development and API Design

Lesson 01: Introduction

Prof. Andrea Arcuri

Goals/Topics

- Develop **Web Applications**, with focus on **Frontend**
- Technical details of JavaScript, but NOT web design
- *Single-Page Applications (SPA)*
  - client-side HTML rendering, using *React* from Facebook
- Intro to *REST* and *GraphQL* web services
  - JS on the server, using *NodeJS*
- *Websockets*
- *Security*

About Me

Prof. Andrea Arcuri

[simula . research laboratory]
- by thinking constantly about it

Course Info

- 12 lessons, once a week
- Check TimeEdit for possible changes of time and rooms
- During the course, do **NOT** send me private messages, but rather use the discussion forum of the course

Class Structure

- “Usually” 2+2
  - 2 hours of lecture: code (and very few slides…)
  - 2 hours in which you should do exercises and get help

- **IMPORTANT**: the 2 hours after lecture is not only for exercises. If you are falling behind, or you need some more revision, you can ask for my help on anything related to coding

If You Skip Class…

- Usually acceptable that a student skips 1-2 classes
- You are supposed to attend, although no strict checks
- If you skip too many classes, it is **YOUR** responsibility to catch up and find out what done in class

Necessary Tools

- YARN
- NodeJS
- Git
- An IDE
  - I recommend *WebStorm*
  - but *Visual Studio Code* is fine as well
- A Bash command-line terminal
  - Mac/Linux: use the built-in one
  - Windows: I recommend GitBash

Git Repository

- https://github.com/arcuri82/web_development_and_api_design

- Note: pull often, as new material will be added during the course

- No book, but plenty of external links to study from

Exam

- 100% home-assignment exam
- 48 hours

JavaScript

JavaScript

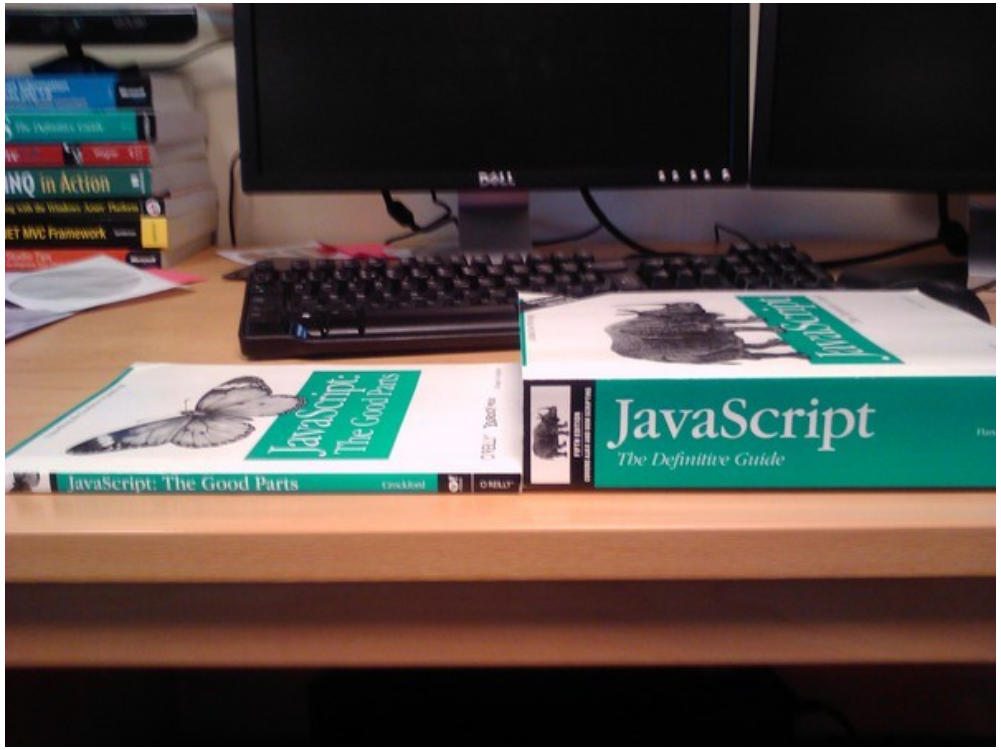
- JavaScript (JS) has nothing to do with Java
- Programming language executed in the browser
  - but now also on the server with *NodeJS*
- JS code referenced by webpages like any other resource (eg images and CSS files), or can be embedded directly in HTML
- JS can manipulate the DOM (Document Object Model) to alter the webpages structure/content based on user's interactions (eg mouse clicks)

JavaScript is King on the Browser

- If web page needs to execute code on browser, you use JS
- But historically there were other options in the (not so long ago) past:
  - Java with Java Applets (practically dead)
  - Flash (still found in some old web pages)
  - Silverligth
  - Etc.
- Those were not natively supported by browser, and you had to install plugins to run them

But JavaScript is a badly designed language…

- When the most famous book is called "*The Good Parts*", that tells you something…
- However, there are other languages that do transpile to JS, like *TypeScript* and *Kotlin*
- … and *WebAssembly* might (hopefully) replace JS one day…



Videos

- https://www.destroyallsoftware.com/talks/wat
- https://www.youtube.com/watch?v=EtoMN_xi-AM

Main Characteristics

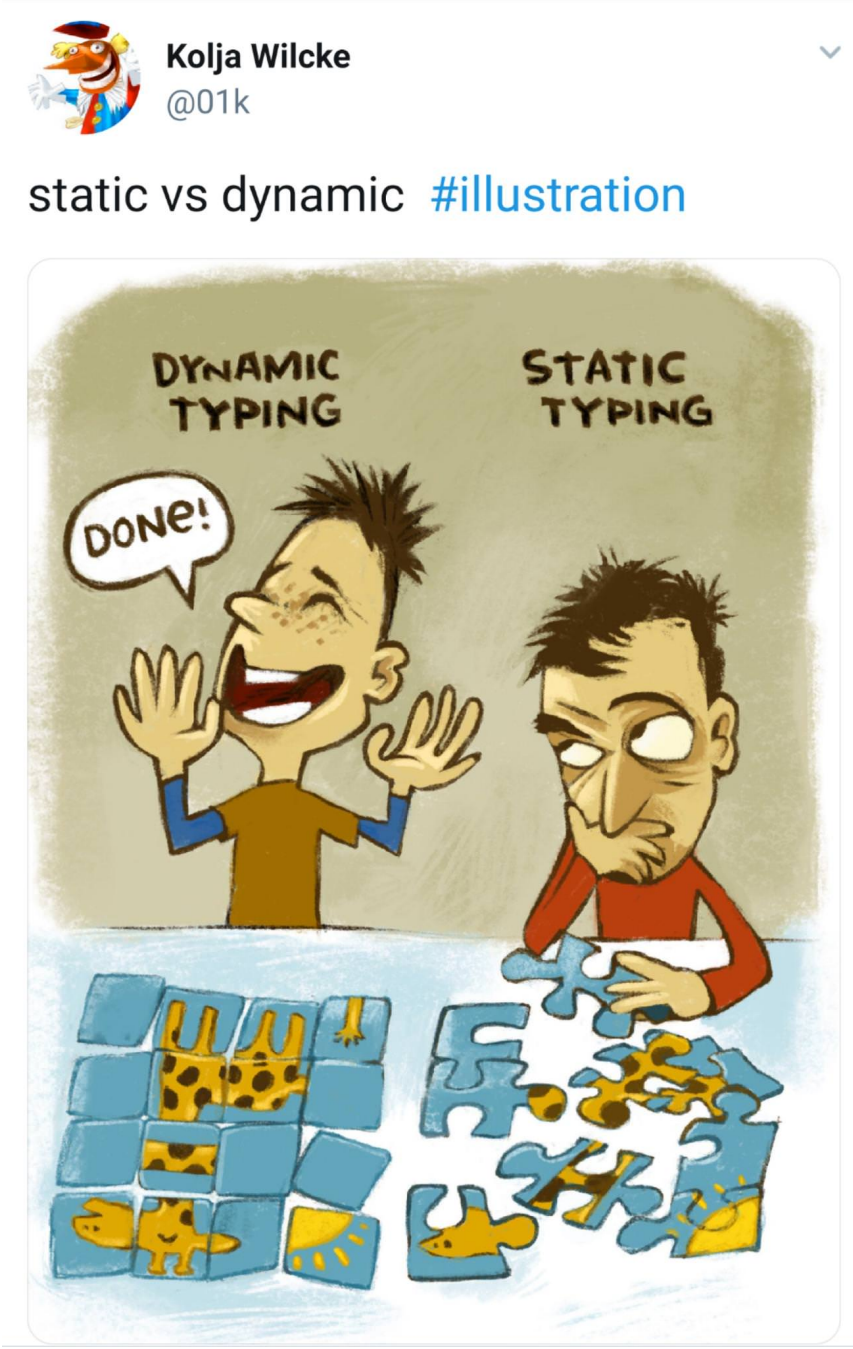
- **Interpreted**: you do not need to compile it (eg, in contrast to Java which is compiled down to bytecode)
  - Note: for performance reasons, the *runtime* (eg a browser like Chrome) will compile JS *on the fly* into machine code
- **Dynamically Typed**: when declaring variables, no need to specify the type, eg *String* or *Numeric*, and can reassign to different types
- **Weakly Typed**: you can use operators like “+” and “-” on different types (eg arrays and strings) without throwing errors

Interpreted

- Can just provide source code directly to the browser
- Can be directly inside HTML, or in separated "*.js*" files imported like any other resource (CSS, images, etc.)
- Note: current practice is to use *transpilation* steps
  - eg, using build tools like *NPM/YARN*
  - bundle dependencies like libraries (*React/Angular/Vue/*etc.)
  - transformations to support old browsers
  - enabling typing with *TypeScript*
  - etc.

Dynamically Typed

- **var x = 1;**
  - declare a variable called **x** with a numeric value equal to **1**
  - note we did not need to specify the “numeric” type
- **var x = 1; var x = “a”;**
  - **x** contains a string in the end. So, we changed the type from numeric to string
- **x = 1**
  - the “**var**” and “**;**” could be omitted, but you should NOT omit them
  - “**var**”: makes a local variable, otherwise is global scope (which is *bad*)
  - omitting “**;**” can lead to subtle bugs…



let/const vs. var

- If you declare a variable like **x = 1**, that will have *global scope*: you must avoid it
- **var x = 1**, does declare it a *function scope*: variable in a block would still be visible after the block inside the same function
- **let x = 1**, the sane way, ie *block scope*
- **const x = 1**, *block scope* like **let**, but cannot change value (similar to **final** in Java)
- In other words, use **let/const**

Weakly Typed

- A string plus a number? Concatenation
  - **“a” + 1** becomes **“a1”**
- A string minus a number? Result is not a number…
  - **“a” – 1** becomes **NaN**
- An empty object plus an empty array? Numeric 0…
  - **{} + []** becomes **0**
- Other dynamically typed languages (eg, *Python*) would throw an exception at runtime
  - They are called *Strongly Typed*
- Statically typed languages (eg, *Java*) would not even *compile*
  - with the only *exception* of “+” on String objects

Quiz: what is the result of this expression?

('b'+'a'+ + 'a' + 'a').toLowerCase()

banana

- “*obviously*”…
- **‘b’+ ‘a’** = **‘ba’**
  - concatenation of strings… that’s OK
- **‘a’ + + ‘a’** is equivalent to **‘a’ + (+ ‘a’)**
- **(+ ‘a’)** does try to convert the content of the string as positive number… but **‘a’** is *not a number*, so get a **NaN** result
- **‘a’ + + ‘a’** = **‘a’ + (+ ‘a’)** = **‘a’ + NaN** = **‘aNaN’**
- **'b'+'a'+ + 'a' + 'a'** = **‘baNaNa’**
- the **.toLowerCase()** just changes the **‘N’** into **‘n’**

Quiz: what is the result of this expression?

+(!![]+!![]+!![]+!![]+[]+(!![]+!![]))

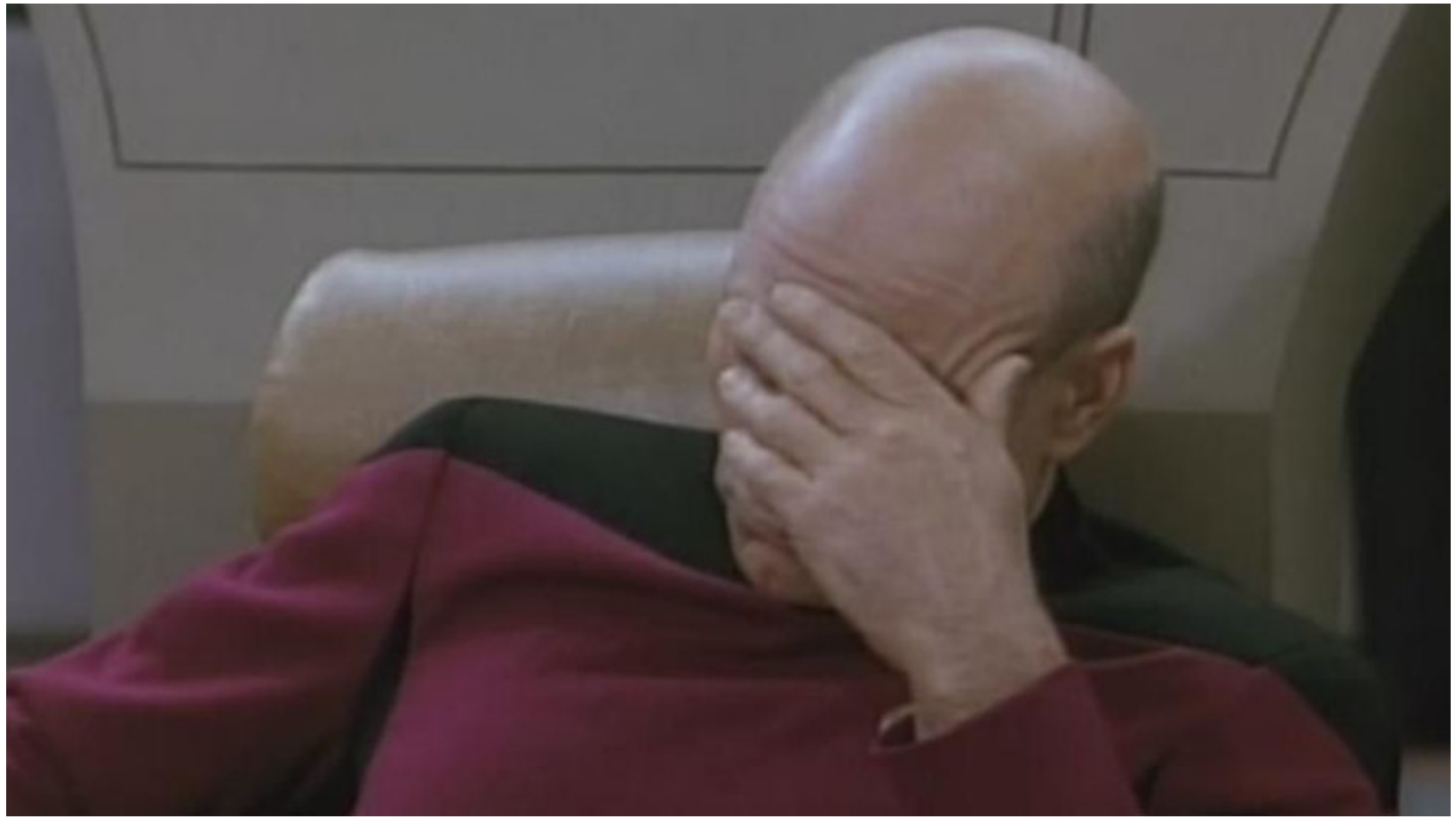
42

- *Obviously*…
- **[]**: empty array
- **![]**: negation of an array, which obviously returns **false**
- **!![]**: equivalent to **!false**, which results in **true**
  - this actually makes sense…
- **!![]+!![]**: equivalent to **true+true**, which JS converts to numbers, and sees **1+1**
- **!![]+!![]+!![]+!![]**: equivalent to **1+1+1+1**, which is **4**

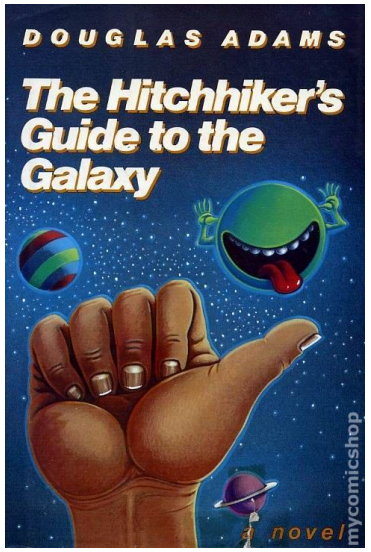
Cont.

- **!![]+!![]+!![]+!![]+[]**: equivalent to **4+[]**, which JS sees as a concatenation of strings, where **[]** is *obviously* coerced into the empty string, so result is **“4”+ “”**, which is just **“4”**
- **!![]+!![]+!![]+!![]+[]+(!![]+!![])**: equivalent to **“4”+2**, which, as a concatenation of strings and not numbers, results into **“42”**
  - ie, **2** is coerced into a string like **“2”**, and NOT **“4”** into a number like **4**
- **+(!![]+!![]+!![]+!![]+[]+(!![]+!![]))**: equivalent to **+(“42”)**, which considers the string as a positive number, and so coerced into **42**

+(!![]+!![]+!![]+!![]+[]+(!![]+!![]))
yes… obviously 42…



Anyway… why **42**?



- You will see **42** all the time…
- Geeky reference to the "*The Hitchhiker's Guide to the Galaxy*"
- It is the "***Answer*** *to the Ultimate Question of Life, the Universe, and Everything*"

Quiz: what happens when you sort an array of integers like the following?

```
[3,18,1,2].sort()
```

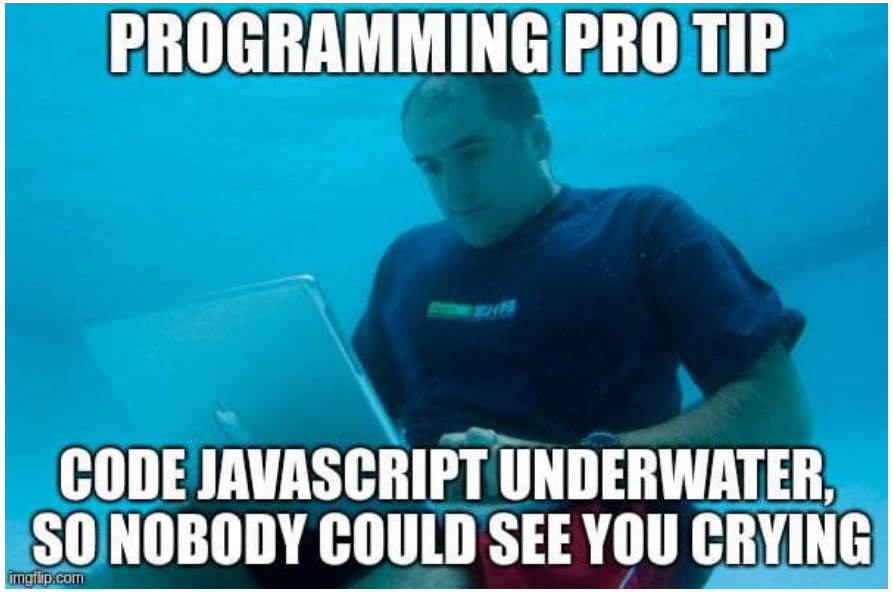
[1, 18, 2, 3]

- *“Obviously”* **18** is *smaller* than **2** and **3**, isn’t it?
- What the heck is happening here?
- JS has no concept of typed array… you could add all different kinds of types in same array
- So, no default way to define ordering on a JS array
- JS, by default, converts all values into STRINGs, and does comparisons based on string ordering
- The string **“18”** is smaller than string **“2”**, as starting with a 1

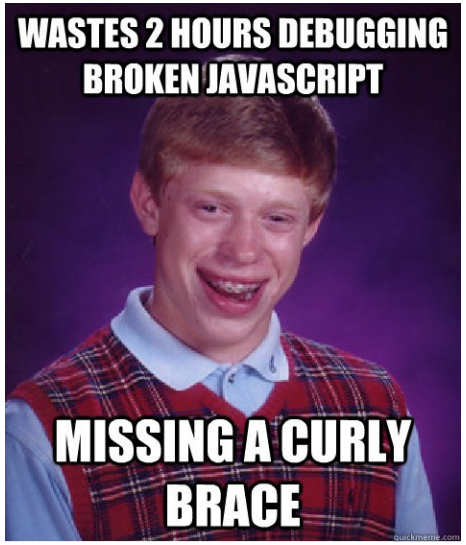
Do Not Do Drugs…

Otherwise, one day you might end up designing languages like JavaScript…

- By now… you should have guessed what is my opinion of JavaScript
- But JS is a *must* to learn if you are dealing with web development…
- … even if you just want to focus on backend
- Until *WebAssembly* will support DOM manipulation, or *Kotlin* transpilation will have better support, unfortunately we need to endure JS
  - *TypeScript* can ease the pain meanwhile…



Keep your employees hygienic



Jokes apart…

- The pain of JS (and other dynamically typed languages) is when working on *large* projects…
- … where you might need to do *refactoring*
  - good luck, you *poor souls*…
- … and/or have to work on code written by others…
- For what you will see in this course, and during your degree, you will be (hopefully) fine, as working only on *small* systems
- You need to get experience in building a project with a dynamically typed language (and so *TypeScript* will not be allowed in the exam)
- Remember: *what does not kill you, makes you stronger*

For equality, use “===“ and not “==”

- **false == 0**
  - result is **true**, ie, boolean **false** is equivalent to numeric **0**, as the **0** gets transformed into a boolean to compare it with **false**
- **false === 0**
  - result is **false**, as a boolean value is not equal to a numeric value
- **0 == []**
  - surprisingly, that is true in JS, ie the numeric **0** is equal to an empty array
  - plenty of these hilarious cases, see https://dorey.github.io/JavaScript-Equality-Table/
- For negation, use **!==** instead of **!=**

Booleans

- 6 values evaluate to "*false*" when used as boolean, all others evaluate to "*true*"
  - **false**
  - **0**
  - **""** (empty string)
  - **null**
  - **undefined**
  - **NaN**
- Ex. **if("foo")** and **if(42)** would execute the *then* branch, but not **if("")** nor **if(0)**

Function Declaration

- **function foo(){ return 1;}**
  - calling **foo()** will return value **1**
- **add = function(x,y){return x+y;}**
  - calling **add(1,2)** will return **3**
  - calling **add(“a”, “b”)** will return **“ab”**
- **add = (x,y) => {return x+y;}**
  - the arrow notation is similar to *function*, but it treats **this** keyword differently, as not defining its own scope
  - this will become more clear when we will define callbacks inside *React* objects

Functions as variables

- **function foo(x,y){return x+y;}**
  - declare a function called **foo**
- **x = foo(1,2)**
  - call the function, and store its result **3** in the variable **x**
- **x = foo; x(1,2)**
  - store the code of the function **foo** in a variable **x**, and then call it by using **()** on such variable with inputs **1** and **2**
- **x = () => foo(1,2); x()**
  - create a new function with no inputs and that just calls **foo(1,2)**, and store it in a variable **x**. Then call such function by using **()** on it
- **addOne = y => foo(y,1); addOne(5)**
  - create a new function that takes an input **y**, and return it with a **+1**. So, **addOne(5)** does return the value **6** here

Code Comments

- To document software, typical case of writing comments directly in the source code
- JS uses similar syntax to other languages (eg Java)
- Single-line comment: **//**
- Multi-line comment: started with **/*** and then closed with ***/**

Map and Filter

- Throughout the course, we will often use *.map()* and *.filter()* operations on *arrays/lists*
- They return a new copy the array
- *Filter*: a subset of the array, according to a predicate
- *Map*: each element is transformed into a new element, according to the provided mapping function

Filter Example

[-2, 1, 4, -7].filter(e => e > 0)

this returns

[1, 4]

Each element **e** in the array is given as input to the arrow function **e => e>0**

Such function must return either **true** or **false**

The element will be part of the output array only if the predicate was **true**

Map Example

["foo", "hello", "hi"].map(e => e.length)

this returns

[3, 5, 2]

Each element **e** is transformed into something else, possibly changing the type (eg from string to number)

Optional Index

["foo", "hello", "hi"].map((e,i) => "" + i +"_" +e.length)

this returns

["0_3", "1_5", "2_2"]

e is the element value, whereas **i** is the index in the array

Note: you can use different variable names instead of **(e,i)**

Index can be ignored

const len = s => s.length

["foo", "hello", "hi"].map(len)

this returns

[3, 5, 2]

The function **len** here accepts only 1 input (called **s**), and so index is ignored

Quiz: what is the result of this expression?

["10", "10","10"].map(parseInt)

where **parseInt("42")** gives **42**, ie from string to number

[10, NaN, 2]

- *obviously*…
- **parseInt** takes 2 inputs: a *string* and a *radix*
- as **map** provides 2 inputs **(e,i)**, the index **i** is going to be used as radix
- **parseInt(“10”, 0) === 10**
- **parseInt(“10”, 1) === NaN**
- **parseInt(“10”, 2) === 2** (ie, **“10”** is read like it was in binary)
- Better to write: **["10", "10","10"].map(e => parseInt(e, 10))**

DOM Manipulation

- Document Object Model (DOM): object representation of the displayed HTML
- One of the main reasons to use JS is to manipulate the DOM, ie altering what is displayed to the user
- To access the DOM, JS can refer to the object called "**document**"
- Call methods on **document** to retrieve object representations of the DOM

```
clearText = function(){

    const textArea = document.getElementById("textId");
    const resultArea = document.getElementById("resultId");

    textArea.value = '';
    resultArea.value = '';
};
```

- Easiest way to retrieve DOM objects is by *id*
- The id needs to be set as HTML attribute, e.g. **<textarea id="textId"></textarea>**

JS Interactions

- There are different ways to execute JS in a page
- One simple approach is to directly register *event handlers* on the HTML tags
  - ** Clear **
  - when user on browser clicks on that button, the JS function "*clearText()*" is going to be executed
- Event handlers:
  - *onclick*, *onchange*, *onmouseover*, *onmouseout*, *onkeydown*, etc.
  - see for example https://www.w3schools.com/js/js_events.asp

JS Console, from Chrome Developer Tools

Useful for debugging and learning by running custom JS directly on page