Contents

The Web Developer Bootcamp 2021

This course has a changelog

The code of this course is in every lecture (and will be saved in this folder) and resources may appear if needed in every lesson

The tools needed for this course are: - Chrome (or every othe web browser) - VS Code (or some other text/code editing software)

Some useful resources: - Mozilla Developer Network - Chome Inspector

Lesson 1: Basic Concepts

The Internet and the Web

The internet is a **global** network of networks. It is the infrastructure that allows services such as e-mails The World Wide eb is the system that allows sharing information over the internet. The current protocol for sharing is *Hyper Text Transfer Protocol* (HTTP) and it works with *Requests* and *Responses.* In every information exchange, we have two agents: 1. The servers are machines (computers) and also softwares that can satisfy requests on the web. 2. The client is a computer that accesses a server, i.e, the one who makes a request to ta server.

At a human level, a web browser is necessary to process the information that servers respond to our request. This processing is mostly rendering a visual representation of the information, using the instructions that the server delivers within every response.

Nowadays, the servers mostly respond with instructions that use three main technologies: HTML, JavaScript and CSS.

Front-end and Back-eand

One suitable analogy to understand the relationshio between the front and the back ends is the kitchen and the tables in a restaurant: 1. At the table, the front-end, the waiters take your order (request) 2. Then, the kitchen, the back-end, prepares your meals (processes your request) and then delivers them with the waiters (response)

In the web development, one can separate this two ends in terms of what technologies are used to implement them: the fron-end develompent focuses on the dynamic trio HTML, JS and CSS, however the back-end may be built upon many languages: Python, C, Ruby, JS (also), Databases languages, etc.

In this course we start by understanding the front-end and then we move into the server-side :)

Front-end: The Purple Dino Danced

The front-end technologies have unique roles as follows:

- HTML: *What*, the contents of the page (the structure)
- CSS: *How it should look*, the content display style
- JS: *Actions*, how do the page behaves and interacts

Section 2 and 3 - HTML Basics

Lest's start with **HTML** HTML is a *markup language* used to describe and annotate text format, styles and annotations. The goal of HTML is to take plain text and give it visual structure and appearances.

HTML Elements

```html
<i> italics </i>
<b> italics </b>
```

HTML Boiler Plate

This is the *Skeleton* for every webpage:

Every page must have ONE `<head>` and ONE `<body>`. The head contains the metadata and the body is the information that will be displayed in the page.

Also, we need some tags to define this document as an HTML one by using:

```html
<!DOCTYPE html>
<html>
```

```
    <head>
        Here goes the metadada
        <title> </tile>
    </head>

    <body>
        Here goes the actual page content
    </body>
</html>
```

In VS Code you can use the *Format this document* to re-arrange the file content. The shortcut for this es `Option+Shift+F`

Lists: ordered and unordered

```
<ul> This indicates an Unordered List opening
    <li>This is an List Item</li>
    <li>This is another List Item</li>
    <li>...</li>
    <li>...</li>
    <li>Etc :)</li>
</ul>
```

Only `<li>` can be used inside a list. But one can nest lists inside an `<li>` tag :)

Anchor tags: provide links

anchor tags are *in-line* elements, which means the do not take their own new line. They can be accomodated inside a paragraph or any other text

Section 4 - HTML next steps

- Undestanding what HTML5 actually is
- Block vs Inline Elements
- `span`and `div`elements
- NTH: `sub` `hr` `br`and `sup`elements

HTML5 is not a *version*

HTML5 is an *evolution* of the HTML Living Standard: There is no *official* version of HTML. There only a "how HTML must work": every company build their own HTML into their applications. These instructions are here. One can always use older HTML's, but it's very uncommon.

Section 5 - HTML Tables & Forms

What we will learn: HTML tables and HTML forms (how to input data), although these days is not a good practice. ### Tables Basics: Tables were used to *layout* content inside a webpage. We can create tables usign upto 15 elemets, but we don't actually need all of them :)

- `<table>`: declare a table block
- `<tr>`: row of a table
- `<td>`: table data (one single element)
- `<th>`: table header

These are just for proper handling of the table - `<tbody>`: isn't - `<thead>`: these - `<tfoot>`: obvious?

Always follow this structure:

```
<table>
    <thead>
        <tr>
            <th> Table Head uses table headers</th>
            <th>...</th>
            <th>...</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td> Table Body uses table data</td>
            <td>...</td>
            <td>...</td>
        </tr>
        <tr>
        <tr>
            <td>...</td>
            <td>...</td>
            <td>...</td>
        </tr>
        <tr>
            <td>...</td>
            <td>...</td>
            <td>...</td>
        </tr>
    </tbody>
</table>
```

How about multiple table headers?

Forms

A form represents a section of the document containing interactive controls for submitting informatio

- The *action* attribute specifies WHERE the data should be sent. e.g.: `/search/`, `/tacos`
- The *method* attribute specifies which HTTP method should be used

What con go inside a form: almost any type of input and text. This is the basic structure:

```
<form action="/tacos">
    <p>
        <label for="username">Enter a username: </label>
        <input id="username" type="text" placeholder="username" name="name">
    </p>
    <button>Submit</button>
</form>
```

The abtributes do the following: - Inside `label`, `for=""` specifies the identifier for the label across the form (or document *citation needed) - The label identifier then is passed to the `input` in the `id` attribute - `name`specifies the variable name that will be passed to the server when the action/query is triggered. It will be reflected in the address bar - `placeholder`specifies the text that is displayed when the user has not typed anything yet. Useful for suggestions like “enter your search here”

Why add labels? Several reasons: - They provide visual information/text/indicators related to the input - Labels allow to click this visual indicators in sync with the actual input, whichc makes it easier and more usable in mobile applications -

Checkboxes

Radio Buttons Note that radio buttons and checkboxes always send `on` when submitted, unless the `value` attribute is specified.

The following won’t work when submit button is clicked

```
<form action="">
    <label for="xs">XS:</label>
    <input type="radio" name="size" id="xs">
    <label for="s">S:</label>
    <input type="radio" name="size" id="s">
    <label for="m">M:</label>
    <input type="radio" name="size" id="m">
    <button>Submit</button>
</form>
``
```

```
This will send
```

?size=on

```
The following, however, will send the data we want to send:
```

XS: S: M:

Submit

~~~
### Dropdown lists: Select and Option

```html
<form>
    <label for="meal">Please Select an Entree</label>
    <select name="meal" id="meal">
        <option value="">-- Please Select Your Meal--</option>
        <option value="fish">Fish</option>
        <option value="veg">Veggie</option>
        <option value="steak">Steak</option>
    </select>
</form>
~~~

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

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>
    <h1>Race Registration!</h1>
    <form action="/race">
        <div>
            <label for="first_name">First Name</label>
            <input type="text" id="first_name" name="first_name" required>
            <label for="last_name">Last Name</label>
            <input type="text" id="last_name" name="last_name" required>
        </div>
        <div>
            <p>Select a Race:</p>
            <div>
                <input type="radio" id="5k" name="race" value="5k">
                <label for="5k">Fun Run 5K</label>
                <br>
                <input type="radio" id="half" name="race" value="half">
                <label for="half">Half Marathon</label>
```

```
                <br>
                <input type="radio" id="full" name="race" value="full">
                <label for="full">Full Marathon</label>
            </div>
        </div>
        <div>
            <label for="email">Email</label>
            <input type="email" name="email" id="email" required>
            <label for="password">Password</label>
            <input type="password" name="password" id="password" required>
        </div>
        <div>
            <label for="age">Select age group</label>
            <select name="age" id="age">
                <option value="age1">Under 18</option>
                <option value="age2">18-30</option>
                <option value="age4">50+</option><html lang="en">

</html>
```

Section 6 - CSS

What it is: Cascading Style Sheets. Decribing *how* documents are presented visually, a.k.a. *the looks*

Everything in CSS follows this pattern

```
selector {
    property: value;
}
```

For example This paints every `<h1>` purple

```
h1 {
    color: purple;
}
```

CSS is huge! - Always have the MDN CSS Resources at hand

Good practices

1. Never include styles inline

```
<h1 style="color: purple">Hello</h1>
<button style="background-color: palegreen;">About</button>
<button style="background-color: palegreen;">Another button</button>
```

2. We may write our css in the `<head>` of the html file inside a `<style>` block, BUT it isn't good either:

```
<style>
    h2 {
        color: palevioletred;
    }
</style>
```

3. Best practice is to create a `.css` file that will be included with a `<link>` tag in the html `<head>`

Properties

Colors

In CSS we have *named* colors, but we can also use *coded* colors

Text

`text-align`: how does the text align in an element. It does not control the alignment of an element respect to the whole page `font-weight`: how to *bold* and *unbold* text. `400` is the same as `normal` `text-decoration`: controls lines for underlining, strike-through, etc. may be useful for "removing" default underlining of `<a>`. `line-height`: how tall a line of text is. Several units: normal, 2.5, etc. `letter-spacing`: horizontal space between letters in our text

Sizes

- Realvite Units: `em`, `rem`, `vh`, `vw`, `%`
- Absolute units: `pt`, `cm`, `mm`
  - `px` not necessarily equal the width of 1 pixel in your machine. Not recommended for responsive websites
    ### Fonts `font-family` is the property that specifies the font of the page, but changing it may not be as easy as it seem. Some fonts may not be supported in a users browser

Every font after the comma is the next alternative font to use if the machine does not support the preferred font family:

```
font-family: Helvetica, sans-serif;
```

Section 7 - CSS Selectors

CSS has a strutured way of computing which selector must go in every element, so having that in mind will help to organize our code better.

- *not very common and not recommended

- element selector is very common and can be grouped using commas

- `id` selectors: appart from `label`-ing, id's are also useful for linking custom and specific selectors
- `.class` selector: similar idea to our id, but it can applied to miltuple elements
- descendant selector: select only elements inside the previous selector. Example: *select `a` only if inside of `li`*

```
li a {
    ...
}
```

- Adjacent selector (Combinators): `h1 + p` select `p`that come right after an `h1`(not nested, immediately right after)
- Child selector: `h1 > p` select `p`that are inmediate children of an `h1`, but not the nested in further children
- Attribute selector: `input[type=password]` select only `input` of type `password`
- Pseudo Classes: keywords added to a selector that specifies a special state of the element: `:active`, `:checked`, `:first`, `:first-child`, `:not()`, `nth-child`, `nth-of-type`.
- Pseudo Elements: Modifiers to our selectors: `::after`, `::before`, `::first-letter`, `::first-line`, `::selection`,

Cascade!

The order matters: the last selector is the one that will be applied

Specificity

The most specific selector wins when in a conflict: ID > Class > Element

Section 8 - CSS Box Model

Width and Heigth

The always state the **content** dimensions

Borders

Every border need at least these three parameters: - `border-width` - `border-color` - `border-style`

These three can be provided in one line with the `border` shorthand property.

```
/* width | style | color*/
border:
```

Additionally, we can set `border-sizing`to specify if the dimensions consider what is inside the border or if it es included in the overall dimensions.

Padding and Margin

The thing you put inside a box carrying fragile stuff to prevent damages. The padding goes between the content and the border. It is colored in green in the code inspector tools

The padding shorthand is just `padding` and it can be written as

```
/* top | right | bottom | left */
padding:
```

The margin works as padding, but outside the box border. By default, the `body` has a fixed margin value, so it is common to override it to cero at the very start of a new project.

Display

The `display`property handles how elements use the space, in relation to other elements

EMs & REMs

EM is a unit in the typography field that is equil to the currently specified point size.

`em` in `font-size` refers to the relative size of the **parent** `em` in `marigin`, `padding`, `border-radius` refers to the relative size of the **element**

REM stands for *Root EM*, which refers to the font size specified for the root element of the page, which often is ‘

“

One may want to use EMs and REMs combined, depending on the context and what the behaviour of specific elements should be.

Section 8 - CSS Box Model

Width and Heigth

The always state the **content** dimensions

Borders

Every border need at least these three parameters: - `border-width` - `border-color` - `border-style`

These three can be provided in one line with the `border` shorthand property.

```
/* width | style | color*/
border:
```

Additionally, we can set `border-sizing`to specify if the dimensions consider what is inside the border or if it es included in the overall dimensions.

Padding and Margin

The thing you put inside a box carrying fragile stuff to prevent damages. The padding goes between the content and the border. It is colored in green in the code inspector tools

The padding shorthand is just `padding` and it can be written as

```
/* top | right | bottom | left */
padding:
```

The margin works as padding, but outside the box border. By default, the `body` has a fixed margin value, so it is common to override it to cero at the very start of a new project.

Display

The `display`property handles how elements use the space, in relation to other elements

EMs & REMs

EM is a unit in the typography field that is equil to the currently specified point size.

`em` in `font-size` refers to the relative size of the **parent** `em` in `marigin`, `padding`, `border-radius` refers to the relative size of the **element**

REM stands for *Root EM*, which refers to the font size specified for the root element of the page, which often is ‘

“

One may want to use EMs and REMs combined, depending on the context and what the behaviour of specific elements should be.

Section 9 - Other Useful CSS Properties

Opacity & Alpha Channel

Setting `color` with `rgba` only sets the color for the selected property, however `opacity` sets the transparecy/opacity for the entire element and its children

Position

The position property specifies how is the

Transitions

So much fun!!

Transform

Section 10 - Flexbox

Flexbox is a **one-dimensional** layout method for laying out items in rows or columns. It's fairly new. Why "Flex"? Because Flexbox allows us to dristibute space dynamically across elements of *unknown* size.

Flex Direction

In Flex model there are two axis - **Main** axis: default `row` - **Cross** axis: default `column`

We can modify the direction using the `flex-direction` property: - `reverse` uses `row`as main axis, but starting from right to left - `column`uses `column`as main axis - `column-reverse` you can figure it out

Justify Content

How the content is ditributed along the main axis. We can use the following parameters; - `flex-start` moves everything to the start of the main axis - `flex-end` moves everything to the end of the main axis - `flex-center` centers everythin aling the main - `space-between` spans every element along the main axis, but not around the borders - `space-around` spans every element along the main axis, including the spaces between the one from start to the first element and from last element to end. - `space-evenly` spans every element evenly

Align Items and contents

Align items distributes the content along the **cross axis**, just like justify.

Also, we can use `align-self`

Flex Wrap

`flex: wrap-reverse` also changes the **cross axis** :grimmacing:

Flexing items

`flex-basis` `flex-grow` is the speed at which an element will occupy available space to fill the main axis `flex-shrink` is the speed at which an element will occupy available space to fill the main axis

Bootstrap

It is a CSS *framework* that provides pre-built CSS code to make page development faster. Not everyone uses it and there are many alternatives, like Bulma, for example. It's not necessary for build a page, but it may come handy in some use cases.

One of the most *annoying* part of creating web pages is laying out the content. Bootstrap provides a grid system that may help to distribute the content using columns that take the specified amount of space, preserving some guidelines, in way that is definitely more structured than flex.

The wonderful world of JavaScript

JS can be run headless in a server or control the logic of html pages in many modern browsers. From interactive menus and galleries, to chatboxes inside a shop web page, JS can manage whatever happens inside a page and we can define the logic of how it happens :)

How to execute JS code

Run JS code from a file using node

We can run JS files from a terminal with `node`.

```
$ node hello_world.js
Hello World!
```

Run JS code from a file using the web browser

Web browsers will only execute code that is bound to an html file, so if we want to run JS scripts, we must add them to our html head. Lets say we have our `app.js` file and we want to run it along with `index.html`. We must include the following:

```
<head>
    ...
    <script src="hello_world.js"></script>
</head>
```

Nothing will be displayed in the browser, however in the console of the developer tools we will see the output ‘Hello World!’

Print outputs

If we run code from files, either within a web-browser or with node, it is always necessary to explicitly print the outputs:

In a file:

```
1 + 4;                  // does not print
console.log(1 + 4);     // prints 5

"Hello!";               // does not print
console.log("Hello!");  // prints 'Hello!'
```

However,in any web browser you can open a console in the developer tools, which come with a powerful feature: they work in a *Read Evaluate Print Loop* (REPL), which means that every statement we input into the console will be evaluated and the result will be printed back at us.

In a *web browser console*, every result of a statement will be printed.

```
1 + 4;          // prints 5
"Hello!";       // prints 'Hello!'
```

Furthermore, there are some functions to prompt results in different ways:

```
console.log("Hello")        // prints 'Hello'
console.warn("Uh oh!")      // prints 'Uh oh!' in yellow background
console.error("Fatal")      // prints 'Fatal' in red background

alert("Hi There!")                // pops a block that the user must accept
prompt("Please enter a number") // pops a bock where the user can enter an input
```

JS Primitive types

The most primitive types in JS are: - Numbers: represent every type of number (except complex) - Strings represent *arrays* of characters and letters - Booleans represent True or False logic values - Null - Undefined - Symbol (uncommon) - BigInt (uncommon)

Numbers & Math

In many languages there are several number types. In JS, we just have `Number`. Any floating point must be written with its decimal point and places.

Alongside with Numbers, we have *math operations*. Basic operations are: - Addition: sum of two numbers - Substraction: substraction of two numbers - Multiplication: product of two numbers - Division: division of two numbers - Modulo: remainder of a division `9 % 2 = 1` - Exponentiation: power of a number '9 ** 2 = 81'

The order of a statement with more than 2 terms is computed following this order, and then from left to right: 1. Parenthesis 2. Exponentials (powers) 3. Multiplications 4. Addition 5. Substraction

Not a Number

There is a value in JS that represent a numeric value that is not a number. It is used mainly in divisions by zero:

```
> typeof(NaN)
< "number"
```

Any operation that takes a `NaN` outputs a `NaN`, so no real math can be done with it, however, it helps us handling some results and catch errors.

Declare & Assign variables

Variables are names we use to reference some data we want to use further in our program.

Any variable can be assigned with this simple syntax:

```
let variableName = value;
```

Then, every time we call the `variableName`, the result will be the `value` we stored, unless we update its value by reassigning it.

When updating variables, we don't need the `let` keyword again:

```
let score = 0;
score = socre + 5;  // 5
score += 5;         // 10
score /= 2;         // 5
score++;            // 6
```

`const`

`const` variables cannot be reassigned, so an error is thrown when some instruction in the code tries to do it.

```
const luckyNum
```

`var`

`var` assignement works very similarly to `let`, however, it has `function` scope, instead of `block` scope. `var` variables can be re-declared (!), so in bigger bigger applications may produce several bugs caused by repeated variable names or by multiple references to variables with the same name.

Also, `let` is not intended for global scopes, so even if a code may run while havin a `let` assignement in the global scope, many linters will complain about it.

Given this restrictions, JS has variable *hoisting* wich enables the use of `var` in

Better explanation here: https://stackoverflow.com/questions/762011/whats-the-difference-between-using-let-and-var

Booleans

Booleans are values that can be used to evaluate logic propositions: `true` and `false`.

```
let isActive = true;
let isInactive = false;
```

Dynamic types

JS is dynamically typed, wich means there are no actual types, but we can build objects on top of the primitive types *and* the type of an object can change through the program:

```
let superVariable = 15;
superVariable = true;
```

This code will not throw errors :)

Naming Conventions

In JS there are *hard rules* that we must abide when coding JS: - Identifiers (variable names) cannot have spaces - Identifiers can have `_`,`$` and digits - Identifiers cannot start with a digit

There are also some conventions in the community: - Use camel-casing for identifiers: `goodNamesAreCamelCased` - Use self-explanatory names instead

```
bad name:  userLoggedIn = false;
good name: isUserLoggedIn = true;
```

Strings

Numbers and booleans are ok, but what if I want to store names, words or phrases? JS has *strings* for that :)

```
let favAnimal = 'Dumbo Octopus';
let age = "23";
```

Note taht the last assignment is not assigning the *number* `23` but the characters `2` and `3` in the `age` identifier.

One should never mix single and double quotes in string declaration, unless there is one of the two characters that is needed in the sentence, e.g.:

```
let greetings = "Hell! I'm John"
```

Here the double quotes `"` are state the variable start and end, and the single quote is just another character.

Strings have indices

In JS, strings are indexed, which meand that every character can be addressed by a number of a list:

```
let animal = "Dumbo Octopus";
animal[0];  // "D"
animal[1];  // "u"
animal[6];  // "O"
animal[99]; // undefined
```

When do we use string indices? One use case is checking the phone area code of a telephone number string:

```
let phone = "(234) 457-9820"
phone[0];   // "("
```

Strings also have *length* that can be accessed via the `length` property:

```
let animal = Phoenix;
animal.length;  // 7
"dog".length;   // 3
```

Also, we can concatenate two or more strings with the `+` operator:

```
let animal = Phoenix;
let place = River;
let combine = place + animal;     // RiverPhoenix
```

The result will always be a new variable; changing the conten of `animal` or `place` will not affect the content of `combine`.

Furthermore, when numbers are added to strings, they are interpreted as strings, so the result is the concatenation of both:

```
1 + "hello";        // "1hello"
typeof(1);          // "number"
typeof(1+"hello");  // "string"
```

¡¡BE CAREFUL WHEN SUMMING NUMBERS TO STRINGS AND EXPECTING NUMBERS!!

String methods

Methods are actions of a string (or object) that perform some calculations and return an output:

```
let msg = "leave me alone";
msg.toUpperCase();  //'LEAVE ME ALONE'

let userInput = "     hello, my name is tim tom     ";
userInput.trim();   // 'hello, my name is tim tom'

let greeting = " hello again!!       ";
greeting.trim().toUpperCase(); " // 'HELLO AGAIN!!'
```

Many string methods return a modified copy of the original string, so they are called "non-destructive" methods.

There are also methods that accept *arguments*, which is stuff we pass through the parenthesis:

```
let tvShow = 'catdog';

tvShow.indexOf('cat');  // 0
tvShow.indexOf('dog');  // 3
tvShow.indexOf('$');    // -1 (does not contain '$')
```

Some other funny methods can be found at MDN: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Template literals

One way to easily format text that depends on variable values is template strings:

```
let product = 'Artichoke';
let price = 3.99;
let qty = 5;
const msg = `You bought ${qty} ${product}. Total price is: ${qty*price}`;
```

VERY HANDY!

Null vs Undefined

`undefined` happens when the JS interpreter cannot find the value for the variable that is evaluating because it has not been assigned or its reference is unknown.

`null` represents a value that is not a value, but it is assigned that.

Math

`Math.random` `Math.floor`

Comparisons

As in many languages, JS has comparisons between numbers and other types. The most basic is comparing numbers, which always return a `boolean`, but characters can also be compared using the unicode mapping:

```
1 > 3;      // false
5 < 9;      // true
'A' < 'a';  // true
```

Equality & Inequality operators

- `<`: less than
- `<=`: less or equal than
- `>`: greater than
- `>=`: greater or equal than
- `==` : equiality that coherces value type, i.e.: convert to the same type and then compare
- `!=` : negate equality

```
1 == '1';    // true
0 == false; // true
```

- `===` : *strict* equality that compares value *and* type
- `!==` : negate strict equality

```
1 === '1';    // false
0 === false; // false
```

Important: It is recommended to always use *strict* equals.

Testing propositions: `if` & `else`

`if` statements work as in many languanges:

```
if (condition){
    /* Do something */
} else {
    /* Do some other thing */
}
```

They can also be nested

```
const password = prompt("please enter new password");

if (password.length >= 6){
    if (password.indexOf(' ') === -1){
        console.log ("Valid password");
    } else {
        console.log("Password cannot contain spaces!")
    }
} else {
    console.log("Password too short!! Must be 6+ characters")
}
```

Truthy and Falsy values

We know that there is only one value that is *true* and one that is *false*. But in JS, every value has a "truth" value, and can evaluated evaluated to `true` or `false`, depending on the following rules:

1. These values are evaluated as `false` (falsy values):
    - 0
    - "" (empty string)
    - null
    - undefined
    - NaN
2. Every other value is evaluated as `true` (truthy).

Logic operators

- AND: `&&`
- OR: `||`
- NOT: `!`

The previous nested example can be re-arranged as:

```
const password = prompt("please enter new password");

if (password.length >= 6 && password.indexOf(' ') === -1){
    console.log ("Valid password");
} else {
    console.log("Incorrect format for password!")
}
```

And the museum example can be re-written as:

```
const age = prompt("How old are you?");

if (age < 5) {
    console.log("Free");
} else if (age < 10 || age >= 65) {
    console.log("Child / Senior: $5");
} else {
    console.log("General: $10");
}
```

Switches

Switch statements allow us to control the flow when several cases are presented for a given value:

```
const day = 2;
switch(day){
    case 1:
        console.log("Monday");
        break;
    case 2:
        console.log("Tuesday");
        break;
    /* ... */
    case 6:
    case 7:
        console.log("Sunday");
        break;
    default:
        console.log("I do not know that day :c");
        break;
}
```

JS Arrays

Arrays allow us to *group* data together in a ordered manner. In other words, they are an ordered collection of information.

Array syntax in JS is the following:

```
let emptyArray = [];
let numbers = [6, 23, 37, 48, 55];
let colors = ["yellow", "orange", "green"];
let stuff = ["spoon", true, 42, null, 8.8234, undefined];
```

One important note is that arrays are of type `object`, which means there is not possible to tell if a variable is a string by simply testing the `typeof`:

```
typeof([]);     // 'object'
```

Array indices

Every element of the array can be referenced by using indices:

```
let dwarves = ["Thorin II", "Fíli", "Kíli", "Óin"];
dwarves[2];      // 'Kíli'
console.log(dwarves);        // ['Thorin II', 'Fíli', 'Kíli', 'Óin']
```

Reassigning array element values

Even when an array may be declared as constant, we can update the content of an element by reassigning it using the index reference:

```
const dwarves = ["Thorin II", "Fíli", "Kíli", "Óin"];
dwarves[1] = "Bombur";
console.log(dwarves);        // ['Thorin II', 'Bombur', 'Kíli', 'Óin']
```

Array methods

Arrays have methods that allow modification and calculations based on its contents. Some of the most common are:

- `push(element)`: appends `element` to the end of the array and returns the length of the resulting array
- `pop()`: returns the last element in the array and removes it
- `shift()`: returns the first element in the array and removes it
- `unshift(element)`: appends `element` to the beginning of the array and returns the length of the resulting array

```
let movieLine = ["tom", "nancy"];
movieLine.push("oliver");   // returns 3
movieLine.pop();            // returns 'oliver'
```

We can also pass several elements to `push` and `unshift` to add the items in one instruction:

```
let movieLine = [];
movieLine.push("ben", "karl", "marie", "joane");    // returns 4
movieLine.unshift("frank", "kat");                         // returns 6
```

IMPORTANT: unlike strings, this methods modify the array. Remember that most string methods return a modified copy of the original string.

These methods allow us to implement data-structures that mimic the behaviour of Queues and Stacks.

Array Concatenation

Arrays can be concatenated, wich means, we can put the contents of two arrays into a new one, preserving the order:

```
let cats = ["blue", "kitty"];
let dogs = ["rusty", "wyatt"];
cats.concat(dogs);      // ['blue', 'kitty', 'rusty', 'wyatt']
```

Note that cats is not modified

Find an element in the array

```
let comboParty = cats.concat(dogs)
comboParty.includes("blue");    // true
comboParty.indexOf("blue");     // 0
```

Slice and Splice

These work similarly to the slice and splice methods that strings have. These methods are descructive, which means that they modify the original object.

Equalities

Testing equality for arrays can be counter-intuitive, because JS does not test what is stored in the array, but the *reference* (address):

```
[1, 2, 3] === [1, 2, 3];    // false
[1, 2, 3] == [1, 2, 3];     // false
```

```
let nums = [1, 2, 3];
let numsCopy = nums;
numsCopy.push(4);
console.log(nums);      // [1, 2, 3 , 4]
```

Important note on `const` and arrays

Given that arrays are the *reference* to a collection, the `const` keyword is often used to keep the reference safe, while the content can be changed whenever we want:

```
const cats = ["kitty", "pearl", "fluff"];
cats.push("tim");           // ok, no error
cats = ["alex", "ditzy"]    // throws an error!
```

Arrays of arrays

Classics of arrays: arrays can contain arrays.

Destructuring arrays

```
const animals = ['cat', 'dog', 'penguin', 'turtle', 'rat', 'bird'];
const [cat, dog, ...everyAnimalElse] = animals;
```

```
console.log(cat);                // prints 'cat'
console.log(everyAnimalElse)); // prints ['penguin', 'turtle', 'rat', 'bird']
```

JS object literals

In JS is very common to store information by using *object literals*, which are collection of data, like arrays, but instead of having slots and indices, we have *properties*, which are *key-value* pairs.

The key is the label necessary to access the data of a property, like the indices in arrays.

```
const fitBitData = {
   totalSteps:          308727,
   totalMiles:          211.7,
   avgCalorieBurn:      5755,
   workoutsThisWeek:    '5 of 7',
   avgGoodSleep:        '2:13'
};
```

As well as arrays, object literals are references and we access the information by using the keys:

```
const person = {firstName: 'Mick', lastName: 'Jagger'};
console.log(person.firstName);  // prints 'Mick'
console.log(person.["firstName"]);  // prints 'Mick'
console.log(person.lastName);  // prints 'Jagger'
console.log(person["lastName"]);  // prints 'Jagger'
```

Why do both syntax exist? 1. JS converts every key to a string when creating the properties, so every word or set of characters can be a key. Thus we can use the dot notation whenever a key is a *word*, but not when it has a number or special characters at the begining 2. To sort this, the brackets notation comes handy, because it can be used with any string 3. Also, bracket notation allows the creation of new properties when the object literal has already been created

```
const person = {firstName: 'Mick', lastName: 'Jagger'};
person['age'] = 78;
console.log(person);    // { firstName: 'Mick', lastName: 'Jagger', age: 78 }
```

Objects values can be anything: primitive data types, arrays or even objects!

```
const product = {
    name: "Gummy Bears",
    inStock: true,
    price: 1.99,
    flavors: ["grape", "apple", "cherry"]
}
```

Two of the most common data patterns is having both arrays and object literals combined:

```
const shoppingCart = [
    {
        product: 'Jenga Classic',
        price: 6.88,
        quantity: 1,
    },
    {
        product: 'Risk',
        price: 24.33,
        quantity: 1,
    },
    {
        product: 'Scrabble',
        price: 16.99,
        quantity: 2,
    },

];
```

```
const student = {
    firstName: 'David',
    lastName: 'Jones',
    strengths: ['Music', 'Art'],
    exams: {
        midterm: 92,
        final: 88,
    }
};
```

Destructuring arrays

```
const student = {
    firstName: 'David',
    lastName: 'Jones'
};

const {firstName, lastName} = user;

console.log(firstName);     // 'David'
console.log(lastName);      // 'Jones'
```

Loops and Iterations

Whenever a process must be repeated several times is always better to use *lopps*. JS has many ways of looping, but the most common are: - `for` - `while` - `for..in` - `for..Of`

`for` loops

Every `for` loop has three parameters that are relevant: - counter variable assignement - break conition - increment counter

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

```
for (let i = 0; i < 10; i++){
    for (let j = 0; j < 10; j++){
        console.log(i, j);
    }
}
```

```
const animals = ['lion', 'tiger', 'bear'];
for (let i = 0; i < animals.length; i++){
    console.log(i, animals[i]);
}
```

```
const animals = ['lion', 'tiger', 'bear'];
for (let i = animals.length - 1 ; i >=0 ; i--){
    console.log(i, animals[i]);
}
```

Be careful with infinite loops! In this case, `i` will never reach the break condition

```
for (let i = 1; i <= 10; i--){
    console.log(i);
}
```

`while`

`while` loops only need one parameter: the break condition.

```
let playerScore = 0;
let gameOver = false;
while (!gameOver){
    playerScore += Math.floor((Math.random() * 10));
    if (playerScore >= 10){
        gameOver = true;
    }
}
```

Iterate through arrays with `for of`

Far more readable <3

```
const topics = ['cringe', 'books', 'chickens', 'funny', 'pics', 'soccer'];
for (let topic of topics){
    console.log(topic)
}
```

This can also use to iterate over *any* `iterables`, like strings. However, object literals are not iterables. For iterating through object literals the proper looping tool is `for in`

Iterate through object literals with `for in`

```
const testScores = {john: 23, damon: 54, karl: 29, vonnie: 37};
for (let student in testScores){
    console.log(`${student}: ${testScores[student]}`);
}
```

There is also an option of getting object literal keys as an array and values as an array:

```
const testScores = {john: 23, damon: 54, karl: 29, vonnie: 37};
Object.keys(testScores);    // [ 'john', 'damon', 'karl', 'vonnie' ]
Object.values(testScores);  // [ 23, 54, 29, 37 ]
Object.entries(testScores); // [ [ 'john', 23 ], [ 'damon', 54 ], [ 'karl', 29 ], [ 'vonnie
```

Functions

Functions are *reusable procedures*, useful for making the code easier tounderstand.

From of the cases we have covered, consider the following example, where we have several dice rols

```
let die1 = Math.floor(Math.random() * 20);
let die2 = Math.floor(Math.random() * 12);
let die3 = Math.floor(Math.random() * 6);
```

Function declaration

With functions we can write the following `rollDice()` function which can also have an argument were we pass the number of sides of a dice:

```
function rollDice(sides) {
    return Math.floor(Math.random() * sides) + 1;
}

let die1 = rollDice(20);
let die2 = rollDice(12);
let die3 = rollDice(6);
```

Functions as variables

There are many different ways of defining functions, apart from the one we defined previously, but one of the most common is assigning it to a variable:

```
const add = function (x, y) {
    return x +y;
}
```

In this case, we do not assign a name to the function, but we assign it to the variable `add`. In order to call the function, we can just call `add(i,j)` and it will execute the function.

This can happen in JS because even functions are treated as variables and one of the most powerful features of the language is that a function can return functions and be passes around like they are anything else. For example:

Arrow functions

Arrow functions are a compact way of defining functions, but without naming them. How is it possible? An arrow function only needs a list of arguments and the result it produces, both linked by an arrow `=>`. For example

```
const succesor = (num) => {return num + 1};
console.log(succesor(4));        // prints 5
```

There are some cases when arrow functions can be written in a more compact way: - When arrow functions recieve only one argument, the parenthesis can be omitted: `js   const succesor = num => {return num + 1};   ```` - When we want the result inmediately *and* there is only one expression to be evaluated in the function, we can use the *implicit return* notation, where the braces and the `return` keyword are omitted;`js const succesor = num => num + 1 ; `If we need to return an object, it must be wrapped in parenthesis:`js const object = params => ({param: "a"}) ; "'

Important note Furthermore, a function can be defined with this syntax and without assigning it to a variable. This kind of functions are called *anonymous functions*, which are very useful when dealing with multiple procedures that share logic, but depend on specific parameter values. More on this topic in the following sections.

Higher-order functions

Every function that recieves one or more function as an argument or returns one or more functions are result is called as a *higher-order* function.

Function as argument

```
function callTwice (func) {
    func();
    func();
}

function rollDice () {
    const roll = Math.floor(Math.random() * 6) + 1;
    console.log(roll);
}

callTwice(rollDice);    // prints two times
```

## Return a function There are several applications where we may want to return functions depending on parameters. The following code example is based on a code pattern called *factory* and is widely for several purposes:

```
function makeBetweenFunc (min, max) {
    return function (num) {
        return (num >= min && num <= max);
    }
}

const isChild = makeBetweenFunc(0, 17);
const isAdult = makeBetweenFunc(18, 64);
const isSenior = makeBetweenFunc(65, 100);

console.log(isChild(5));    // true
console.log(isAdult(17));   // false
console.log(isSenior(95));  // true
```

Add methods to objects

Combining the information above, we can see that object literals can have functions as property values, which then behave as methods:

```
const myMath = {
    PI: 3.1415,
    square: function (num) {
        return num * num;
    },
    cube: function (num) {
        return num * num * num;
    }
}
console.log(myMath.PI);          // prints 3.1415
console.log(myMath.square(2))    // prints 4
console.log(myMath.cube(4))      // prints 64
```

Recently, there is a shorthand for omitting the `function` keyword:

```
const myMath = {
    add (x, y) {
        return x + y;
    },
    multiply (x, y) {
        return x * y;
    }
}
```

Accessing arguments of a function

This applies to every function except arrow ones. Every function comes with an array-like variable that can be accessed through the `arguments` keyword *inside* the function. It is an *array-like* because it behaves as an array in the way it stores the data and how we can access each element, but it has not the buil-in methods that every other iterable has.

In order to deal with this "issue", we can use the *rest params* syntax:

```
function sum (start_number, ...rest) {
   console.log(start_number, rest);
}
```

This syntax means that all the parameters we enter after the mandatory ones, will be stored in the array `rest` (we can choose any other name; the three dots are the ones that matter).

Destructuring arguments

```
const student = {
    firstName: 'David',
    lastName: 'Jones'
};

function fullName({firstName, lastName}){
    return `${firstName} ${lastName}`
}

console.log(fullName(student));     // prints 'David Jones'
```

This is possible because in the declaration we state that the function expects an object, so using the destructuring feature of the objects we can use the keys that we know and use them directly in the function body.

Scopes

The *scope* of a variable refers to its *visibility* or *where in the code* can the variable be used. JS has three scopes: function, block and lexical.

Function Scope

Variables that are defined (declared) inside a function cannot be referenced out of the function.

```
function birdWath() {
    let bird = "Nighingale";
    console.log(bird)
}

birdWatch();         // prints 'Nighingale'
console.log(bird)    // prints undefined
```

Variables defined out of the function can be used inside a function, but also, a new variable that uses the same namespace as an outer variable can be declared inside a function:

1. Variable declared out of the function and used in the function

   ```
   let bird = "Nighingale";
   function birdWatch() {
       bird = "Sparrow";
       console.log(bird)
   }

   console.log(bird)   // prints 'Nighingale'
   birdWatch();        // prints 'Sparrow'
   console.log(bird)   // prints 'Sparrow'
   ```

2. Variable with same name declared inside the function

   ```
   let bird = "Nighingale";
   function birdWatch() {
       let bird = "Sparrow";
       console.log(bird)
   }

   console.log(bird)   // prints 'Nighingale'
   ```

```
birdWatch();        // prints 'Sparrow'
console.log(bird)   // prints 'Nighingale'
```

Block Scope

A *block* is every piece of code that is inside a pair of braces `{ }`. Functions, for loops and conditionals are blocks. Every variable declared with `let` and `const` inside a block will have *block scope*. `var`declarations, however, have function scope.

Lexical Scope

Any inner block or function has access to the variables defined in a parent or any of its succesive parents, but not the other way around.

```
function bankRobery(){
    const heroes = ['Spiderman', 'Wolverine', 'Black Panther'];
    function cryForHelp() {
        for  (let hero of heroes) {
            console.log(`Please help us, ${hero}!`)
        }
    }
    cryForHelp();
}
```

`this`

The keyword `this` refers to the object itself. When called inside an object, the keyword will hold every variable and method that it has been bound to (via property-value assignements).

However, `this` value depends on the context where it is run. Lets see the following example when run in the terminal with node:

```
const cat = {
    name: "Blueberry",
    meow () { console.log(`${this} says meow!!`)}
}

const justMeow = cat.meow;  // note that the function has not been called!

cat.meow();      // prints 'Blueberry says meow!!'
justMeow();      // prints 'undefined says meow!!'
```

However, if the same code is run in a browser, the last result is very different:

```
justMeow();      // prints 'usi_aff=1=1646771350149; says meow!!'
```

Why? Every method that calls `this` will assert the value of `this` as the object that is calling the method. But 'justMeow' is not called from an object in the previous example… or is it?

Well, everything in JS is an object. Even an application, written in a plain `.js` file, has an object that serves as a container for the variables, functions and objects it creates when run.

In node, the *top level object* is the following;

```
Welcome to Node.js v16.1.0.
Type ".help" for more information.
> this
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  performance: [Getter/Setter],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}
```

But in Chrome, the top level object is filled with lots of parameters:

```
> this
Window {0: Window, 1: Window, 2: Window, 3: Window, 4: global, window: Window, self: Window
```

Array Callbacks

Arrays have powerfull functions that iterate throw the array and recieve another function as an argument, called *callback*. The callback will be executed with every element of the array as an argument.

`forEach` method

```
const numbers = [1, 2, 3, 4, 5];
function print(thing){
    console.log(thing);
}

numbers.forEach(print);     // prints every element of numbers in a single line
```

Moreover, we can use anonymous functions if it is a function that we won't call again

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach( e => console.log(e) );
```

`map` method

The `map` method allows the *reproduction* of an array, but with an added process that is computed for each element:

```
const movies = [
    {
        title: "Kill Bill vol. 1",
        score: 95
    },
    {
        title: "Bug's Life",
        score: 89
    },
    {
        title: 'Avengers Endgame',
        score: 91
    },
]
const titles = movies.map(movie => movie.title);
console.log(titles);    // returns [ 'Kill Bill vol. 1', "Bug's Life", 'Avengers Endgame' ]
```

`filter` method

One useful method for filtering data is the `filter` method, that only accepts callbacks that return boolean values:

```
const goodMovies= movies.filter(m => m.score >90);
console.log(goodMovies);        // returns [ { title: 'Kill Bill vol. 1', score: 95 }, { ti
```

Test elements with `every` and `some` methods

These methods allow to check if *every* or *some* elements complies with a certain condition, passed as callback:

```
movies.every( m => m.score > 90);   // returns false
movies.some( m => m.score > 90);   // returns true
```

`reduce` method

Executes a reducer function on each element of an array and returns only one value, for example, when we sum all the elements:

```
const numbers = [1, 2, 3, 4, 5];
numbers.reduce( (accumulator, currentValue) => {
    return accumulator + currentValue;
});
```

The Document Object Model

The DOM is a JavaScript representation of the elements that are contained in a page. It is a tree where the nodes are the html elements.

In a browser document object can be accessed `document`. The actual JS object can be seen with `console.dir(document)`.

Selecting elements

- `document.getElementById(id)`: fetch the element of the document that matches the `id` (as a string). Returns the object that represents the element.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Unicorn</title>
</head>
<body>
    <h1 id="mainheading">I &hearts; unicorns</h1>
    <img src="https://devsprouthosting.com/images/unicorn.jpg" id="unicorn" alt="unicorn">
</body>
</html>
```

Fetching the image and the heading of the html above is accomplished by the following js script:

```
const image = document.getElementById('unicorn');
const heading = document.getElementById('mainheading');
```

- `document.getElementsByTagName(tag)`: fetch *all* the elements of the document that match the specified tag (as a string). Returns an *html collection* which is an array-like of objects that represent all the matching html elements. Since it is not an array, we do not have acces to array methods we saw before (filter, map, etc.), but we can still iterate through them:

```
const allImages = document.getElemetsbyTagName('img');
console.log(allImages.length);      // prints the number of images in the document
```

```
for (let e of allImages) {
    console.log(e.src)          // prints the 'src' attribute of every image
}
```

- `document.getElementsByClassName(class)`: fetch *all* the elements of the document that match the specified class (as a string). Returns an *html collection* which is an array-like of objects that represent all the matching html elements.

Query Selector

A modern way of fetching the elements of a document in a more concise way. It returns the first element in the DOM tree that matches the specified one of the identifiers:

```
document.querySelector('p a');
```

The `querySelectorAll()` variant returns all the elements that match the specified identifiers.

Manipulating

Once we have fetched an element, then we can alter its properties by simply setting them:

```
document.querySelector('p').innerText = 'This is the new text :)';
```

```
const allLinks = document.querySelectorAll('a');
for (let link of allLinks) {
    link.innerText = "I AM A LINK!"
}
```

It is important to know common attributes that can be edited: - `innerText`: all the text that belongs to an element - `innerHTML`: all de *markup* that belongs to an element *and* that is being displayed. - `textContent`

`getAttribute` and `setAttribute`

Change Styles

Inline styles

```
const h1 = document.querySelector('h1');
h1.style.color = 'green';
h1.style.fontSize = '15em';

const allLinks = document.querySelector('a');
```

```
for (let link of allLinks){
    link.style.color = 'rgb(100,100,100)';
    link.style.fontFamily = 'sans-serif';
}
```

Manipulate classes

We can do this:

```
const h1 = document.querySelector('h1');
h1.getAttribute(class);
h1.setAttribute(class, 'border');
```

But its cumbersome, so we can actually access the list of classes:

```
h1.classList.add('purple')
h1.classList.remove('purple')
h1.classList.toggle('purple')   // adds or removes
```

Traverse through elements

```
const firstBold = document.querySelector('b');
firstBold.parentElement;    // accesses the parent element
firstBold.children;         // returns an HTMLCollection of children (array-like)
```

```
const squareImg = document.querySelector('.square');
squareImg.previousElementSibling;
squareImg.nextElementSibling;
```

Create and remove elements

Append child

```
const newImg = document.createElement('img');  // creates an *empty* image
newImg.src = 'hhttps://upload.wikimedia.org/wikipedia/commons/7/75/Cute_grey_kitten.jpg';
document.body.appendChild(newImg)
```

Insert as sibling

```
const h2 = document.createElement('h2');
h2.append("This is an h2");
const h1 = document.querySelector('h1');
h1.insertAdjacentElement('afterend', h2);   // inser h2 after h1 ends
```

There are for `position` values that can be used: - `beforebegin`: just before the target element - `afterbegin`: insert inside target element as first child - `beforeend`: insert inside target element as last child - `afterend`: just after the target element

remove and removeChild

```
const b = document.querySelector('b')
b.parentElement.removeChild(b);
```

This does the same job but its called directly in the node:

```
b.remove();
```

DOM Events

This section covers how to respond to user inputs and actions: the most powerful features of the web!

By default, we can access events in the attributes of some elements, but it is not recommended to do so:

```
<button onclick="alert('you clicked me!')">Click Me!</button>
```

With JS we can achieve that behaviour in a much more friendly and mantainable way:

```
const btn = document.querySelector('button');
btn.onclick = () => {
    console.log('clicked!')
}
```

But wait! There is a much better way to handle events using event listeners with the `addEventListener` method:

```
const btn = document.querySelector('button');
btn.addEventListener(
    'click',
    () => { console.log('clicked!') },
    { once: true }
);
```

Why is it better? Some of the reasons are: - `addEventListener` allows to have as many functions as we want associated with the same event type. On the contrary, the property assignement will always override the function if we assign more than one (the last will prevail). - `addEventListener` also accepts `options` that allow better control of how the functions are bound, for example: `js const btn = document.querySelector('button'); btn.addEventListener( 'click', () => { console.log('clicked!') }, { once: true } );`

Event listeners and `this`

Supose we have several elements in a document that we want to *colorize* when clicked, i.e.: change its background color:

```
const buttons = document.querySelectorAll('button');
for (let button of buttons){
    button.addEventListener('click', function () {
        button.style.backgroundColor = randomColor();
    })
}

const paragraphs = document.querySelectorAll('p');
for (let p of paragraphs){
    p.addEventListener('click', function () {
        p.style.backgroundColor = randomColor();
    })
}
```

We could be repeating the same logic inside the callbacks, but there is a better way:

```
function colorize() {
    this.style.backgroundColor = randomColor()
};

const buttons = document.querySelectorAll('button');
for (let button of buttons){
    button.addEventListener('click', colorize);
}

const paragraphs = document.querySelectorAll('p');
for (let p of paragraphs){
    p.addEventListener('click', colorize);
}
```

Then we can use this to extend the behaviour to any othe element we wnant and the `this` keyword will do de work of identifying the context object from which it has been called :)

Event objects

When working with keyboard events, such as key press or release, or mouse events, the `addEventListener` provides an object that contains the information of the key that triggered the event:

```
const input = document.querySelector('input');
input.addEventListener('keydown', function (e) {
    console.log(e)
})
```

Important: This binding will trigger the event only when the element is focused, which only work with inputs!

If we want to trigger the event *globally*, we must bind the callback to the `window` object

Form events

By default, forms perform an acation right after being submitted. Commonly, we don't want that behaviour because it generally takes the user to another page and reloads the browser. To prevent this, we can add `preventDefault()` to the event that is being called when we define the callback for an event listener:

```
const form = document.querySelector('form');
form.addEventListener('submit', function (e) {
    e.preventDefault();
    /** do something else **/
});
```

Input and Change events

Changes when we leave the user clicks outside the input element, after editing its value:

```
input.addEventListener('change', function (e) {
    console.log(input.value);
})
```

Changes whenever the value of the input element changes:

```
input.addEventListener('input', function (e) {
    console.log(input.value);
})
```

Event bubbling

Some times, the same event can trigger events of an element an one or more of its succesive parents.

```
<div class="container">
    <button> Click me!</button>
</div>
```

```
const container = document.querySelector('.container');
container.addEventListener('onclick', function (e) {
    /* Do something */
});

const button = document.querySelector('button');
button.addEventListener('onclick', function (e) {
    /* Do other things */
    e.stopPropagation();
});
```

Event Delegation

Bind events to the parent of an element that *might* exist in the future, but still be bound to the child when it is created:

```
list.addEventListener('click', function (e) {
    e.target.nodeName === 'LI' && e.target.remove();
})
```

JS async

The call stack

JS, as many other languages, has a call stack that stores all the functions that have been called and yet to be resolved. Its data structure is a stack, so last in first out.

JS is single threaded

But browsers implement behaviours where JS can hand off the job to the browser, keep executing the code and then the browser answers with the result.

Handling the timing

This code will wait during *almost* the same period of time to change colors, so we will only see 1 color change

```
setTimeout( () => {
    document.body.style.backgroundColor = 'red'
}, 1000);

setTimeout( () => {
    document.body.style.backgroundColor = 'orange'
}, 1000);
```

Instead, what we can do is *nest* the callbacks

```
setTimeout( () => {
    document.body.style.backgroundColor = 'red';
    setTimeout( () => {
        document.body.style.backgroundColor = 'orange'
        }, 1000);
    }, 1000);
```

Or in a more “““*readable*”““ way:

```
const delayedColorChange = (color, delay, doNext) => {
    console.log(color);
    setTimeout( ()=> {
        document.body.style.backgroundColor = color;
        doNext && doNext();
    }, delay);
}

delayedColorChange('red', 1000, () => {
    delayedColorChange('green', 1000, () => {
        delayedColorChange('yellow', 1000, () => {
            delayedColorChange('blue', 1000, () => {
            });
        });
    });
});
```

This is a mess and in web applications there are lots of jobs that need to wait for something to be completed, for example, lets say we have a function that makes a request to a fake server url and executes a `success` callback or `failure` callback, depending on if it reached the timeout:

```
const fakeRequestCallback = (url, success, failure) => {
    const delay = Math.floor(Math.random() * 4500) + 500;
    setTimeout(() => {
        if (delay > 4000) {
            failure('Connection Timeout :(')
        } else {
            success(`Here is your fake data from ${url}`)
        }
    }, delay)
}
```

If we try to build a simple application of two succesive requests, things get real messy:

```
fakeRequestCallback('books.com/page1',
    (response) => {
        console.log("It worked :)")
        console.log(response)
        fakeRequestCallback('books.com/page2',
            (response) => {
                console.log("It worked :)")
                console.log(response)
            },
            (err) => {
                console.log("Failed :c")
            });
    },
    (err) => {
        console.log("Failed :c")
    }
);
```

And we have only two nested requests!!

The `Promise` approach simplifies this with the following syntax:

```
const fakeRequestPromise = (url) => {
    return new Promise((resolve, reject) => {
        const delay = Math.floor(Math.random() * (1500)) + 500;
        setTimeout(() => {
            if (delay > 1500) {
                reject('Connection Timeout :(')
            } else {
                resolve(`Here is your fake data from ${url}`)
            }
        }, delay)
    })
}

fakeRequestPromise('favkitten.com/page1')
    .then(data => {
        console.log(data);
        return fakeRequestPromise('favkitten.com/page2')
    })
    .then(data => {
        console.log(data);
        return fakeRequestPromise('favkitten.com/page3')
    })
    .then(data => {
        console.log(data);
        return fakeRequestPromise('favkitten.com/page4')
    })
    .catch(err =>{
       console.error(err)
    });
```

Promises

```
const delayedColorChange = (color, delay) => {
    return new Promise( (resolve, rejecct) => {
        setTimeout( () =>{
            document.body.style.backgroundColor = color;
            resolve()
        }, delay)
    })
}

delayedColorChange('red', 500)
    .then(() => delayedColorChange('orange', 500))
    .then(() => delayedColorChange('yellow', 500))
    .then(() => delayedColorChange('green', 500))
    .then(() => delayedColorChange('lightblue', 500))
    .then(() => delayedColorChange('blue', 500))
    .then(() => delayedColorChange('indigo', 500))
```

async functions

:sparkles: Syntax sugar for promises :sparkles:

`async` function declaration makes the function return a `Promise`. The status of the `Promise` is determined through the following rules: - If the function returns something, *anything*, the `Promise` will be considered resolved - If the function throws an error, the `Promise` will be rejected.

```
const login = async (username, password) => {
    if (!username || !password) throw 'Missing Credentials';
    if (password === 'kittycats') return 'Welcome';
    throw 'Invalid password'
}

login('Rue')
    .then(msh => console.log(msg))
    .catch(err => console.log(err))
```

`await`

```
async function rainbow() {
    await delayedColorChange('red', 500)
    await delayedColorChange('orange', 500)
    await delayedColorChange('yellow', 500)
    await delayedColorChange('green', 500)
    await delayedColorChange('blue', 500)
    await delayedColorChange('indigo', 500)
    await delayedColorChange('violet', 500)
}

async function printRainbow() {
    await rainbow();
    console.log("Finished rainbow");
}
```

Handling rejected `Promises` inside `async` functions

Use `try/catch` :)

```
async function makeTwoRequests() {
    try {
        let data1 = await fakeRequest('/page1');
        console.log(data1);
        let data2 = await fakeRequest('/page1');
        console.log(data2);
    } catch (e) {
        console.log("Error:", e)
    }
}
```

HTTP Requests with JS

The Request-Response data cycle

nice drawing of request sent to a server, made by a client, and the response the server sends back to the client, after processing the data

Requests

Requests are always made to a *server* and can include several parameters and user-specific information, such as passwords, tokens and or keys to validate identity or permissions to access the data.

Responses

Responses always contain one *status code*, which is a number that state the result of the request. The are defined by the HTTP standards.

Also, a response can have HTML description of a page, so the client browser can render it to the screen.

But in modern applications, we want to reload sites as few times as possible. To accomplish this, many servers implement web *Application Programming Interfaces* that allow the retrieval of the bits of data we need, upon requests. Then the client browser only needs to update that specific data, instead of re-rendering the whole page.

AJAX & JSON

In past days, most requests were done using *Asynchronous JavaScript and XML* (AJAX) that worked with JS requests an HTTPXML formats, however, in present days, the *XML* format is very uncommon and has been replaced with *JavaScript Object Notation*, which is *almost* the same as the way we write JS object literals. It is not exactly the same because it does not support `undefined` and other types, but it support most common primitive data types.

Common JSON methods used in applications: - `JSON.parse` - `JSON.stringify`

The Fetch API

It's useful and far more easy to use than *XML HTTP Requests* (XHR). But it does not provide the data straight away. We have to do at least two steps: fetch and wait for the data.

```
fetch('https://pokeapi.co/api/v2/pokemon/pikachu')
    .then(response => response.json())
    .then(data => console.log(data.name))
    .catch(error => console.error('Request Failed'));
```

To overcome this *issue*, we can wrap the fetch in another async function:

```
const fetchPokemon = async (pokemon_name) => {
    try {
        const res = await fetch(`https://pokeapi.co/api/v2/pokemon/${pokemon_name}`);
        const data = await res.json();
        console.log(data.name)
    } catch (e) {
        console.error('Something went wrong', e)
    }
};
```

It's not the best, but it is ok :)

Axios

Axios is an open sourced JS library that allows to do the fetching far easier. The request has only one step:

```
axios.get('https://pokeapi.co/api/v2/pokemon/pikachu')
    .then(res => console.log(res.data.name))
    .catch(err => console.log("Something went wrong :c", err))
```

And it can be wrapped not because of the step, but for error handling:

```
const fetchPokemon = async (pokemon_name) => {
    try {
        const res = await axios.get(`https://pokeapi.co/api/v2/pokemon/${pokemon_name}`)
        console.log(res.data.name);
    } catch (e) {
        console.log('Somthing went wrong :c', e);
    }
};
```

Prototypes, Classes and Object Oriented Programming

In this chapter we cover the JS flavour of Object Oriented Programming (*OOP*). Most important

Object Prototypes

Quoting MDN article: > *Object prototypes* are the mechanism by which JS objects inherit features from one another. … >Objects can have a *prototype* object that acts as a *template*

For example, if we look at arrays that are created with the squared brackets notation, they have no associated methods, however, it is possible to call several methods like `push()`, `length`, etc. This is possible because the `Array` has a `__proto__` property that is the *template*, which contains the methods that the `Array` can inherit. Take a look at the following console inputs and outputs:

```
> const a = [2, 3, 4]
3
```

```
> a.sing = () => console.log("La La La!");
() => console.log("la la la")
```

```
> a
[2, 3, 4, sing: f]
```

```
> a.__proto__
[constructor: f, concat: f, copyWithin: f, fill: f, find: f, ...]
```

Prototype definitions & `__proto__` instances

Prototypes definitions can be accessed by their *class* names using the syntax `Class.prototype`. for example, `String.prototype` is the definition for `String` *template* objects and `Array.prototype` is the definition for `Array` *template* objects.

If we want to access the prototype *instance* (not the class) of an object, the syntax is `obj.__proto__`, like we previously did for the array `a`.

Add and modify prototype methods

Prototypes can be edited so methods can be overriden and new methods can be created. This changes will remain until the application is closed, when running with node, or when the browser refreshes:

```
String.prototype.yell = function () {
    return `${this.toUpperCase()}!!!!!1`;
};

console.log("hi there".yell()) // prints 'HI THERE!!!!!1'
```

One important note is that even though redefining and/or adding new methods to prototypes is possible, it it nod a good practice to do so.

Factories

```
function makeColor(r, g, b) {
    const color = {}
    color.r = r;
    color.g = g;
    color.b = b;
    color.rgb = function () {
        const { r, g, b } = this;
        return `rgb(${r}, ${g}, ${b})`;
    };
    color.hex = function () {
        const { r, g, b } = this;
        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);
    };
    return color;
}
```

The problem with this approach is that every object has its own definition for every method, i.e., every time an object is created, all functions are created from scratch and are saved. This is bad for our cpu and memory resources.

If you exectute at the following code, you will notice that every color object has its own rgb, but `Strings`, for example, have the exact same methods (or method references, to be exact).

```
const black = makeColor(0, 0, 0);
const white = makeColor(255, 255, 255);

black.rbg === white.rgb // returns false!
```

```
"hi".slice === "bye".slice  // returns true!
```

This is possible because `Strings` hav their methods defined in their prototypes! The feature that makes it possible is the `new` keyword.

Constructors (the old ways)

Take a look at the following function. It returns nothing, more over, the `this` points to the *top-level object*, because the function scope is global.

```
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b
}
```

However, we can use this function as a *constructor*, which is a *waaay* better solution for creating objects than factories. This is accomplished by calling the function with the `new` keyword just before:

```
const black = new Color(0, 0, 0);
```

The `new` keyword perform the following steps: 1. Create a *blank*, plain JS object 2. Link (*set the constructor of*) this object to another object, using the function as *constructor* 3. Pass the newly created object. from step 1, as the `this` context 4. Returns if the function does not return its own object.

Note that the `black` object has `r`, `g` and `b` as properties, as well as `__proto__`.

If we want to add methods, we can do as we did when modifying the `String.prototype`:

```
Color.prototype.rgb = function () {
        const { r, g, b } = this;
        return `rgb(${r}, ${g}, ${b})`;
    };

Color.prototype.hex = function () {
        const { r, g, b } = this;
        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);
    };
```

If we now test the method reference, we see that the functions for two objects, built by this constructor, share the same reference, but the methods return values depending on the `r`, `b` and `g` properties of each object, thanks to the `this`:

```
const black = new Color(0, 0, 0);
const white = new Color(25, 255, 255);

black.rgb === white.rgb;     // returns true!
console.log(black.rgb());    // prints 'rgb(0, 0, 0)'
console.log(white.rgb());    // prints 'rgb(255, 255, 255)'
```

Classes

:sparkles: Syntax sugar for constructors :sparkles:

```
class Color {
    constructor(r, g, b, name){
        this.r = r;
        this.g = g;
        this.b = b;
        this.name = name;
    }

    rgb () {
        const { r, g, b } = this;
        return `rgb(${r}, ${g}, ${b})`;
    };

    hex () {
        const { r, g, b } = this;
        return "#" + ((1 << 24) + (r << 16) + (g << 8) + b).toString(16).slice(1);
    };

};
```

Inheritance

```
class Cat {
    constructor (name) {
        this.name = name
    }
    eat() {
        return `${this.name} is eating`;
    }
    meow() {
        return `Meooowww!!!`
    }
}


class Dog {
    constructor (name) {
        this.name = name
    }
    eat() {
        return `${this.name} is eating`;
    }
    bark() {
        return `Woof Wooof!!!`
    }
}
```

`extend`

```
class Pet {
    constructor (name) {
        this.name = name;
    }
    eat() {
        return `${this.name} is eating`;
    }
}

class Cat extends Pet {
    meow() {
        return `Meooowww!!!`;
    }
}

class Dog extends Pet {
    bark() {
        return `Woof Wooof!!!`;
    }

}
```

`super`

```
class Cat extends Pet {
    constructor (name, age, livesLeft = 9){
        super(name, age);
        this.livesLeft = livesLeft;
    }
    meow() {
        return `Meooowww!!!`;
    }
}
```

Example Summary

Converting Colors & Crazy Math

The *Beloved* Terminal

The goal of this chapter is to know and to understand the *power* of terminal tools.

If haven't opened a terminal, I hope this guide is clear enough. Terminals and

Operating Systems can be complex topics, but don't panic; there is nothing that can't be fixed.

What is a terminal

A terminal is a text-based interface to your computer. Within the terminal one can access *almost* any part of our computer's operating system. The programs that run in the terminal may vary depending on the system, but most computers are shipped with one or more of the following: - `sh` - `bash`: *the standard for several years* - `zsh`:*the best so far*

Some systems may have their own propietary shell

Navigate through your files and directories

Most operating systems have *file systems*, which means, all the data and we have stored and applications we have installed, *are* in some *place* in the file system. The file system uses mainly two types of *entries*: 1. Files: the blocks of *information* or *applications* 2. Directories: also called *folders*, are the *locations* where entries can be *placed* in the file system.

File systems are usually *trees*, from a data structure perspective, where directories represent *nodes* and files are *leaves* of the tree.

Whenever you open a terminal, it starts in a location by default: the *home* directory. On a Mac, terminals would show some information about the current user (technically is the *host*) and, next to it,the directory where the terminal is placed. Most machines use the ~ symbol as abreviation for the home directory, but to know the *full path*, we can use the `pwd` command, with stands for *print current directory.* Open a terminal and use the `pwd` command (hit Enter to execute the command).

The terminal should show something like this:

```
username@ComputerName ~ $ pwd
/Users/username
```

If you want to see what inside the directory, use the `ls` command, which stands for *list.* Here you may see folders and files

```
~ $ ls
Applications/           Movies/
Desktop/                Music/
Documents/              Pictures/
Downloads/              Public/
Library/                file.txt
```

To change to another directory, we use `cd`, which stands for *change directory*, and next to it, separated by a space, the path to the folder. By default, every command that doesn't prints can be considered *successful*, which means it did not throw errors. `cd` is one of the commands that returns nothing, except when it does not find the specified directory *or* the specified path resolved to a file, intead of a directory:

```
~ $ cd Documents
~/Documents $ ls
file1.txt
file2.txt
file3.txt
~/Documents $ pwd
/Users/username/Documents
~/Documents $ cd
~ $
```

To go back to the home, we can execute `cd` without arguments. It returns to the home by default :)

Important Note >For now on, all code snippets will not include the directory or the `$` symbol. They will be just the commands, with the outputs in separate snippets if necessary

Adding entries

Add directories

To create new directories (or folders), the command is `mkdir`, which stands for *make directory*:

```
mkdir folderA
mkdir .hidden_folder
mkdir folderA/folder_inside_folderA
```

Add files

To create brand new empty files, we can use the `touch` command and the name of the file or its path

```
touch file.txt
touch folderA/file_inside_forlderA.md
```

Copy and Move

To copy files or folders, the command is `cp`, which stands for *copy*. The arguments it recieves are the source entry and the destination entry. When copying a folder we must use the `-r` flag, that stands for *recursive mode*. This flag tells the command to copy all the contents of the directory:

```
cp file.txt file_copy.txt
cp file.txt folderA/file.txt
cp -r folderA folderA_copy
```

To move files or folders, the command is `mv`, which stands for *move*. The arguments it recieves are the source entry and the destination entry. When moving a folder and its contents we must use the `-r` flag, same as with `cp`.

The `mv` command is also used to *rename* entries, as seen in the following example:

```
mv file.txt renamed_file.txt
mv file.txt folderA/moved_file.txt

mv folderA renamed_folder
mv -r folderA
```

Removing entries

To remove entries, the command is `rm`, which stands for *remove*. The arguments it recieves are the source entries, separated by spaces. When removing a folder and its contents we must use the `-r` flag, same as with `cp` or `mv`.

```
rm file.txt
rm folderA/file.txt
rm -r folderA
```

Sometimes, files and directories are protected agains accidental (user-)deletions, so `rm` may ask manual confirmation for each protected entry. To bypass the confirmation we can add the `-f` flag, which stands for *force.*:

```
rm -f protected_file.txt
rm -rf protected_folder
```

Node JS

Node JS is a JavaScript *runtime* environment. Until recently, we could only use JS in a web browser. Node is the most popular implementation of JS runtime that executes code out of a browser.

This allows to have *server-side* applications that can do lots of things, using *packages*, which are libraries built by the community -*or yourself!*-, and having the ability to make applications in the same language as the *front-end* does. And, since it is supported in all operating systems, apps can be built and ported without having to re-compile for each platform.

Most common applications of Node are: - Web servers - Command line tools - Native Apps (like VSCode!) - Video Games - *and much more!*

Install Node

Go to the Node download page, get the installer and click to the end. Then open a terminal and enter `node`. If everything went right, it should look like this:

```
node
Welcome to Node.js v17.4.0.
Type ".help" for more information.
>
```

Running JS code with Node

As seen in previous sections, JS can be run with Node in two ways: the REPL command terminal and file execution.

The Node Read-Evaluate-Print Loop

When we type `node` in the terminal, the program that starts running is a *Node* command terminal which has a REPL. The Node REPL

File execution

`process` & `argv`

File System