JavaScript-theory

- JavaScript can run either in browser for animated components or in NodeJS
- JavaScript(programming language conforming to ECMAScipt) vs ECMAScript(specification-defining standards)
- ECMAScript version 1 in 1997 and in 2015 ES2015 or ES6, and each year after is released a new version.
- The javascript is used between the <script></script> and it can be used either in head(for third party libraries) or usually at the end of the body section
- NodeJS is a runtime environment for executing JavaScript code
- Types in JavaScript: **string, boolean, number, undefined, null**
- Reference Types: object, array and function

JavaScript is a dynamic language: The variable name can be changed at runtime

arrays

- the arrays in javascript can store different types of data

- the arrays are dynamic , can be changed at runtime

functions

-Functions are used to calculate a value or to perfume multiple statement to run a task

- The functions have parameters and when calling that functions the same parameters are called arguments

==OPERATORS ==

::Arithmetic, Assignment, Comparison, Logical, Bitwise

==Logical operators==

//Falsy(falsy) values in JavaScript

//undefined

//null

//0

//false

//''

//NaN

// !!!! Anything that is not Falsy -> Truthy

-----Function should have only one responsibility----------

- Break a function into multiple functions (calculateGrade split in calculateGrade and Calculate averaged of values)

//**this** keyword is a reference to the object that execute that piece of code

There are two ways to create a new object: **Factory Function and Constructor Function.**

```
//------Factory Functions----------------
//camel notation oneTwoThreeFour
//Pascal Notation: OneTwoThreeFour
function createCircle(radius){
    // const circle = {
    return {
        // radius: radius,
        radius,
        draw(){
            console.log('draw');
        }
        //  draw: function(){
        //      console.log('draw');
        //  }
        };
    //   return circle;
}
const circle1 = createCircle(1);
console.log(circle1);
circle1.draw();

const circle2 = createCircle(2);
console.log(circle2);
circle2.draw();
```

```
//----------Constructor Function-Pascal Notaion---------------
function Circle(radius){
    //this keyword is a reference to the object that execute that piece of code
    this.radius = radius;
    this.draw = function (){
        console.log('draw');
    }
}

const circle = new Circle(1);
```

Objects are dynamic:

```
//Objects are dynamic
const circleD = {
    radius:1
};

//we can add new properties
circleD.color = "yellow";
circleD.draw = function(){}

//we can DELETE properties
delete circleD.color;
delete circleD.draw;

console.log(circleD);
```

Every Object created have a constructor: let x={}(new Object()). There are constructors for string -> new String()//can be used just ‘’ “” ``, new Boolean(), new Number()…

!!!Functions are objects in JS

Reference Types

Object

Function

Array are Objects in JS

Objects and Primitives have different behaviour

Primitives are copied by their value

Objects are copied by their reference

… Spreading an array mean taking every individual element

Functions

Function declaration and Function Expression

```
Hoisting is the process where the JavaScript engine move the
// function declaration on top of the file
```

… Rest operator look the same as the Spread operator

In ES6 the function can be added to an object without the function keyword

//"This" keyword is referencing the object that executes current function

```
//The rest operator transform the parameters into an array
function sum(...items){
    console.log(items);
    if(items.length ===1 && Array.isArray(items[0]))
    //copying the array in array to a flat array with the spread operator (copy)
        items =[...items[0]];
    return items.reduce ((a,b) => a+b)
}
```

UDEMY -JAVASCRIPT

1. Javascript is a high-Level(We don't have to worry about complex stuff like memory management), object-oriented(based on objects, for storing most kinds of data), multi-Paradigm(We can use different styles of programming-imperative,declarative) programming language(instruct computer to do things)
2. Js- Allowsus to add dynamic effects and web applications in the browser
3. Modern JavaScript -> from 2015(ECMA script 6) and every year is a new release with new features (ES6->ES7->ES8…)
4. Data Types: Number, String, Boolean, Undefined(value taken by a variable that is not yet defined('empty value'), Null(also mean 'empty value', Symol(ES2015):value that is unique and cannot be changed, BigInt(ES2020):larger integers than the Number type can hold
5. JavaScript has dynamic typing: We do not have to manually define the data type of the value stored in a variable. Instead, data types are determined automatically.
6. In JavaScript the value have the type , NOT the variable. That means that the value can change from number to string. It also can generate problems
7. Should use const for variables first and change after to let if needed
8. Operators in JavaScript: Arithmetic(+,-,*,/,**), Assignment operators ("=,+=,-=,++,- -"), Comparison operators("<, > , >=,<=")
9. Type conversion(we do it explicitly) and coercion(when the java convert the data to another type for us): Conversion => console.log(Number("234"),243,"243");, String(23);; Coercion => console.log("I am " + 23 + "years old") transforming numbers to string
10. 5 Falsy values : 0, "", undefined, null, NaN(they are converted to Boolean false). Anything else is truthy(1, "sadf", etc)(converted to true)
11. Are used by the JS when using type coercion(implicit transforming the types-IF..else)
12. Strict equality operator("===") and it doesn't do the type coercion
13. Loose equality("==") does type coercion: if ("18" == 18) console.log("using loose equality and JS using type coercion")
14. JavaScript Releases: -
    - Released in 1995(Brendan Eich-Mocha in 10 days).
    - 1996 renamed in LiveScript(and the name JavaScript was there to marketing reasons to attract the java developers).
    - In 1996 Microsoft launched IE, copying JavaScript from Netscape and caling it Jscript.
    - In 1997 we need to standardize the language, it was implemented by ECMA and it released ECMAScript 1(ES1), the first official standard for JavaScript(ECMAScript is the standard, JavaScript the language in practice)

- In 2009 was released ES5(ECMAScript 5) with lots of great new features;
- In 2015 was released the ES6/ES2015(ECMAScript 2015) and it was the **biggest update to the language ever!**
- After that ECMAScript changed to an annual release cycle in order to ship less features per update.
- Release of ES2016, ES2017, ES2018, Es2019, Es2020…..ES2102
- It has backwards compatibility:Don't break the web(from 2020 works on 1997)

15. Don't break the web! :
    a. Old features are never removed
    b. Not really new versions, just incremental updates(releases)
    c. Websites keep working forever
    d. There are some old bugs but it made to work around in modern javascript
    e. It is used to keep the websites build forever woking
    f. What you can build with it? Fontend, Backend, Mobile.
16. During the development : Simply use the latest Google Chrome or Firefox.
17. During production: Use Babel to transpile and polyfill your code(converting back to ES5 to ensure browser compatibility for all users)
18. ES5 -Fully supported in all browsers (down to IE 9 from 2011). Ready to be used today
19. ES6/ES2015-ES2020: ES6+ => Well supported in all modern browsers; No Support in older browsers; Can use most features in production with transpiling and plyfilling
20. ES20201- ** : ESNext: Future versions of the language (new feature proposal that reach Stage4). Can already use some features in production with transpiling and polyfilling.
21. Functions in JS are values: Function declaration (func name(name){return name}) and Function expression(name = func(name){return name;}). **Calling the functions with the same sintax: name(name).**
22. Arrow function used primarly for one line functions: const calcAge3 = birthYear => 2037 - birthYear;; the arrow function does not get "this" keyword
23. Functions are values in javascript
24. Keep the code dry. Don't repeat yourself principle.
25. 4 Stepps to solve any problem: 1. Make sure you 100% understand the problem. Ask the right questions to get a clear picture of the problem.
    2. Divide and conquer: Break a big problem into smaller sub-problems.
    3.Don't be afraid to do a much research as you have to.
    4. For bigger problems, write pseudo-code before writing the actual code.

26. DOM manipulation: Document Object Model: Structured representation of HTML documents. Allows javascript to access HTML elements and styles to manipulate them
27. The HTML page is generated by the browser as a DOM tree structure(child, parent, sibling of ELEMENTS)
28. DOM Methods and Properties for **DOM Manipulation are not a part of JavaScript(ECMA)**.
29. DOM Methods and properties is a library available through the API implemented by the Web Browesers and they are already ready to use. And the JavaScript can interact with the methods. Also there are many more libraries available through the APIs like: Timers, Fetch …
30. Doom elements are saved in string… We need to convert them
31. DOM:The style property is added as an inline style. The property in css need to be with camel naming
32. DRY(don't repeat yourself) principle

Java-under the hood

33. Javascript is a **high-level**, **prototype-based object-oriented**, **multi-paradigm**, **interpreted or just-in-time compiled(0s and 1s)**, **dynamic**, **single-threaded**, **garbage-collected** programming language with **first-class functions** and a **non-blocking event loop** concurrency model.
34. High-level the programmer don’t worry about creatin variables in memory(abstractization) vs low-level (C project) which are potentially faster;
35. Paradigm is an approach and mindset of structuring code, which will direct your coding style and technique: Procedural programming, Object-oriented programming(OOP), Functional programming(FP). Javascript have all that paradigms. Javascript is very flexible
36. In a language with first-class functions, functions are simply treated as variables. We can pass them into other functions, and return them from functions.
37. Dynamically-typed language: No data type definitions. Types becomes known at runtime. Data type of variable is automatically changed.
38. Concurrency model: hot the JavaScript engine handles multiple taks happening at the same time. Why do we need that? Javascript runs in one single thread, so it can only do one thing at a time. So what about a long-running task? Sounds like it would block the single thread. However, we want non-blocking behavior! How do we achieve that? By using an event loop: takes long running tasks, executes them in the “background”, and puts them back in the main thread once they are finished.

39. What is a javascipt engine? It is a program that executes javascript code. Example : V8 Engine empower google and nodeJS. Each browser have its own engine.
40. Js engine is composed by CALL STACK(Where our code is executed using the execution context) and HEAP(Object in memory, where the objects are stored)
41. Compilation:Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer. (java, desktop application etc.)
42. Interpretation: Interpreter runs through the source code and executes it line by line.(old javascript)
43. **Just-in-time(JIT) compilation:** Entire code is converted into machine code at once, then executed immediatle. (**modern javascript**).
44. Modern just-in-time compilation of javascript: Parsing(reading the code) into AST-Abstract syntax tree, Compiling the code into 101001010, and right after compilation the code is executed in the CALL STACK. First the compilation is poor and during the execution of the program the compilation is optimized in special threads that we can't access from code.
45. Javascript runtime composition: JS engine(with HEAP and CALL STACK), Web APIs(DOM ,timers, Fetch API etc) functionalities provided to the engine, accessible on global window object. CALLBACK QUEUE -> callback functions from DOM event listener(event functions). The callbacks function are put on the call stack when it is empty to be executed. The event loop is essential for non-blocking concurrency model.
46. What is an execution context? Environment in which a piece of JavaScript is executed. Stores all the necessary information for some code to be executed. (pizza box is the execution context, the pizza is the javascript code , and the fork and bill are the helpers so that javascript code to be executed).
    After compilation the code is executed top-down. After compilation -> Creation of gloval execution context(for top-level code)-not inside a function -> Execution of top-level code(inside global EC) -> Execution of functions and waiting for callbacks(click event callback).
47. Exactly one global execution context(EC): Default context, created for code that is not inside any function (top-level). One execution context per function: For each function call, a new execution context is created;(this function are provided by the event loop).
48. What is inside execution context? 1. Variable Environment: let, const and var declarations, Functions, arguments object. 2. Scope chain. 3. This keyword. These are generated during "creation phase", right before execution.
49. Arrow function don't have arguments object and this keyword.
50. The javascript code is executed in call stack. The principle is LIFO. The functions , variables are pushed on the call stack on top on each other and after they are finished

they are pop out (removed) from call stack and after, the function below continue the execution.

51. The functions are pushed on the CALL STACK only when the functions are calle, if not the functions are not pushed
52. Scope Concepts: 1. Scoping : How our program's variables are organized and accessed. "Weher do variables live?" or "Where can we access a certain variable, and where not?". 2. Lexical scoping: Scopin is controlled by plancement of funcions and blocks ain the code. 3. Scpace or environment in which a certain variable is declared (variable environment in case of efunction). There is global scope, function scope, and block scope; 4. Scope of a variable: Region of our code where a certain variable can be accessed.
53. The 3 types of scope in JavaScript: 1. Global Scope->-Outside of any function or block – Variables declared in global scope are accessible everywhere, 2. Function Scope -> - Variables are accessible only inside function, NOT outside, - Also called local scope , 3.Block Scope(ES6)-> -Variables are accessible only inside block(block scoped), - HOWEVER, this only applies to let and const variables!(the var variables are not block scoped), -Functions are also block scoped(only in strict mode)
54. The scope of a function declared into a block is available only in that block if the "use strict" mode is used. If not the functions are available outside the block scope.
55. The scope chain: The inner scope can access outer declared variables from all outer scopes. It is called variable lookup in scope chain. The outer scope variables can't use inner declared variables.
56. The VAR variables are function scoped: even if is declared into a block the var variables behave like it is declared as a function variable(unlike the const and let variables)
57. The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all. (Scope chain vs. Call Stack)
58. The scope of a function declared into a block is available only in that block if the "use strict" mode is used. If not the functions are available outside the block scope.
59. If the global or outer variables are redeclared in the inner scoped functions there is not a problem because the javascript look for the first declaration of the variable and only if it doesn't find it it will looks outside of its scope.
60. Hoisting makes some types of variables accessible/usable in the code vefore they are actually declared. "Variables lifted to the top of their scope". Behind the scene = > Before execution, code is scanned for variagble declarations, and for each variable, a new property is created in the variable environment object. function declarations(hoisted, initial value:actual function, scope: block(in strict mode, oderwise function); var variables(hoisted, initial value: undefined(a lot of bugs), scope: function);let and const variables(not hoisted(technically yes, But not in practice),

uninitialized,TDZ(Temporal Dead Zone->error), scope:block. For function expressions and arrows : depends if they are declared var or let/const.

61. The temporal dead zone for let and const is the zone from the scope where the variable is declared to the initializing the variable. The error will be ReferenceError: Cannot access “job” before initialization instead of ReferenceError: x is not defined\
62. Why TDZ? Makes it easier to avoid and catch errors: accessing variables before declarations is bad practice and should be avoided; Makes const variables actually work.
63. Why hoisting? Using functions before actual declaration; var hoisting is just a byproduct(it is old and can be replaced now by let/const)
64. How the “this” keyword works? “this” keyword/variable: Special variable that is created for every execution context (every function). Takes the value of (point to) the “owner” of the function in which the this keyword is used. “this” is NOT static. It depends on how the function is called, and its value is only assigned when the function is actually called.
65. Types of calling functions. Method - this = <Object that is calling the method>; Simple function call – this = undefined(In strict mode! Otherwise it calls the global object window(in the browser); Arrow function – this = <this of surrounding function (lexical this)> The arrow function don’t have “this” keyword and when calling this for an arrow function the outer object is called; Event listener – this = <DOM element that the handler is attached to>; more ways to call functions with this(new, call, apply, bind).
    - “this” keyword does NOT point to the function itself, and also NOT to its variable environment
66. Don’t use arrow function for method because it doesn’t have this keyword.
67. Primitives(Number, String, Boolean, Undefined, Null, Symbol, BigInt) are stored in execution context in CALL STACK and objects(object literal, arrays, functions etc) are stored in HEAP.
68. When creating a primitive in call stack and changing the value it creates a new object to a new address because it is immutable . If a new primitive is creating to point to a value, and then changing the previous variable to a new value, a new address is created.
69. When creating an object it creates an address in call stack with reference to memory address(the value of the address) in the HEAP which have the value of the object. Object to large to stack in the call stack. When creating an object pointing to the same object it can manipulate the object.
70. Only primitive values are immutable, not CONST.
71. SPREAD OPERATOR(PACK the VALUE) and it is after the “=” => REST OPERATOR (UNPACK the VALUE) and it is before the “=”
72. Sets is a modern data structure added in ES6. It contains only unique elements, doesn’t have indexes. It has the methods: add, delete, has,clear

73. Maps are pairs of key: values. The difference between the maps and object is that the keys can be anything like: objects, maps, Booleans, arrays …. Instead of only strings. Maps have methods like: set, get, has, size, clear, delete.
74. When we have to store a simple list we use Arrays or Sets and when we have Key/Value Pairs(ex. JSON) we use Objects or Maps.(Keys allow us to describe values)
75. Data structures build in javascript: arrays, set, object, maps, weakMap, WeakSet. NOT BUILT IN: Stacks, Queues, Linked lists, Trees, Hash Tables.
76. Arrays: - Use when you need ordered list of values(might contain duplicates); - Use when you need to manipulate data as arrays have handy methods.
77. Sets: -Use when you need to work with unique values; -Use when high-performance is really important; -Use to remove duplicates from arrays
78. Objects: -More “traditional” key/value store (“abused” objects); -Easier to write and access values with . and []; -Use when you need to include functions (methods); - Use when working with JSON(can convert to map)
79. Maps: -Better Performance; -Keys can have any data type ; -Easy to iterate ; -Easy to compute size; -Use when you simply need to map key to values; - Use when you need keys that are not strings
80. Boxing. Javascript is smart and takes the primitive string and converts behind the scene to an object string so we can call the methods. After calling the method it returns a primitive string again(unboxing)
81. String methods: indexOf(); lastIndexOf(); slice()-returning a string from a given position and an optional end position, slice(1),slice(-5), slice(2,6); toUpperCase(); toLowerCase(); trim()-removing spaces from end and start; replace(“x”,”x”.toUpperCase); replacesAll(all words); startsWith(“air”), endsWith(“r”) -returns Booleans; includes(); split(“ “)-returns an array based on the split sequence ; join(“+”) join the splitted words with the (“+”); padStart(25,”+”), padEnd(35,”-“); repeat(10) repeat a message n times
82. In javascript there is just passing by value. We still can use the reference if its address is passed as value. In javaScript there is no passing by reference like in the C++ programming.
83. When we want to copy an object to another object we can use Object.assign(where,from) or spread operator(…objectToCopy). But this methods only offer a shallow copy(works only for first level, if there is an object in the object, when changing the innerObject it will change the object everywhere). To obtain a deep copy we should use some external library(LoDash).
84. JavaScript have First-class functions(they are values and is a concept). This means: =JavaScript treats functions a first-class citizens; =This means that functions are simply values; = Functions are just another “type” of objects; = Store functions in variables or

properties; = Pass functions as arguments or OTHER functions(event handlers); = Return functions FROM functions; = Call methods on functions(bind)

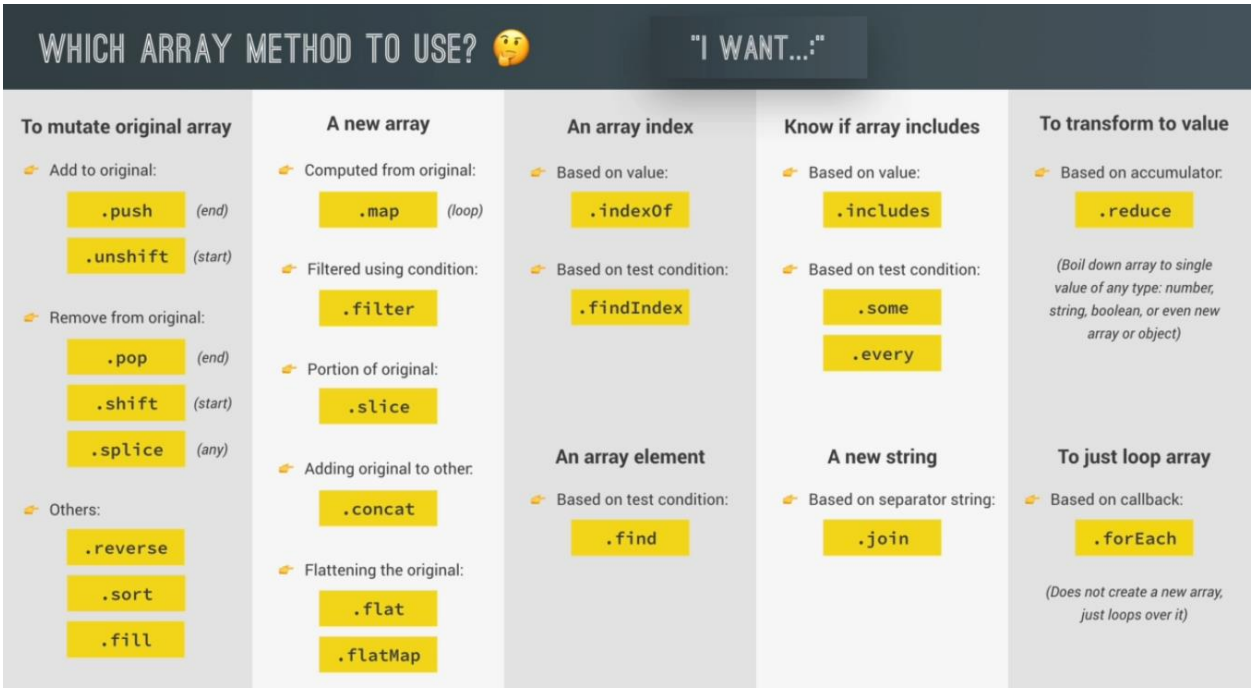
85. JavaScript have Higher Order functions: = A function that receives another function as an argument, that returns a new function, or both; = This is only possible because of first-class function. 1.Function that receives another function (eventHandler-Higher-order function, and the function return on click(Callback function); 2. Function that returns a function(function name -Higher-order function and the return of that function is a returned function)
86. Higher-order function it take as a parameter a function
87. It adds a layer of abstraction- the high-order is at a higher level of abstraction, and it need a lower order function in order to do some processing
88. Call , Apply methods are used for function that need to set a context for “this”. Call uses a list of arguments and the apply uses an array of arguments.
89. Bind method returns a function. Bind method can be use to set a context for “this” but also can set it to null. It can be used to preset some arguments in order to create a more specific new function based on mother function.
90. **CLOSURES**-we don't create explicitly
    a closer is a function that remembers all the elements of the function that created the inner function(closer) at birth date
    every function have access to the variable environment ov the execution context where it was created even the function that created the inner function is no longer active in call stack
    the variables are attached to the inner functions(closers)
    Variable environment is attached to the function somewhere in the engine, exactly as it was at the time and place the function was created,
    the closure doesn't loose connection with the environment where it was created in the first place
    priority to look for variable in the closure variable environment even bigger priority than scope chain
91. CLOSURES definitions:
    a. 1. A closure is the closed-over variable environment of the execution context in which a function was created, even after that execution context is gone;
    b. A closure gives a function access to all the variables of its parent function, even after that parent function has returned. The function keeps a reference to its outer scope, which preserves the scope chain throughout time.
    c. A closure makes sure that a function doesn’t loose connection to variables that existed at the function’s birth place;

d. A closure is like a backpack that a function carries around wherever it does. This backpack has all the variables that were present in the environment where the function was created.

We do NOT have to manually create closures, this a JavaScript feature that happends automatically. We can't even access closed-over variables explicitly. A closure is NOT tangible JavaScript object.

92. More arrays methods: slice(return a shallow copy of an array), splice(mutate the original array), join(join the original array with extra characters and return a new copy), reverse(mutate the original array and return the elements reversed by index), concat(return 2 arrays concatenated, doesn't mutate the arrays)
93. More arrays methods: foreach(callback function): it has 3 parameters -> foreach(currentValue, currentIndex, wholeArray).
94. Maps and set also have forEach(currentValue, currentKey, map/set)
95. Other modern methods: **map**-> returns a new array containing the results of applying an operation on all original array elements; **filter** - > returns a new array containing the array elements that passed a specified test condition; **reduce**(it have an accumulator initialized at the end of the function) -> reduce boils("reduces") all array elements down to one single value (ex. Adding all elements together) .
    -Each function have access to currentValue, currentIndex and whole array, like foreEach
    -All this functions call the callback function for each element of an array
96. -ForEach function create sideEffects(ex. Printing intermediate values for each iteration of elements) and it doesn't return anything, it just mutate the data wanted.
    -Map function doesn't produce sideEffects(it stores all the elements into a new array and we can use that later)
97. Other array functions: **find** method-> return first element that satisfy the condition; **findIndex**->return the index where the condition is met; **some**->(is like include but the condition can be a complex algorithm to met) , **every**-> it returns true if all the elements meet the function condition; **flat** -> return an array without nested arrays, the deep level can be set; **flatMap**-> return an array without nested array(better performance), but only on 1 level deep
98. Sorting an array: sort=> the default sort method sort the elements based on strings; it can be changed using a callback function: sort((a,b)=>a-b) || sort((a,b)=>b-a). // //return < 0, A,B(keep order);ascending order: movements.sort((a, b) => {if (a > b) return 1; if (a < b) return -1});
99. More ways of creating and filling arrays: new Array(length), array.fill(23,4,6); Array.from({length:23},(_,i)=>i+1); Array.from to transform from maps, sets to an array, and even node lists movementsUI = Array.from(document.querySelectorAll (".movements__value"), (el) => Number(el.textContent.replace("€", "")));

100. All the array methods:

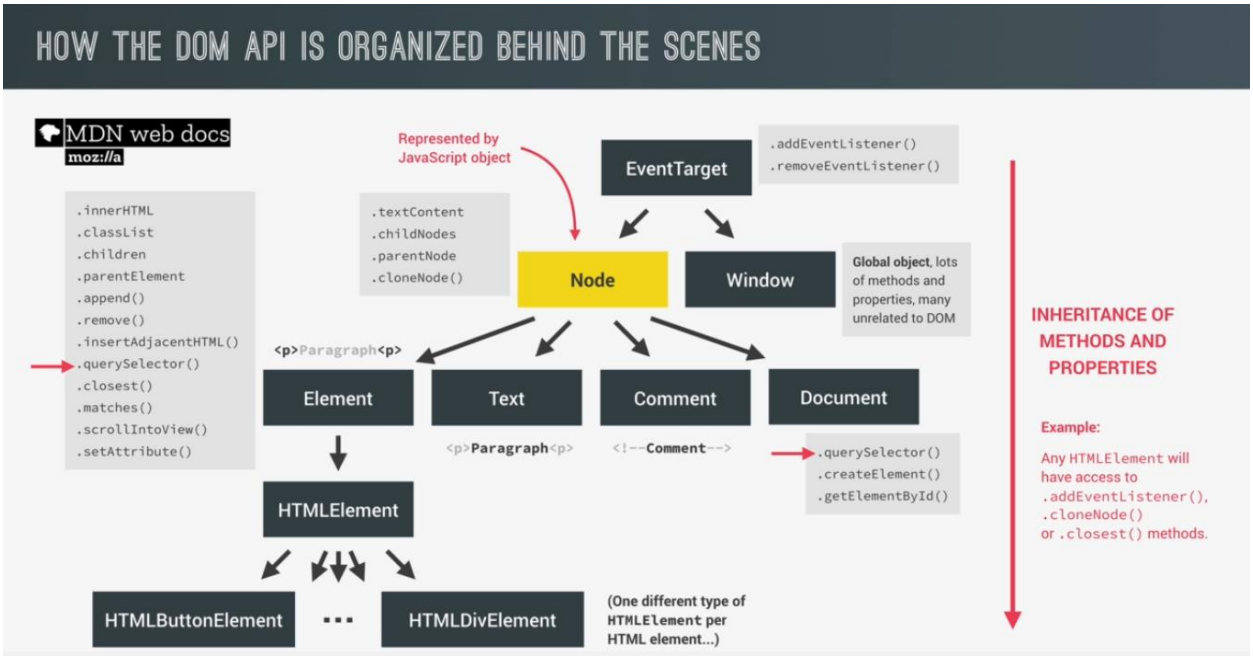


101. Numbers are always represented as floating point numbers. Numbers are stores in base 64 binary format. Hard to display fraction in base 2 and it can't show the truncated number (0.1+0.2=0.30000000000000004)—can't do precise operations
102. Numbers and Math methods: console.log(+("23")), Number("23"), Number.parseInt, Number.parseFloat, Number.isFinite(is a number or not), Number.isInteger, Math.sqrt(), Math.max(),Math.min(), Math.PI, Math.random, Math.random(min-max =>const randomInt = (min, max) => Math.floor(Math.random() * (max - min) + 1) + min;); Math.trunc; Math.round(); Math.ceil(); Math.floor()(better to round); console.log((2.345).toFixed(2)); //return a string with 2 decimals console.log(+(2.345).toFixed(2)); //return a number with 2 decimals
103. In 2020 it was introduces bigInt.They are good for very big numbers. They are represented with ending n. Can do operation between bigInt numbers only, math methods doesn't work,
104. Date and time. JavaScript parse the data even if we throw different formats and wrong dates. console.log(new Date(2037, 10, 19, 14, 23, 5)); Different methods: getFullYear, getMonth, getDate, getDAte, getHours, toISOString(), getTime, passing timestamps as arguments ->console.log(new Date(2142246180000)); Date.now(), date.setFullYear etc..
105. Adding 0 before date if is only one digit: const day = `${now.getDate()}`.padStart(2, 0);
106. We can do operations with dates using timestamps.
107. For displaying the dates in different country formats we use internationalizing dates : using Intl with the method DateTime format with the language-Country,we can add

options what to display, and than format and the date we want to format->Intl.DateTimeFomat("en-US,options).format(now)

108. SetTimeout : calling a callback function after a {number of milliseconds}, and we can pass arguments to it. We can clear the timeout with clearTimeout.
109. SetInterval: calling a callback function every {number of milliseconds}, and we can stop the timer with closeInterval. To do that we need to give it a name and to declare it globally.
110. DOM : Allows us to make JavaScript interact with the browser; We can write JavaScript to create, modify and delete HTML elements, set styles, classes and attributes, and listen and respond to events; Dom tree is generated from an HTML document, which we can then interact with

    DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree
111. How the DOM api is organized behind the scenes:

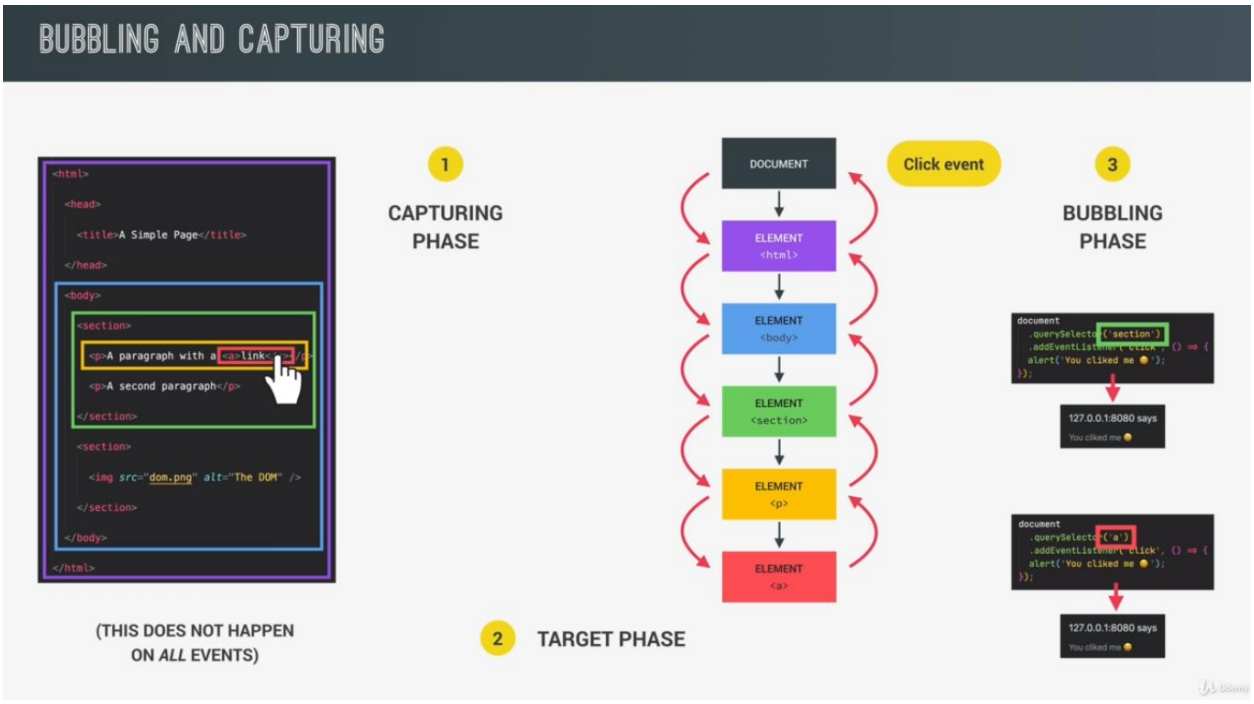


112.

-Each Node of the DOM API can be the type of: Element; Text; Comment;Document.
-Node have methods like .textContent, .childNodes, .parenNode, .cloneNode()...
-Elements have methods like: .innerHTML, .classList, .children, .parentElement, .append(), .remove(), . insertAdjacentHTML(), .querySelector(), .closest(), .matches(), .scrollIntoView(), .setAttribute() etc
-Document also have methods like : .querySelector(), .createElement(), .getElementById()

-Each Element has a child HTML Element that have one different type of HTMLElement per HTML element because it has special properties... : HTMLButtonElement , HTMLDivElement, HTMLImage, HTMLLink(and each of these can have some special attribute that only that element has(ex.href))
-all the bottom element have all the methods that the parents element have
-there is a special node :EventTarget that has the .addEventListener() and .removeEventListener() that every node inherit and also the Window(Global object, lots of methods and properties, many unrelated to DOM)

113. **DOM-Propagation: Bubbling and Capturing**: When setting an event listener the signal is received at the root of the document and then there are 3 phases: 1. Capturing phase where the event travel down through all the parents roots to the event target(not the siblings). 2.Target phase: At the event target the event is executed. 3.Bubbling phase: After the event is executed the event listener travel back to the root. This phases means how the event is **propagated**. This doesn't happen for all events. This is important because if the event is attached to the parent too, the event will be triggered there too with only one click.(this happens in the target phase and in the bubbling phase only, can be set to listen to the event in the capturing phase too)



114. In the bubbling phase the target event listener triggers the events attached in the parents nodes too. It can create problems. It can be stopped e.stopPropagation();, but is not a good idea. Adding a parameter true it changing the triggering the event from bubbling to capturing phase.
115. e.preventDefault();- using when we don't want any default action to happen

116. Event delegation is used to handle events that are happening for child elements, instead of attaching events for each element, and in the case we add programmatically buttons and we need events for them. Event delegation in 2 steps: 1. Add event listener to common parent element.; 2. Determine what element originated the event.
117. Intersection observer API(for smooth scrolling) is created using an object IntersectionObserver(callback,options) and than the object call the method to observe an element as a target. Options contains the root(element we want to intersect, if is null it means we observe how the element intersect with the entire viewport), threshold(0…1=>0 means 0% of the target, and 1 means 100% of the target); threshold: at which percentage the function callback is intersecting the root element at a specific threshold-> is represented by intersectionRatio ;and the rootMargin(if we want to add some margin to the intersection -> in pixels). The callback function have 2 parameters : entries(the object IntersectionObserverEntry)->this has multiple properties(isIntersecting ,intersectionRatio etc.) , and the observer object(IntersectionObserver));
118. Regular vs. Async vs. Defer: Regular at the end of body: - Scripts are fetched and executed after the HTML is completely parsed; - Use if you need to support old browsers; Async in Head: - Scripts are fetched asynchronously and executed immediately; - Usually the DOMContentLoaded event waits for all scripts to execute, except for async scripts. So, DOMContentLoaded does not wait for an async script; - Scripts not guaranteed to execute in order; -Use for 3rd-party cripts where order doesn't matter (ex Google Analytics); DEFER in head: -Scripts are fetched asynchronously and executed after the HTML is completely parsed; DOMContentLoaded event fires after defer script is executed; - Scripts are executed in order; - This is overall the best solution! Use when order matters (including a library)
119. What Is **Object-oriented programming**? (OOP)
    a. Object-oriented programming(OOP) is a programming paradigm(style of code, "how" we write and organize code) based on the concept of objects;
    b. We use objects to model(describe)real-world(user or to do list item) or abstract features(HTML component or data structure).
    c. Objects may contain data(properties) and conde(methods). By using objects, we pack data and the corresponding behavior into one block.
    d. In OOP, objects are self-contained pieces/blocks of code.
    e. Objects are building blocks of applications, and interact with one another;
    f. Interactions happen through a public interface (API): methods that the code outside of the object can access and use to communicate with the object.
    g. OOP was developed with the goal of organizing code, toe make it more flexible and easier to maintain (avoid "spaghetti code").

120. Classes and instances: A Class is like a blueprint from which we can create new objects. Every time we create a new blueprint with data is called Instance. It is like a new object created from the class(Like a real house created from an abstract blueprint).
121. The 4 fundamental principles of **Object-Oriented Programming**: Abstraction, Encapsulation, Inheritance and Polymorphism.
122. **Abstraction**: Ignoring of hiding details that don't matter, allowing us to get an overview perspective of the thing we're implementing, instead of messing with details that don't really matter to our implementation.(like a phone, we don't need all the low level mechanism, turnOnVibration() when the there is a call etc.)
123. **Encapsulation**(there is **no private** keyword in javaScript): Keeping properties and methods private inside the class, so they are not accessible from outside the class. Some methods can be exposed as a public interface(API). Why? -Prevents external code from accidentally/non-accidentally manipulating internal properties/state; - Allows to change internal implementation without the risk of breaking external code.
124. **Inheritance**: Making all properties and methods of a certain class available to a child class, forming a hierarchical relationship between classes. This allows us to reuse common logic and to mode real-world relationships. The child class extend(inherit) the parent class properties and methods but it also can have its OWN methods and properties.
125. **Polymorphism**: Is the capacity of an object to take multiple forms. A child class can overwrite a method it inherited from a parent class, or overload a method.(overwriting login method in Admin inherited from User in). Up Casting, Down Casting, overloading, overwriting.
126. **OOP in JavaScript: PROTOTYPES:** In javaScript there are no classes and instances. There are Prototype that contain methods and properties and Objects that can access methods and properties. – Objects are linked to a prototype object; -> Each object has a prototype.
127. **Prototypal inheritance**: The prototype contains methods(behavior) and properties that are accessible to all objects liked to that prototype. Objects inherit methods and properties from the prototype->Prototypal inheritance(different than classical inheritance, here is an instance inherit from a class)
128. Behavior of the created object is **delegated** to the linked prototype object. (prototypal inheritance/ Delegation). In classic OOP when a new object is created the methods and properties are copied to the instances.
129. Ex. Array: Array.prototype is the prototype of all array objects we create in JavaScript. Therefore, all arrays have access to the map method.
130. 3 Ways of implementing prototypal inheritance in JavaScript:

a. **Constructor functions**: -Technique to create objects from a function. -This how built-in objects like Arrays, Maps or Sets are actually implemented.
b. **ES 6 Classes:** -Modern alternative to constructor function syntax; - “Syntactic sugar”: Behind the scenes, ES6 clases work exactly like constructor functions; - ES6 classes do NOT behave like classes in “classical OOP”
c. **Object.create():** - The easiest and most straightforward way of linking an object to a prototype object.

131. How the javaScript create an new object behind the scenes when the word “new” is used: 1. New object{} is created; 2. Function is called and the "this" keyword points the the previous created object this ={}; 3. {} The new object is linked (__proto__property) to the constructor function’s prototype property; 4. function automatically return the empty object in the beginning{}(not necessarily empty)
132. In JS is better to use prototypes and prototypal inheritance instead of methods, because of the fact that the methods(functions) declared in constructor function are copied for each object in JS , there would be a performance issue. So all the objects created using the "Person" will have access to the methods defined in the Person.prototype and we create only one copy of the function and each of the created object can call the method.
133. The prototype in Person.prototype is not the prototype of the Person, it is the prototype of all the object that are created using that Person: console.log(grig.__proto__); console.log(grig.__proto__ === Person.prototype); Prototype is bad naming(should be prototypeOfLinkedObjects). Each object created will have the __proto__ property attached that always points to an object’s prototype
134. We can set properties also. The properties are not stored in the object created, but in the prototype.
135. When we call a method for an object JS will look into the object, but if it is not found it will look into the prototype(delegation)
136. Prototype chain: There is a series of links between objects, linked through prototypes. The highest level is the Object(It is built-in constructor function for objects. This is used when we write an object literal:{…}=== new Object(). This Object has a prototype , that has implemented its own methods, that any object lower level can have access to, because each prototype is an object and the low level prototype is linked to the Obect.prototype. console.log(grig.__proto__.__proto__); The prototype is also an object
137. ES6 Classes: They are just modern syntax. They work the same as the construction functions. The Classes have a constructor where the properties are initialized , and also we can add methods directly to the class, but they are still added automatically to the prototype property behind the scene. (if we want we can declare new method using the Prototype.prototype.method)

138. ES6 Classes Important notes: 1. Classes are NOT hoisted like function declarations.; 2. Classes are first-class citizes(we can pass them to another functions and can return from the functions)-just a special kind of functions; 3. Classes are executed in strict mode always.
139. We can use setters and getters when we need data validation. The setters and getters are treated like a property when setting new data or getting the data.
140. Object.create():We can set the prototype of an object to any object do we want; When we create a new object with the Object.create the prototype of the new object will be the same with the prototype of the object we passed in
141. ES6 Classes Summary:

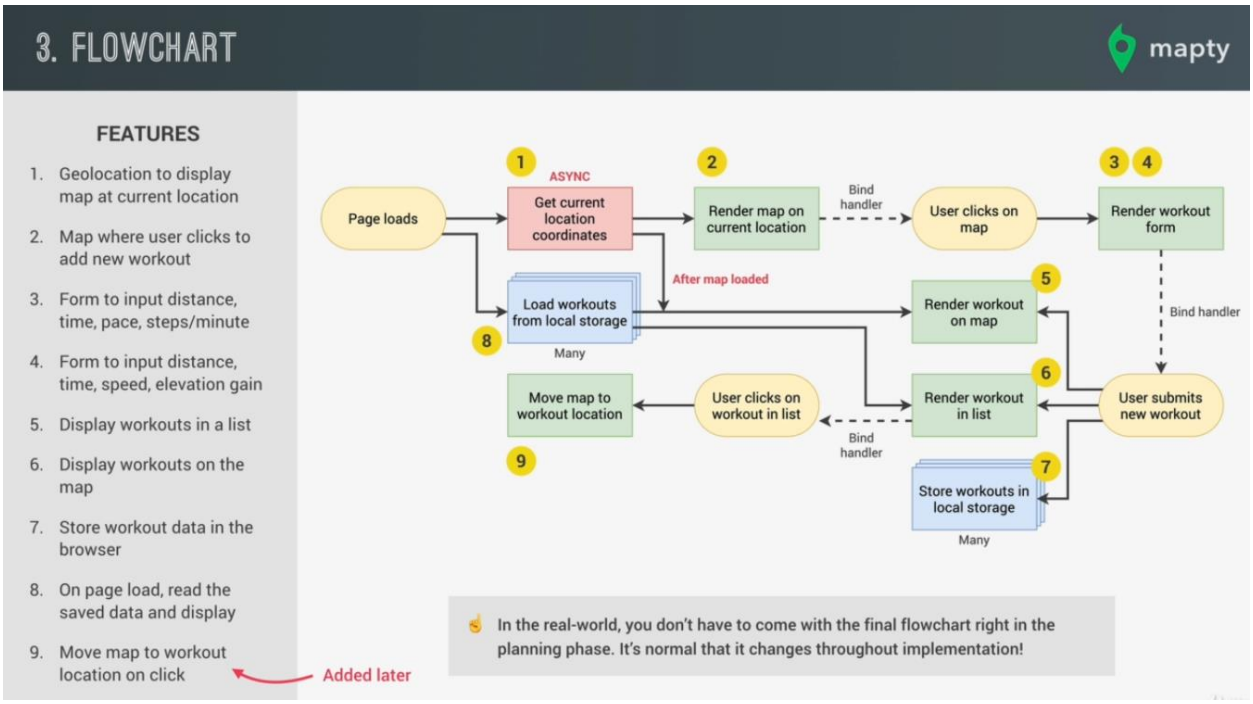
142. Planning a project steps: 1) User stories -> Description of the applications' functionality from the user's perspective. All user stories put together describe the entire application; 2) Features of the application; 3) Flowchart ->WHAT we will build ; 4)Architecture->HOW we will build it; 4) Development step-> Implementation of our plan using code
143. 1)User Stories: -User story: Description of the application's functionality from the user's perspective. – Common format: As a [type of user](Who? User, admin etc), I want [an action](what?) so that [a benefit](why?)
    a. As a user, I want to log my running workouts with location, distance, time, pace and steps/minute, so I can keep a log of all my running
    b. As a user, I want to log my cycling workouts with location, distance, time, speed and elevation gain, so I can keep a log of all my cycling.

c. As a user, I want to see all my workouts at a glance, so I can easily track my progress over time.
d. As a user, I want to also see my workouts on a map, so I can easily check where I work out the most
e. As a user, I want to see all my workouts when I leave the app and come back later, so that I can keep using there app over time

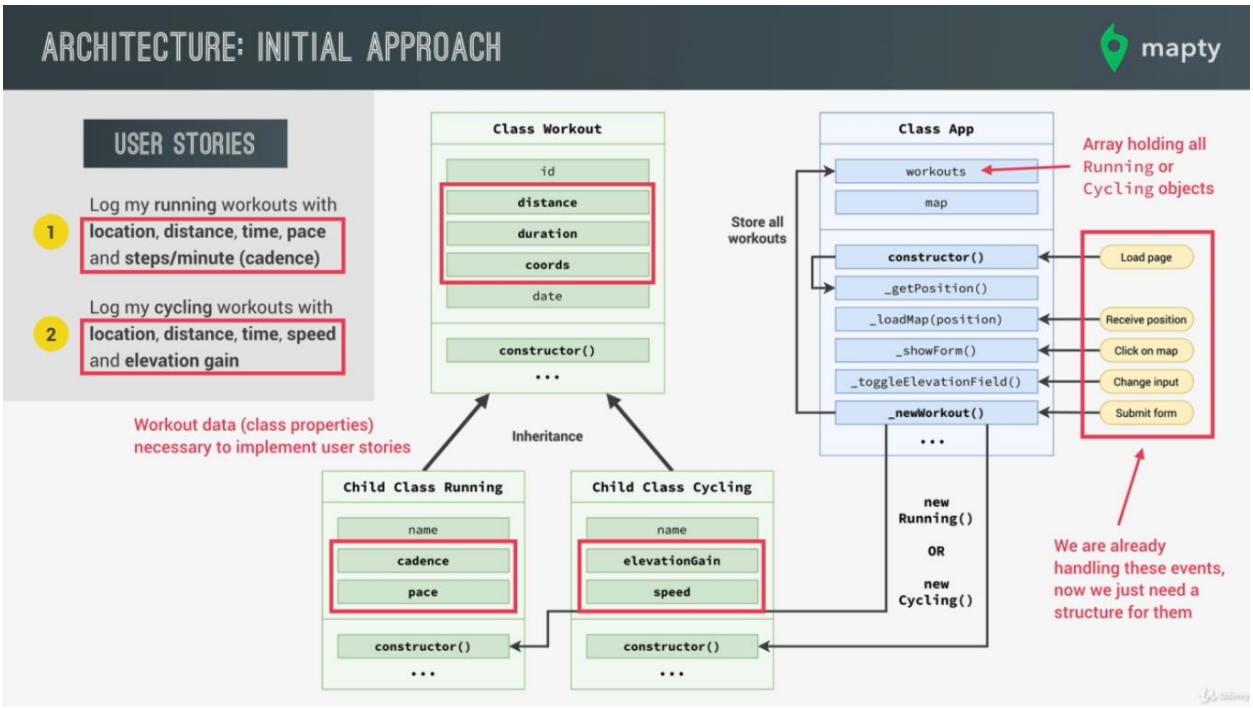
144. 2) FEATURES

a. Map where user clicks to add new workout (best way to get location coordinates).
   Geolocation to display map at current location (more user friendly)
   Form to input distance, time, pace ,steps /minute
b. Form to input distance, time speed, elevation gain
c. Display all workouts in a list
d. Display all workouts on the map
e. Store workout data in the browser using local storage API
   On page load, read the saved data from local storage and display

145. 3) FLOWCHART



146. 4) ARCHITECTURE-> don’t need to have the architecture right in the beginning, but we can make a sketch and figure out things after we start the implementation.

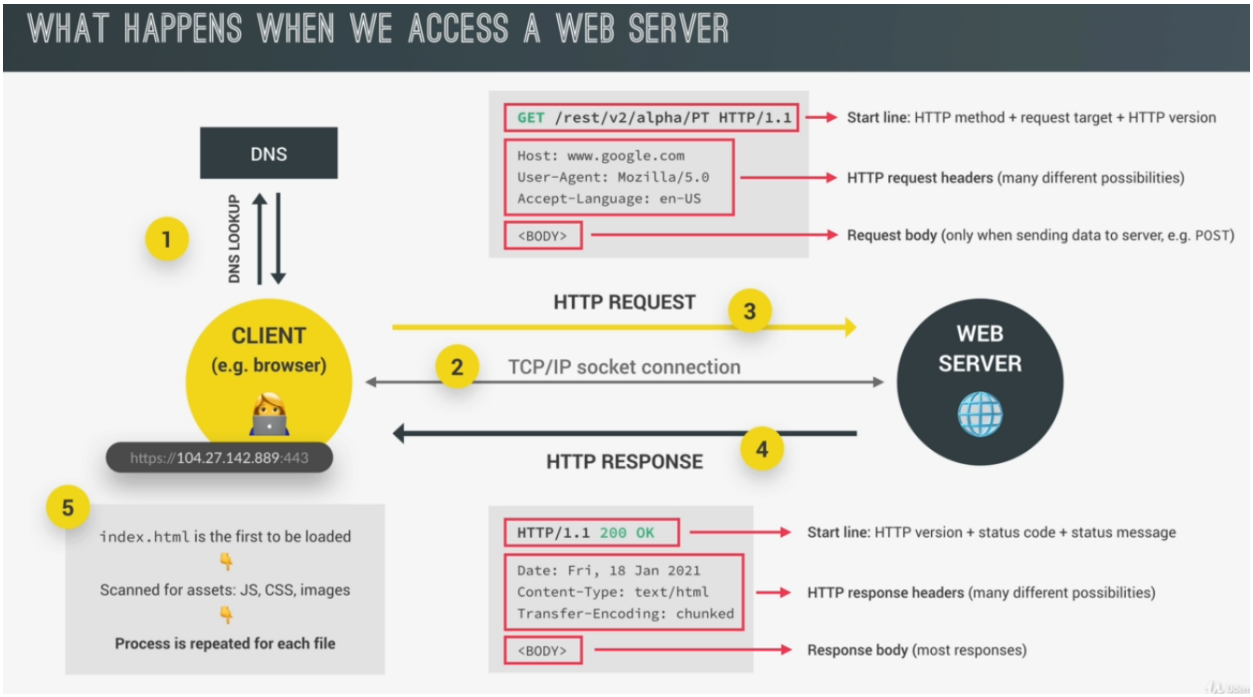


147. Synchronous: -Most code is synchronous ; -Synchronous code is executed line by line; -Each line of code waits for previous line to finish; Long-running operations block code execution(ex. Alert prompt)
148. Asynchronous(Coordinating behavior of a program over a period of time): -Asynchronous code is executed after a task that runs in the “background” finishes(setTimeout); -Asynchronous code is non-blocking; -Execution doesn’t wait for an asynchronous task to finish its work; Callback function alone don NOT make code asynchronous ; Some methods are implemented as asynchronous(img.src = “dot.src”) and we can listen when the image was loaded and we can perform an callback function when the loading event have happened(img.addEventListener(“load”, function(){…});-addEventListener does NOT automatically make code asynchronous; Other examples: Geolocation API or AJAX calls
149. JavaScript . AJAX :Asynchronous JavaScript And XML: Allows us to communicate with remote web servers in an asynchronous way. With AJAX calls, we can request data from web servers dynamically(API).
150. API: Application Programming Interface: Piece of software encapsulated that can be used by another piece of software, in order to allow applications to talk to each other.; -There are be many types of APIs in web development(DOM API, Geolocation API, Own Class API), “Online” API; “Online” API(Just “API”): Application running on a server, that receives requests for data, and sends data back as response; -We can build our own web

APIs (requires back-end development, ex. With node.js) or use 3rd-party APIs.(weather data, Data about countries, Flights data,Currency conversion data, Apis for sending email or SMS, Google Maps, More possibilities)

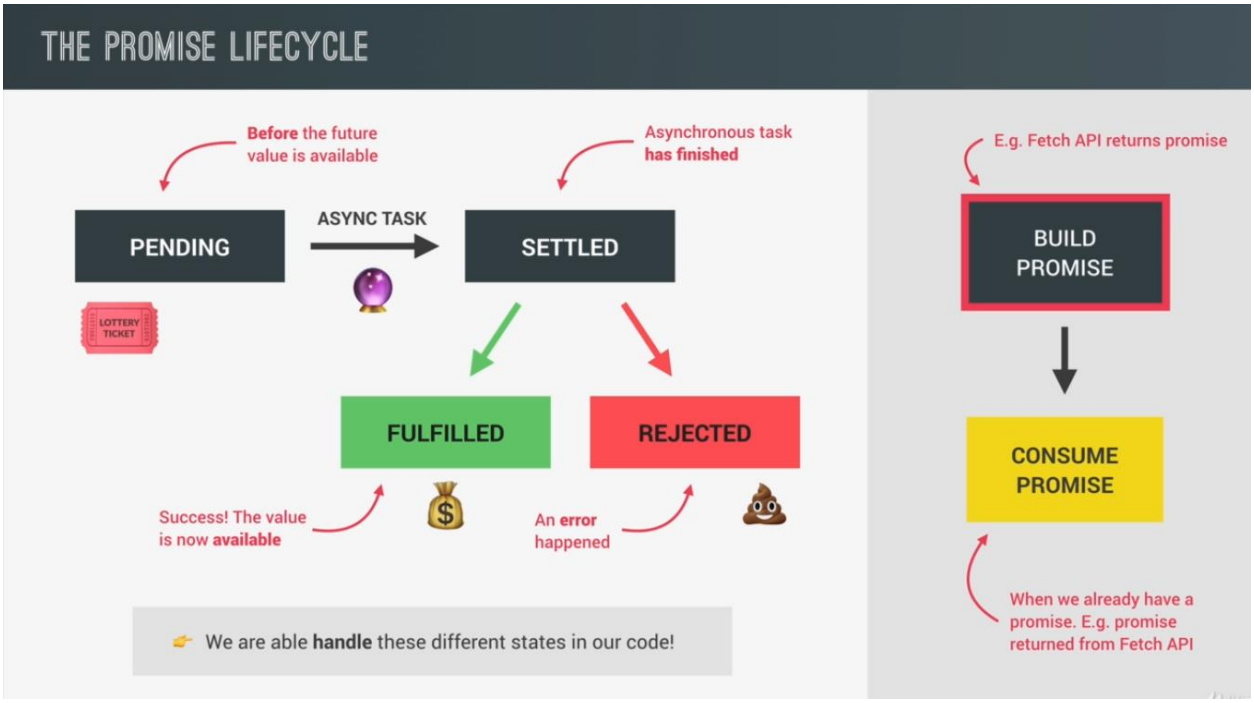
151. JSON data format is the most popular API data format and is what uses now AJAX(x stands for XML but is too old now)
152. How the web Works: Requests and Responses. What happends when we access a web server? Request-response model or Client-server architecture.
153. url website: https->Protocol; grigorenath.com -> Domain name; and after / -> Resource. **1.)**The domain name is not the real location of the site, it must look into the DNS(Domain name server(like a phonebook of the internet) and when the client(browser) want to go at that web server the DNS return the matched IP address of that web address url(the DNS converted the domain name to an IP address)-> it will look like: protocol; IP address; and Port number(443 for HTTPs and 80 for HTTP).;**2.)** To connect the Client to the Web Server it is used a TCP/IP(Transmition control protocol/internet protocol->how data travels) socket connection that is kept alive for all the time it takes to transfer all the data or the files.; **3)**The client send a HTTP REQUEST(communication protocol, how the parties communicate) (ex. GET /rest/v2/alpha/PT HTTP/1.1), there are HTTP request headers (many different possibilities) and Request body(only when sending data to server (ex.POST).;**4)**After the response is finished the Web Server sends back a HTTP Response with Start line(HTTP/1.1 200 OK): HTTP version + status code +status message, HTTP response headers(many different possibilities), and a Response body (most responses).; **5)** First page to get loaded is the index.html, after that the index is scanned for assets: JS, CSS, images -> and the process is repeated for each file(page).;**2)**TCP transform the request in multiple packets and when it reaches the destination the tcp will reassemble the packets into a whole original request and the role of the IP is to route the packets to

reach the destination(client, web server) using IP addresses on each packets

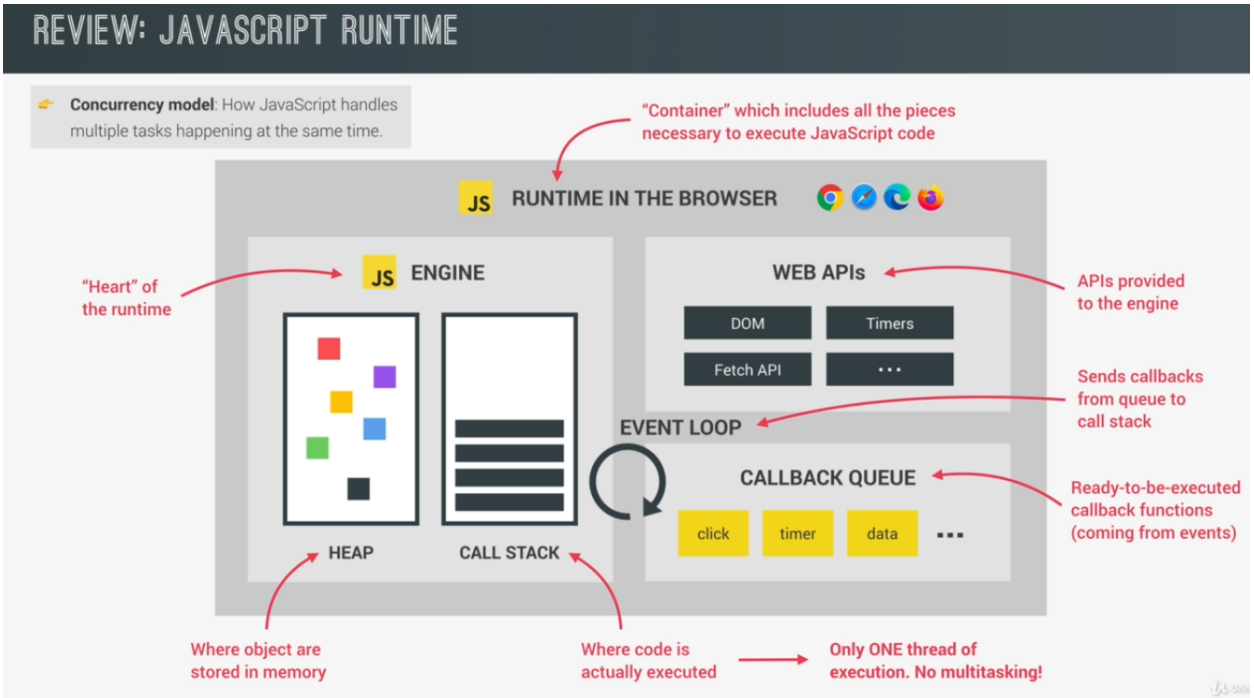


154. **Promises**. = An object that is used as a placeholder for the future result of an asynchronous operation. = A container for a synchronously delivered value. = A container for a future value(Response from AJAX call).; -We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results.; -Instead of nesting callbacks, we can chain promises for a sequence of asynchronous operations: escaping callback hell.
155. The promise Lifecycle: Pending(Before the future value is available)->Settled(Asynchronously task has finished) -> It can be Fulfilled(Success! The value is now available) or it can be rejected(An error happened): We are able to handle these different states in our code. IN order to use promises we need to consume promises(When we already have a promise.ex promise returned from Fetch API-most

used case), but if we don't have the promise we need to build the promise.

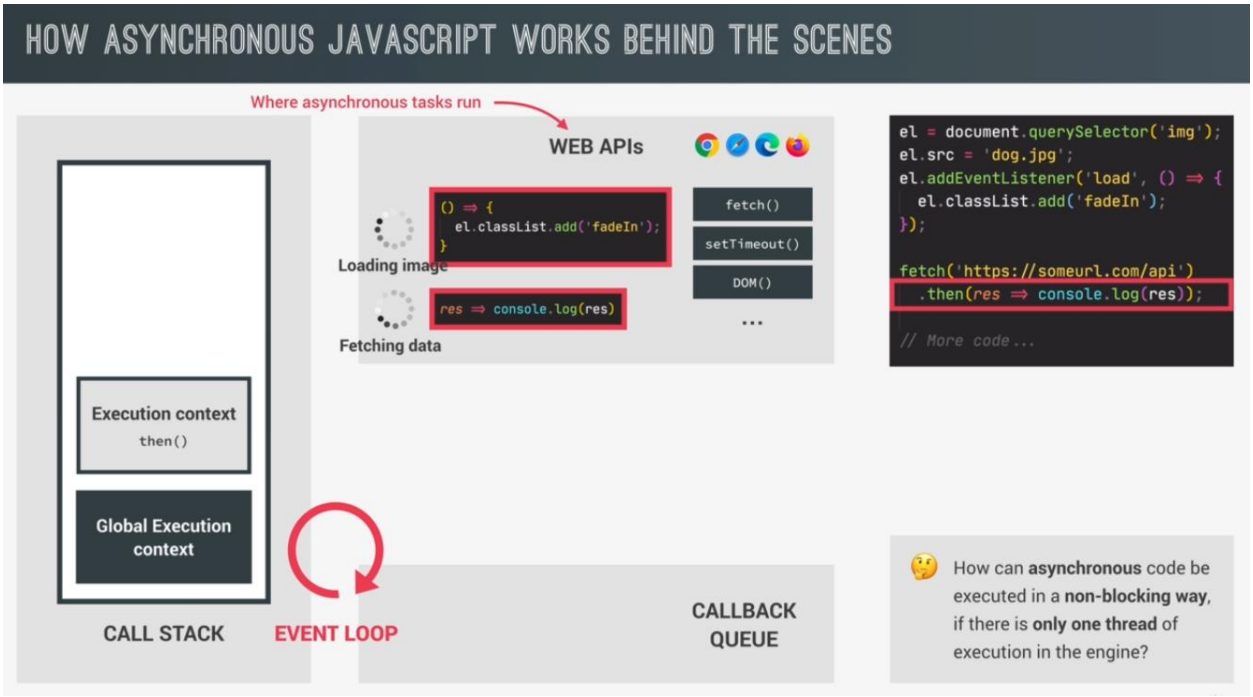


156. JavaScript Runtime

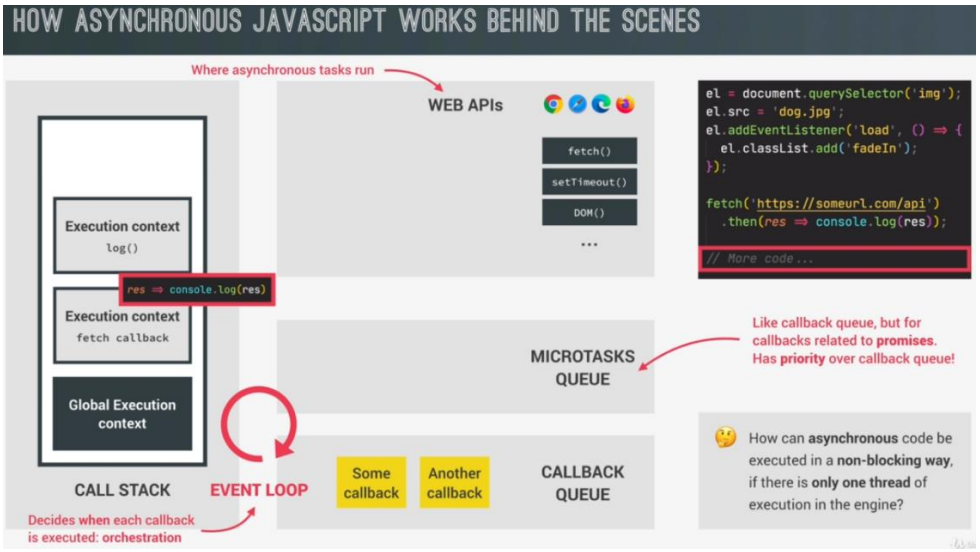


157. How Asynchronous JavaScript works behind the scenes: When an asynchronous function is called than it will be executed on the WEB API environment and it will have attached the callback function when the asynchronous function is finished. The callback function are registered in the call stack but after that they are attached to the loading

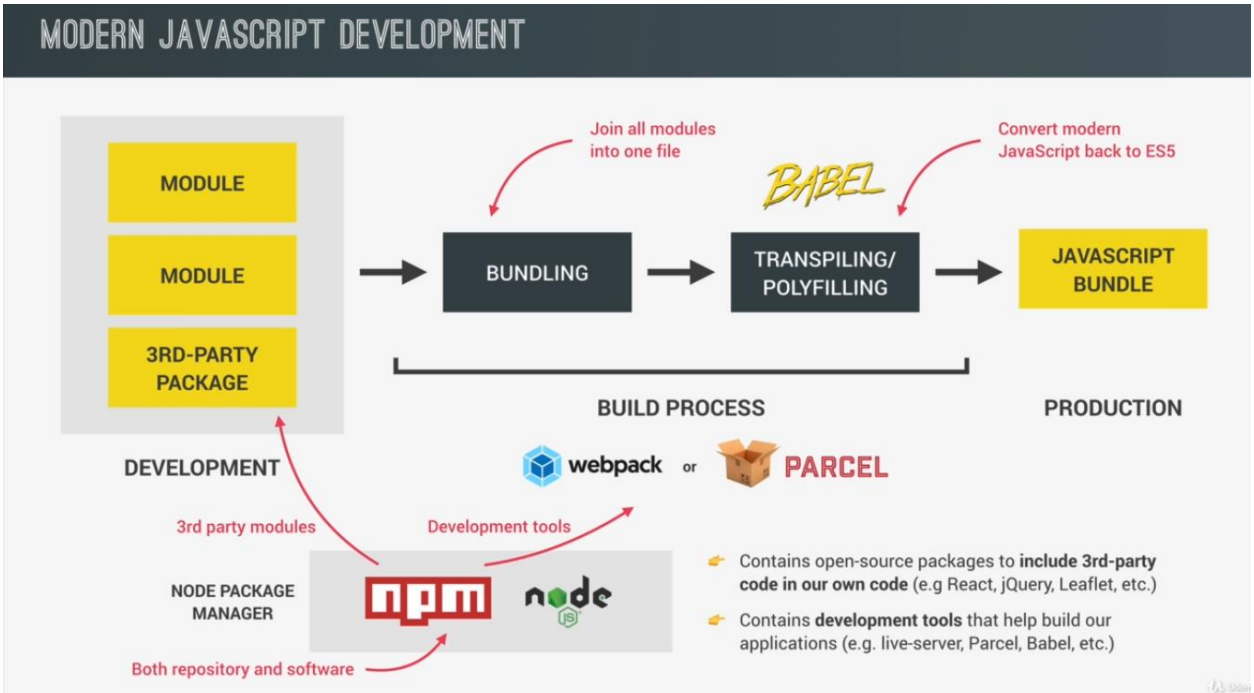
asynchronous function in the WEB API environment.



158. After the asynchronous function is finished loading it is added to the Callback Queue(all events to execute:DOM, functions etc.) in the back(if we have a callback of 1 sec to execute the setTimeout(5 sec) is executed after 6sec).
159. Event loop check if the Call Stack is empty and if it is, it takes the events from the callback queue and adds them to the call Stack = It is called a Event Loop Tick.(it orchestrate the runtime)
160. The callbacks to react to the promises from the fetch calls are added to the Microtasks Queue after they are finished loading. This microtasks have priority over Callback Queue for being added in the Call Stack by the Event loop when there is an space in the Stack



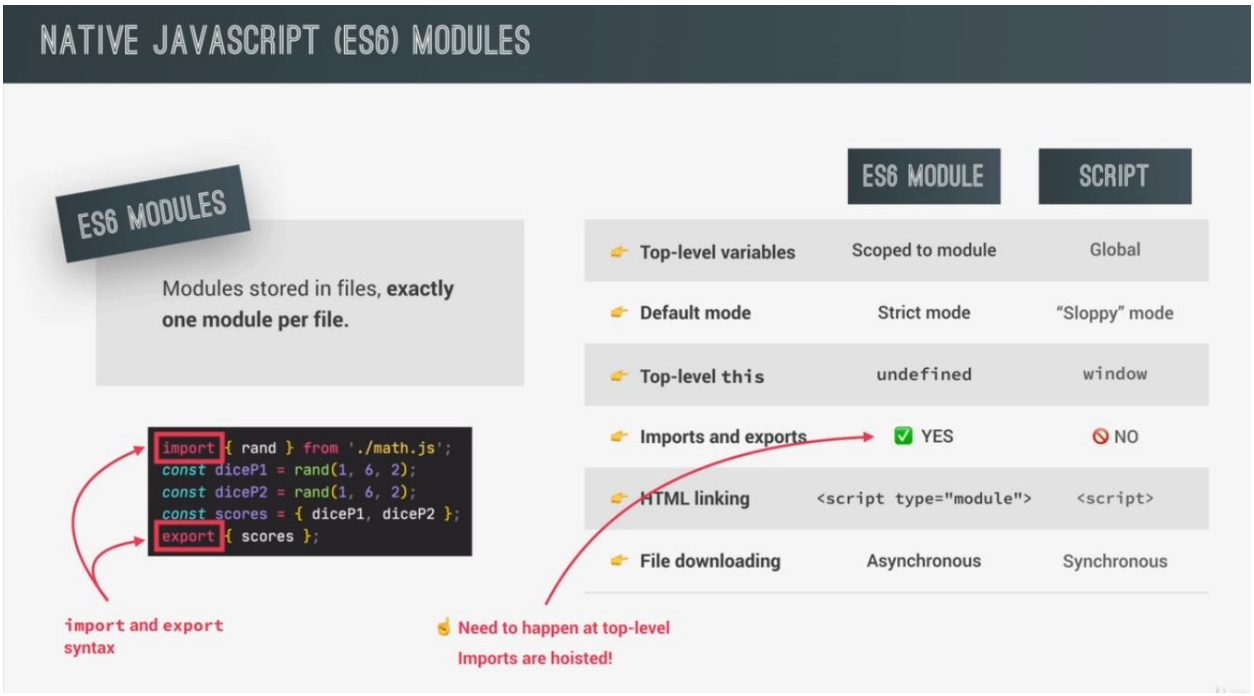
161. The first code that is executed is the top-level code (NOT inside of any function) from the global execution context
162. ASYNC function and AWAIT promise response. When marking a function with async the execution is starting in background, and when encounter the await then the execution is stopped and the return of a promise is waited. The result of await promise can be stored in a variable, and it must be converted in an object with await promise from method .json(). It is just syntax sugar over the promises and returning the promises
163. When we want to handle the errors in async functions we need a try..catch block and we have to check the returned await variable for .ok property and manually catch the error. Or we can catch all the errors in the try…catch block in the catch
164. Running all the promises functions in parallel using Promise.all. If 1 promies is rejected than it will short circuit the function Promise.all. Use for loading multiple async function in parallel(much more faster). This is a combinator function
165. Other promise combinators: Promise.race(returns the first fulfilled promise, doesn't matter the result); Promise.allSettled(ES2020)(returns all the promises fulfilled, ignore the rejects); Promise.any(ES2021)(returns the first fulfilled promise)
166. Module:-Reusable piece of code that encapsulates implementation details; -Usually a standalone file, but it doesn't have to be.
167. Modern Javascript development:



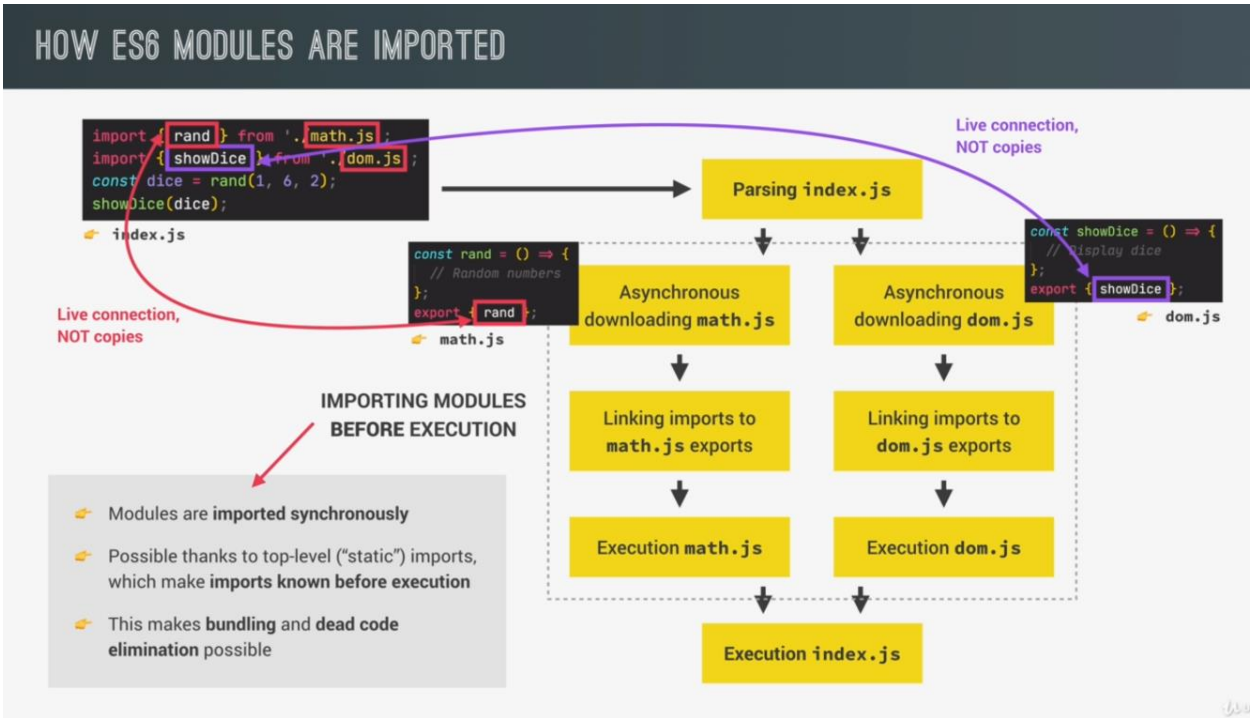
168. Why modules: -**Compose software**: Modules are small building blocks that we put together to build complex applications; -**Isolate components**: Modules can be developed in isolation without thinking about the entire codebase; -**Abstract code**; Implement low-level code in modules and import these abstractions into other

modeules; -**Organized code**: modules naturally lead to a more organized codebase; -**Reuse cod**e: Modules allow us to easily reuse the same code, even across multiple projects;

169. Moules in JavaScript(Native in ES6): Modules are stored in files, exactly one module per file.



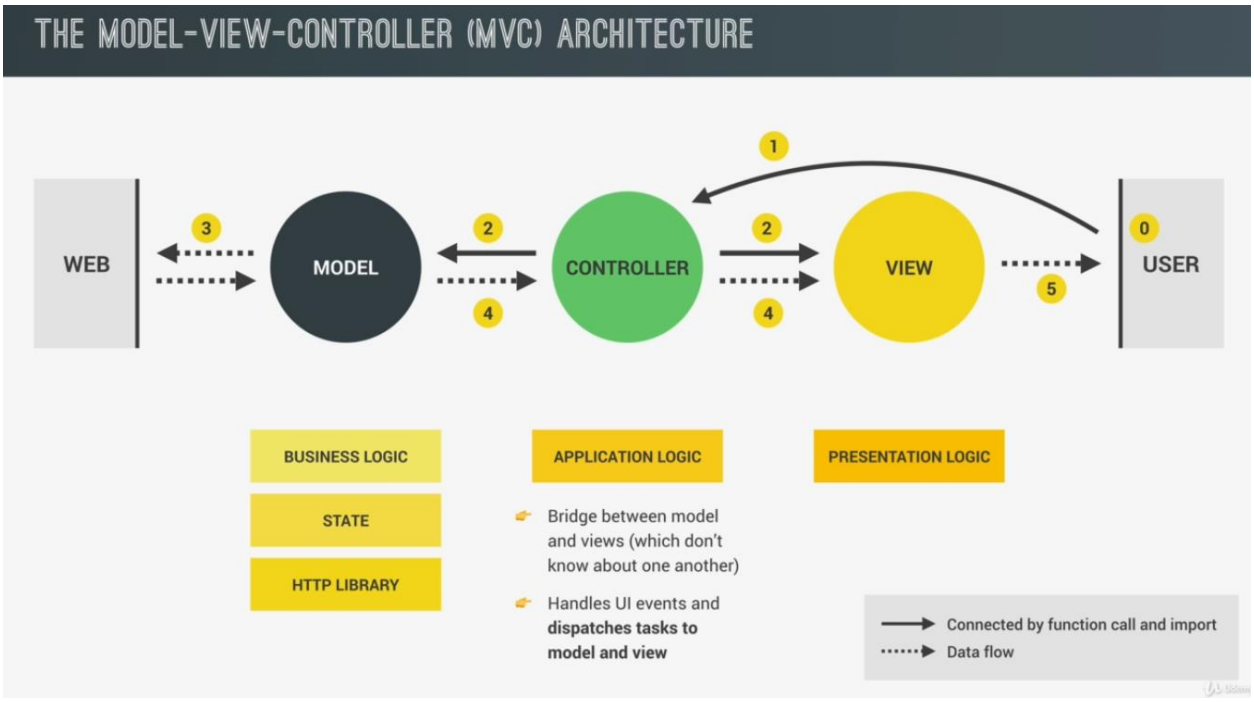
170. How ES6 modules are imported



171. **Modern and clean code**: **READABLE CODE**:-Write code so that others can understand it; -Write code so that you can understand it in 1 year; -Avoid too “clever” and overcomplicated solutions; -Use descriptive vaiabble names: what they contain; -Use descriptive function names: what they do; **GENERAL**: -Use DRY principle (refactor your code); -Don’t pollute global namespace, encapsulate instead ; -Don’t use var;-Use stron type checks(=== and !==); **FUNTIONS**: -Generally, functions should do only one thing; -Don’t use more than 3 function parameters; -Use default parameters whenever possible; -Generally, return same data type as received; -Use arrow functions when they make code more readable; **OOP**: -Use ES6 classes; -Encapsulate data and don’t muatte it from outside the class; -Implement method chaining; -Do not use arrow functions as methods(in regular objects); **AVOID NESTED CODE**:-Use early return(guard clauses); -Use ternary(conditional_) or logical operators instead of if; -Use multiple if instead of if/else-if; -Avoid for loops, use array methods instead; -Avoid callback-based asynchronous APIs; **ASYNCHORNOUS CODE**: -Consume promises with async/await for best readability; -Whenever possible, run promises in parallel(Promise.all); Handle erros and promise rejections.
172. **IMPERATIVE** VS. **DECLARATIVE CODE**: Two fundamentally different ways of writing code (paradigms): IMPERATIVE: -Programmer explains “HOW to do things”; -We explain the computer every single step it has to follow to achieve a result; Ex. Step-by-step recipe of a cake.; DECLARATIVE:-Programmer tells “WHAT to do” ; -We simply describe the way the computer should achieve the result; -The HOW (step-by -step instructions)gets abstracted away; Ex. Description of a cake(const arr=[2,3,5,6]; const doubled = arr.map(n=>n*2))
173. Subtype of Declarative : **FUNCTIONAL PROGRAMMING(modern way):-**Declarative programming paradigm;-Based on the idea of writing software by combining many pure functions, avoiding side effects and mutable data; Side effect: Modidfication(mutation) of any data outside of the function (mutating external variables, loggin to console, writing to DOM, etc).; Pure function: Function without side effects. Does not depend on external variables. Given the same inputs, always returns the same outputs. **FUNCTIONAL PROGRAMMING TECHNIQUES**: -Try to avoid data mutations; -Use built-in methods that don’t produce side effects; -Do data tranformations with methods such as .map(), .filter() and .reduce().; -Try to avoid side effects in functions: this is of course not always possible. **DECLARATIVE SYNTAX**: -Use array and object destructuring; -Use the spread operator(…);-Use the ternary(conditional) operator; -Use template literals.
174. Why worry about Architecture? -Like a house, software needs a structure: the way we organize our code;.- A project is never done! We need to be able to easily change it in the future; -We also need to be able to easily add new features;. We can use a well-

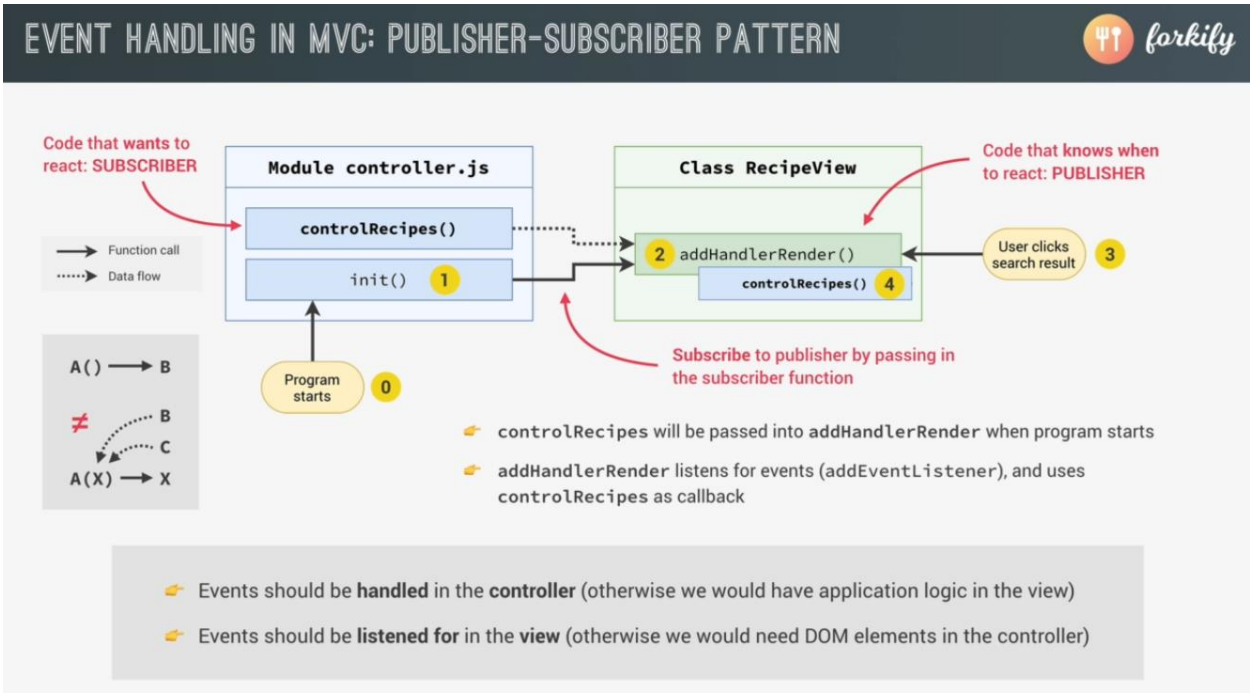
establised architecture pattern like MVC, MVP, Flux etc. We can use a framework like React, Angular, Vue, Svelte etc.

175. Architecture Components: **BUSINESS LOGIC**:-Code that solves the actual business problem; -Directly related to what business does and what it needs; -Ex. Sending messages, storing transactions, calculating taxes…; **STATE**: -Essentially stores all the data about the application; -Should be the "single source of truth"; -UI should be kept in sync with the state; -State libraries exist reudux, mobEx; **HTTP LIBRARY**: -Responsible for making and receiving AJAX requests; -Optional but almost always necessary in real-world apps; **APPLICATION LOGIC (ROUTER)**: -Code that is only concerned about the implementation of application itself; -Handles navigation and UI events; **PRESENTATION LOGIC (UI LAYER)**: -Code that is concerned about the visible part of the application; -Essentially displays application state;
176. The Model-View-Controller(MVC) Architecture: **Model** takes care of all the operation that handle data(Business logic, state and http library), **Controller** takes care of application logic(it is a bridge between model and view, which don't know about one another; Handles UI events and dispatches tasks to model and view) and the **View** that takes care of Presentation logic and displaying the data to the user.



177. Publisher-Subscriber pattern: The publisher doesn't know about the subscriber. The subscriber function must be passed as an argument for the publisher. When some event happen, addHandlerRender listen and than the function contolREcipes is called. The programs starts, with the help of the init() the subscriber subscribe to the publisher by passing the function controlRecipes()(SUBSCRIBER) as an argument of the

addHandlerRender()(PUBLISHER).



178. Event delegation. Selecting the parent element and add a function that handles the element that have the parent a class (e.target.closest(".className)) do the operations wanted.

```
this._parentElement.addEventListener("click", function (e) {
      const btn = e.target.closest(".btn--tiny");
      if (!btn) return;
      console.log(btn);
      handler();
    });
```

179. Writing documentation for javascript functions(jsdoc.app) /** */ : -adding the function description, @parameters used, what it @returns, @author, @todo etc.
180. Free sited for deploying static projects(html, css, javascript): netlify.com, surge.sh
181. Git fundamentals: git init; git status -> to see the files untracked or the changes not staged for commit; git add ./git add -A(add all files); git reset --hard HEAD(going back to the previous commit); git log(seeing all the commits made); q/:q(for quit); git reset --hard c20c651ebc1d02bdb9e5fc0119ba2ec0b228468a -> moving to a previous commit; git branch->showing available branches; git branch new-feature-> create a new branch; git checkout new-feature; git merge new-feature -> merging the new code to the main branch; git remote add origin githubRepository_URL; git push origin mainBranch; github git cheat sheet ->google