Important Notes

# CSS

Positioning:

* Position relative: relative to it’s normal position (that was determined by the html order).
* Position absolute is relative to its ancestor.
* Position sticky: toggle between relative and fixed (depending on the scrolling).
* **Parent** element is **relative**, **child** element is **absolut** ---> then you can **change** the positioning of the **child** with tbrl, **relative** **to** the **parent**. If you move the parent, the child moves with it.
* **Z index influences only non static elements. Use relative for it.**
* **Separation:** Style file, menu file, variable files, normalize : separated files.

## Box-shadow

box-shadow: 0 1px 8px rgb(0,0,0,.25);

## Classes

* Specify more- higher class. Pseudo classes (e.g. hover) would not work when the a higher class applies to the button. יותר ספציפי יותר חזק!

## Responsiveness

* Min-height, max-width, clamp(min value, ideal value, max value)
* Font size 62,5%, then use rem
* Mobile first!

# Git

- open new file: "repo test", go to the file with VIS code

- create new repo online

- open terminal in VIS and past the content of the repo to it (these first lines)

Text

Description automatically generated- To add all stuff in the file: add . ; commit -m „some input“, push

### Important git commands

* **Pull**: getting commits from other branches into the current branch. “git pull origin <branchName>”
* **Checkout**: the command used to move from commit to another commit (i.e. from one version of the file to another) or from branch to branch.
* **Checkout -b <branchName>**: creating a new branch locally. Then you commit the changes and get a response: sth sth upstream. Then you copy this command to your line and press enter, so the branch goes online.
* **Merge**: combining 2 branches.
* **Diff**: show the difference between 2 commits or 2 branches. Adding the file name afterwards.
* **Log:** see all the commits. Then you can copy the Id and with checkout go to it.
* **Delete**: branch --delete <branchname>
* **Delete all after the current version**: setting a new head.

## Javascript

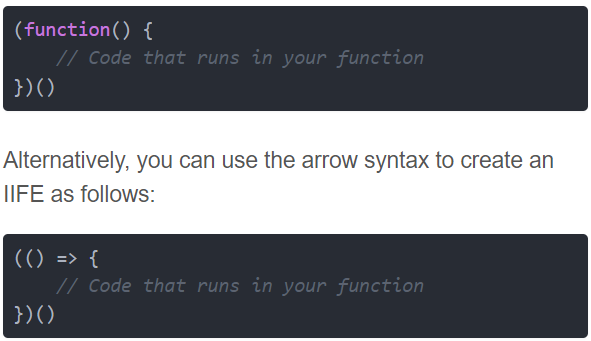
### Primitive Data types

* Number
* Boolean
* Symbol
* Bigint
* Undefined
* Null
* String

### Unsorted Notes (assign topics)

* While declaring a function in the brackets: parameters, while invoking it you give the function arguments!
* Function definition or statement: 1. Declaration. 2. Expression
* There’s a difference between a function declaration: function myFunction() {}. Function declaration means the function is hoisted. So you can use the function, and define further down.
* and function expression (assigning the function to a variable). These functions are not hoisted.

const variableName = functionName(){}. IT WORKS BECAUSE FUNCTIONS ARE OBJECTS. It means we can store them in arrays, pass them around etc. You can name the function in the expression or leave it anonymous.

* **IIFE**: you can make BOTH declaration or expression IIFE by: putting the function in () and putting () at the end. Like this:
* 
* **The use of ! and of !!:** name=‘Talal‘

console.log(!!name) ->> true. It gives you the direct boolean value of the variable.

It’s like: Boolean(name).

* **The reduce function:**

const nameOfReduceFunction (arr) => {

const reducedResult = nameOfReduceFunction.reduce((acc, curr) => acc + curr, 0);

console.log(`The result is ${reducedResult}.`

* **Destructor: get elements out of array :** if the [] is on the left of =, its a destructor and not an array! Allows you to extract values out of array, and at the same time declare them as new variables!

Text

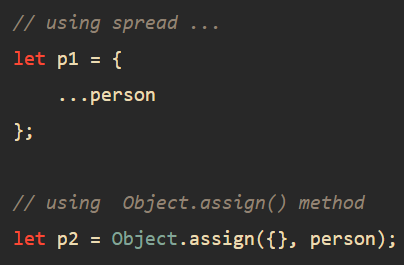
Description automatically generated

* **Shallow copy of array**

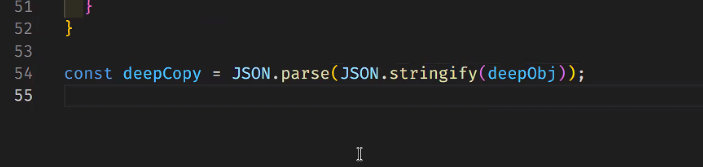
Graphical user interface, text

Description automatically generated

* **Shallow copy of an object:**

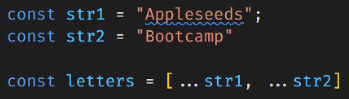


* **Deep copy of an object or array (same same):**

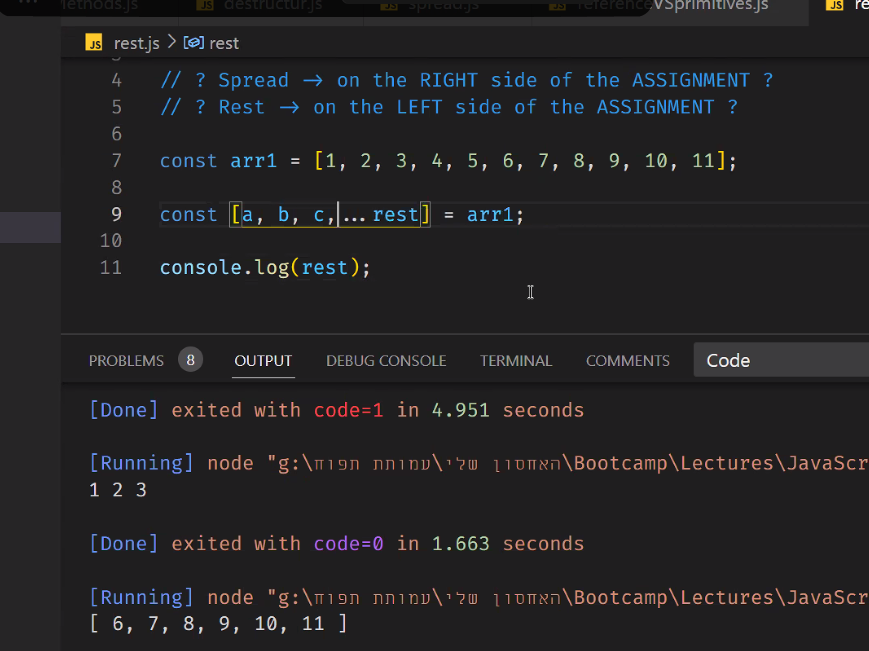


A deep copying means that value of the new variable is disconnected from the original variable while a shallow copy means that some values are still connected to the original variable. A shallow copy copies the first layer by value, but the second layer (z.b object in an object, or array in array) is copied by reference => it means that if you change the second layer of the shallow copy, you will change the original! Soooo always make a deep copy!

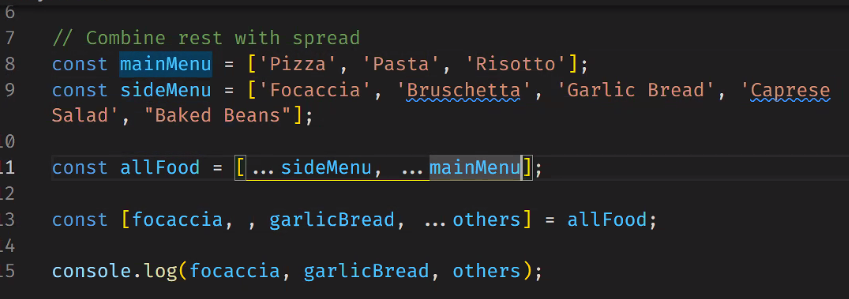
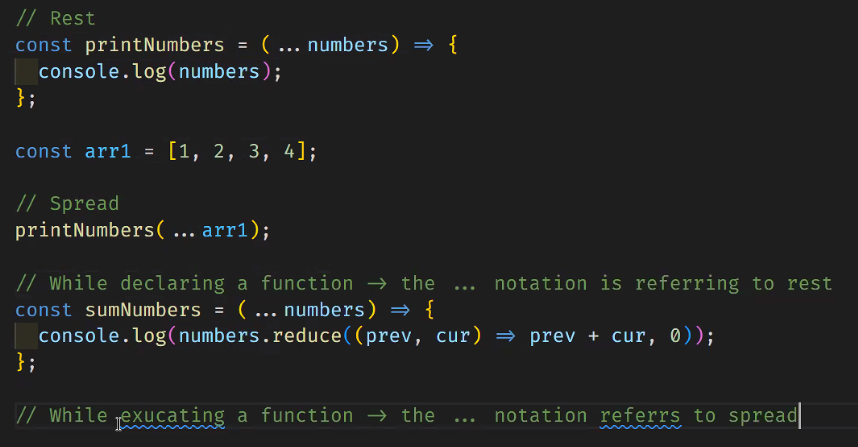
* **Spread: ... : extracts information from strings and arrays, and puts it in a new array.**

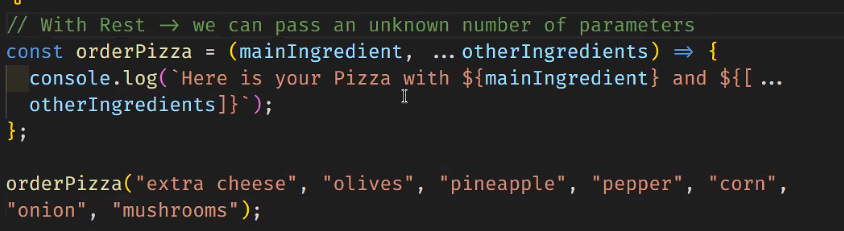


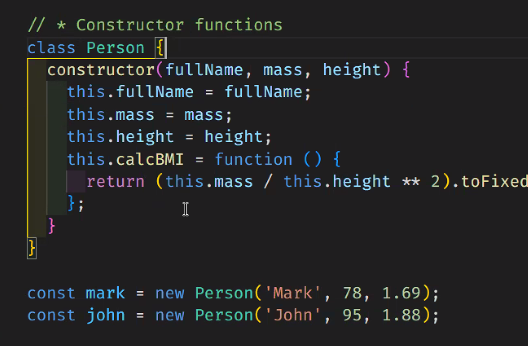
-**Rest**



* **Rest VS spread:** rest takes everything left on the left side and puts it in a new array (rest can have another name, it’s a variable). Spread reffers to the right side of the decleration (here allFood).





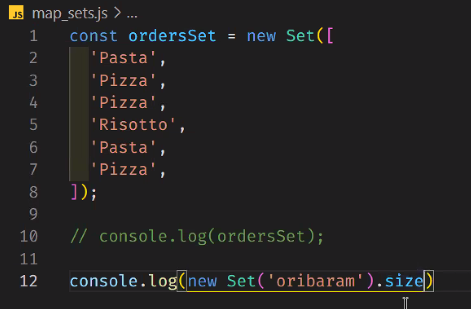
* **Accesess to keys and values of objects:**
  + To acces values: objName.keyName
  + To acces values that are numbers or variables: objName[keyName]
* **Iterating through objects (objects are not iterable):**
  + For loop through Object.keys() our Objects.values()
  + For in loop
* **Using constructor functions:** creating a class that is easy to repeat:
* 
* **Maps and sets data bases**

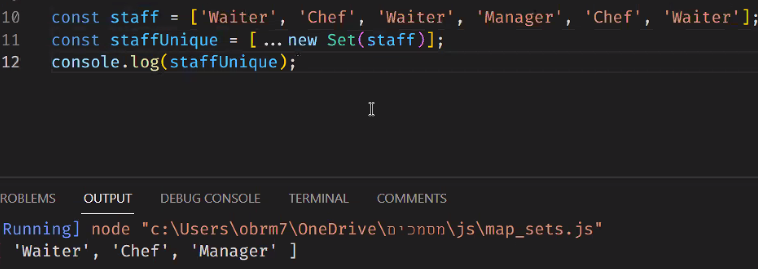
A map takes information and stores it in a new array or object or string. You have to declare it, because you create sth new!

* 

**You can use a callback function (usually nameless) while creating the map. This is a GREAT WAY TO EXTRACT INFORMATION FROM ARRAYS AND OBJECTS.**

* **Sets:**

Have only one repetition of each of the components : in the following examples, it returns only one repitition, or in the ‘oribaram’ case, only 6 letters.

Another example of returning unique valuese: 

### Hoisting

**var let const**: var decleration is hoisted to the global scope. Const and let are hoisted to the Temporal Dead Zone.

**In a function**: var let and scope are all not hoisted out!

**In a block**: only var decleration is not enclosed by block, but is enclosed by function.

**Function decleration** (function funName() {} ) is also hoisted to the global scope.

**Lexical scope:** nested functions have accrs to variables initialized in the outer function scope, BUT NOT THE OTHER WAY AROUND.

### Closure

One kind of closure is when sth inside has access to variables outside. Important: you also have access to the inside scope, after the scope was closed -using closure- defining a function or sth that calls it, that you can then call. Sth like that. **Check out again in the lecture code! Important for interviews.**

**פונקציה שמוחזרת מפונקציה, ששומרת על הסקוופ של הפונקציה שהחזירה אותה**

Text

Description automatically generated

### Text Description automatically generated

### This

* The important cases of this:
  + Object: this is the object.
  + Function invocation: this is global
  + Method invocation: Function invocation in a method (in object). this is the object.
  + Arrow function: if you use this in an arrow function, it inherits the context of the this above it (not like a normal function invocation). So if the arrow function is a method inside an object, this gets the value of global – the context of the this of the object! (and not the this that would be inside the object).
    - Sooo in a nested arrow function, this will get the value of the outer function -which is the value of this with the context of object.
  + Nested function inside an object: this gets the value of global. Because... we use the this of the nested function, and the one inside the object.
  + In a class (of a constructor): when you create an instance of the class, only then you give the class constructor a meaning, and the this that you write in the class gets the context of the instance of the class.
  + Indirect invocation (call, apply, bind): If you call a method outside of the object: this becomes the value of undefined.
    - You can use **bind** to give this the context of the object that it was in, and define a new function expression with it. Then you need to invoke the function (because you just defined it).
    - Another method to call a method outside the object without loosing the context of this: using **call** (for variables as arguments) or **apply** (for an array as argument). In these cases as well, this becomes the context of the object.

**BOTTOM LINE OF THIS IN FUNCTIONS**: in invocation: this becomes the context of the function itself (of where it’s existing). In an arrow function: this becomes the context of the parent.

### OOP

**Exporting and importing classes:**

When you export you export as an object, then you need to destructor it in the impoirt { <imported sth> }.

When you write export default you don’t need to destructor, you export the entire class as is.

Three ways to create a prototype of a certain class: Objecet.create(), constructor function and classes.

### Classes functions

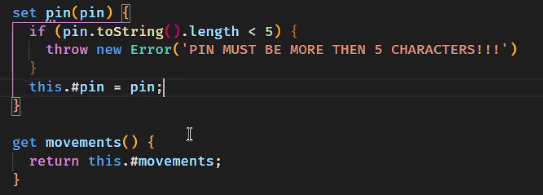
**get** : get funcName(){

you insert here a function or some kind of filter or whatever you want to do with the information.

}

Then you can call funcName like a **property** of the instance of the class. You don’t have to invoke funcName when you call it. You can also use it to get a secret property.

**set** : set a property, you can also use it to unravel a secret property



### Deep dive to event loop

* Just in time runtime :
  + The code is being parsed to an AST file
  + Then compiled to binary, ready to be executed.
  + functions and variable are passed out to the callstack, where it’s executed, and then it’s being stored in the heap.
  + API functions like event listeners and set timeout are being passed to the API, to keep track. When they’re ready to execute they are passed to the callback queue, and being put in the callstack when it’s empty again.
* Arrow functions don’t have their own argument object. It’s like this.
* Once a function was executed, it pops out of the callstack. After all was executed, the global is waiting at the bottom of the callstack.
* Variable look up: if the variable is not in the scope, the engien will look up and up in the scope and look for it.
* Lexi

### Call apply and bind

* call: binds the this value, invokes the function, and allows you to pass a list of arguments.
* apply: binds the this value, invokes the function, and allows you to pass arguments as an array.
* bind: binds the this value, returns a new function, and allows you to pass in a list of arguments.

### Asyncronous

* asyncronous functions run in the background via API, while js countinous runs its calls stack. When it’s time, they are bein put in the js callback queue. **setTimeout(), fetch(), new Promis((resolve, reject) => {**

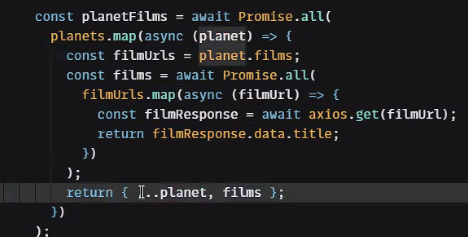
**}).**

* **async await**



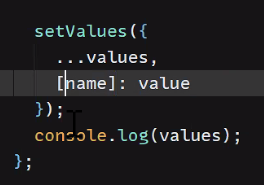
### Promise.all()

If you’re waiting for a few promises, you can use Promis.all( ), then you get as a result an array with the results of all promises.



## Tips and tricks

### Approaching dynamically keys of an object



Using the squared brackets here to updated the value of the chosen key. name can be different every time here.

### Easily converting a string to a number

Adding a plus before the string

