Buzz words:

|  |  |
| --- | --- |
| Isomorphic code =  Universal code =  Server Side Rendering (SSR) | Works everywhere (client & server side) |
| SEO - Search Engine Optimization |  |
| dynamic vs static language | Variables can change types vs must remain in the same type as it was declared at first. |
| type safety | Helps you not changing types of a variable after it was made. |
| Compiler vs. transpiler | Compiling is taking high level to low level. Transpiler is simply making one high level into another high level. |
| JIT language | Just In Time. |
| primitive | The most basic and first things in a language, like string, int, bool… |
| Shallow / deep copy | Shallow |
| IIFE | Immediately Invoked Function Expression – A technique to make "global" functions "local" to the file itself (put it in a scope) |
| Module Pattern | Uses IIFE to make functions "local" to the single file itself, in order to avoid relying on global scopes. |
| For **in** and for **of** | In is all the keys (object keys), of is all the values |
| Event propagation / bubbling | An event that works on one element will also work on all it's children For example, clicking on a button inside a div will click both the button and the div. |
| This = the one firing the event, e.target = the element that was the target of the event.  For example: this = div, e.target = button. |
| CSS Box Model | Content, padding, border and margin |
| CSS sizes: Px em  Rem: ROOT em. vh % | Px are usually on borders, and usually 1 px.  1 em = 16 px BY DEFAULT! Em is relative to font-size. If font-size is changed from 1 to 2em, and width is 80px, the width will be 80px, but width 5em (=80 px on default) will be 160px.  Rem: em that is not relative to the font-size of the element, only the font of the root document (=html).   vh: relative to screen size. 100vh = the entire screen. 50vh: half the screen. If you put html height = 100vh, the html will always take the entire screen, even if the content is not enough. |
| EcmaScript 6/7/8  Let/const  Class  (Fat) Arrow Function /  lambda expression  Async/await  Destructing object  ES6 modules import/export | Const must be initialized, it cannot be declared alone. Let/Const are BLOCK scoped, while var is FUNCTION scoped.  There is NO hoisting on Let/Const like there is on var!  They are not available in the global scope.  Class came from the OOP view, and it's not native, but JS can support it. A class needs a "constructor(par1, par2…)", which will be called for every new Class. Then, we put this.Par1 = Par1 and so on.  In order to avoid that kind of code duplication, we can write in TypeScript and add "private" to each parameter. TS will make the "private" a "this.par1 = par1" after compiling.  Inheritence: class B extends A. in order to refer to A elements in B, we use "super". Super(par1…) for constructor, super.func for functions.  normal functions (that act likes classes) can work without "this", but there are performance issues, because each function is cloned to every new instance. But if you use this in "other" functions (like Timeout), the "this" is what that function decides. A "this.x" of the "main" function is not the same as "this.x" of that other function.  Enter Arrow func => they keep the "this" context to the same "overall this" of the function. Written like this: =>{} instead of function(){}.  P.S: we can use const that = this and use "that" in regular functions to keep referencing the right "this".  Make a function "async" (has no meaning, just to show we can use await in it) and then we can use "await" on "Promises" to "wait" some time, thus resembling a synchronous delay.  Make an object with properties, and then make "local" variables with the object's properties.  Const obj = {id:1} and const {id} = obj. Then, write id=2, and obj.id will stay 1 (object property remain untouched) and id will be 2.  In one file: "export function…" in the other: "import from \*place\*". **UNUSABLE IN THE BROWSER**! Only in Node! |
| Typescript:  Type  Interface  Accessibility |  |
| Boiler plate | Anything that is not the main issue |
| The value of This | f()-> Window. f() & "use strict" -> undefined. obj.f() -> obj. |
| "use strict" | Setting a variable without declaring it first.  The value of this inside global function -> undefined (normally is Window).  Trying to modify a readonly property. |

Trainologic link: https://github.com/Trainologic/JSBootcamp3.git

WINDOWS TRICK: Shift + Right click a file to get a nifty little option: "Copy as path" under the "Open with >".

Memory mix-up: 1 KB = 1024 Bytes, but in the "real world" we count it as 1000 bytes.  
1 KiB is an "actual" 1024 bytes.  
1 KiB > 1 KB (by 24 bytes).

Extra reading:

1) prototype chain

23/04/18 – 9:15

JS is a JIT language – Just In Time.

23/04/18 14:30

Hoisting : in a **scope**, any variable that would be declared inside the scope will actually, behind the scenes, be declared at the beginning of the scope. If the variables gets a value while being declared, the **declaration** will be performed first, and the **hasama** is made at the respected written line.

When doing a logical test of && between 2 "true" things (strings, numbers, so on), it will return the last thing evaluated.

1 && "bye" && "hello" && 0 && "ok"

Will return ok.

1 && "bye" && "hello" && **null** & 0 && "ok"

Will return null, because it's like "false", and thus the evaluation stops there.

24/04/2018 9:00

Because there are several scenarios, like null, 'undefined' and such, the best way to check for something if it's true or false is with !!var.

Bitwise (p. 56 in slide 1):

A binary action between 2 variables that results in true or false.

Binary numbers example / trick to calculate:

|  |  |
| --- | --- |
| 0110 |  |
| 2^3 2^2 2^1 2^0 |  |
| 8421 | 8\*0 + 4\* 1 + 2\* 1 + 1\* 0 = 4 + 2 = 6 |

|  |  |
| --- | --- |
| 1111 |  |
| 2^3 2^2 2^1 2^0 |  |
| 8421 | 8\*1 + 4\* 1 + 2\* 1 + 1\* 1 = 8 + 4 + 2 + 1 = 15 |

|  |  |
| --- | --- |
| 1 | 1 = 2^0 = 1 |
| 2 | 10 = 2^1 = 2 |
| 3 | 11= 2^1 + 2^0 = 3 |
| 4 | 100 = 2^2 = 4 |
| 5 | 101 = 2^2 + 2^0 = 5 |
| 6 | 110 = 2^2 + 2^1 = 4 + 2 = 6 |
| 7 | 111 = 2^2 + 2^1 +2^0 = 4 + 2 +1 = 7 |
| 8 | 1000 = 2^3 = 8 |
| 9 | 1001 = 2^3 + 2^0 = 9 |
| 10 | 1010 = 2^3 + 2^1 = 8 + 2 = 10 |

25/04/2018 10:00

Synchronous code runs in the Stack, every line goes in the stack, and then executed.

Asynchronous code, however, goes in the stack, and then goes to the Runtime to be executed there. Even if it's finished, it will not be executed until the synchronous code in the stack is done and over with. Instead, it will go in a queue, and will be executed first in first out according to the "answers" from the runtime.

13:00

An array is saved like an object, with key – 0 : value …, key – 1 : value … and so on.

Array have an important function named map. It will go over the array, one element at a time, and will replace the element with a new one, according to the callback.

arr.map (function (element) {

return new\_element;

});

15:30

Delete an element in an object: delete myObj['key'];

26/04/18 9:20

setTimeOut doesn't happen after X milliseconds, that time is only WHEN it will get **in the queue!**

14:20

**Data structures** – p. 124

1) Static array – FIXED SIZED. Can access elements easily (=random access). In JS it is represented by Array(num).

2) Dynamic array (A.K.A ArrayList) – Can change in size. Add, Remove. Can access elements easily. When size changes, it will reallocate the memory using different methods (+1, \*2 and so on).   
Meaning, adding an item when the array is full will result in it being copied into another place in the memory and point there, and the old array is discarded.  
In JS, it is represented by [].

3) Stack – Last In First Out. LIFO. Push (insert) and Pop (remove last).  
Cannot access elements easily, only the last element in it.  
In JS it is represented by a plain array.

4) Queue – First In First Out. FIFO.

5) Linked List – no random access. Can add / remove items in the beginning and end. Items are not necessarily put one next to another, they can be anywhere in the memory, and each will "point" to the next (and former in case of Double Linked List), thus allowing to "flow" through them if given the first (last) item.  
Because they are randomly placed in the memory, we can add and remove items without changing the size of the list. Each new item will point to the former/latter, and thus the increase of size is formed.

29/04/18 9:30

A capital letter in a function is SUPER important. It means that it gives back a NEW object, and not overwrite an existing object.

13:00

Array is good because:

1) random access – provided we know the index of the value, we can easily get the value in the array.

Array is bad because:

1) Bad allocation: if the array is full, adding a new element to it will cause a re-allocation to a different part of the memory + more space for the added value.

2) Searching – at best it will be O(N). Might be good, but another thing is better (O(1))…:

Linked list is good because:

1) can add elements easily, no extra use of memory, simply link the new one to the previously last element.

Linked list is bad because:

1) no random access! Must "travel" the entire list from the beginning each search.

9.5.2018

CSS: To get the whole box model and not just content : CSS pp 36.

10.5.2018 10:40

Flex: the ratio which elements grow or shrink next to other elements.

Grow – the element will be X times bigger than another element.

Shrink – the opposite of grow.

basis – a certain minimum, of which it will never be lower than.

direction - defaults to Left to Right.

The way to make an element flex:

Display: flex.

Flex-direction: column / row

Flex: 0 1 10em (grow 0, shrink 1, basis 10em)

This is equal to:

Flex-grow: 1  
flex-shrink: 0  
flex-basis:10em

14.5.2018 10:00

Typescript:

Interface is a blueprint, a skeleton, a "shablona" for types of our creation.

Interface \*name\* {

Key: type – the items of such interface **MUST** contain this kind of key of that type

Key?: type – same as above, only it's **optional**!

}

When making new variables, we can give them the interface as a type.

12:00

React advantages:

1) Good connection between components. We start with a source and can easily access all of its children.

2) Virtual DOM: every little change to the DOM can happen tons of times.

In Virtual DOM, React takes all of the changes that need to happen and shoot them all at once, and not each change in its turn.

3) Using JSX (superset of JS), we can write HTML in our JS!

Component = React element. It holds HTML, CSS and JS of itself. We can reuse that component in different places, and each one would have the same HTML, CSS and JS.  
Each component can work in a "vacuum", relying on no-one but itself.

14:30

When writing HTML in JSX, the "class" attribute is named className, and id is key.

When writing HTML in JSX, if we want to refer to the normal JS, we write it in {}.

Const item = "one";

<li> {item} </li>

21/5/2018 15:00

Unit Testing: Test every single function on its own.

in Jest: we need to make a directory named "\_\_tests\_\_" and put tests JS files there.

Functions:

describe (description, callback full of "it" functions) – a group of tests.

it (what should happen in the test, callback with the test itself) – a single test.

Inside "it" there are 2 functions tied together:

expect().toBe() – what needs to be returning.

expect(() => {}).toThrow() – when an error thrown is actually the good thing to expect

beforeEach – before every "it" test, it will perform something (like create an object instance to test on).

beforeAll ( () => { \*CONTENT\* } ) – something to happen only once per group of tests.

expect.assertions(number) – when we expect a few assertions in a single test (like, when we need to assert an array, so we expect to test each and every element in it).

Import Enzyme, {mount} from 'enzyme'

Enzyme: can test components.

const wrapper = mount (component).

Now we can use expect(wrapper.prop\*(\*PROP NAME\*)).toBe(\*PROP VALUE\*);

\*prop gives A SINGLE PROP (according to its name).

propS gives all of the props as an object of key – value.

27.5.2018 10:00

In React, render only applies to the Virtual DOM, so when we have React activate its Render(), we don't really re-fresh the entire app, only the Virtual DOM is changing, and then any difference is written into the actual DOM. This operation saves a lot of time, taking few milliseconds.

A component's state is both the this.state and the this.props it gets from a parent.

Webpacks:

1) npm install webpack

2) node\_modules\.bin\webpack – this will tell us to install the webpack cli

3) npm install webpack-cli

4) same as 2 – this will tell us it defaults to production because we lack a configuration file.

We will find how to make one in the webpack docs. We'll just… copy-paste…. In a new .js file named webpack.config.

'Entry' should be the main file that bundles everything in it (like index.js or app.js)

But it's not enough! We need to add 'mode': 'development'.

5) Again 2. Now, put the output in the HTML / Node and they will both "eat it".

In order to avoid writing node\_modules\.bin\webpack all the time in order to bundle, we can use a package.json with a

**"scripts"** : {  
**"dev"**: **"webpack –w"**,  
**"build"**: **"webpack"**},

and then only use npm build. We can use "dev" to only do it once, and it will update for every little change.

7) What if we want to write in TS?

This won't work with typescript, because it only looks for .js. so to fix it, we add in webpack.config:   
resolve : { extensions: ['.ts','.tsx','.js'] } and so on.

But even this is not enough… the bundling won't know what to do with TS only stuff (like interface). We need ts-loader for that! npm i ts-loader . (don't forget to install ts as well).  
and finally, we must add a tsconfig.json.

Webpack & react

1) npm i react react-dom

2) npm i @types/react @types/react-dom

3) turn on jsx flag in tsconfig and lib: ["es6","dom"]

4) webpack.config.js – extensions : [".tsx"]…

Webpack & CSS

1) npm i css-loader style-loader

2) add to rules:

28.5.2018 9:00

Advanced JavaScript

Prototype is pointing to an Object that holds the function.

11:15

Function obj {

this.a = 'a' this.b = 'b'

Func f(arg1, arg2) {

Print(this.a + this.b + arg1 + arg2)

}

}

Call:

Function.call ( {} ) – this calls the function (like f(); ) but add an object to be used as a context for "this" if the function uses it.

f() -> will result in undefined.

f.call({'a' : 'aaa'}) -> will result in aaa.

f.call({'a' : 'aaa'}, 'b') -> will result in aaab.

f.call({'a' : 'aaa'}, 'b', 'c', 'd') -> will result in aaabc (d is ignored).

Apply:

Pretty much the same as call, but the other arguments are in an array.

f.apply({'a' : 'aaa'}, ['b']) -> will result in aaab.

f.apply({'a' : 'aaa'}, ['b', 'c', 'd']) -> will result in aaabc (d is ignored).

Bind:

Gives a context from another source.

f.bind(obj) takes the obj (with the a and b defined in it) and gives their context to the f, thus:

f() -> 'ab'

3.6.2018

Closure and Scopes.

When we make a function Point(x,y), which has a function print(), and then do:

Const pt1 = Point(1,2);  
Const pt2 = Point(2,3);

We create 2 pointers.

the pointers reside in the Stack and they point to a place in the memory (Heap) where the function and its variables reside.

A pointer takes 8 bytes (4 if it’s a 32 bit OS or browser).

In Chrome, an Object's size is 24 bytes (at the bare minimum).

Meaning, our Point has 24 bytes from being an object + 8 from itself = 32.

If we add additional fields to Point, but **not** use them, they **won't** appear in the scope.  
If we **use** them in functions, but **don't** return them (thus making them un-reachable), they **will** appear.

**Conclusion**: As long as something is referenced to in the scope, it will appear in the scope.

Because of those un-reachable fields, it can result in a memory leak.

Shallow size: the size of the object itself and only itself. Even if it points to different things (variables and functions and objects), it will still only show the "pure" size, disregarding them.

Retained size: the "actual" size of the object, including everything it points to and references. If other objects reference things that I reference, it **won't** appear in the retained size. if we delete the object, we will get that amount of memory.

5.6.2018 Node.js