Buzz words:

|  |  |
| --- | --- |
| Isomorphic code | Works everywhere |
| 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 variables 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 |

Assignment 1 link: https://github.com/Trainologic/JSBootcamp3.git

Read on 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.