**Basics**

* console.log may use multiple values

console.log (variable1, variable2, …);

* x2  may be written as x \*\* 2 and similarly other powers of x.
* console.log (`string interpolation available: ${x}`);
* null, 0, NaN, undefined,’’ considered “falsy” values;
* ‘+’ operator always concatenates, even if they are two numbers but

‘-’, ‘\*’, ‘/’ operators convert to number format if possible and then perform

operations between the numbers.

let a = ‘10’ + 2; //a = ‘102’

let b = ‘10’ – ‘2’; // b = 8

* const x = ‘10’; let b = Number(x).
* typeof x gives the type of x.
* ‘===’ checks the type and the value. ‘==’ checks only the value.
* Dynamic nature of variables:

let x = 5; //valid

x = ‘sabya’; //valid

x = true; //valid

* Functions can be called before they are defined. But Function expressions cannot be.

const x = CalAge(15); //Invalid to call a function expression before it is initialized.

const CalAge = function (birthYear){ return 2021 - birthyear; }

* Arrow function: same as C#

const CalAge = x => 2021 – x;

CalAge(15);

If multiple lines of code then we must include them in ‘{‘ and ‘}’ brackets and return statement must be used if returning something.

const CalAge = x =>

{

console.log(‘This is a normal statement’);

return 2021 – x;

}

**Execution contexts**

Execution context represents the code and the memory in which the code is executed.

Basically, the “context” or the “environment” in which the code is to be executed.

Rules:

1. Once the code is compiled during execution, javascript engine scans through the code and starts “picking up” variables with ‘*var’*, ‘*let’* and ‘*const’* and functions (regular ones, neither arrow nor anonymous functions). After completion then moved to the top. This is called **Global Execution Context (GEC)**.
2. i)Variables with ‘*var*’ are assigned an initial value of *undefined*. Functions are assigned the code inside them.

ii)Variables with ‘const’ and ‘let’ are also moved to the top but stored in a different memory (not the **GEC**).

iii)Remember in the following code, **const x = (y) => y\*y\*y;** ‘**x**’ is treated like a variable of type ‘*const*’ and moved on the top to a different memory.

1. **Now, the code starts execution of the code**. **Until now, only scanning and execution context creation was done**. As and when the execution reaches a particular code, it assigns the value to that variable. Before that if the variable is accessed, it would give *undefined* as the value.
2. This process of creation of execution context happens for every function call. Every function has an execution context. If there is function call within another function call, the outer function is halted and inner function execution context “starts”. Every function has access to its own and its parents’ execution context (and therefore all the variables in them, too).

**The ‘*this’* keyword**

Rule 1: ‘*this*’ points to the object that is currently being referred.

Rule 2: Inside a function, it refers to the object that is currently calling the function.

Consider:

function playVideo(){

console.log(this);

}

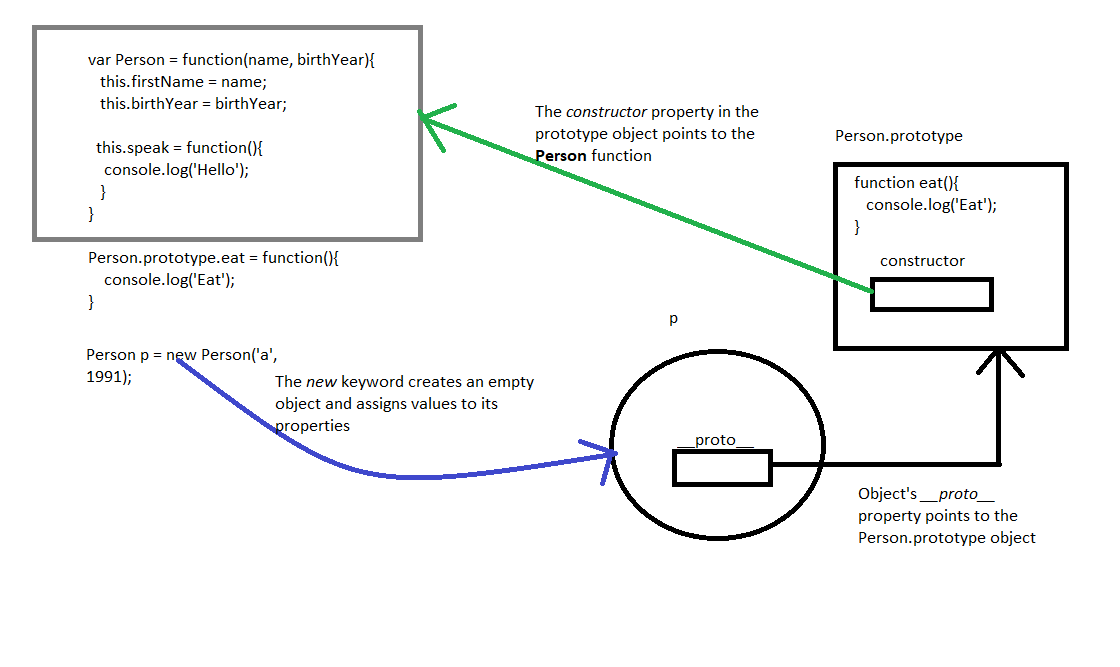
playVideo(); //scenario 1

var p = new playVideo(); //scenario 2

In scenario 1, ‘*this*’ points to the ‘window’ object.

In scenario 2, ‘*this*’ points to the ‘playVideo’ object.

**Objects and Prototypes**



Quick points:

* Classes are not hoisted