**Java Script**

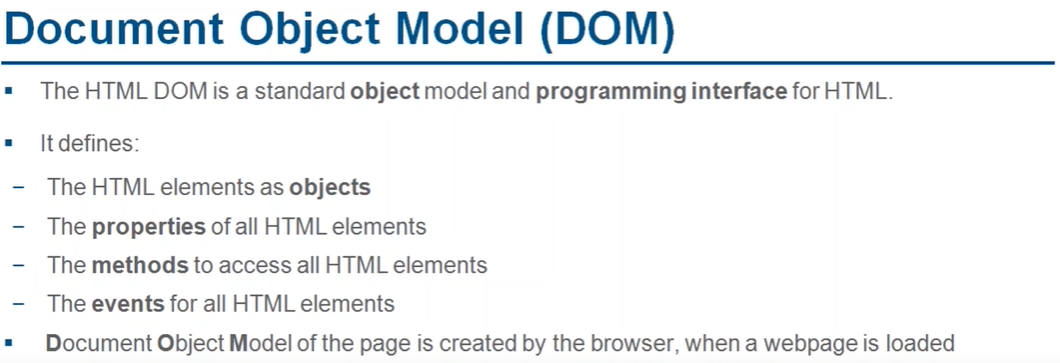
->Browser and Node provides run time env for JS.

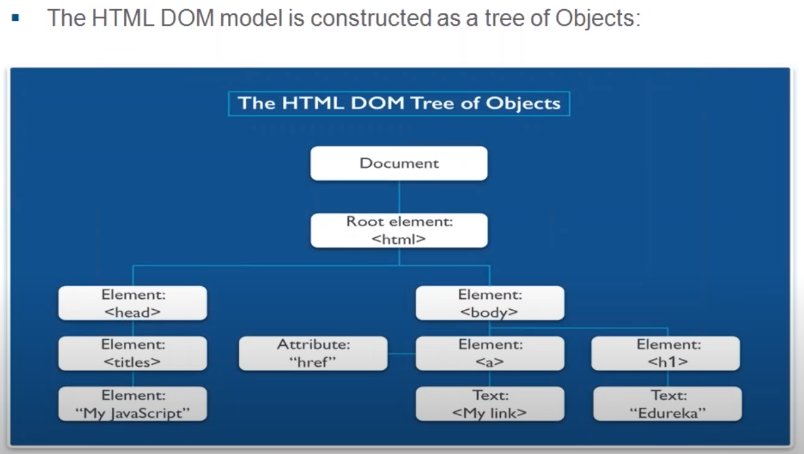
->JavaScript runs in Browser then. Now JS also runs outside the browser using NODE.

NODE: A C++ program that includes V8 JS engine.

->ECMA script is a specification and it defines standards whereas JS is Programming Language.ES6 provides many specifications for JS.

**DOM(Document Object Model):**

****

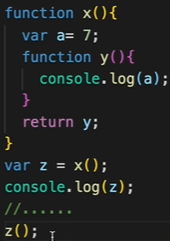
****

* Java Script is a Synchronous, Single-threaded language. It is a loosely typed language.
* Execution Context: Everything in JS happens in Execution Context

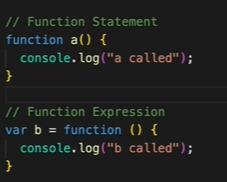
Memory (Variable Environment): variables are stored as key-value pairs.

Code (Thread of Execution): code is executed line by line.

* A Call Stack is maintained by JS to execute execution contexts in a particular order.
* Execution flow: Before executing the code JS allocates memory to all variables, arrow functions as “undefined” and actual function copy to functions.
* **Window Object**:
* Window object (global object) consists of many variables and functions which we can use anywhere in the JS program
* This window object is loaded by JS engine into the global space.
* JS engine also creates “this” keyword which points to the window object.
* **Scope and Lexical Environment**:
* Every EC of a function creates reference to the lexical environment of the parent function in the memory/variable environment.
* The lexical environment of a function contains the memory of the execution context and the lexical env of the parent.
* The chain of all the lexical environments and parent references is called “Scope Chain”.
* Let, const can also be hoisted but differently than var hoisting.
* In let, const hoisting the memory is allocated to the variables in other than global object unlike var hoisting.
* Temporal Dead Zone: The time between hoisting and the initialisation of the let variable. Reference error occurs when we try to access let variable before initialisation.
* Unlike var, let and const are block scoped where var can be accessed outside the block. Let assigns different value to a variable every time in each iteration (creates separate copy at different locations).

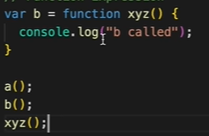
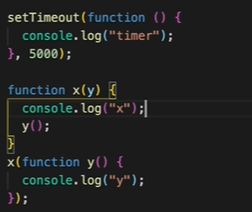


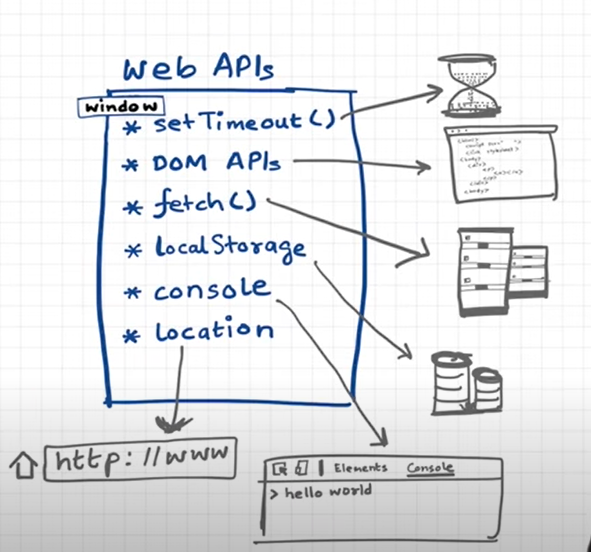
* **Closure**: function along with its lexical environment bundled together.

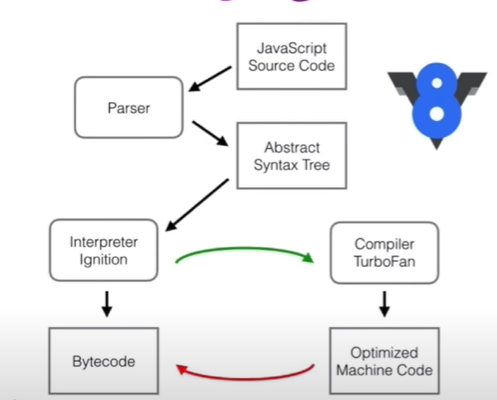
Here x() returns the function y along with its lexical scope whereas z() can access variable a without x() is in the scene.

* Provides data encapsulation or data security
* Disadvantage is that it requires much memory and the variables are not garbage collected after the program expires.

**Self-invoking function or IIFE:**

* A Self-invoking function can be created by wrapping the event handler assignment inside an immediately invoked function expression (IIFE). The IIFE takes the current value of referring element as an argument and assigns the event handler within its scope.
* EX: (function(button) {
* button.onclick = function() {
* handleClickedButton(button.id);
* };
* }) (button);
* **Functions**:
* The difference between function statement or declaration and expression is based on the hoisting.
* Anonymous functions: can be used when a function should act as a value to a variable and used to achieve a function expression.
* Named function expression: xyz() function call throws reference error because xyz() declaration is not in the global scope.
* First Class Functions: The ability of functions which can be used as values assigned to a variable and can passed as arguments to another function and can be returned from a function. This ability makes functions as first-class citizens.
* Callback Functions: The functions that are passed as arguments to another function. Here y() is a call back function. These makes JS as asynchronous. Callback functions prevents the mainthread(call stack) from blocking. The functions passed to an eventlistner is also a callback function.
* Web API’s:



* Event Loop:
* Event loop is responsible for checking the callback queue or microtask queue for any callback functions which are previously registered in the Web API’s environment and push them into call stack which is empty.
* Microtask Queue has higher priority than the call back queue.
* The elements of the microtask queue can be the callback functions from promises(fetch()), or from mutation observers of the DOM object.
* Starvation : The phenomena of a task inside callback queue waiting for a long time for execution because of the high priority of the microtask queue.
* V8 Engine: