These are the complete notes, You can view any one of the chapters from the .md files above :D

Episode 1 : Execution Context

Everything in JS happens inside the execution context.

Assume execution context to be a big box where everything takes place. It has 2 components in it:

- **Memory :** The place where all the variables and functions are stored as (key:value) pairs. Memory component is also known as *variable environment*.
- **Code :** The place where code is executed one line at a time. Code component is also known as *Thread of Execution*

JS is a synchronous single-threaded language.

- By single threaded, we mean JS can only run 1 command at a time
- By synchronous single threaded, we mean it can run 1 command at a time, *in a specific order*

Episode 2 : Execution & Call Stack

Everytime you run a program, an execution context is created. When a variable or function is encountered, it is stored in the memory area.

```
var n=2;
function square(num){
    var ans = num*num;
    return ans;
}

var square2 = square(n);
var square4 = square(4);
```

Now first, for this entire code a **Global** execution context is created.

In the first phase (memory creation)

- Memory is allocated to variables and functions.
- For variable name(which is key) it assigns a value of **undefined**
- For the function name(which is key) it assigns the entire function code as value.

```
n:undefined
square:{...entire-code...}
square2:undefined
square4:undefined
```

In the second phase (code execution)

- The variable name is replaced with its actual assigned value from code. So now n:2
- Skips over function code as there is nothing to assign there.
- We encounter a function call in square2. So a brand new local EC is created inside the code part of global EC and this will have the same 2 components: Memory and Code.
- In the local EC, ans and num are both undefined (in first phase). Then, the n value in global EC is passed to num, replacing undefined. num is the parameter and n is the argument.
- ans = num*num (calculated in code part of local EC and returned) replaces undefined in local EC (memory part) and the final value is returned from local and is assigned to square2 var in global. After returning, local EC is removed form global EC and control goes back to global.
- One more fun. call is met. Same thing happens here. Once square4 value is replaced from undefined to 16, global EC will also be deleted.

To manage all these EC, a call **stack** is created. Everytime code is run, the EC is pushed in. So first global EC is pushed. Then e1 EC(for square2) is pushed, and then after value returned, is popped. Similarly e2 EC(for square4) is pushed, and then popped and finally Global is also popped and stack is empty.

> Call Stack maintains the order of execution of execution contexts

Call stack aka Execution control stack, program stack, control stack, runtime stack ad machine stack

Episode 3 : Hoisting

```
// code example 1
```

```
var x = 7;

function getName(){
    console.log("Namaste JavaScript");
}

getName();
console.log(x);
```

Output:

> Namaste JavaScript

> 7

```
// code example 2

getName();      // in most languages, both lines which are above their
declaration will give error. Not in JS though.
console.log(x);

var x = 7;

function getName(){
    console.log("Namaste JavaScript");
}
```

Output:

> Namaste JavaScript

> undefined

```
// code example 3

getName();
console.log(x);

function getName(){
    console.log("Namaste JavaScript");
}
```

Output:

> Namaste JavaScript

> Error: x is not defined // note that not defined here and "undefined" in sample 2 are totally different.

- Not defined: We have not initialised the value for variable anywhere in the entire code and in memory space.
- Undefined:

Hoisting is a concept which enables us to extract values of variables and functions even before initialising/assigning value without getting *error*

```
// code example 4

function getName(){
    console.log("Namaste JavaScript");
}

console.log(getName)
```

Output:

> f getName(){
>
> console.log("Namaste JavaScript);
>
> }

```
// code example 5

getName();
console.log(x);
console.log(getName)

function getName(){
    console.log("Namaste JavaScript");
}
```

Output:

> Namaste JavaScript

> ReferenceError: x is not defined

> f getName(){

```
    console.log("Namaste JavaScript);


}
```

```
// code example 6

console.log(getName)

var getName = function () {
    console.log("Namaste JavaScript");
}

var getName = () => {  // use fat arrow function
    console.log("Namaste JavaScript");
}
```

Output:

> undefined //it is because they behave as variable and not function.

REASON OF WEIRDNESS

- The answer lies in the Global Exection Context. In the memory phase, the variables will be initialized as *undefined* and functions will get the whole function code in their memory.
- This is the reason why we are getting these outputs.

Episode 4 : Functions and Variable Environments

```
var x = 1;
a();
b();   // we are calling the functions before defining them. This will work
properly, as seen in Hoisting (Ep3)
```

```
console.log(x);

function a() {
  var x = 10;
  console.log(x);
}


function b() {
  var x = 100;
  console.log(x);
}
```

Outputs:

> 10

> 100

> 1

Code Flow

- The Global Execution Context (GEC) is created (the big box with Memory and Code subparts). Also GEC is pushed into Call Stack
  > Call Stack : GEC
- In first phase of GEC (memory phase), variable x:undefined and a and b have their entire function code as value initialized
- In second phase of GEC (execution phase), when the fun is called, a new local EC is made. After x = 1 assigned to GEC x, a() is called. So local EC for a is made inside code part of GEC.
  > Call Stack: [GEC,a()]
- For local EC, a totally different x variable assigned undefined(x inside a()) in phase 1 , and in phase 2 it is assigned 10 and printed in console log. After printing, no more commands to run, so a() local EC is removed from both GEC and from Call stack
  > Call Stack: GEC
- Cursor goes back to b() function call. Same steps repeat.
  > Call Stack :[GEC, b()] -> GEC (after printing yet another totally different x value as 100 in console log)
- Finally GEC is deleted and also removed from call stack. Program ends.

Episode 5: Window and this keyword

Everywhere JS is run, it is done with a JS execution engine. For Chrome: v8

- Shortest JS program is nothing but an Empty JS file
- Even for this program, JS engine does a lot behind the scenes
- It creates the GEC, the "window" and the *this* variable
- Window is a big global object that has a lot of functions and variables. All of these can be accessed from anywhere in the program
- *this* points to *window*
  > this === window -> true (at global level)

```
var a = 10;      // not inside any fun. So global object
function b() {    // this fun not inside any function. So global.
  var x = 5;    // not global
 }
console.log(window.a); //gives us "a" value
console.log(this.a); //this points to window so it returns "a" value
console.log(a); //also gives same "a" value. (if we dont put any . in front
of variable, it **assumes variable is in global space**
console.log(x); // x is not defined. (tries to find x inside global space,
but it isn't there)
```

- Global space is anything in JS which isn't inside a function. All these global objects will be present inside the windows schema. But non globals ones won't be there (here, x)
- When a GEC is made, *this* is also created with it (even for functional(local) EC). Global object provided by the browser engine is the window, so *this* points to window.

Episode 6: Undefined vs Not Defined

- In first phase (mem alloc) JS assigns each variable to a placeholder called *undefined*
- *undefined* is when memory is allocated for the variable, but no value is assigned yet.
- If an object/variable is not even declared/found in mem alloc phase, and tried to access it then it is *Not defined*

> When variable is declared but not assigned value, its current value is undefined. But when the variable itself is not declared but called in code, then it is not defined.

```
console.log(x);
var x = 25;
```

```
console.log(x);
console.log(a);
```

> undefined
> 25
> Uncaught ReferenceError: a is not defined

- JS is a loosely-typed / weakly-typed language. It doesn't attach variables to any datatype. We can say var a = 5, and then change the value to bool (a = true) or string (a = 'hello') later on.
- **Never** assign *undefined* to a variable manually. Let it happen on it's own accord.

Episode 7 : Scope and Lexical Environment

```
This is why JS is confusing (Case-1)

function a() {
    console.log(b); // surprisingly instead of printing undefined it prints
10.
    //So somehow this b could access the b outside the fun.
}

var b = 10;
a();

----------------------

Another case: (Case-2)

function a() {
  c();
  function c() {
  console.log(b); // when cursor comes here, it still prints out 10
somehow!!
  }
 }
 var b = 10;
 a();

 --------------------

 Another one (DJ KHALED!) (Case-3)

 function a() {
  var b = 10;
```

```
  c();
  function c() {
    console.log(b); //it prints the right value. How? See ans below Summary
part
  }
 }

 a();
 console.log(b); // now when cursor comes here, it prints NOT DEFINED!
```

- This is the intuition behind **scope**
- Scope is directly dependent on the lexical environment
- **Lexical Environment** : local memory + lexical env of its parent
- Whenever an EC is created, a Lexical environment(LE) is also created and is referenced in the local EC(in memory space)
- Lexical means hierarchy. In the DJ KHALED (xD) code, function c is lexically inside function a.
- So in EC of c(), variables and fun in c (none) + reference of lexical env of parent a() is there
- LE of a() in turn is its memory space + reference to LE of parent (Global EC)
- LE of Global EC points to *null*

```
   To summarize the above points:

   call_stack = [GEC, a(), c()]

   Now lets also assign the memory sections of each execution context in
call_stack.

   c() = [[lexical environment pointer pointing to a()]]

   a() = [b:10, c:{}, [lexical environment pointer pointing to GEC]]

   GEC =  [a:{},[lexical_environment pointer pointing to null]]
```

For case -3

- First JS engine searches for b in local mem of c(). Nothing is there.
- So it goes to the reference of Lexical env of parent a(). Here b = 10 is here. So it takes this value, goes back to c() and console prints it.
- Had b not been in a(), then pointer would have gone to a()'s parent (Global EC and searched there). Had b not been there too, then it goes to LE of global's parent which is

null. Now JS engine stops and says b is NOT DEFINED.

- **Lexical env of c = Local memory of c + LE of A + LE of Global**
- This process of going one by one to parent and checking is called **scope chain**

Episode 8 : let, const, temporal dead zone, types of errors

> let and const declarations are hoisted. But its different from var

```
console.log(a); // ReferenceError: Cannot access 'a' before initialization
console.log(b); // prints undefined as expected
let a = 10;
console.log(a); // 10
var b = 15;
```

It looks like let isn't hoisted, **but it is**

- Both a and b are actually initialized as *undefined* in hoisting stage. But var b is inside the storage space of GLOBAL, and a is in a separate memory(script), where it can be accessed only after assigning some value to it first.
- ie. one can access 'a' only if it is assigned. Thus, it throws error.
- **Temporal Dead Zone** : Time since when the let variable was hoisted until it is initialized some value.
- So any line till before "let a = 10" is the TDZ for a
- Since a is not accessible on global, its not accessible in *window/this* also
  > window.b or this.b -> 15; But window.a or this.a ->undefined, just like window.x->undefined (x isn't declared anywhere)

```
let a = 10;
let a = 100;  //this code is rejected upfront as SyntaxError. (duplicate
declaration)
------------------
let a = 10;
var a = 100; // this code also rejected upfront as SyntaxError.
(can't use same name in same scope)
```

Let is a stricter version of var. Now, **const** is even more stricter than let.

-const holds all above properties of let.

```
let a;
a = 10;
console.log(a) // prints 10 properly. Note declaration and assigning of a
is in different lines.
------------------
const b;
b = 10;
console.log(b); // SyntaxError: Missing initializer in const declaration.
(This type of declaration won't work with const. const b = 10 only will
work)
------------------
const b = 100;
b = 1000;
//this gives us TypeError: Assignment to constant variable.
```

- Till now 3 types of errors have been covered: Syntax, Reference, and Type.
- Uncaught ReferenceError: x is not defined at ...
    - This Error signifies that x has never been in the scope of the program. This literally means that x was never defined/declared and is being tried to be accesed.
- Uncaught ReferenceError: cannot access 'a' before initialization
    - This Error signifies that 'a' cannot be accessed because it is declared as 'let' and since it is not assigned a value, it is its Temporal Dead Zone. Thus, this error occurs.
- Uncaught SyntaxError: Identifier 'a' has already been declared
    - This Error signifies that we are redeclaring a variable that is 'let' declared. No execution will take place.

```
//code example 1.1
let a=10;
let a=100;
```

```
//code example 1.2
let a=10;
var a=100;
```

Will throw this Syntax error and no code will be run and be rejected affront. 'let' is a strict form of declaration and thus can be done only once.

- Uncaught SyntaxError: Missing initializer in const declaration
    - This Error signifies that we haven't initialized or assigned value to a const declaration.
- Uncaught TypeError: Assignment to constant variable
    - This Error signifies that we are reassigning to a const variable.

Type Error:

The Errors that occur due to conflicts with the declaration type. For example re-assigning const type declaration will throw this.

Syntax Error:

The Errors that occur due to wrong syntax that doesn't match with JS Engine syntactical rules.

For example, if const is not initialized, it will throw syntax error as by syntax, it must initialize if it sees a const declaration.

Also, if variable that is assigned with 'let' declaration is tried to re-declared, then it throws Syntax Error.

Reference Error

The Errors that occurs if no reference is available for access. Can occur when the variable is no where in scope or maybe it is in temporal dead zone.

SOME GOOD PRACTICES:

- Try using const wherever possible.
- If not, use let.
- Avoid var.
- Declare and initialize all variables with let to the top to avoid errors to shrink temporal dead zone window to zero.

PS: If in any interview when asked "Are let and const hoisted?" explain fully about temporal deadzone and all the above concepts too

Episode 9 : Block Scope and Shadowing

What is a block?

- Block aka *compound statement* is used to group JS statements together into 1 group. We group them within {...}
- The purpose is to group multiple statments at a place where JS expects only 1 statement.

```
//code example 1

if(true)some statement
```

But if we want to write more statements to execute after if condition; then:

```
//code example 2

if(true){
    statement 1
    statement 2
    ...
}
```

- The {} block treats all the statements as one statement.
- The if doesnt have any curly braces in syntax.

BLOCK SCOPE

- What are the variables and functions that can be accessed inside the block.

```
//code example 3

{
    var a = 10;
    let b = 20;
    const c = 30;
}

console.log(a);
console.log(b);
```

Outputs:

> 10

> Uncaught ReferenceError: b is not defined

- Behind the Scenes:

    - In the BLOCK SCOPE; we get b and c inside it initialized as *undefined* as a part of hoisting (in a seperate memory space called block)
    - While, a is stored inside a GLOBAL scope.

- Thus we say, *let* and *const* are BLOCK SCOPED. They are stored in a separate mem space which is reserved for this block. Also, they can't be accessed outside this block. But var a can be accessed anywhere as it is in global scope.

- Thus, we can't access them outside the Block.

What is SHADOWING in JS?

```
//code example 4

var a = 100;
{
    var a = 10; //same name as global var
    let b = 20;
    const c = 30;
    console.log(a); // 10
    console.log(b); // 20
    console.log(c); // 30
}

console.log(a); // 10 instead of the 100 we were expecting. So block "a"
modified val of global "a" as well.
// In console, only b and c are in block space. a initially is in global
space(a = 100), and when a = 10 line is run, a is not created in block
space, but replaces 100 with 10 in global space itself.
```

- If one has same named variable outside the block, the variable inside the block *shadows* the outside variable.
- So, a is reassigned to 10. Since both refers to same memory space i.e GLOBAL SPACE. **This happens only for var**

Instead of var let us use let

```
//code example 5

let b = 100;
{
    var a = 10;
    let b = 20;
    const c = 30;
    console.log(b);
}
```

```
console.log(b);
```

Outputs:

> 20

> 100 // this was what we were expecting in this first place. Both b's are in separate spaces (one in Block(20) and one in Script(another arbitrary mem space)(100))

- In the Scope, we have b in two places. The b outside of the block is in *Script* SCOPE (seperate memory space for let and const)
- The b inside the block is in *Block* scope.
- Thus, when in Block scope, 20 is printed and 100 when outside.
- This concept is called "Shadowing".
- It is also true for *const* declarations.

Same logic is true even for functions

```
const c = 100;
function x() {
  const c = 10;
  console.log(c);
}
x();
console.log(c);
```

Output:

> 10

> 100

What is Illegal Shadowing?

```
// code example 6

let a = 20;
{
    var a = 20;
}
```

Outputs:

> Uncaught SyntaxError: Identifier 'a' has already been declared

- We cannot shadow let with var. But it is valid to shadow a let using a let.
- However, we can shadow var with let.
- All scope rules that work in function are same in arrow functions too.
- Since var is function scoped, it is not a problem with the code below.

```
// code example 7

let a = 20;
function x() {
    var a = 20;
}
```

Episode 10 : Closures in JS

Important Interview Question

Closure : Function bundled together with its lexical environment/scope.

```
JS is a weird language. You can pass functions as parameters to another
function, assign a variable to an entire function, or even return a
function.
eg:

function x() {
  var a = 7;
  function y() {
  console.log(a);
}
  return y;   // instead of y();
}
var z = x();
console.log(z);  // value of z is entire code of function y.
```

When functions are returned from another fun, they still maintain their lexical scope.

- When y is returned, not only is the fun returned but the entire closure (fun y + its lexical scope) is returned and put inside z. So when z is used somewhere else in program, it still remembers var a inside x()

Uses of Closure

Module Design Pattern, Currying, Functions like once(fun that can be run only once), memoize, maintaining state in async world, setTimeout, iterators...

Episode 11 : setTimeout + Closures Interview Question

Time, tide and Javascript wait for none

```
function x() {
  var i = 1;
  setTimeout(function() {
  console.log(i);
  }, 3000);
  console.log("This is Hari");
 }
 x();
```

> This is Hari

> 1 //after waiting 3 seconds (3000ms)

We expect JS to wait 3 sec, print 1 and then go down and print the string. But JS prints string immediately, waits 3 sec and then prints 1.

- The fun inside setTimeout forms a closure (remembers reference to i). So wherever fun goes it carries this ref along with it.
- setTimeout takes this callback function & attaches timer of 3000ms and stores it. Goes to next line without waiting and prints string.
- After 3000ms runs out, JS takes function, puts it into call stack and runs it.

Print 1 after 1 sec, 2 after 2 sec till 5 : Tricky interview question

We assume this has a simple approach as below

```
function x() {
 for(var i = 1; i<=5; i++){
   setTimeout(function() {
     console.log(i);
    }, i*1000);
   }
   console.log("This is Hari");
 }
```

```
x();
```

> This is Hari
>
> 6
>
> 6
>
> 6
>
> 6
>
> 6

- This happens because of closures. When setTimeout stores the function somewhere and attaches timer to it, the fun remembers its reference to i, **not value of i**
- All 5 copies of fun point to same reference of i.
- JS stores these 5 functions, prints string and then comes back to the functions. By then the timer has run fully. And due to looping, the i value became 6. And when the callback fun runs the variable i = 6. So same 6 is printed in each log
- **To stop this from happening, use let instead of var** as let has black scope. For each iteration, the i is a new variable altogether(new copy of i).
- Everytime setTimeout is run, the inside fun forms closure with new variable i

Using let instead of var is the best option. But if asked to use var only..?

```
function x() {
 for(var i = 1; i<=5; i++){
   function close(i) {
    setTimeout(function() {
     console.log(i);
    }, i*1000);
    // put the setT fun inside new function close()
   }
   close(i); // everytime you call close(i) it creates new copy of i. Only
this time, it is with var itself!
  }
   console.log("This is Hari");
 }
 x();
```

Only the important new concepts

- Closures are used in encapsulation and data hiding.

```
(without closures)
var count = 0;

function increment(){
  count++;
}

in this code, anyone can access count and change it.

(with closures) -> put everything into a function

function counter() {
  var count = 0;

  function increment(){
    count++;
  }
}
console.log(count); // this will give referenceError as count can't be
accessed.

(inc with function using closure)

function counter() {
  var count = 0;
  return function increment(){
    count++;
    console.log(count);
  }
}
var counter1 = counter(); //counter fun has closure with count var.
counter1(); // increments counter

Above code is not good and scalable for say, when you plan to implement
decrement counter at a later stage.
To address this issue, we use constructors

Adding decrement counter and refactoring code:

function Counter() { //constructor function. Good coding would be to
capitalize first letter of ctor fun.
  var count = 0;
  this.incrementCounter = function(){ //anonymous function
    count++;
    console.log(count);
  }
   this.decrementCounter = function(){
    count--;
    console.log(count);
  }
```

```
}

var counter1 = new Counter();  // new keyword for ctor fun
counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();

// returns 1 2 1
```

Disadvantages of closure

- Overconsumption of memory when using closure as everytime as those closed over variables are not garbage collected till program expires. So when creating many closures, more memory is accumulated and this can create memory leaks if not handled.
- **Garbage collector** : Program in JS engine or browser that frees up unused memory. In highlevel languages like C++ or JAVA, garbage collection is left to the programmer, but in JS engine its done implicitly.

```
function a() {
  var x = 0;
  return function b() {
    console.log(x);
   }
 }

 var y = a(); // y is a copy of b()
 y();
```

Once a() is called, its element x should be garbage collected ideally. But fun b has closure over var x. So mem of x cannot be freed. Like this if more closures formed, it becomes an issue. To tacke this, JS engines like v8 and Chrome have smart garbage collection mechanisms. Say we have var x = 0, z = 10 inabove code. When console log happens, x is printed as 0 but z is removed automatically.

Episode 14 : First class and Anonymous functions

(there is no Episode 13 idk why lol)

Function statement : Just your normal function definition

```
function a() {
  console.log("a called");
}
a();
```

Function Expression : Assigning a function to a variable. Function acts like a value

```
var b = function() {
  console.log("b called");
}
b();
```

Difference btw function statement and expression is Hoisting

- You can put "a();" before "function a()" and it will still work. But putting "b();" before "var b = function()" throws a typeError.
- Why? During mem creation phase a is created in memory and function assigned to a. But b is created like a variable (b:undefined) and until code reaches the function() part, it is still undefined. So it cannot be called.

Function Declaration : Exactly same as function statement

Anonymous Function : A function without a name

- They don't have their own identity. So an anony function without code inside it results in an error.
- Anony functions are used when functions are used as values eg. the code sample for function expression above

Named Function Expression : Same as Function Expression but function has a name instead of being anonymous

```
var b = function xyz() {
  console.log("b called");
}
b(); // prints "b called" properly
xyz(); // Throws ReferenceError:xyz is not defined.
```

> xyz function is not created in global scope. So it can't be called.

Parameters vs Arguments

```
var b = function(param1, param2) {  // labels/identifiers that get the arg
values
  console.log("b called");
}
b(arg1, arg2); // arguments - values passed inside function call
```

First Class Function aka First Class Citizens

- You can pass functions inside a function as arguments(WOW!)

```
var b = function(param1) {
  console.log(param1); // prints " f() {} "
}
b(function(){

});

this can also be done:

var b = function(param1) {
  console.log(param1);
}
function xyz(){

}
b(xyz); // same thing as prev code

you can return a function from a function:

var b = function(param1) {
  return function() {

  }
}
console.log(b()); //we log the entire fun within b.
```

Arrow Functions (latest in ES6 (ECMAScript 2015) -> coming in future lecture

Episode 15 : Callbacks and Event Listeners

Callback Function : Functions are first class citizens (see prev lecture) ie. take a fun A and pass it to another fun B. Here, A is a callback function

- JS is a synchronous and singlethreaded language. But due to callbacks, we can do async things in JS.

> setTimeout(function () {}, 1000) -> here the anony function is a callback function as it is passed to setT and called sometime later in code after certain time (here 1000ms).

- This is how we do async things. JS is a synch language, but it doesn't wait 1 sec for setT to finish before going to code below it. It stores the function, attaches timer and goes down the code.

```
setTimeout(function () {
  console.log("timer");
 }, 5000);

function x(y) {
  console.log("x");
  y();
}

x(function y() {
  console.log("y");
});
```

> x

> y

> timer

- In the call stack, first x and y are present. After completion, they go away and stack is empty. Then after 5 seconds(from beginning) anonymous suddenly pops up in stack ie. setTimeout
- All 3 functions are executed through call stack. If any operation blocks the call stack, its called **blocking the main thread**
- Say if x() takes 30 sec to run, then JS has to wait for it to finish as it has only 1 call stack/1 main thread. *Never block main thread*.
- **Always use async for functions that take time eg. setTimeout**

Event Listener

- When we create a button in HTML and attack a clickListener in JS :

```
in index.html
```

```
<button id="clickMe">Click Me!</button>

in index.js

document.getElementById("clickMe").addEventListener("click", function xyz()
{ //when event click occurs, this callback function is called into
callstack
    console.log("Button clicked");
});
```

Suppose we want to increase count by 1 each time button clicked.

- Use global variable (not good as anyone can change it)

```
let count = 0;
document.getElementById("clickMe").addEventListener("click", function xyz()
{
   console.log("Button clicked", ++count);
});
```

- Use closures for data abstraction

```
function attachEventList() {  //creating new fun for closure
   let count = 0;
   document.getElementById("clickMe").addEventListener("click", function
xyz(){
   console.log("Button clicked", ++count);  //now callback fun forms
closure with outer scope(count)
});
}
attachEventList();
```

Garbage Collection and removeEventListeners

- Event listeners are heavy as they form closures. So even when call stack is empty, EventListener won't free up memory allocated to *count* as it doesn't know when it may need *count* again.
- **So we remove event listeners when we don't need them (garbage collected)**
- onClick, onHover, onScroll all in a page can slow it down heavily.

Episode 16 : Asynchronous JS and Event Loops

> Note that call stack will exeeute any exeeution context which enters it. Time, tide and JS waits for none. TLDR : Call stack has no timer

Browser has JS Engine which has Call Stack which has Global exec context, local exec context etc

- But browser has many other *superpowers* - Local storage space, Timer, place to enter URL, Bluetooth access, Geolocation access and so on
- Now JS needs some way to connect the callstack with all these superpowers. This is done using **Web APIs**

WebAPIs

None of the below are part of Javascript! These are extra superpowers that browser has. Browser gives access to JS callstack to use these powers.

> setTimeout(), DOM APIs, fetch(), localstorage, console (yes, even console.log is not JS!!), location and so many more..

- setTimeout() : Timer function
- DOM APIs : eg.Document.xxxx ; Used to access HTML <script>..... DOM tree. (Document Object Manipulation)
- fetch() : Used to make connection with external servers eg. Netflix servers etc.

We get all these inside call stack through *global object* ie. **window**

- Use window keyword like : window.setTimeout(), window.localstorage, window.console.log() to log something inside console.
- As window is global obj, and all the above functions are present in global object, we don't explicity write *window* but it is implied

Workflow

```
// First a GEC is created and put inside call stack.
console.log("Start"); // this calls the console web api (through window)
which in turn actually modifies values in console.

setTimeout(function cb() {  //this calls the setT web api which gives
access to timer feature. It stores the callback cb() and starts timer.
  console.log("Callback");
 }, 5000);

console.log("End"); // calls console api and logs in console window. After
this GEC pops from call stack.
```

- While all this is happening, the timer is constantly ticking. After it becomes 0, the callback cb() has to run.
- Now we need this cb to go into call stack. Only then will it be executed. For this we need **event loop and Callback queue**

Event Loops and Callback Queue

- cb() cannot simply directly go to callstack to be exeeceuted. It goes through the callback queue when timer expires.
- Event loop checks the callback queue, and if it has element puts it into call stack. It is a *gate keeper*.
- Now cb() in callstack is run. Console API is used and log printed

Final console output:

> Start

> End

> Callback

Same happens for any other event as well (Click, Hover etc). This is the basic workflow.

```
console.log("Start");
document. getElementById("btn").addEventListener("click", function cb() {
// cb() registered inside webapi environment and event(click) attached to
it. ie.
// REGISTERING CALLBACK AND ATTACHING EVENT TO IT.
  console.log("Callback");
});

console.log("End"); // calls console api and logs in console window. After
this GEC pops from call stack.
```

In above code, even after console prints "Start" and "End" and pops GEC out, **the eventListener stays in webapi env**(with hope that user may click it some day) until explicitly removed, or the browser is closed.

Why need callback queue?

- Why can't event loop directly take cb() and put it in callstack? Suppose user clciks button x6 times. So 6 cb() are put inside callback queue.

- Event loop sees if call stack is empty/has space and whether callback queue is not empty(6 elements here).
- Elements of callback queue popped off, put in callstack, executed and then popped off from call stack.

fetch() works a bit different than the rest

```
console.log("Start"); // this calls the console web api (through window)
which in turn actually modifies values in console.

setTimeout(function cbT() {
  console.log("CB Timeout");
 }, 5000);

 fetch("https://api.netflix.com").then(function cbF() {
  console.log("CB Netflix");
 });

console.log("End");
```

- Same steps for everything before fetch() in above code.
- fetch registers cbF into webapi environment along with existing cbT.
- cbT is waiting for 5000ms to end so that it can be put inside callback queue. cbF is waiting for data to be returned from Netflix servers.
- fetch requests data from Netflix servers, and get data back and now cbF ready to be executed. **We have something called a Microtask Queue**
- It is exactly same as Callback queue, but it has higher priority. Functions in Microtask Q are executed earlier than Callback Q.
- cbF goes inside the Microtask Q and not callback Q. Once call stack is empty, Event loop gives chance for elements in **both** Microtask Queue and Callback Queue to enter Call Stack.
- In console, first *Start* and *End* are printed in console. First cbF goes in callstack and "CB Netflix" is printed. cbF popped from callstack
- Next cbT is removed from callback Q, put in Call Stack, "CB Timeout" is printed, and cbT removed from callstack.

What enters the Microtask Queue ?

- All the callback functions that come through *promises* go in microtask Q.
- **Mutation Observer :** Keeps on checking whether there is mutation in DOM tree or not, and if there, then it exeeeutes some callback function.
- Callback functions that come through *promises and mutation observer* go inside Microtask Queue.
- All the rest goes inside **Callback Queue aka. Task Queue**
- If the task in microtask Queue keeps creating new tasks in the queue, element in callback queue never gets chance to be run. This is called **starvation**

Some Important Questions

1. **When does the event loop actually start ? -** Event loop, as the name suggests, is a single-thread, loop that is *almost infinite*. It's always running and doing its job.
2. **Are only asynchronous web api callbacks are registered in web api environment? -** YES, the synchronous callback functions like what we pass inside map, filter and reduce aren't registered in the Web API environment. It's just those async callback functions which go through all this.
3. **Does the web API environment stores only the callback function and pushes the same callback to queue/microtask queue? -** Yes, the callback functions are stored, and a reference is scheduled in the queues. Moreover, in the case of event listeners(for example click handlers), the original callbacks stay in the web API environment forever, that's why it's adviced to explicitly remove the listeners when not in use so that the garbage collector does its job.
4. **How does it matter if we delay for setTimeout would be 0ms. Then callback will move to queue without any wait ? -** No, there are trust issues with setTimeout() 😅. The callback function needs to wait until the Call Stack is empty. So the 0 ms callback might have to wait for 100ms also if the stack is busy.

Episode 17 : JS Engine and Google v8 architecture

JS runs literally everywhere from smart watch to robots to browsers because of Javascript Runtime Environment (JRE)

- JRE consists of a JS Engine (❤️ of JRE), set of APIs to connect with outside environment, event loop, Callback queue, Microtask queue etc.
- JRE is a container that can run JS code.

- ECMAScript is a governing body of JS. It has set of rules followed by all JS engines like Chakra(Edge), Spidermonkey(Firefox), v8(Chrome)
- JS Engine is **not a machine**. Its software written in low level languages (eg. C++) that takes in hi-level code in JS and spits out low level machine code

In all languages, code is compiled either with **interpreter** or with **compiler**. JS used to have only interpreter in old times, but now has **both** to compile JS code.

Interpreter : Takes code and executes line by line. Has no idea what will happen in next line. Very fast. Compiler : Code is compiled and an optimized version of same code is formed, and then executed. More efficient

- Code inside JSE passes through 3 steps : **Parsing, Compilation and Execution**

1. **Parsing** - Code is broken down into tokens. In "let a = 7" -> let, a, =, 7 are all tokens. Also we have a **syntax parser** that takes code and converts it into an **AST (Abstract Syntax Tree)** which is a JSON with all key values like type, start, end, body etc (looks like package.json but for a line of code in JS. Kinda unimportant)(Check out astexplorer.net -> converts line of code into AST)
2. **Compilation** - JS has something called **Just-in-time(JIT) Compilation - uses both interpreter & compiler**. Also compilation and execution both go hand in hand. The AST from previous step goes to interpreter which converts hi-level code to byte code and moves to exeception. While interpreting, compiler also works hand in hand to compile and form optimized code during runtime.
3. **Execution** - Needs 2 components ie. Memory heap(place where all memory is stored) and Call Stack(same call stack from prev episodes). There is also a *garbage collector.* It uses an algo called **Mark and Sweep**.

Companies use different JS engines and each try to make theirs the best.

- v8 of Google has Interpreter called *Ignition*, a compiler called *Turbo Fan* and garbage collector called *Orinoco*

Episode 17 : Trust Issues with setTimeout()

(Episode number mixup. This is the actual 17th episode)

```
console.log("Start");

setTimeout(function cb() {
  console.log("Callback");
```

```
 }, 5000);

 console.log("End");

setTimeout sometimes does not exactly guarantee that the callback method
will be called exactly after 5s.
Maybe 6,7 or even 10! It all depends on callstack
```

While execution of program

- First GEC is created and pushed in callstack.
- Start is printed in console
- When setT is seen, callback method is registered into webapi's env. And timer is attached to it and started. cb waits for its turn to be exeeceuted once timer expires. But JS waits for none. Goes to next line
- End is printed in console.
- After "End" suppose we have 1 million lines of code that runs for 10 sec(say). So GEC won't pop out of stack. It runs all the code for 10 sec.
- But in the background, the timer runs for 5s. While callstack runs the 1M line of code, this timer has already expired and cb fun has been pushed to Callback queue.
- Event loop keeps checking if callstack is empty or not. But here GEC is still in stack so cb can't be popped from callback Q and pushed to CallStack. ** Though setT is only for 5s, it waits for 10s until callstack is empty before it can execute**(When GEC popped after 10sec, cb() is pushed into call stack and **immediately executed** (Whatever is pushed to callstack is executed instantly)
- This is called as the **Concurrency model of JS**. This is the logic behind setT's trust issues

The First rule of JavaScript

- **Do not block the main thread** (as JS is a single threaded(only 1 callstack) language)
- This raises a question. *Why not add more call stacks and make it multithreaded?*
- JS is a synch single threaded language. And thats its beauty. With just 1 thread it runs all pieces of code there. It becomes kind of an interpreter lang, and runs code very fast inside browser (no need to wait for code to be compiled) (JIT - Just in time compilation). And there are still ways to do async operations as well.

Now what if the timeout = 0sec

```
console.log("Start");

setTimeout(function cb() {
```

```
 console.log("Callback");
}, 0);

console.log("End");
```

- Even though timer = 0s, the cb() has to go through the queue. Registers calback in webapi's env , moves to callback queue, and execute once callstack is empty

> Start

> End

> Callback

- This method of putting timer = 0, can be used to defer a less imp fun by a little so the more important fun (here printing "End") can take place

---