Advanced JS

Lecture 1

→ Javascript execute one command at a time.

→ Memory component contain variable function as a key value pairs

→ Code component where whole Javascript code is executed.

Lecture 2 (How code is executed)

→ First memory execution face created and then code execution face created.

→ In case of function it store whole code inside function in memory execution.

→ Functions are heart of Javascript and behave differently than any other language.

→ When you run a function or invoke a function, a brand new execution context is created.

→ When we executed square(n) we concern about piece of code inside function square(num){ }

→ "return keyword" state that return the control of the program to the place where it was invoked.
→ Call stack is a stack everytime in the bottom of this we have global execution context.
→ Call stat is only for managing execution context.
→ whenever execution context is created it will pushed into the stack and whenever execution context is deleted it will popped out of the stack.
→ After all code is executed GEC is removed from call stack.

Lecture 3: (Hoisting in Javascript)

→ invoke function before creating
→ Hoisting is a phenomenon in Javascript by which you can access these variables and functions even before you have initialized it. you can access it without any error.
→ even before the code executed memory is allocated to each of function and variable.

→ We have not reserved memory for x. then it is not defined.

→ In case of arrow functions it's says getHome is not a function becaus it's behave like just another variable. This time getHome behave just like another variable.

Lecture 4: (How functions work in JS):

→ When same variable x in whole program.

→ execution context is independent of everything.

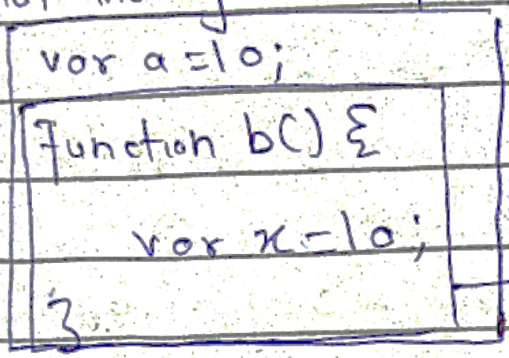
Lecture 5: (Window and this keyword).

→ empty file is a shortest Javascript program.

→ Debugger is in source.

→ Window (big object with lot of functions and variables)

→ This points to the window object
→ Window is an global object which is created along with the global execution context.
→ Always global object created.
→ this === window
true.
→ What the global space



→ Anything outside function is called global space.
console.log(x); → ~~undefined~~ not defined
↳ global space.

Lecture 6 (undefined vs not defined):
→ Javascript is a loosely type language.
→ Javascript is a weekly type language.

Lecture 7 (Scope chain)

→ scope chain is directly related to its lexical environment.

```
function a(){
  Console.log(b);
}
  var b=10;
  a();
```

→ Javascript try to find out whether b exist in its local memory space or not

→ Where & can access this variable

→ b is called scope.

→ scope is directly dependent on lexical environment.

→ Whenever an execution context is created lexical environment is also created.

→ lexical environment is a local memory along with lexical enviroment to its parent.

→ c function is lexically inside a function. and a is lexicaly inside global space.

→ If is not find b then it goes to lexical environment of its parent
→ Null mean no more environment to search for - so we get that error b is not defined.

Lecture 8 (Leet and const) + Temporal Dead Zone
→ let & const declarations are hosted but these are in temporal deadzone for the time being.
→ consol e.log(a); output:
let a = 10; connot access a
var b = 100; before initiolizing.

→ In case of var it is in global space but in case of let we see some script.

→ Temporal dead zone since when this let variable was hosted until it is initallized some variable.

```
console.log(a);
let a=10;
```

that is when temporal dead zone ends and value a is assighed

→ from hoisting to it inilize some value, that is called temporal dead zone.

→ separate space for let and const.

→ const is more strict then let.

→ Type error (const type)

```
const b=1000;
b=10000;
```

→ syntax error `const b;` → missing syntax

→ Refereece: When Javascript engine to find out inside memory space

the non is known.

you cannot access it then it give us reference error.

→ Always put initialization on the top of code → Avoid temporal dead zone

→ we shrink temporal dead zone movement to zero.

Lecture 9 (Block SCOPE AND SHADOWING).

→ {

}

→ This is a block.

→ Block is also known as compound statement.

→ what all variables and function we can access inside this block is called block.

→ If we have some named variable outside the block then this variable shadowing that variable.

→ illegal shadowing

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

→ lexical work some inside the block.

→ Scope for normal and arrow function same.

Lecture 10 (closures)

→ A function bind together its lexical environment

→ A closure is a combination of a function bounded together (enclosed) with references to its surrounding environment (the lexical environment).

→ A closure gives you access to another function scope from its inner function.

```
function x() {
    var a = 7;
    function y() {
        console.log(a);
    }
    return y;   → function returned along with its lexical environment
}
```

FANG

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

Lecture 11 (Settimeout + closures)

→ Javascript do not wait for anything

→ let creates new copy everytime.

Lecture 12 (First class functions)

→ Function statement

→ Function expression:

→ The main difference b/w function statement and expression is hoisting.

→ A function without name is called anonymous function.

→ Giving a named to a function is called named function.

→ We can pass function inside another function as an argument

→ We return function from a function.

→ The abity of function to used as value is called first class function and it is in other languages also.

1) → used as values
2) → can be passed as argument
3) can be executed inside a closured function
4) can be fetch as return.

Lecture 13 (Callback Functions)

→ You can fate a function and pass it into another function. This function is known as callback function.

→ It give us access to the whole asynchronous world using a synchronous single threaded language.

→ function x(y){

}

x(function y(){ → is called callback function.

})

→ If any of the operating blocking the main thread is called blocking the main thread.

→ store somewhere and automatically come into call stack.

→ callback function forms a closure with count and this and it remember where this count is present.

Lecture 15 (Event loop)

→ Set Timeout is not a part of Javascript.

→ even console.log is not a part of Javascript.

→ fetch() make connection to other servers.

→ window is global object.

→ window.setTimeout()

→ window.console.log();

→ Event loop has only one job continuosly monitor the callback and call stack queue.

→ Why we need call back queue.
For example user pushes button five to eight times click, click...

→ Browsers do lot of things behind the screne.

→ All be callback functions which come through promises will go outside the of Microstack Queue

→ All other callback functions goes inside callback Queue

→ Callback Queue == Task Queue.

Lecture 16 (JS engine Artitecture).

→ Every Browser has Javascript run-time environment

→ e.g water cooler.