Global, Block, function scope

Strict mode removes a lot of problems.

Scope sets boundaries for the variables.

Scope Chain.

var, let, const

When we declare a variable using const, we must assign a value to it.

Variable declaration is statement (perform some actions in the memory).

Variable assignment is expression.

undefined is system value. It is assigned automatically.

**impure** function:

when we change a variable value that is declared outside that function.

Pure function: Only change local variables.

**Block scope**: Variables declared with ‘**let**’ and ‘**const**’ inside of the block are limited to the scope of this block.

Will be covered in section on Variable lifecycles.

function myFn () {

a = true;

console.log(a); // **true**

}

myFn ();

console.log(a); // **true**

Variable ‘a’ is looked for in the global scope. But there is no variable a in the global scope.

So variable ‘a’ will be automatically declared in the Global scope.

This is how javascript handles undeclared variables.

This behaviour can lead to problems and is not desired.

Using strict mode, we can block such behaviour.

“**use strict**”: String literal

If we add a semicolon ; then this becomes an expression statement.

“**use strict**”;

This allows all the browsers that don’t support ES5 features to simply ignore this line.

This will be processed by the engine and gone.

New browsers that understand what this line means will execute js code below in the strict mode.

It is possible to execute a specific function using strict mode by using “use strict”; inside the function.

In strict mode,

Also, we cannot delete any variable using ‘delete’ operator.

**Section 12: Advanced Topics**:

**Mutable vs Immutable**:

String is immutable.

let c = “abc”

c[1] = “d”

c won’t change. Cannot change the individual character but can make ‘c’ point to a new string.

In javascript, primitive types such as Number, String or boolean are immutable.

This means that we cannot change value of primitive value type.

But we can reassign value to the variable.

let b =30;

b = 50; // can do this.

New memory location is created if we reassign the reference.

These values 30 and 50 were not changed. 30 was deleted. We have replaced 30 by 50.

We are not able to mutate primitive value types (memory locations).

mutate (change in form or nature)

There is single reference type in Javascript which is **Object**.

Arrays are objects and have **Object** value type.

We can mutate/change the values in Object type by using dot notation even if the reference variable is a const.

**Objects in Javascript are mutable**.

We can add or delete any property of an object using dot notation.

With reference type in JS, variable type holds pointer.

The reference variable still point to the same location. The content of that location can be changed.

Similarly, with array.

Array is also mutable.

We can delete a property of an object using ‘delete’ operator.

We say mutable or immutable **values** and not variables.

Variables cannot be mutable or immutable.

“**typeof**” and “**instanceof**” operators

These are used in expressions. Each expression produces a value.

null is a primitive value type.

typeof null gives “object”: due to nature of object. object can point to nowhere, that’s why we get ‘null’.

typeof undefined: “undefined”

typeof returns a string.

typeof a === “number” (small n)

typeof has higher precedence than ===, so typeof is evaluated first.

To check specifically for Array, we cannot use “object” with typeof.

“object” is returned for some object, array and null, so we cannot be sure about it.

We can use instanceof operator for this.

If we print an array variable, then we can see first Array comes and then Object comes on expanding.

const c = [ ]

c instanceof Array: true

c instanceof Object: true

We can see the **prototype chain**.

This information is used by **instanceof** operator.

In prototype chain of Object, there is Object only.

const a = “abc”;

a instanceof String: false

String is a primitive and it does not have a prototype chain.

That is why instanceof gives false for primitive value types.

There is another prototype: null which we will see later.

instanceof expects a reference type and returns boolean.

Array, String, Object, Number, Boolean: these are function constructors. These are used to create new instances of these types using **new** keyword.

These are global function variables and can be used with instanceof operator.

Number instanceof Object: true

**typeof Number: function**

Object instanceof Object: true

Number: used with instanceof

number: used with typeof

(small n)

“number”

With arrays, use instanceof operator.

window.Array, Array is one of the properties of global ‘window’ object.

If we print window, then on expanding we can find this Array property.

**Lecture 102**:

In node.js, Array will be the property of the global variable. It is called a function constructor because it is used to create new Arrays.

const a = [ ] ; (Under the hood, Array function constructor is called)

const b = new Array ();

Both of these expressions are equivalent.

Both the arrays will have same structure.

const a = [1, 2, 3]

const b = new Array(1, 2, 3)

const string1 = “abc”

const string2 = new String(“def”);

Printing string1 gives “abc”

Printing string2 gives String {“def”}

Both of them differ.

These 2 syntaxes are not equivalent.

string2 is instanceof String and Object and its type is “object”.

string1 type is “string”. It is not an instance of String or Object.

We can call some functions on string2.

The same functions can be called using string1. We won’t get any error.

string1 holds primitive values. There are no methods.

But if we try methods of String prototype on string1,

string1 is converted to String object under the hood. This happens only when we call any method of the String prototype.

string1 still holds primitive value.

Conversion is hidden and temporary. It is done only for the purpose of function calls.

string primitive and string object behave similarly but look different.

Primitive strings and number are generally created using literal notation.

Array instanceof Function: true

**Primitive**:

null, undefined, boolean, number, string, null

**Reference types**: Array, Object, Function

Boolean, Number, String are reference types for boolean, number and string. These add properties to its own primitives.

“function” type

Compound type or “object”

Similarly, with number and Number.

let a =1;

a.valueOf () gives 1.

“a”.length is evaluated as new String(“a”).length by the interpreter.

Instances of Object and Function.

const myObject = new Object({a: 10, b: true})

const c = function (a) {

console.log(a);

}

c instanceof Function: true

c is instance of both Object and Function as can be seen from prototype chain.

const b = new Function ("a", "console.log(a);");

b("hello")

console.log (b instanceof Function, b instanceof Object, typeof b)

// true, true, “function”

**IMP**:

This version may be useful when we get function contents from an external server and we need to create a function on the fly.

This syntax helps to achieve that.

It is an anonymous function. If we do console.dir (b), we can see that.

**Lecture 104**:

**Execution Contexts**:

Global execution context and function execution context.

Stack of execution context.

When a javascript file is executed in the Javascript engine, a new execution context is created.

There is only 1 global execution context and it is created as soon as Javascript engine starts execution of the javascript file.

If we have just 1 file and we want to console.log something, then in our case global execution context will be created.

In this context, we can write some code.

Let’s create a function.

function a1 (a, b) {

return a+b;

}

console.log (a1(2,4)); // line alpha

function a1 is declared in the global execution context and will be a property of “window” object.

When Javascript engine reaches the line alpha, function a1() is called.

When a function is called, a new execution context is created.

This context is called function execution context.

Function execution context is created for each particular function at the time of that function call.

This new function execution context is created at top of the global execution context.

Code inside the function is executed in this function execution context.

When the function returns its result, its execution context is automatically deleted. And we get back to the global execution context.

The code after the function call is executed in the global execution context.

When all the code of the file is execution, the global execution will be automatically deleted.

Now, suppose we have 2 function calls. Then the function execution context will be created 2 times.

Function execution context is created as many times as the function is called. It would be a new execution context each time.

Next execution context is created on the top of the previous one.

Stack of execution contexts.

**Lecture 105**:

**this variable**:

In the global execution context “this” is equal to the window object.

‘this’ is different in different execution contexts.

this.console.log(“Hello”) // prints “Hello”

console is a property of window object.

function a () {

console.log(this);

}

a ();

window.a();

‘this’ inside the function ‘a’ is equal to the object on which the function a () is called.

a () is one of the functions of window object.

If we call a () standalone, then also ‘this’ is the window object.

In strict mode however, these won’t be equivalent.

“**use strict**”; // By putting ; we convert this expression into a statement.

In strict mode, ‘this’ is not passed automatically when we call a function standalone.

a() will give undefined.

In html code,

<button onclick= “console.**log**(this);”> My Button </button>

On clicking the button, this prints the whole button tag.

<button onclick= “console.**dir**(this);”> My Button </button>

On clicking the button, this prints an object with many properties.

<button onclick= “console.**dir**(this.textContent);”> My Button </button>

Clicking the button will print the text “My Button”

Every time we click a button, a new execution context is created for the onclick event handler.

Can check properties of a tag by logging it to the console.

<input onchange= “console.log(this.value);”/>

Whenever the input is changed, a new execution context is created for the function call.

const myObject = {

name: “abc”,

age: 20,

greeting: function(){

console.log(this)

}

}

Is it possible to call ‘greeting’ function defined inside myObject from another context different from ‘myObject’ ? Yes

**Lecture 106**:

**bind, call, apply methods of the Function**:

These methods are available for each Function object.

const myObject = {

a: 10,

b: null

};

function myFunction () {

console.log(this)

}

myFunction () // “this” is window (not strict mode)

myFunction.call (myObject)

// ‘this’ is the object myObject.

We can invoke any function with **custom this**.

The call() function is called immediately with no delay.

Now let’s pass arguments to the call() function.

const myObject = {

a: 10,

b: null

};

function myFunction (a, b) {

console.log(a+b);

console.log(this)

}

myFunction.call (myObject, 10, 3)

// 13 and the myObject will be printed.

const person1 = {

city:"New York",

name:"Bob",

info: function() {

console.log(this.name+" lives in " + this.city);

}

};

person1.info();

const person2 = {

city="Paris",

name: "Alice"

}

Suppose we want to borrow info () function of person1 for person2.

person1.info.call (person2);

info is not located in the global scope. It is a property of person1 object.

In apply method, array is passed.

From previous example,

myFunction.apply (myObject, [10, 3]);

We pass arguments as an array rather than just comma separated values.

First argument is custom ‘this’.

call() and apply() are called immediately.

In both call() and apply(), first argument is custom “this”.

We can iterate over key-value pairs of an object using for-in loop.

**“bind” method**:

With bind we can call the function later on.

Before the call, we can assign custom ‘this’ and arguments to the function.

const c = myFunction.bind (myObject, 10, 3);

the bind expression creates a brand new function.

myFunction.bind (myObject, 10, 3); does not call the function.

To call it later,

we need to assign the result of bind () to some variable.

c() // this executes the myFunction() function with ‘this’ as myObject.

We can also do this:

const c2 = myFunction.bind(myObject); // preset only custom this.

c2(12, 2);

Using bind(), we can either preset all arguments or just certain arguments.

Atleast pass the myObject in bind, that is ‘this’ reference.

Can also do this:

var a = myFunction.bind()

a = myFunction.bind(myObject, 1 , 2)

**Lecture 108**:

**Pass by Value vs Pass by Reference**:

Primitive values are immutable.

Object values are mutable.

// **Pass by Value**.

function myFunction(a) {

a=20;

return a;

}

const a = 10;

console.log(myFunction(a)); // 20

console.log(a); // 10

If variable that holds value of primitive value type is passed to the function as argument, it’s value cannot be changed inside of the function.

// When 10 is passed to myFunction(), a separate variable ‘a’ is created in the function execution context.

// **Pass by Reference**.

const myArray = [1, 2, 3];

console.log(myArray); // [1, 2, 3]

function myFunction (arr) {

arr.push(4);

console.log(arr); // [1, 2, 3, 4]

return arr;

}

myFunction(myArray);

console.log(myArray) // [1, 2, 3, 4]

arr inside the function and myArray point to the same memory location.

**VIMP**:

function addCustomGlobalMethod (globalObject) {

globalObject.customMethod = function() {

console.log ("Hello World");

}

}

addCustomGlobalMethod(this);

‘this’ is equal to ‘window’ as we are in global execution context.

this.customMethod (); // Hello World

customMethod (); // Hello World

window.customMethod(); // Hello World

It’s like we are creating a new function property for window object.

We made a local variable as global using this.

This is pass by reference and that’s why it impacts the window object.

window object will now have a new property called as customMethod which is a function.

true for “object” type.

**Lecture 109**:

**IIFE**: Immediately Invoked Function Expression

Adding custom method to the global ‘window’ object without polluting global scope, without creating any new function such as addCustomGlobalMethod().

In previous example, we created the function addCustomGlobalMethod() in the global scope.

Hence, it is a property of window object.

This leads to creation of 2 methods for the global window object.

Using IIFE, we can just add 1 function.

(function addCustomGlobalMethod (globalObject) {

globalObject.customMethod = function () {

console.log ("Hello World");

}

}) (this);

this.customMethod (); // Hello World

customMethod (); // Hello World

For window object, addCustomGlobalMethod won’t be created for the global window object.

**IIFE** does not add any variable to the scope in which it is defined.

That is, it does not pollute scope in which it is created and called.

We don’t need any name for the function. This function can be simply anonymous.

We can turn this function declaration into a function expression.

(function (globalObject) {/// Function expression

globalObject.customMethod = function () {

console.log ("Hello World");

}

})(this);

this.customMethod (); // Hello World

customMethod (); // Hello World

Function expression cannot be used standalone.

Function expression must be either assigned to a variable or passed as a callback function or be immediately invoked.

We can have space between ‘function’ keyword and opening parentheses.

//Option 2

(function () {/// Function expression

this.customMethod = function() {

console.log ("Hello World");

}

})();

this.customMethod (); // Hello World

customMethod (); // Hello World

// Using Arrow function

( () => {/// Function expression

this.customMethod = function () {

console.log ("Hello World");

})();

this.customMethod (); // Hello World

customMethod (); // Hello World

We cannot use function expressions standalone.

IIFEs are no longer popular. We use Modules.

**Synchronous Code Execution**:

Javascript code is executed synchronously.

It is a single threaded language.

console.dir (Date)

// We can see the expandable object.

function waiting (timeInMs) {

const futureTime = Date.now () + timeInMs;

while (futureTime > Date.now ()) {

// waiting ....

}

}

waiting (5000);

console.log ("Function call just ended");

// This code has to wait for 5 seconds. JS code is executed synchronously.

/\*

JS is a single threaded language.

We can only get back to the global execution when the function execution context is **popped** out from the stack.

\*/

Stack of execution contexts.

**Events and Event Queue**:

Suppose we create a button and print a message when we click it.

This message will only be printed after the waiting() function in the index.js file has finished.

First "Function call just ended" will be printed after 5 seconds.

And then, “button clicked” will be printed in the console.

When there is something in the stack of execution contexts, events are waiting and they wait in the events queue.

When the function waiting() is running for 5 seconds, we are not able to process events at that time.

When the stack becomes empty that is global execution context is removed from the stack, only then we are able to process any events that are waiting in the queue.

If we again click the button, then this event will be processed without any delay.

This is because now the execution context stack is empty.

Events can only be processed when the stack of execution contexts is empty.

JS is single threaded language. Events can only be processed when stack of execution contexts is empty.

**Callback Functions**:

Callback is a function that is passed as an argument in the call to another function.

Let’s see what happens if we will simultaneously use callbacks, events in the queue and long running running functions.

setTimeout() is a global method.

setTimeout ( ()=> console.log ("Callback is executed" ) , 1000 ) ;

console.log("last statement in the Global execution context")

Call to setTimeout is non blocking and we don’t wait 1 second.

Separate execution context is created for setTimeout() function call.

The callback function is placed in separate place in the memory called as web APIs.

The web browser knows that setTimeout() function should be called in 1 second.

Call to setTimeout function ends and its function execution context is deleted from the stack and we get back to the global execution context and in the global execution context we get back to the console.log() statement.

Separate function execution context is created for the purpose of the log() function call.

When the log() method is executed, we delete global execution context from the stack.

The callback function is there in the web API place in the memory and the web browser counts 1 second.

**IMP NOTE:**

Callback function is placed in the event queue.

When this 1 second passes by, this callback function is placed in the event queue.

Events in the event queue are processed only when the execution stack is empty and the moment it becomes empty, the function call event will be processed without any delay.

To execute this callback function, a new execution context will be created in the stack.

On top of this callback function execution context, another execution context will be created for the log() function call inside setTimeout() function.

Inside this log() function execution context, “Callback is executed” message will be printed to the console.

Execution context of log() will be popped out from the stack and execution context of callback function will be popped out as well and stack will become empty again.

When timeout is 0, nothing will change in terms of the output.

If we use a delay smaller than the specified minimum for a browser, then the browser will ignore the delay and use the minimum.

Here the callback function will be placed in the event queue as soon as the execution context of setTimeout() is deleted from the stack.

Events in the event queue are processed only when the execution stack is empty.

function waiting(timeInMs) {

const futureTime = Date.now() + timeInMs ;

while (futureTime > Date.now() ) {

// waiting...

}

}

waiting(5000); // first wait for 5 seconds.

setTimeout( ()=> console.log("Callback is executed" ) , 1000 ) ;

console.log("Global")

If we swap the call to waiting() and setTimeout, then the output will remain same.

Now the output of setTimeout will be executed immediately after log() as 2 seconds have already passed.

Total execution time is now 5 seconds instead of 7 seconds as 2 seconds are counted by the browser in parallel and now waits in the event queue.

function waiting(timeInMs) {

const futureTime = Date.now() + timeInMs ;

while (futureTime > Date.now() ) {

// waiting...

}

}

setTimeout( ()=> console.log("Callback is executed" ) , 1000 ) ;

waiting(5000);

console.log("Global")

Suppose we also click on the button.

Callback function is added to the event queue first but the code of button click is executed first.

Events in the event queue have priority.

Events bound to UI items such as button, or are part of the visible part of the web page have high priority.

Users don’t like to wait.

Short response time is part of good UI experience. That’s why web browser prioritizes those events in the queue.

setTimeout() is not part of the visible part of the UI.

Callbacks are stored in web APIs and then they are placed in event queue.

webAPI: Events related to API are stored there.

Through APIs, we request data from servers. Such requests require some amount of time to complete.

Web browser waits for response in the Web API area. When response from the remote server is received, corresponding event that will process this response is placed into the events queue.

Suppose we swap setTimeout() and waiting() calls.

This won’t make difference on the output.

**Closures**:

Closure is created when a specific function returns another function.

function outerFn() {

 const a = 10;

function innerFn(b) {

    return a + b;

}

return innerFn

}

const result  = outerFn() ;

console.log(result(1)) ;

outerFn is created in the global execution context.

When we call result(1), there is no variable ‘a’ in innerFn() and we look for it in outer scope.

Outer scope for innerFn() is outerFn() function.

After outerFn() call, its execution context was deleted.

How can innerFn() still access the value of variable ‘a’?

Scope Chain

In the scope chain of innerFn we can see that there is a closure which is outerFn function.

innerFn will have access to all the variables that were created in the execution context of outerFn function.

If we add a parameter to outerFn, then this will also be accessible by innerFn and will be in the closure of innerFn.

function outerFn(mult) {

 const a = 10;

function innerFn(b) {

    return (a + b)\*mult;

}

return innerFn

}

const result1  = outerFn(2) ;

console.log(result1(1)) ;

Variables ‘a’ and ‘mult’ will be present in the closure of result1.

**Section 13: ES6 Variable Lifecycles**

// Scope

// Reassignment

// Redeclaration

// Hoisting

With ‘var’, the variable is either created in global or function scope.

If we define a new variable with same name inside the function scope, then it won’t affect the variable in the outer scope.

**Hoisting with ‘var’:**

e=4;

var e;

console.log(e) // 4

Javascript before code execution has found all variable declarations and puts them at the beginning of the code.

This is called **Hoisting**.

Root of the javascript file: Global scope.

Global scope, function scope and block scope.

Block scope:

{

var c;

}

Variable created in the block scope using ‘**var**’

is created in the block scope.

It will be available in the outer scope in case of ‘var’.

Other examples of block scope:

if-else, for loop.

**Lecture 118**:

Using **let**:

{

let b1 =1;

}

If we try to access b1 on the console, we will get a not defined error.

This behaviour is different from var.

Difference comes in the block scope.

Behaviour of both let and var is same inside a function.

for ( let i=0; i <5; i++ ) {

    console.log(i) ; // 0, 1, 2, 3, 4

}

console.log(i) ; // not defined error.

for ( var i=0; i <5; i++ ) {

    console.log(i) ; // 0, 1, 2, 3, 4

}

console.log(i) ; // 5

We can redeclare a variable ‘a’ using var again.

But cannot do this using ‘let’ again.

We cannot redeclare a variable using let **in the same scope**.

var a = 3 ; var a=4 ; console.log(a); // 4

With ‘let’ hoisting does not work. That is we should declare variable with ‘let’ before its usage.

‘let’ is limited to global, function and block scope.

‘var’ is limited to global and function scope.

const: variable that cannot be reassigned.

Must assign a value to const declaration.

If we have a const array or object then we can add or delete elements but we cannot reassign it.

We can mutate it but cannot reassign it.

const e;

e = 5;

This gives an error of Missing initializer in const declaration.

Hoisting does not work with let and const.

**Lecture 120**:

**Variables Usage Guidelines**:

* Readable Code
* “use strict”
* Declare variables before first usage.
* Don’t expose to the outer scopes locally used variables. All variables should be as hidden as possible.

Without strict mode, the following code works:

b=10;

function fn() {

d = b;

return d;

}

fn();

Both d and b will be created in global scope.

‘d’ is created in the global scope automatically if do not specify anything such as ‘var’, ‘let’, etc.

“use strict”; is a directive.

‘**let**’: For temporal variables (in a block scope). Signal to the other developers that this variable will be gone and its safe to redeclare it outside the previous block scope.

const sum = function (a, b) { // **anonymous function**

return a+b;

}

var a = function() {

{

i=2 ;

}

}

a()

console.log(i) // 2

variable i will be defined in the global scope.

var a = function() {

{

var i=2 ;

}

}

a()

console.log(i) // Error ‘i’ is not defined

var a = function() {

{

{

{

{

{

i=2 // defined without using ‘var’, etc.

}

console.log(i) // 2

}

console.log(i) // 2

}

console.log(i) // 2

}

console.log(i) // 2

}

console.log(i) // 2

}

a()

console.log(i) // 2

var a = function() {

{

var i=2 ;

}

console.log(i) // 2

}

a()

console.log(i) // error

{

var a = 4;

}

console.log(a) // 4

In this example, variable ‘i’ defined in block is only accessible to its outer scope and not to the outer scope of this outer scope.

In the first case, it is accessible to the function scope (outer scope) and not to the global scope.

In the second case,

Variable ‘a’ is accessible in the global scope (outer scope).

function fn2(){

myArray = [1, 2, 3]

for(i=0; i<myArray.length; i++){

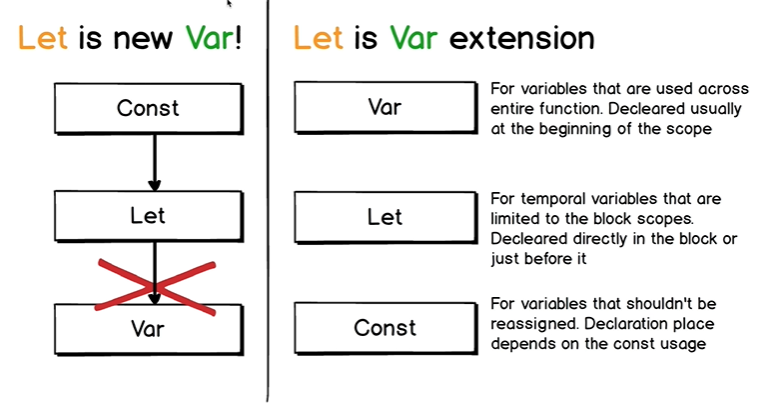
console.log(myArray[i]);

}

}

We have access to the variable ‘i’ in the global scope.

**Lecture 121**: var, let or const ?



**Case 1**: Most of the time const and in some cases ‘let’

2nd point of view is better.

‘var’, ‘let’ and ‘const’ act as a signal to the developer.

‘const’ is not accessible outside block and function scope.

Understanding concept of hoisting and scope rules is important.

Use const when we want to prevent re-assignment.

Example: const path = require(“path”);

Function scope: Use var

Block scope: Use let

Keep the rules in mind.

Using ‘var’ could be a signal that the variable can be changed several times in the function.

If we have “use strict”, then for a variable we will have to use ‘var’, ‘let’ or ‘const’.

Else we will get an error of ‘variable not defined’.

When we assign a function to a variable, then we should use ‘const’ as we won’t be re-assigning this variable.

We use direct signals that tell others about what happens inside a function and outside of it.

Callback functions are placed in Web Apis.

const menuItems = document.querySelectorAll(".nav-link");

const LEN = menuItems.length; // 4

for(var i=0 ; i<LEN; i++){

menuItems[i].onclick = function(){

for(let j=0 ; j < LEN ; j++){

menuItems[j].classList.remove("active");

}

console.log(i); // 4

menuItems[i].classList.add("active");

console.log(menuItems[i].innerHTML);

}

}

**IMP NOTE**:

The loop has been executed before the click the html items. When we click the items, then ‘i’ is 4 and this same ‘i’ is shared among all 4 callback functions.

// If we use let i=0 instead of var,

// then we get correct expected values of i.

With var, we have same variable 'i' for all 4 callback functions.

Hence when the outer loop finishes, i becomes 4 and this value is also changed for the

callback functions.

If we want to use var i=0, then we have to take help of another variable,

for example, inside the outer for loop, we should have const current =i and use “current”

instead of i.

Important case of var and let with closure and for-loop.

**Lecture 125**:

With **let**, a brand new variable is created for each iteration because it is **limited to the block of statements**.

With var, variable remains the same.

‘var’ is not limited to the block scope and that’s why there will be single variable in the scope where loop itself sits.

With let, ‘i’ will be placed separately in the memory for each iteration of each function.

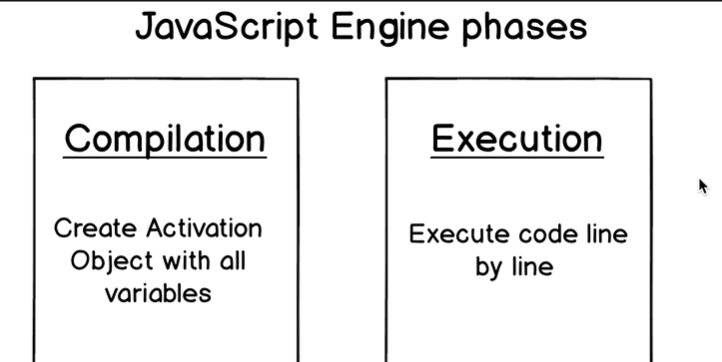
The variable will be closed inside the function for later execution.

This is called Closure.

With var, we have single variable for all onclick event handlers and it would have last value of i in the loop.

If we want to use ‘**var i**’, then we would need to use another variable, ‘const’ and assign it equal to i.

**Javascript Engine Phases**:



Before Javascript code is executed, it is compiled.

Javascript engine looks through the entire code that we supply to it. It finds alll variable declarations that are present in our code.

During compilation, the javascript engine looks through all the block, function and global scopes.

It creates special **activation objects** with all variables and then engine starts execution of the code.

In execution phase, code is executed line by line.

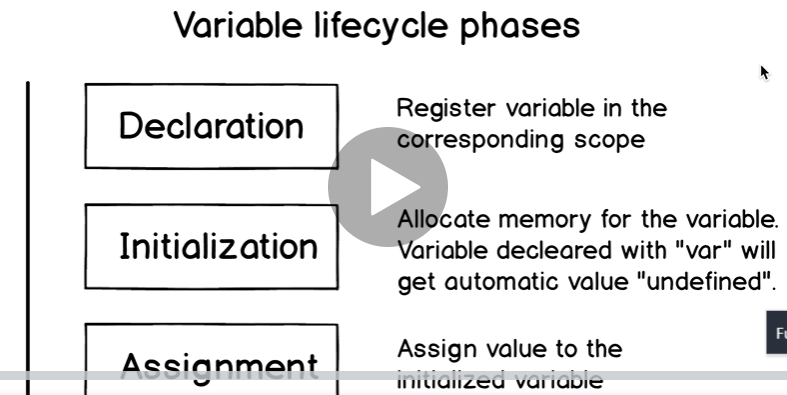
During execution, all the variables are already declared in the internal memory of the javascript engine. And it knows about scopes for each variable.

Javascript engine runs code in 2 phases:

Compilation

Execution

**Variables Lifecycle Phases**:



var a;

a=23;

In javascript code there is declaration and assignment.

In javascript engine there are three different phases in a variable lifecycle.

They are Declaration, Initialization and Assignment.

This declaration: **“var a”** ; and the Declaration phase in javascript engine are different.

“var a” declaration is just used to declare variables in the Javascript file and nothing else.

In this phase, variable is registered in the corresponding scope.

Initialization phase: Allocate memory for the variable. Variables declared with “var” will get automatic value “undefined”.

Assignment: Assign value to the initialized variable.

function fn(a) {

console.log(a);

}

fn(); // undefined

Here the variable ‘a’ has been declared and initialized by javascript engine.

But we have not used ‘var’, ‘let’ or ‘const’ keywords. This is just a parameter inside a function.

Here the variable ‘a’ is declared inside function scope and is initialized with “undefined” value during compilation phase.

It is not necessary to call a Javascript function by passing all the parameters.

**Here, In the compilation phase, the function parameter behaves as a “var”.**

function(){

console.log(a)

}

// a is not defined error

function(){

a

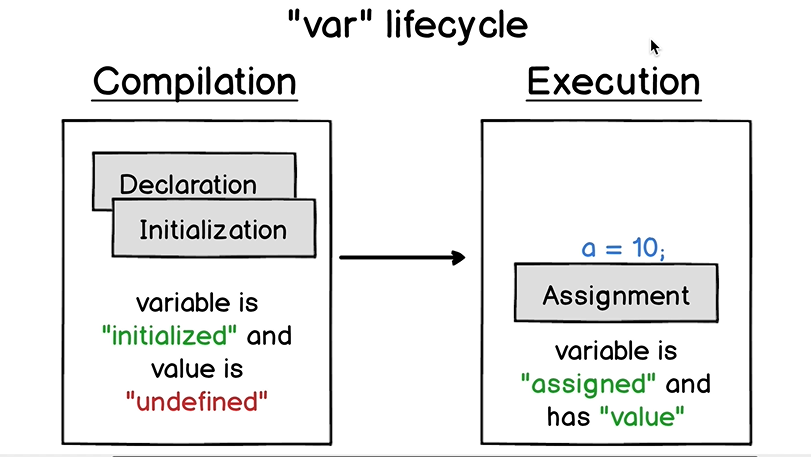
console.log(a)

}

// a is not defined error

After declaration phase, variable is said to be declared. After initialization phase, variable is said to be initialized and after assignment phase, variable is said to be assigned.

**var lifecycle**:



Declaration and Initialization of the ‘var’ occurs in the compilation phase.

Assignment occurs in the execution phase.

When javascript engine registers the variable in the scope, immediately after this, initialization takes place and the variable is initialized.

With ‘var’ hoisting takes place.

When Javascript engine performs compilation phase, it scans this code and finds all variable declarations.

And when it finds the line ‘var a’, then declaration and initialization phases are performed on the javascript engine and the variable gets a value of undefined.

console.log(a) // undefined

var a;

a=3;

console.log(a); // 3

var a ;

**During compilation phase** a is given the value undefined. During execution phase, it is set to 3.

console.log(b); // undefined

var b;

function fn () {

console.log(a); // undefined

var a;

}

During compilation phase, ‘a’ is bound to function scope and gets the value of undefined.

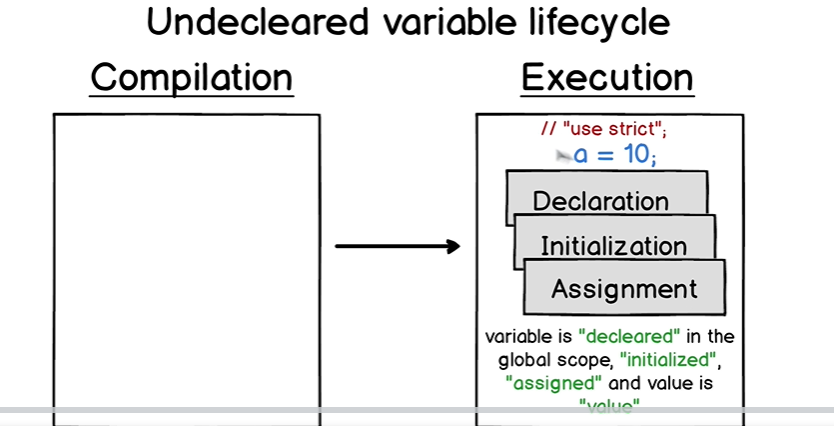
**During execution phase**, javascript engine already has the variable declarations.

So it doesn’t matter if we declare a variable after its assignment.

What will happen if we don’t declare variable inside the function scope but have an assignment for it?

It will be created globally.

Undeclared variable lifecycle:



With undeclared variable, all three phases of Declaration, Initialization and Assignment are bundled together. So we cannot write a variable just as ‘a’. We must assign a value to it, like a = 23. Else we will get an error.

**NOTE**:

Undeclared variables can appear in code if we don’t use “use strict” option.

If we use the strict mode then we will get an error if any variable is not defined.

If we don’t use strict mode, then nothing happens in the Compilation phase.

When javascript engine reads our code, it will do nothing during the compilation phase.

When the code is executed and the engine reads the assignment, the variable will be immediately declared, initialized and assigned. All three phases together.

Variable is declared in the **global scope**.

function fn() {

function fn2() {

console.log(a);

a = 3 ;

}

fn2();

}

fn();

// Uncaught ReferenceError: a is not defined

If we have,

console.log(a) ; a=4 then also we get

“a is not defined” error.

Hoisting does not take place with undeclared variables.

When we do console.log(a) before a=4, then Javascript engine does not find any declaration for variable ‘a’. That is why variable ‘a’ is not declared and not initialized.

If we do console.log(a) below a=3 or below it in any of the outer scope, then it works perfectly fine.

This is because javascript engine reads the line a=3 and immediately declares, initializes and assigns the value to variable ‘a’.

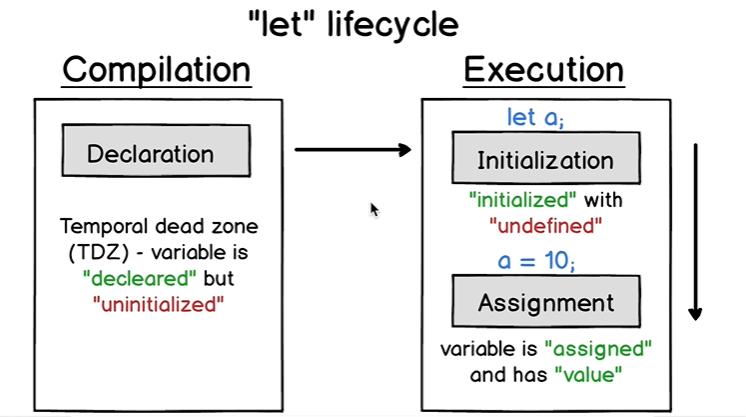
The engine tries to find the declaration of variable ‘a’ in the current scope. If it does not find it, then it looks for it in the outer scope and finally reaches the global scope.

If the variable declaration is not in the global scope, then it is automatically declared, initialized and assigned **in the global scope**.

(For case a=10; console.log(a))

This is what happens when we don’t use strict mode.

**let lifecycle**: **Lecture 130**



Why we cannot use variable declared with ‘let’ before its actual declaration in the code?

With ‘let’, hoisting does not happen.

Declaration phase occurs during Compilation.

Initialization and Assigment occur during Execution.

When the engine finds a ‘let’ declaration, it declares that variable in the corresponding scope.

Variable becomes declared but uninitialized.

Time between start of code and this line of code: **“let a”**, is called **temporal dead zone**.

Temporal dead zone: Variable is declared but uninitialized (not stored in the memory).

That’s why we cannot use the let variable before its actual declaration in the code as we hit the temporal dead zone.

console.log(a) ; // “a is not defined” error

let a

When the engine hits the line ‘let a’, the variable a is initialized with ‘undefined’ and before this line, it is in temporal dead zone.

let a;

console.log(a) // undefined

a=10 // ‘a’ is assigned

console.log(a) // 10

Main difference with variable lifecycle between ‘let’ and ‘var’ comes because of Hoisting.

**One Important Difference**:

With ‘let’, variable ‘a’ does not become the property of global window object.

With ‘var’, variable ‘a’ becomes property of global window object.

If we write, a=3 then variable ‘a’ is declared in the global scope using ‘var’.

**IMP**: Interesting case

a=3

let a

console.log(a)

// ReferenceError: Cannot access ‘a’ before initialization

(If we try to access let variable before its declaration in the code, then we hit the temporal dead zone.)

**Hoisting does not happen with let**.

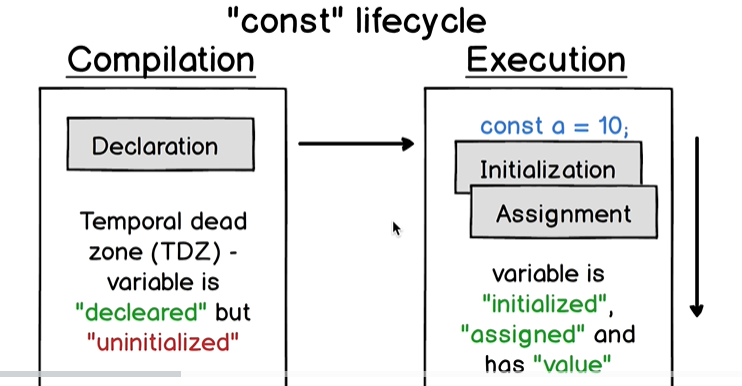
a=3

let a

a=2

console.log(a) // same error as above

**const lifecycle**: Lecture 131



Temporal dead zone is a special interval between start of the code execution and till declaration of the variable using **let or const**.

(Not var)

Accessing variable in the Temporal Dead zone (TDZ) will lead to the following error:

ReferenceError: Variable is not defined.

Access means assigning value to the variable or simply console.log().

This happens because the variable is declared in the compilation phase but is not yet initialized in the execution phase.

**With let**, initialization and assignment phases may be separate.

But with const, initialization and assignment are bundled together.

This is because, with const, we have to assign the value at the time of declaration.

Before execution phase, the variable is located in temporal dead zone.

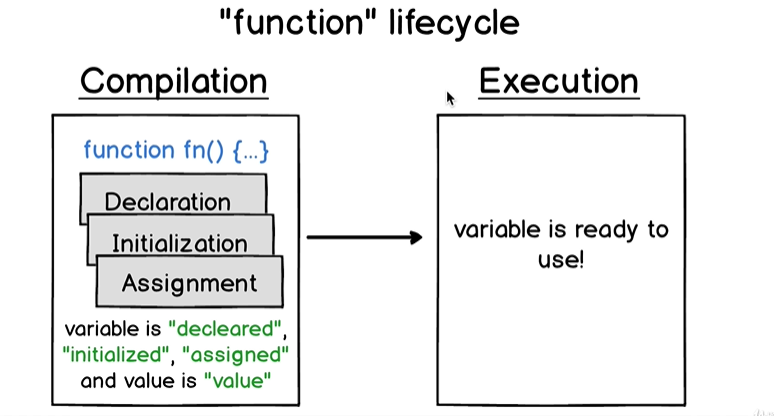
a=3;

const a;

// Error

With undeclared variables, all three stages are bundled together, so just writing ‘a’ leads to an error.

**Function Life Cycle**:



Function is a variable. It is declared, initialized and assigned in the compilation phase. All these 3 phases are bundled together.

function fn() {

…

}

When Javascript engine finds function declaration somewhere in the code, it immediately performs declaration, intialization and assignment.

In the execution phase, the variable is ready to use.

Value of the function is the list of statements located in the body of the function.

Function declaration consists of name of the function and its contents.

We can also call the function before the declaration of the function. This is because all 3 phases happen in Compilation phase.

a() // “Function a” ; **Function declaration**

function a () {

console.log("Function a")

}

a() // “Function a”

To see expandable function, use console.**dir**()

Inside of the javascript engine, function declaration moves at the top of the current scope.

Function expression: this function has no name.

function(){

console.log(“Function expression”)

}

// a() // gives error.

{

function a()

{

console.log("a")

}

a() // prints a

}

In case of **IIFE**, nothing happens from those 3 phases. The function is simply executed and gone.

Javascript engine simply goes through it and does not perform any actions.

Assign function expression to a variable.

Here it is anonymous function expression.

const sum = function(a, b) {

return a + b;

}

console.log(sum(5, 10));

Here the steps similar to steps in lifecycle of ‘const’ take place.

Now we cannot do this:

sum(12, 11); // sum is not defined.

const sum = function(a, b) {

return a + b;

}

‘sum’ is accessed during the temporal dead zone.

This is because we used const to declare the function.

And const has temporal dead zone.

Suppose we use var,

sum(12, 11); // sum is not a function

var sum = function(a, b) {

return a + b;

};

Here ‘sum’ is undefined when we try call sum() before declaring it in the code.

This is the difference between function expressions and function declarations.

With var, declaration and initialization take place in the compilation phase.

And var has undefined value before execution. So “sum” is undefined at that point.

That is why we get an error of sum is not a function.

This is **Function Expression**.

**Lecture 134**:

**Hoisting** = Compilation + Execution

Hoisting describes process of compilation and execution phases.

Different variables behave differently in terms of hoisting because of different behaviour in compilation and execution phases.

let and const are only declared in the compilation phase. (Temporal dead zone comes into play).

var is declared and initialized in the compilation phase.

Functions are also hoisted.

**Lecture 135** is amazing.

Before doing the execution phase, the Javascript engine checks whether the compilation was successful or not.

**IMP EXAMPLE**:

let a = 1

let a = 1

// Identifier ‘a’ has already been declared. **Why this gives error** ?

var a= 1

var a=1

// No Error

Reason is the way lifecycle phases work and hoisting.

**Line by line** code execution.

let and const hoist but we cannot access them before the actual declaration is evaluated at runtime because that phase is of temporal dead zone.

**In the compilation phase, the function parameter behaves as a “var”.**

function mult(a, b){

result = a\*b

return result

var result

}

During compilation phase, this behaves as

mult{

var a; // declared and initialized with “undefined”

var b;

}

**VIMP EXAMPLE**:

const sum = function(a, b){

return a + b ;

}

In compilation phase, sum is declared.

In execution phase, sum will be initialized and assigned.

Value of the “sum” is “function(a,b){ return a+b }”

sum {

parameter a and b will be declared and initialized in the function scope with “undefined”.  
}

Anonymous function expression.

Scope of the anonymous function is assigned to the const “sum”.

When we call mult() function with both parameters, in the execution phase, these variables become assigned.

First Javascript Engine looks for the variable in the scope.

Compilation phase occurs before execution phase.

Compilation Phase: Create Activation Object with all variables.

Execution Phase: Execute code line by line.

TDZ: Declared but not initialized.

Function is also hoisted along with ‘var’.

**Section 14: ES6 Arrow Functions**

**Lecture 143**:

// Function declaration

function fn(a, b) {

return a+b;

}

During compilation phase, declaration, initialization and assignment occurs for this function.

// Anonymous function expression

Anonymous function expression cannot be used standalone. It must always be used in the context, either assigned to another variable or used as a callback function.

// does not have a name.

function(a, b) { // Anonymous function.

return a+b

}

const fn1 = function(a, b){

return a+b;

}

Assigning anonymous function to a variable.

We can also execute this anonymous function as an IIFE.

**With IIFE, an expression is converted to a statement**.

IIFE:

( function(a, b){

return a+b;

} )(2, 3)

// Callback function

setTimeout( function(a, b) {

return a+b;

}, 1000 )

**NOTE**:

The following is not function declaration. It is assigned to another variable.

const fn1 = function sum(a, b) {

return a+b;

}

We can use this anonymous function as a callback function.

If we try to print sum, then it will “sum” not defined error.

The function “sum” is not declared. It is assigned to another variable.

No variable with name of “sum” is ever created. The value of “fn1” is the function.

This is named function. We can also give a name to our IIFE.

Similarly, can also give a name to our callback.

Using above named functions or named callback function, we can give signal to other developers.

setTimeout ( function sum(a, b) {

return a+b;

} , 1000);

The function is created just for the purpose of setTimeout() call. Javascript engine will delete “sum” function after setTimeout() function call finishes.

This is the behaviour of a named function when it is used as an argument.

**What is the purpose of using named function instead of anonymous function expression** ?

If we give a name to the function, we give a signal to other developers about the purpose of the callback function.

Without the name, it is not fully clear. We have to go through the code to understand what the function is doing and what it returns.

Function declaration and function expression.

**Lecture 144**:

Arrow functions are anonymous.

There are no arrow function declarations because arrow functions don’t have any name.

There are only arrow function expressions and all of them are anonymous.

We cannot use named arrow functions as we did in previous examples.

We can assign arrow functions to a variable. We can use them as IIFE and can use them as a callback.

Anonymous arrow functions expressions *have different variations of syntax*.

// anonymous function expression

function(a, b) {

return a+b ; // this is a statement.

}

// anonymous arrow function expression.

(a,b) => a + b

(return is implicit here)

Here the function body has only expression, so we can omit the return keyword.

// In programming, expressions don't require a ;

// Expressions are separated using , and statements are separated using ;

Javascript does not require semicolons.

We cannot use anonymous function expression or anonymous arrow function expression standalone without any context.

Return is **implicit** in arrow functions. The value is returned automatically.

(a,b) => a + b is equivalent to

(expression)

(a, b) => {

return a+b // Here it is statement

}

If the arrow function has only 1 parameter then we don’t need to use parentheses, ().

a => a\*a

When arrow functions don’t have any parameters, then we do this:

() => 2

This function implicitly returns 2.

To return an object from an arrow function,

(a, b) => {

return {

a1: a,

b1: b

}

}

To return an object without using explicit return keyword, we should add parentheses ().

(a, b) => ({

a1:a,

b1:b

})

If we add parentheses around an object literal, then Javascript engine will not treat it as a function body but it will correctly recognize it as an object literal.

Syntax of arrow functions is much shorter than the syntax of traditional functions.

If arrow function has multiple statements, then we should embed them in { }, same as in traditional functions.

**Lecture 146**:

const post = {

title: "Sample title",

comments: 10,

shared: true,

published: true,

postId: 5134

}

const processedPost = (post) => {

return {

title: post.title,

comments: post.comments,

popular : post.comments > 5 ? true : false

}

} // can also return the object implicitly using ()

console.log(processedPost(post))

If we want to return an object implicitly,

const processedPost = (post) => ({

title: post.title,

comments: post.comments,

popular : post.comments > 5 ? true : false

})

We can remove the parentheses around post since we are only passing a single parameter.

**Arrow function as IIFE**:

(() => {

function greet() {

return "Hello"

}

this.greet = greet ;

return this ;

})()

Immediately invoked anonymous arrow function expression.

The above function adds greet() to window object and returns the window object. “this” is the window object in the arrow function.

Suppose we use an anonymous function inside the IIFE instead of a named function.

(() => {

this.greet = function greet(){

return “Hello”

}

//console.log(this);

return this;

}

(() => {

this.greet = () => “Hello”

// console.log(this)

return this;

})()

// this will also add ‘greet’ as the property of window object.

Most often, the arrow functions are used in the array helper methods.

Such as map, reduce, forEach, etc.

Main difference between traditional and arrow functions is with behaviour of “this” keyword.

Arrow functions don’t have own “this”.

“**this**” in arrow functions is always statically defined by the surrounding or outer lexical scope.

**IMP Example**: Traditional function

const num = {

value: 100,

info: function info() {

console.log(this) // num object

return this.value ; // 100

}

};

console.log(num.info());

// this, in this case refers to the object.

const num2 = {

value: 100,

info: () => { // using arrow function.

console.log(this) // window

return this.value ; // undefined

}

};

console.log(num2.info())

Arrow function does not have its own this. And that is why “**this**” refers to the outer scope.

And outer scope for num2 is window.

If we want to refer to properties of the object using “this”, we must not use arrow functions.

const num = {

value: 100,

info: function info() {

console.log(this)

return this.value ;

}

};

const newNum = {

value:17

}

We can use call() method to pass newNum as the “this” object to info() method of num object.

num.info.call (newNum) // 17

If we use an arrow function instead,

const num = {

value: 100,

info: () => {

console.log(this) // Window

return this.value ; // undefined

}

};

num.info.call (newNum) // undefined

This means that the arrow function does not have its own “this”. That is why it takes “this” from the surrounding scope which here it is the ‘window’ object.

const str = {

value: "Delayed greeting",

greet: function greet() {

setTimeout( function() {

console.log(this.value); // undefined

}, 1000 );

}

}

str.greet() ; // undefined

The inner callback function passed to setTimeout() has lost the ‘str’ object.

Inside the callback function of setTimeout() function, “this” refers to the window object.

There are several ways to use “this” for the object str.

One way is to use another variable.

const str2 = {

value: "Delayed greeting",

greet: function greet() {

const self = this;

setTimeout( function() {

console.log(self.value);

}, 1000 );

}

}

str.greet(); // “Delayed greeting”

Outside the setTimeout(), “**this**” points to the str2 object.

But in the callback function, “this” reference is lost and “this” points to the window object.

Also we can use bind() function to bind ‘this’ to the str2 object.

If we do this:

greet: function greet() {

setTimeout( function() {

console.log(this); // str

console.log(this.value); // “Delayed greeting”

}.**bind** (this), 1000 );

}

**IMP**:

Simple call to setTimeout function is equivalent to the window.setTimeout because setTimeout is one of the global methods.

That is why inside of the setTimeout function call, ‘this’ will be window.

Now let’s use arrow functions.

greet: function greet() {

setTimeout( () => {

console.log(this); // str

console.log(this.value); // “Delayed greeting”

}, 1000 );

}

str.greet();

Using arrow function, we achieved the same result that we got by using bind and an extra variable.

This happens because arrow functions don’t have own “this”.

‘this’ is taken from the surrounding scope.

Surrounding scope for arrow function is “str” object.

**Lecture 148**:

Function constructor

Common convention to start the function constructor names with a capital letter.

function GroceryItem (title, kind) {

this.title = title ;

this.kind = kind;

// console.log(this)

}

const apple = new GroceryItem ("Apple", "fruit");

// prints GroceryItem object.

Function constructor returns ‘this’ which referes to the function scope of GrocerytItem function. And we added ‘title’ and ‘kind’ to ‘this’ reference in the function.

function b (title, kind) {

this.title = title ;

this.kind = kind;

console.log(this)

}

b(1,2 )

// prints ‘window’ object. Here we do not create an object.

Function constructor creates a new object.

Now lets use arrow function instead of the function constructor.

const GroceryItem = (title, kind) => {

this.title = title;

this.kind = kind;

}

const apple = new GroceryItem ("Apple", "fruit") ;

This gives an error that GroceryItem is not a constructor.

It is not possible to use an arrow function expression as a function constructor and create new objects from it.

We must always use traditional functions.

function GroceryItem (title, kind) {

this.title = title;

this.kind = kind;

}

GroceryItem.prototype.info = function() {

return this.title + " is " + this.kind;

}

This method will be shared among all objects created using ‘GroceryItem’ function constructor.

const apple = new GroceryItem ("Apple", "fruit") ;

2nd piece of code adds a function info() as part of GroceryItem object.

This info() method can be accessed by all references of GroceryItem object.

Cannot do this by using an arrow function.

Inside the arrow function, “this” will point to the Window object.

Closure: variable enclosed and retained in the memory of the function.

**Lecture 151**:

Another difference between traditional and arrow functions is in handling arguments.

function sum() {

console.log(arguments);

}

sum(1, 2, 3);

// prints an object called "**Arguments**" and it is array

// of elements that are passed to the sum() function.

We can convert this ‘arguments’ object to array and get those elements as array.

function sum() {

console.log(arguments);

const argumentsArray = **Array**.**from** (arguments);

console.log(argumentsArray);

// this prints array of 3 elements.

}

“arguments” is a special variable.

sum(1, 2, 3);

The sum() function does not have any parameters. **We can call sum() function** using any number of arguments.

Using an arrow function will give not defined error.

const sum2 = () => {

….

Uncaught ReferenceError: arguments is not defined

Arrow function expression does not have access to “arguments” variable.

By using ES6 **REST operator,** we can access “arguments” variable inside of the arrow function.

const sum2 = (**…arguments**) => {

console.log(arguments);

const argumentsArray = **Array**.**from** (arguments);

}

Using traditional function, “arguments” is not an array. It is an Object.

Array.isArray(arguments) gives false.

With the arrow function, arguments is an array.

Array.isArray(arguments) gives true.

Arrow functions are

* Anonymous
* Expressions
* Don’t have own “this”

We cannot arrow functions in the function constructors. We are also not able to use them in the methods of the prototype.