1. The this keyword allows you to reuse functions with different contexts. Said differently, **the “this” keyword allows you to decide which object should be focal when invoking a function or a method.**
2. Every function in JavaScript has a prototype property that references an object.

Scope chain

Generator

Modules

var VS let VS const

If you declare a variable withour ‘var’, let or const keyword, then JS engine is going to hoist it to top. Since does not have any keyword, JS engine will take it to the Top i.e., global scope.

That is why it will get attached with global scope.

Therefore it is recommended to never use a variable without var/let/const.

var:

function scoped

undefined when accessing a variable before it's declared

can be reassigned

let:

block scoped

ReferenceError when accessing a variable before it's declared

can be reassigned

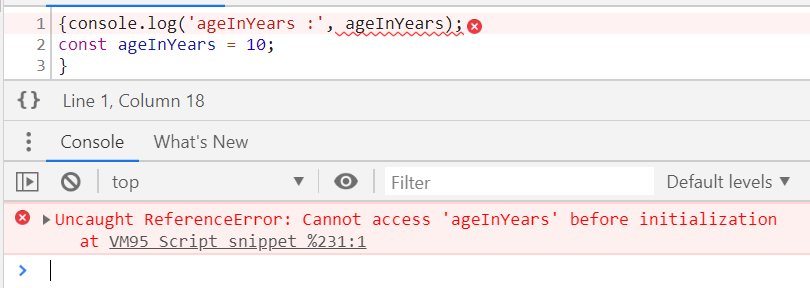
const:

block scoped [anything inside { } ]

ReferenceError when accessing a variable before it's declared

can't be reassigned

If we try to access a variable before it is declared with const keyword.



Execution Context, Scoping and Hoisting

Just like functions, modules or packages allows you to manage the complexity of writing code, Execution context allows JavaScript Engine to manage the complexity of interpreting the code.

Global Execution context: It is the first execution context gets created when JS engine runs your code.

Rules

#1. Function declarations are hoisted before any statement is hoisted or executed by JS Engine, which means if we write:

function getName() { return ‘Hennry’; }

then this function name with function body is going to be hoisted.

#2. Variable declarations are hoisted, after function declaration, which means if you write

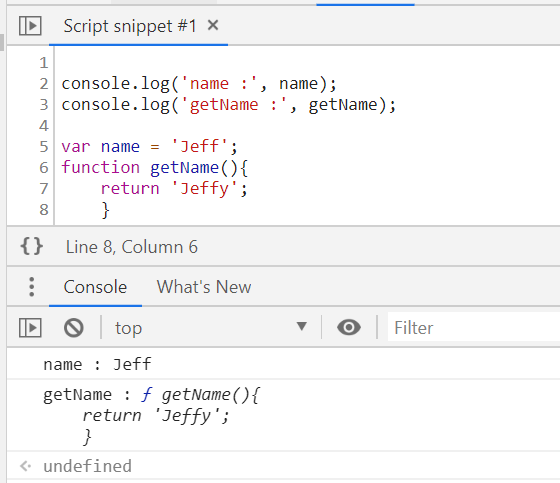
var a = 10;

Then, it is going to be interpreted as below in lexing phase,

var a; // Value of ‘a’ is Undefined

Below is interpreted during execution phase, when the interpreter reaches that particular line of code.

a =10; // Value of ‘a’ is 10



Scope Chain

It is a chain of nested scopes. It is created when we define a function or block of code inside another function or block of code.

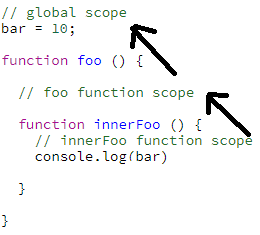
To Determine the Scope Chain, we need to look at the code first.

At top: there is global scope, which is window object in the browse.

Then there are functions which have their own scopes.

# Resolving Scope:

When resolving a variable, inner functions first look at their own scope. If the variable cannot be found in its own scope it will **climb up the scope chain** and looks for the variable name in the environment where the function was defined. This look like this:



Slice()

The **slice()** method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included) where begin and end represent the index of items in that array. It returns a new array containing the extracted elements. The original array will not be modified.

var animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));

// expected output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4));

// expected output: Array ["camel", "duck"]

console.log(animals.slice(1, 5));

// expected output: Array ["bison", "camel", "duck", "elephant"]

[] vs new Arrar()

### Array Methods

We talked in depth above about how if you want to share methods across instances of a class, you should stick those methods on the class’ (or function’s) prototype. We can see this same pattern demonstrated if we look at the Array class. Historically you’ve probably created your arrays like this

const friends = []

Turns out that’s just sugar over creating a new instance of the Array class.

const friendsWithSugar = []

const friendsWithoutSugar = new Array()

One thing you might have never thought about is how does every instance of an array have all of those built in methods (splice, slice, pop, etc)?

Well as you now know, it’s because those methods live on Array.prototype and when you create a new instance of Array, you use the new keyword which sets up that delegation to Array.prototype on failed lookups.

We can see all the array’s methods by simply logging Array.prototype.

console.log(Array.prototype)

/\*

concat: ƒn concat()

constructor: ƒn Array()

copyWithin: ƒn copyWithin()

entries: ƒn entries()

every: ƒn every()

fill: ƒn fill()

filter: ƒn filter()

find: ƒn find()

findIndex: ƒn findIndex()

forEach: ƒn forEach()

includes: ƒn includes()

indexOf: ƒn indexOf()

join: ƒn join()

keys: ƒn keys()

lastIndexOf: ƒn lastIndexOf()

length: 0n

map: ƒn map()

pop: ƒn pop()

push: ƒn push()

reduce: ƒn reduce()

reduceRight: ƒn reduceRight()

reverse: ƒn reverse()

shift: ƒn shift()

slice: ƒn slice()

some: ƒn some()

sort: ƒn sort()

splice: ƒn splice()

toLocaleString: ƒn toLocaleString()

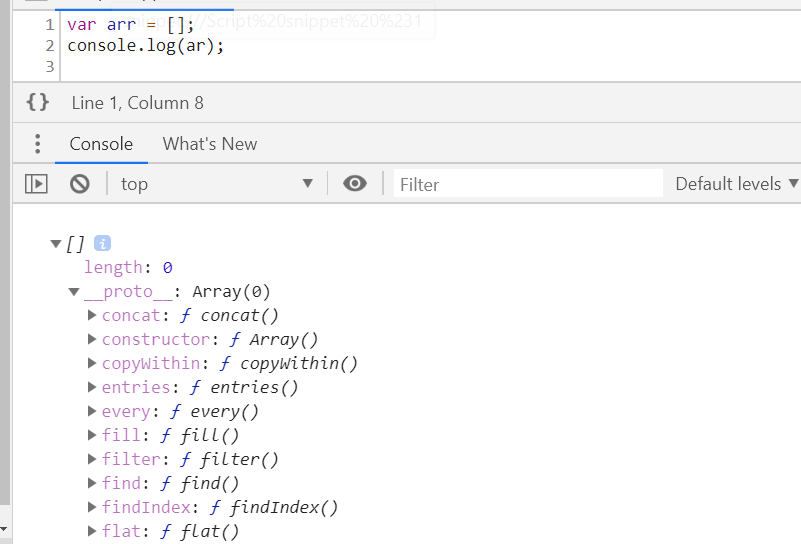
toString: ƒn toString()

unshift: ƒn unshift()

values: ƒn values()

\*/

The exact same logic exists for Objects as well. Alls object will delegate to Object.prototype on failed lookups which is why all objects have methods like toString and hasOwnProperty.



Sorting an Array

The sort() method sorts an array alphabetically:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();        // Sorts the elements of fruits

## **Reversing an Array**

The reverse() method reverses the elements in an array.

You can use it to sort an array in descending order:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();        // First sort the elements of fruits  
fruits.reverse();     // Then reverse the order of the elements

Numeric Sort

By default, the sort() function sorts values as **strings**.

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

To fix this, we use Callback function passed as argument to sort() function.

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a});

If the result is negative a is sorted before b.

If the result is positive b is sorted before a.

If the result is 0 no changes are done with the sort order of the two values.

**Example:**

The compare function compares all the values in the array, two values at a time (a, b).

When comparing 40 and 100, the sort() method calls the compare function(40, 100).

The function calculates 40 - 100 (a - b), and since the result is negative (-60),  the sort function will sort 40 as a value lower than 100.

Array Methods:

1. Push()

method adds a new element to an array (at the end)

Returns the new array length

1. Pop()

This method removes the last array element

Returns the value that was "popped out"

1. Shift() :

this method removes the first array element

Returns the string that was "shifted out"

1. Unshift() :

add an element at beginning of the array

Returns the new array length.

1. Splice()

method can be used to remove / add new items to an array:

Reurns an array with the deleted items

The first parameter: position where new elements should be **added** (spliced in).

The second parameter: **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);        // Removes the first element of fruits

1. Delete

delete fruits[0]; // Changes the first element in fruits to **undefined**

Slice()

It creates a new array. It does not remove any elements from the source array.

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1, 3); // Orange,Lemon

var lastThree = fruits.slice(1); // Orange,Lemon,Apple,Mango

1. Concat()

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments.

Argument Object

The arguments object is a local variable available within all **non-**[**arrow**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) functions. You can refer to a function's arguments inside that function by using its arguments object. It has entries for each argument the function was called with, with the first entry's index at 0.

For example, if a function is passed 3 arguments, you can access them as follows:

arguments[0] // first argument

arguments[1] // second argument

arguments[2] // third argument

Each argument can also be set or reassigned:

arguments[1] = 'new value';

The arguments object is not an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). It is similar, but does not have any Array properties except [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length). For example, it does not have the [pop()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop) method. However, it can be converted to a real Array:

var args = Array.prototype.slice.call(arguments);

// Using an array literal is shorter than above but allocates an empty array

var args = [].slice.call(arguments);

As you can do with any Array-like object, you can use ES2015's [Array.from()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from" \o "The Array.from() method creates a new, shallow-copied Array instance from an array-like or iterable object.) method or [spread syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) to convert arguments to a real Array:

var args = Array.from(arguments);

var args = [...arguments];

The [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof" \o "The typeof operator returns a string indicating the type of the unevaluated operand.) operator returns 'object' when used with arguments

console.log(typeof arguments); // 'object'

The type of individual arguments can be determined by indexing arguments:

console.log(typeof arguments[0]); // returns the type of the first argument

Argument without rest, default or destructure

If arguments DOESN’T contain – rest, default or destructured arguments, then the values in the arguments object **do** change in sync with the values of the argument variables.

function func(a) {

arguments[0] = 99; // updating arguments[0] also updates a

console.log(a);

}

func(10); // 99

function func(a) {

a = 99; // updating a also updates arguments[0]

console.log(arguments[0]);

}

func(10); // 99

Argument with rest, default or destructure

If arguments DOES

contain – rest, default or destructured arguments, then the values in the arguments object **do** change in sync with the values of the argument variables.

function func(a = 55) {

arguments[0] = 99; // updating arguments[0] does not also update a

console.log(a);

}

func(10); // 10

// An untracked default parameter

function func(a = 55) {

console.log(arguments[0]);

}

func(); // undefined

var arr1 = ["Cecilie", "Lone"];  
var arr2 = ["Emil", "Tobias", "Linus"];  
var arr3 = ["Robin", "Morgan"];  
var myChildren = arr1.concat(arr2, arr3);

1. toString()

converts an array to a comma separated string

All JavaScript objects have a toString() method.

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.toString(); // Banana,Orange,Apple,Mango

for..of vs for..in

Both for..of and for..in statements iterate over lists; the values iterated on are different though, for..in returns a list of keys on the object being iterated, whereas for..of returns a list of values of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

Example 1:

let list = [4, 5, 6];

for (let i in list) { // I contains property names

console.log(i); // "0", "1", "2",

}

for (let i of list) { I contains property values

console.log(i); // "4", "5", "6"

}

Example 2:

Object.prototype.objCustom = function() {};

Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];

for (let i in iterable) {

console.log(i); // logs: 0, 1, 2, "arrCustom", "objCustom"

}

for (let i in iterable) {

if (iterable.hasOwnProperty(i)) {

console.log(i); // logs: 0, 1, 2,

}

}

for (let i of iterable) {

console.log(i); // logs: 3, 5, 7

}

Value, denumerable : they are property descriptor.

Arrays : are iterable

Objects: not iterable by default => that is why for …of fails on it

An iterator is built in object in several data types like:

Arrays, String, Maps, Sets, NodeLists have built-in iterator object