Javascript

typeof (2)
>number
To create a variable use “let”
let x=3;   <!-- js will append “;” if missing
x+=5;
e.g
5*’5’
>>>25
clear()
num--;   <!-- js return the value b4 decrementing by 1.
const pi=3.14;
Booleans
let z=true;
js data-types are totally mutable

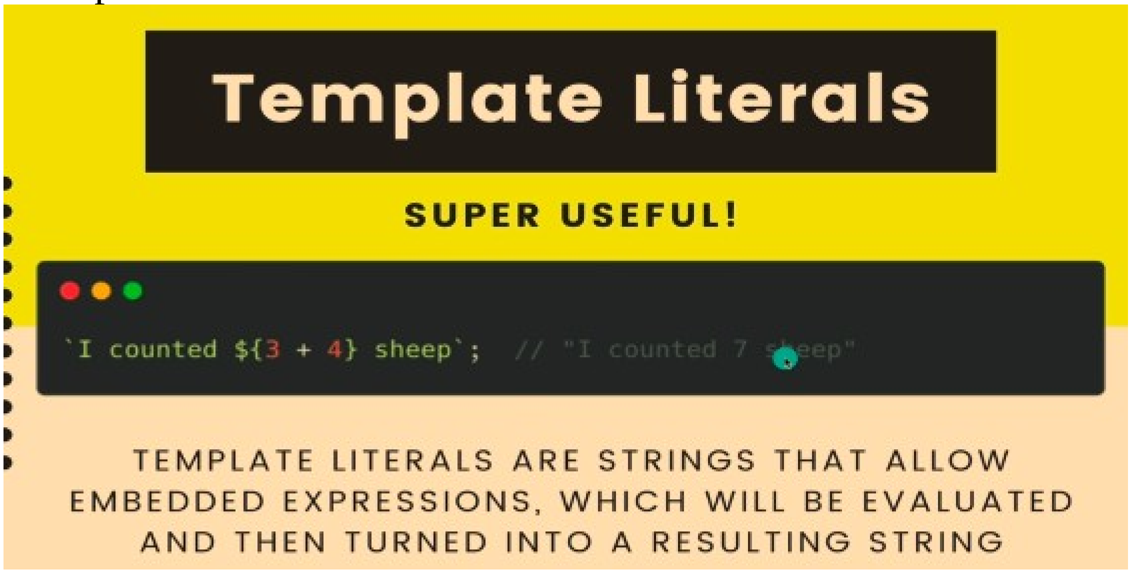
Strings
strings are indexed,starting from index no ‘1’
name[0];
>>> k
Concatenation
“kazim” +” “ +”syed”
>>> kazim syed
e.g
let x=’River’
x=’river’
>>>river   <!-- here u can’t update the first char but u could reassign the var.
e.g
let x=1;
x+=’kazim’
typeof(x)
>>> string

String methods
x.length   <!-- property
x.toUpperCase()   <!-- method

x.trim().toUpperCase()   <!-- applying multiple func

x.indexof(‘’)   <!-- since 0 is for index 0,so -1 for not found.
>>>4
slice is [)
“kazim”.slice(begin-index[,end-index])
<!-- This is the way docs tell us this is optional i,e [,.....] .
        begin-index:0 by default
        end-index:somvalue-1
“kazim”.replace(“kaz”,”kas”);
>>>kasim   <!-- it’s only gonna replace the first matching
                            instance
“kazim”.repeat(10)
>>>kazimkazimkazim.........

Template literals

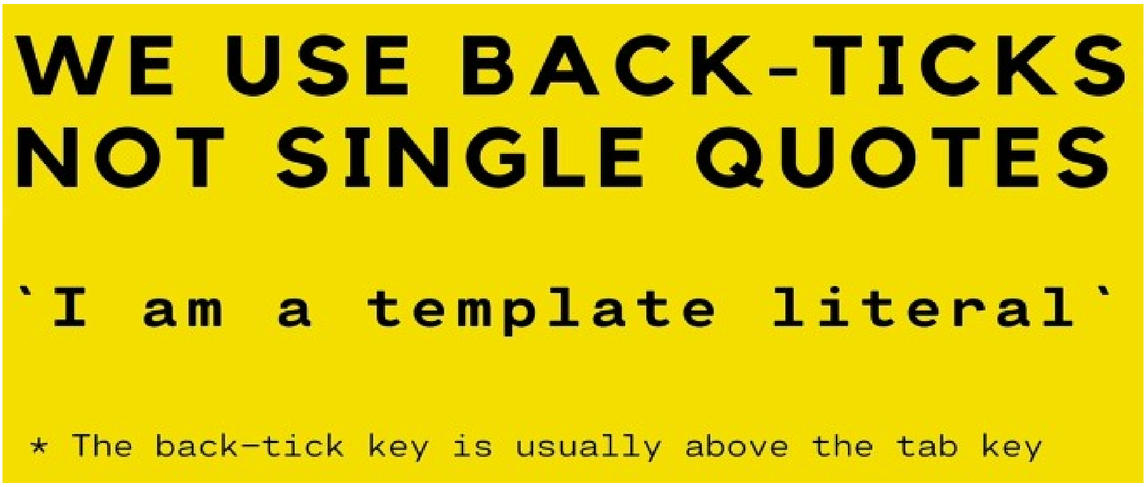


Template Literals

SUPER USEFUL!

```
`I counted ${3 + 4} sheep`;   // "I counted 7 sheep"
```

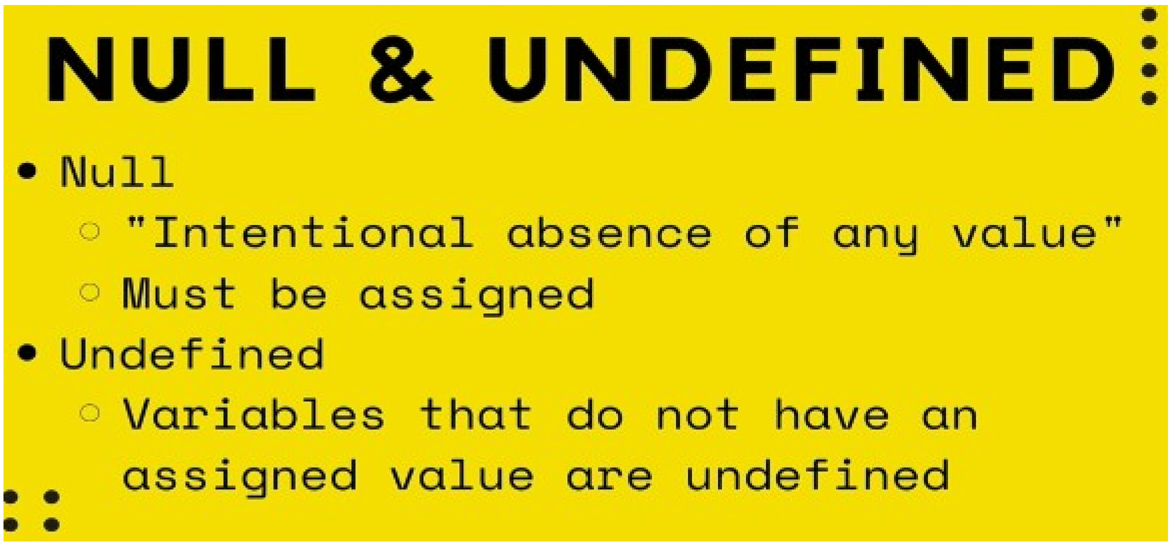
TEMPLATE LITERALS ARE STRINGS THAT ALLOW EMBEDDED EXPRESSIONS, WHICH WILL BE EVALUATED AND THEN TURNED INTO A RESULTING STRING



WE USE BACK-TICKS NOT SINGLE QUOTES

```
`I am a template literal`
```

* The back-tick key is usually above the tab key



NULL & UNDEFINED:

- Null
  - "Intentional absence of any value"
  - Must be assigned
- Undefined
  - Variables that do not have an assigned value are undefined

Math.PI   <!-- use math class for math related calc
Math.random() <!-- this return bet 0 and 1
e.g
  Math.floor((Math.random)*4)) +1
                    <!-- bet 1 and 5
or
Math.floor((Math.random)*6)
                    <!-- bet 0 and 5
why is 1 not inside arg cuz
5.99+1=6.999   i.e 6
(5)+1=6
so why then, lol.
‘A’ < ‘a’
>>>true

== equal
=== strictly equal
!== not equals strictly
e.g
5==’5’
>>>true

0==false | null==undefined
>>>true | >>>true

console.log(“hey”,”there”)
>>> hey there   <!-- like a print statement.

alert(“”)   <!-- just draws attention

parseInt(prompt(“enter a no”))   <!-- takes a input

Note: Always put script at end of the body.Reason is you want ur html load before ur js.

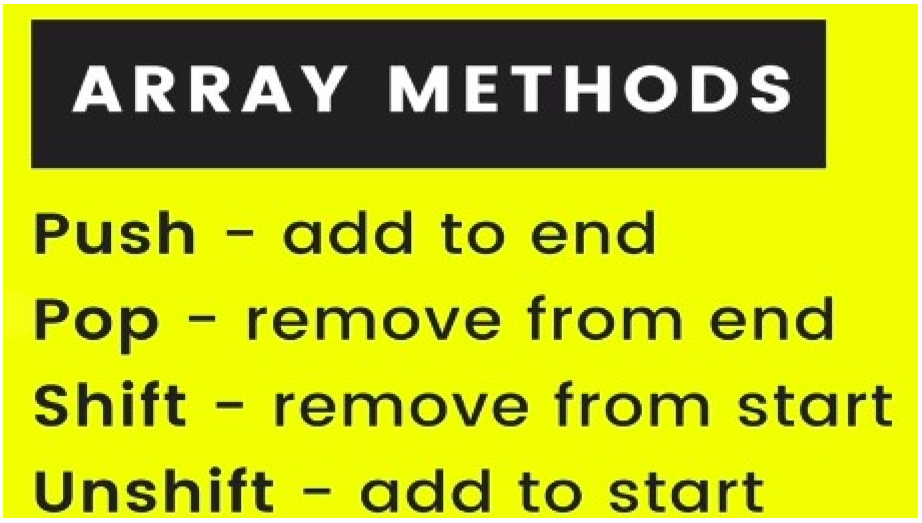
```
If(){
}
else if(){
}
else{
}
```

```
&&, ||, !            <!-- and ,or, not
let x=[]             <!-- array
```

e.g

```
x=[’jon’,’danerys’,’bran’];
x[10]=’khal drogo’;
x
>>>  [’jon’,’danerys’,’bran’,empty x 7, ‘khal drogo’ ]
x.length
>>>11
```

even though index [3,9] is undefined, the length is still 11.

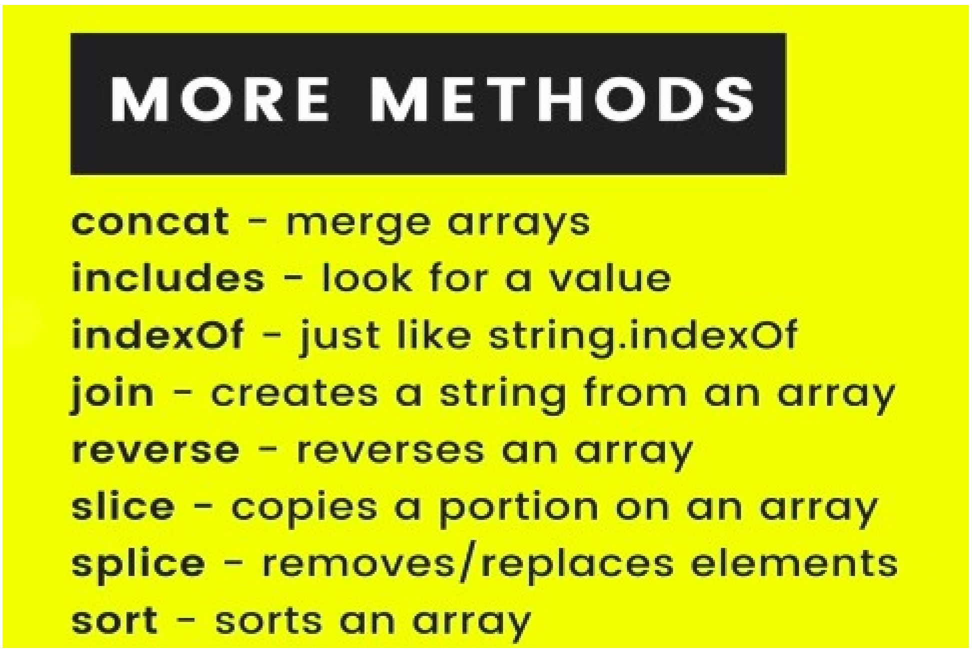

ARRAY METHODS

Push - add to end

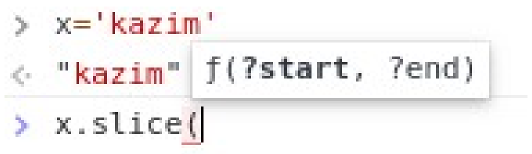
Pop - remove from end

Shift - remove from start

Unshift - add to start


MORE METHODS

concat - merge arrays
includes - look for a value
indexOf - just like string.indexOf
join - creates a string from an array
reverse - reverses an array
slice - copies a portion on an array
splice - removes/replaces elements
sort - sorts an array

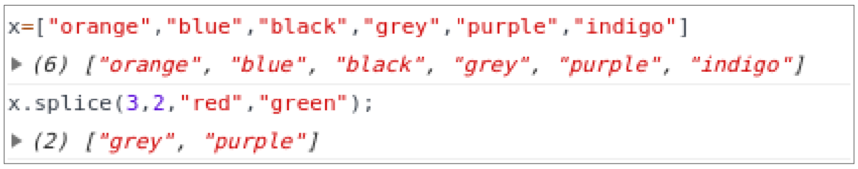

```
> x='kazim'
< "kazim"  f(?start, ?end)
> x.slice(
```

slice: is like give me a slice it cuts offs value within a range.

Push pop	add , remove from end
unshift shift	add, remove from start

splice() method changes the content of an array by removing or replacing existing elements and/or adding new elements in place

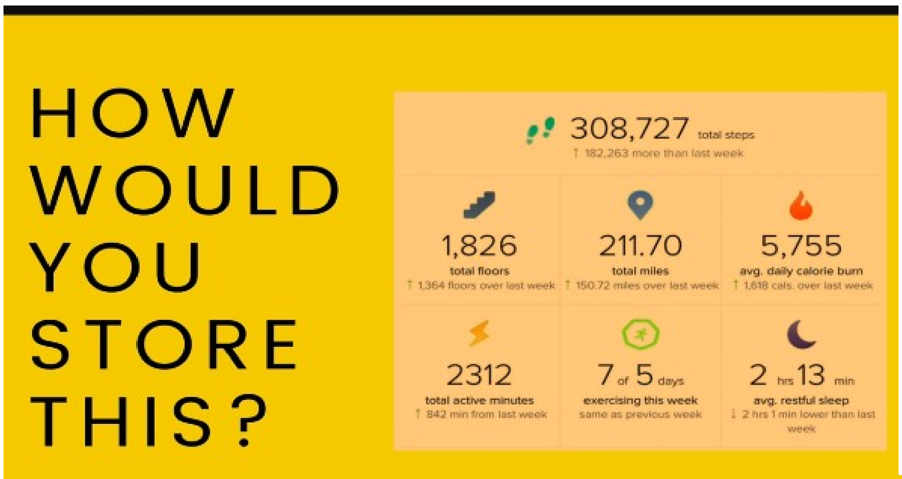
Array
slice(start,?delete count, ...items);


```
x=["orange","blue","black","grey","purple","indigo"]
▶ (6) ["orange", "blue", "black", "grey", "purple", "indigo"]
x.splice(3,2,"red","green");
▶ (2) ["grey", "purple"]
```

x.sort() is stupid.

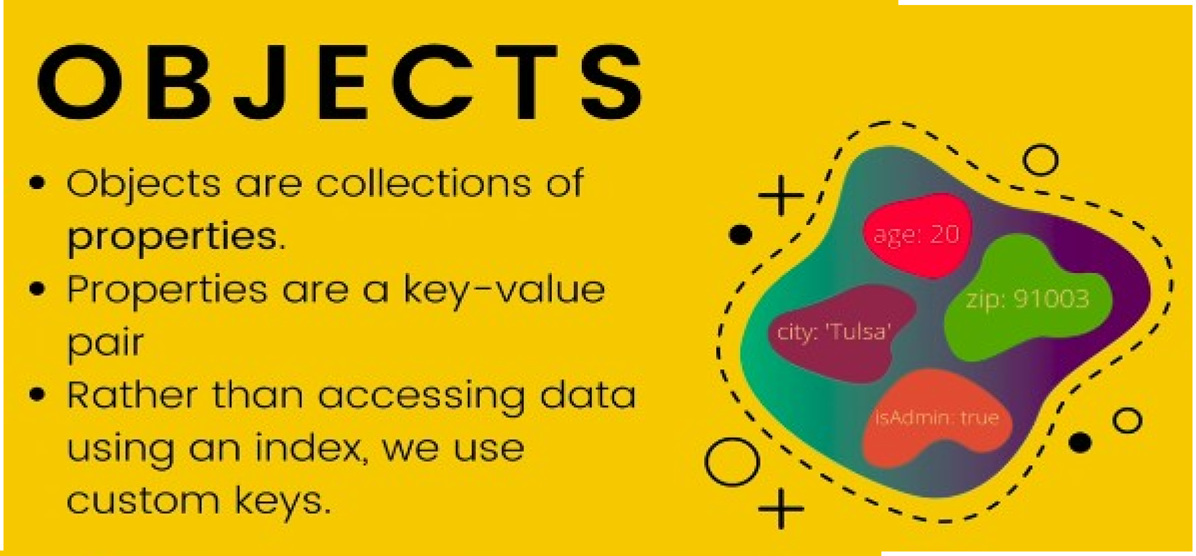
Identifier: seq of char in code,identifies a var,f(),property
naming conventions:
What is valid identifier?
- $, _ , digits
- dont start with a digit

Javascript object literals.


HOW WOULD YOU STORE THIS?

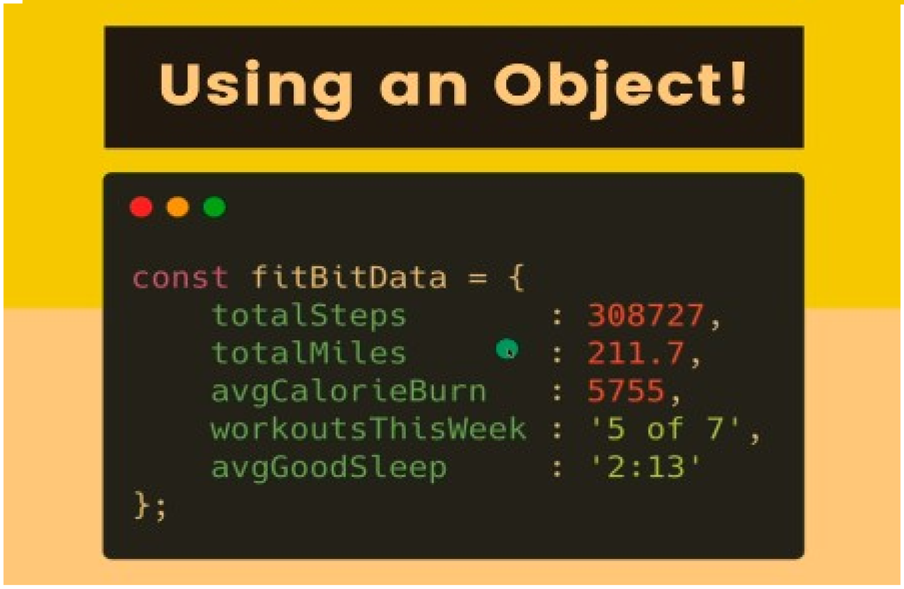
308,727 total steps
↑ 182,263 more than last week

1,826 total floors ↑ 1,364 floors over last week	211.70 total miles ↑ 150.72 miles over last week	5,755 avg. daily calorie burn ↑ 1,618 cals. over last week
2312 total active minutes ↑ 842 min from last week	7 of 5 days exercising this week same as previous week	2 hrs 13 min avg. restful sleep ↓ 2 hrs 1 min lower than last week


OBJECTS

- Objects are collections of properties.
- Properties are a key-value pair
- Rather than accessing data using an index, we use custom keys.

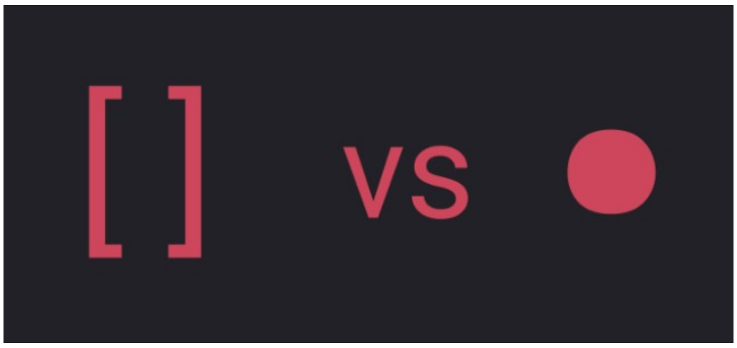
age: 20
zip: 91003
city: 'Tulsa'
isAdmin: true


Using an Object!

```
const fitBitData = {
    totalSteps        : 308727,
    totalMiles        : 211.7,
    avgCalorieBurn    : 5755,
    workoutsThisWeek  : '5 of 7',
    avgGoodSleep      : '2:13'
};
```

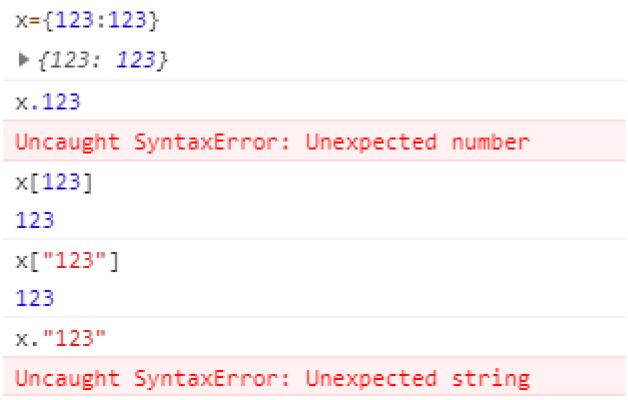
weird
fitbitData[‘total’+’Steps’]
>>>308727

Dot vs Bracket Notation in Accessing Javascript object. [Dot vs bracket]



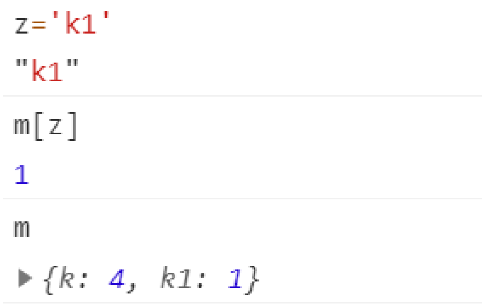
e.g

var obj={
123:123
}
Here, this is a vaild identifier in array somehow

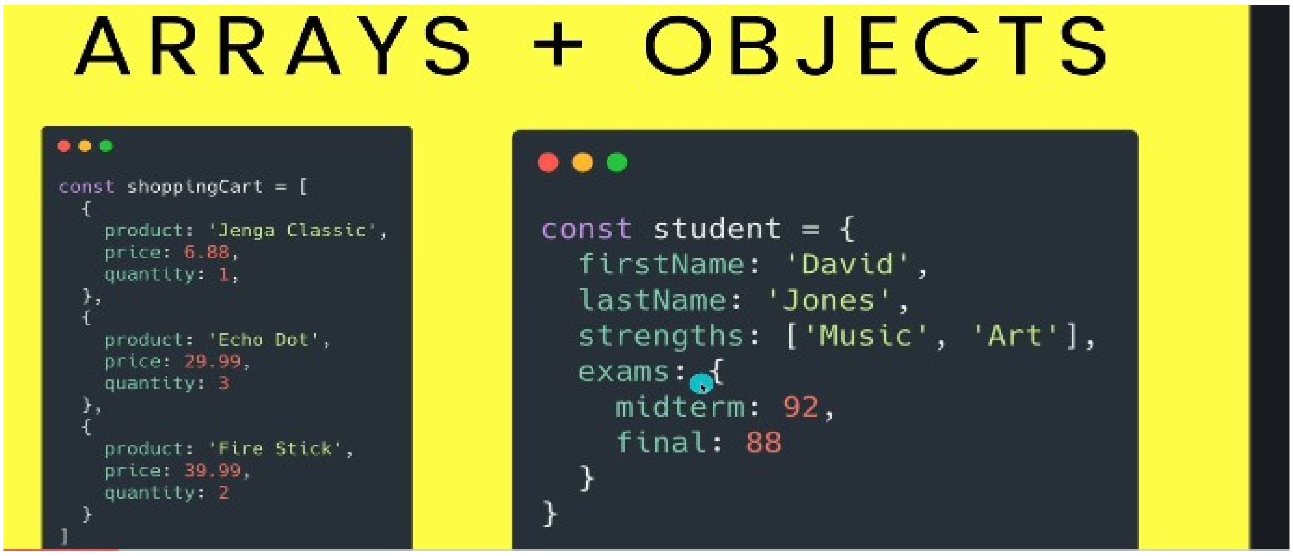


```
x={123:123}
▸{123: 123}
x.123
Uncaught SyntaxError: Unexpected number
x[123]
123
x["123"]
123
x."123"
Uncaught SyntaxError: Unexpected string
```

every key is turned into string.
Bracket notation is use whenever property name is a invalid identifier
[Dot vs bracket Accesing data]
use ‘’ in creation and accessing.
You can access variables in bracket notation aka no quotation also tells why actual numbers works cuz they can’t be variables, n convert it to string.



```
z='k1'
"k1"
m[z]
1
m
▸{k: 4, k1: 1}
```

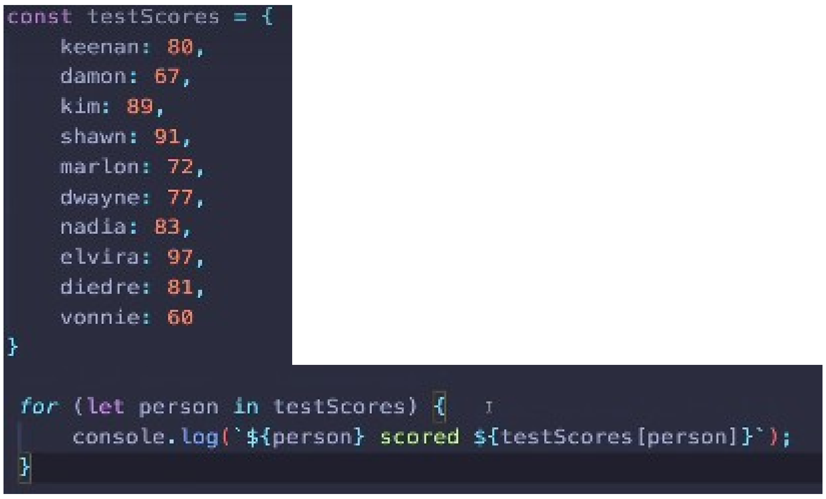


ARRAYS + OBJECTS

```
const shoppingCart = [
  {
    product: 'Jenga Classic',
    price: 6.88,
    quantity: 1,
  },
  {
    product: 'Echo Dot',
    price: 29.99,
    quantity: 3
  },
  {
    product: 'Fire Stick',
    price: 39.99,
    quantity: 2
  }
]
```

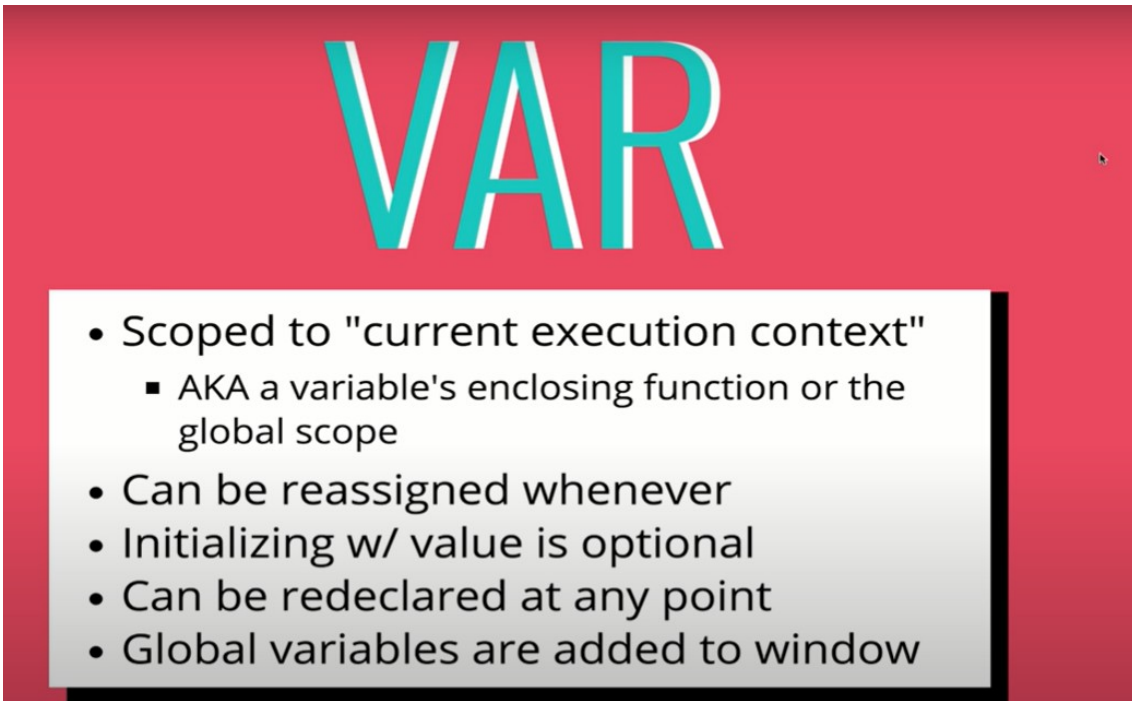
```
const student = {
  firstName: 'David',
  lastName: 'Jones',
  strengths: ['Music', 'Art'],
  exams: {
    midterm: 92,
    final: 88
  }
}
```

In js, objects are considered iterables.
So use
for (let e in x){statement}   <!-- Array negative, Obj positive
instead of
for (let e of x){ statement}   <!-- common now
e.g
to get all the keys use
Array Object.keys(testscores)
Array Object.values(testscores)
nestedArray Object.entries(testscores)



```
const testScores = {
    keenan: 80,
    damon: 67,
    kim: 89,
    shawn: 91,
    marlon: 72,
    dwayne: 77,
    nadia: 83,
    elvira: 97,
    diedre: 81,
    vonnie: 60
}

for (let person in testScores) {
    console.log(`${person} scored ${testScores[person]}`);
}
```

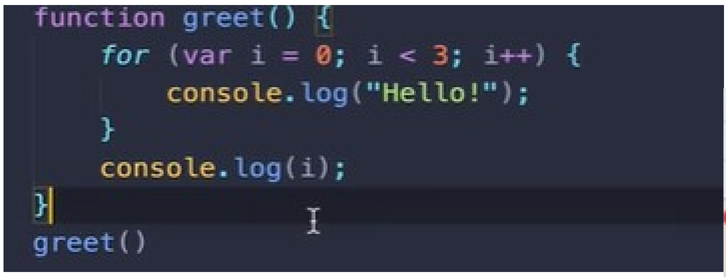
Hoisting is js default behaviour to move declaration to the top.But you should always put declaration before calling.



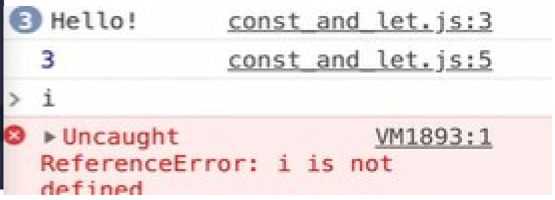
VAR

- Scoped to "current execution context"
  - AKA a variable's enclosing function or the global scope
- Can be reassigned whenever
- Initializing w/ value is optional
- Can be redeclared at any point
- Global variables are added to window

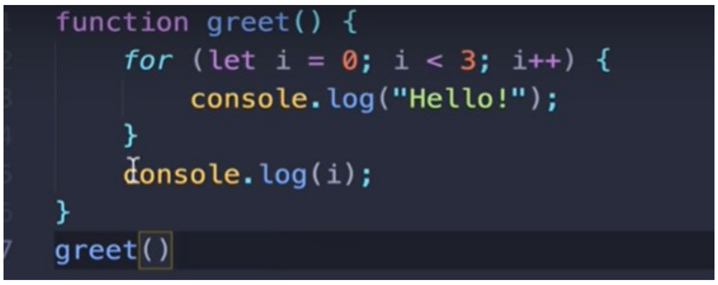
let and const are block scope and var is not.



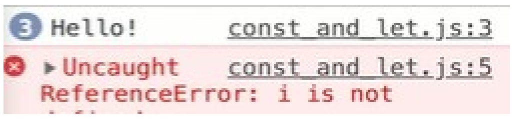
```
function greet() {
    for (var i = 0; i < 3; i++) {
        console.log("Hello!");
    }
    console.log(i);
}
greet()
```



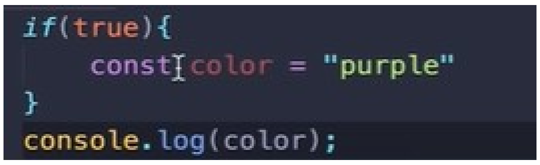
```
3 Hello!        const_and_let.js:3
  3             const_and_let.js:5
> i
  ▶Uncaught             VM1893:1
  ReferenceError: i is not
  defined
```



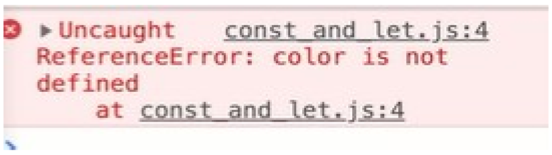
```
function greet() {
    for (let i = 0; i < 3; i++) {
        console.log("Hello!");
    }
    console.log(i);
}
greet()
```



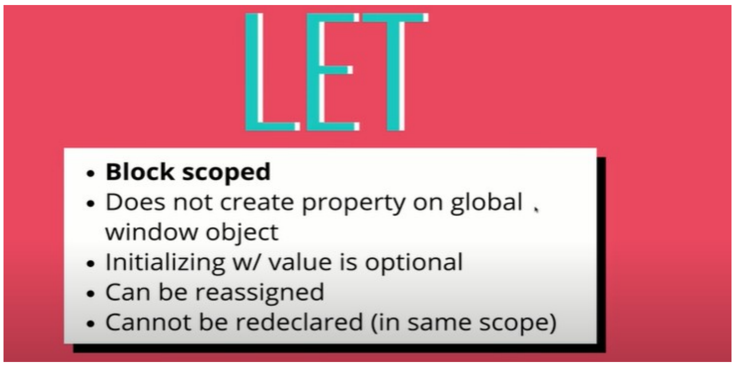
```
3 Hello!        const_and_let.js:3
  ▶Uncaught     const_and_let.js:5
  ReferenceError: i is not
```



```
if(true){
    const color = "purple"
}
console.log(color);
```

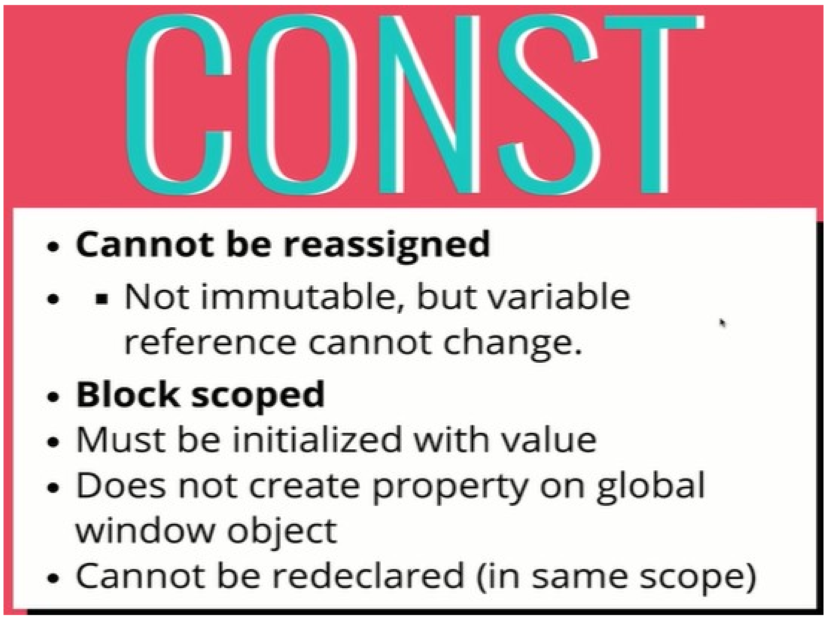


```
  ▶Uncaught    const_and_let.js:4
  ReferenceError: color is not
  defined
      at const_and_let.js:4
```



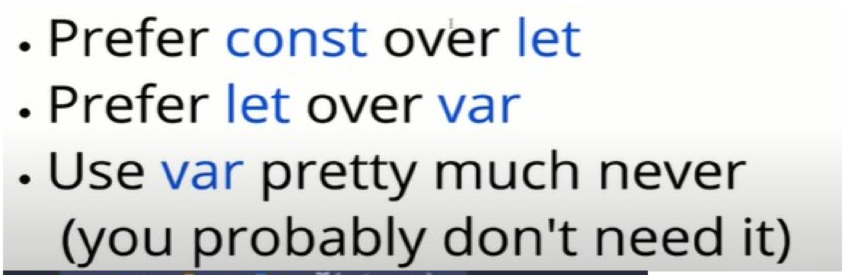
LET

- **Block scoped**
- Does not create property on global window object
- Initializing w/ value is optional
- Can be reassigned
- Cannot be redeclared (in same scope)

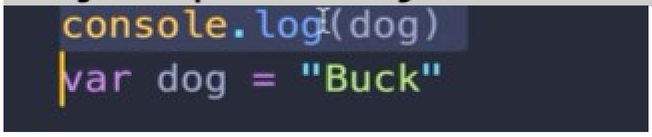


CONST

- **Cannot be reassigned**
-
  - Not immutable, but variable reference cannot change.
- **Block scoped**
- Must be initialized with value
- Does not create property on global window object
- Cannot be redeclared (in same scope)

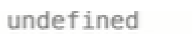


- Prefer const over let
- Prefer let over var
- Use var pretty much never (you probably don't need it)



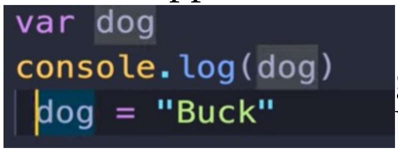
```
console.log(dog)
var dog = "Buck"
```

>No error



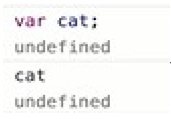
```
undefined
```

THIS WORKS
what happen is



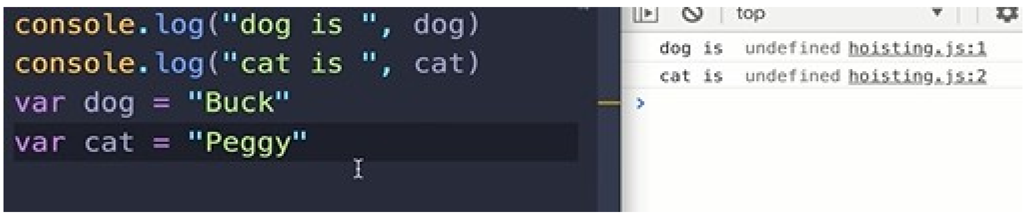
```
var dog
console.log(dog)
 dog = "Buck"
```

What happen is js is aware of this variable and gives it a value of undefined at the begining before this console.log before it's initialized. Your variables are not physically move to the top of scope, Your variable declaration happens first.



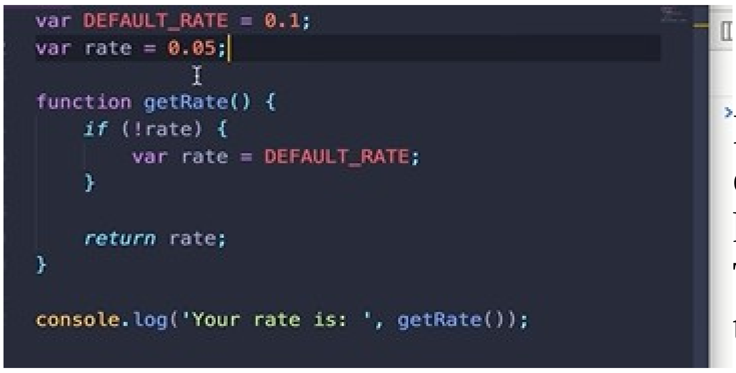
```
var cat;
undefined
cat
undefined
```

Same as this, its just making space in memory whatever variable you have. Its not that code is re-arranged. It's just multiple-step process, when your code is run, there is a step of creating space for all the variables in memory and thats is done before they are given a value.

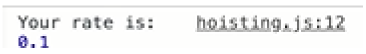


```
console.log("dog is ", dog)
console.log("cat is ", cat)
var dog = "Buck"
var cat = "Peggy"
```

```
dog is  undefined hoisting.js:1
cat is  undefined hoisting.js:2
```

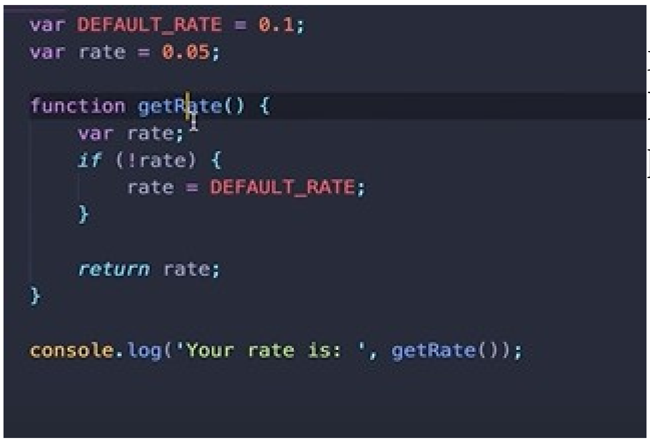


```
var DEFAULT_RATE = 0.1;
var rate = 0.05;

function getRate() {
    if (!rate) {
        var rate = DEFAULT_RATE;
    }

    return rate;
}

console.log('Your rate is: ', getRate());
```



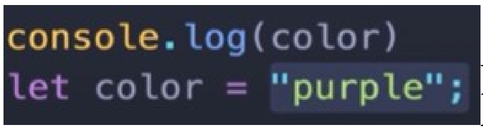
```
Your rate is:    hoisting.js:12
0.1
```

Here, 2 things are happening, one is scope and other is hoist.
The var rate is not scope to this conditional not this block but this entire func getRate().



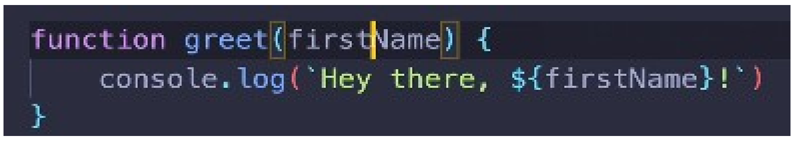
```
var DEFAULT_RATE = 0.1;
var rate = 0.05;

function getRate() {
    var rate;
    if (!rate) {
        rate = DEFAULT_RATE;
    }

    return rate;
}

console.log('Your rate is: ', getRate());
```

Two var define, the one define inside take precedence.
Easiest way of to get around this problem is to let and const.



```
console.log(color)
let color = "purple";
```

Alot of ppl thing let and cost are not hoisted, thats not true. They are hoisted but they are not assigned undefined.
Ik its confusing.



```
function greet(firstName) {
    console.log(`Hey there, ${firstName}!`)
}
```

Here, the firstname is called a parameter, the name that we use in function definition.
And when we actually call greet n pass the value e.g
greet('kazim')

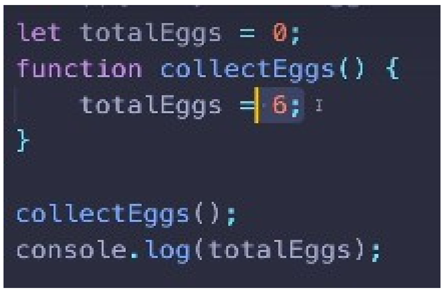
argument

for e.g
repeat(name,numTimes){ statement}
<!-- js doesnt care if u didnt give an argument, but if there is use of arg in code it will cause an error.

- when you use ur <u>return statement,</u> You can only return a single value. Well that one could be an array or object n u have whole bunch of values.
- the func stops when a func return's.

<u>Levelling up our functions</u>

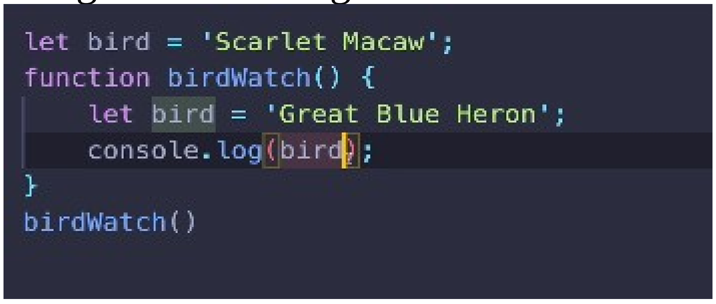
Scope: variable visibity: where we define our varibles in js impacts where we have access to it.



```
let totalEggs = 0;
function collectEggs() {
    totalEggs = 6;
}

collectEggs();
console.log(totalEggs);
```

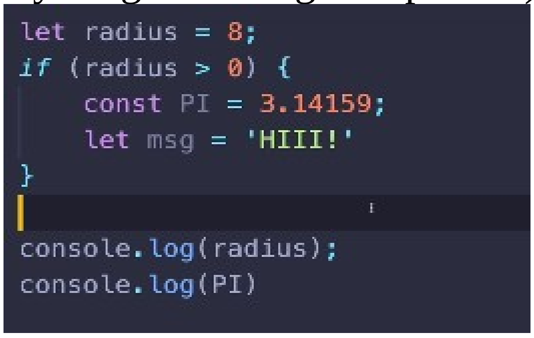
```
>>>  0
     6
```

changes made on a global variables in a func is global.



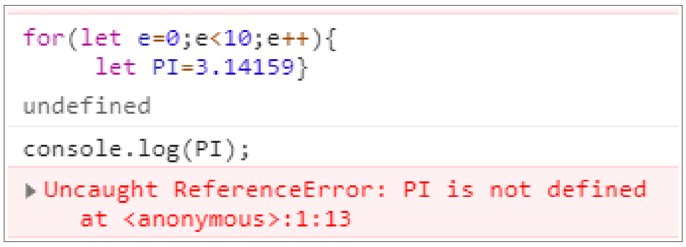
```
let bird = 'Scarlet Macaw';
function birdWatch() {
    let bird = 'Great Blue Heron';
    console.log(bird);
}
birdWatch()
```

if there is a variable define with the name “bird” in the closest enclosing, we will reference to that variable.if not walk through the closest enclosing brackets all the way to see global variables.
Block scope:
Block scope where the visibility of declared var is within blocks only i.e (anything enclosing except func) like loops and conditional.

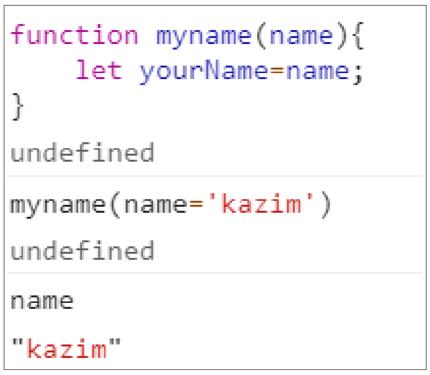


```
let radius = 8;
if (radius > 0) {
    const PI = 3.14159;
    let msg = 'HIII!'
}

console.log(radius);
console.log(PI)
```

>>> 8
error pi isn’t defined

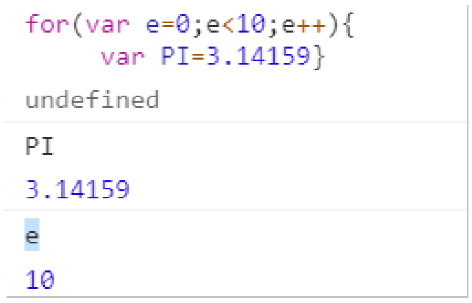


```
for(let e=0;e<10;e++){
    let PI=3.14159}
undefined
console.log(PI);
▶Uncaught ReferenceError: PI is not defined
    at <anonymous>:1:13
```



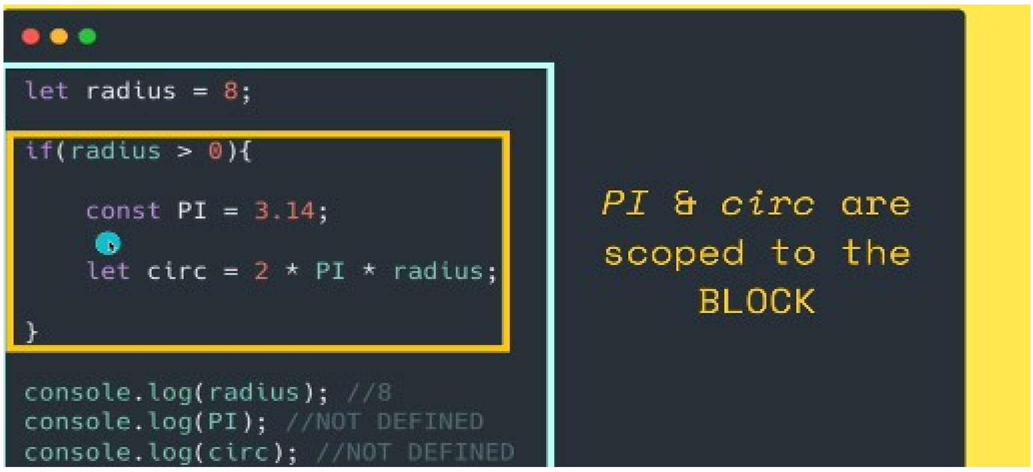
```
function myname(name){
    let yourName=name;
}
undefined
myname(name='kazim')
undefined
name
"kazim"
```

Why we use let and const instead of var ?



```
for(var e=0;e<10;e++){
    var PI=3.14159}
undefined
PI
3.14159
e
10
```

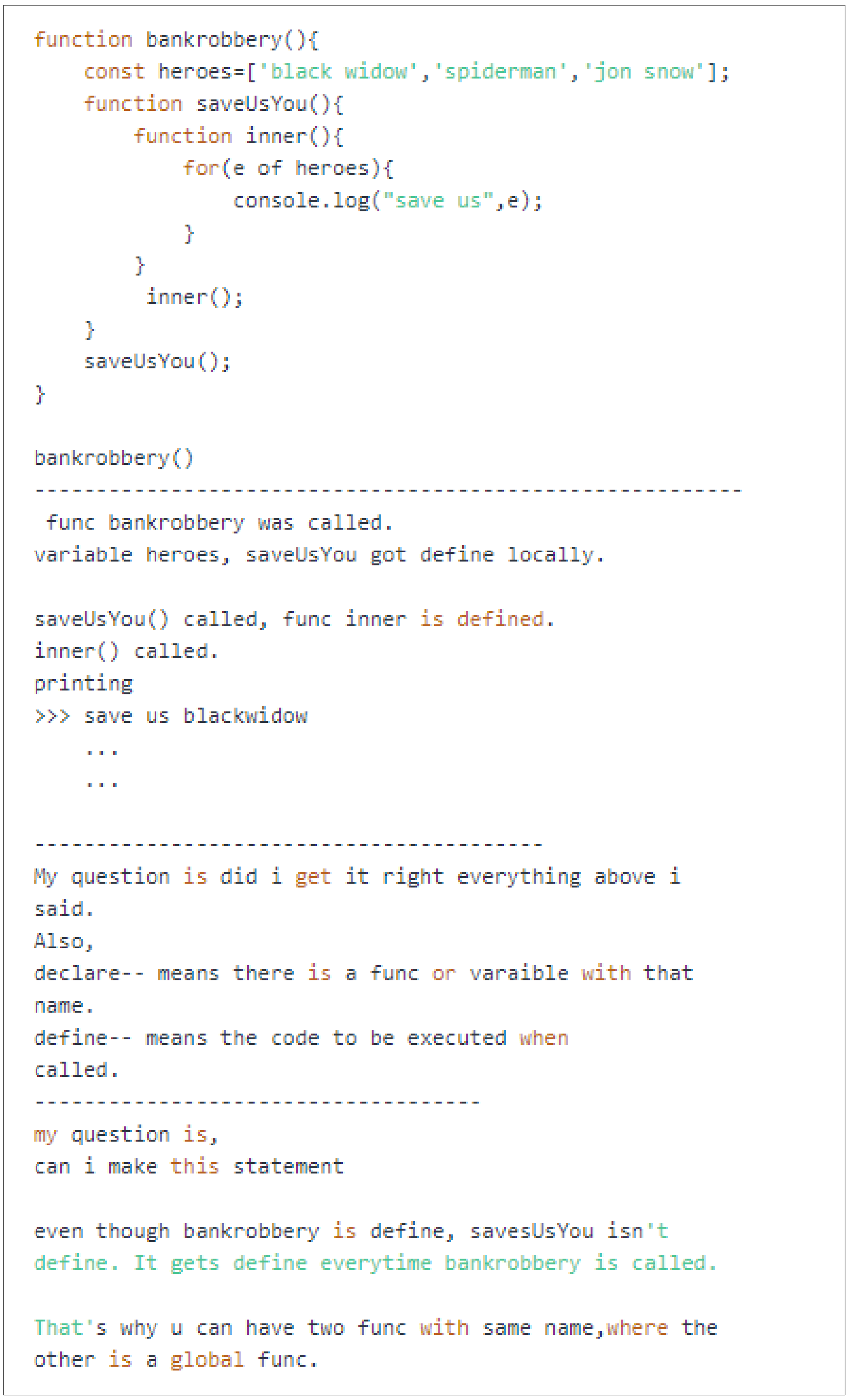
Let and const were introduced in order to follow this scoping rules.
If you use variables without declaration they are consider declared using var



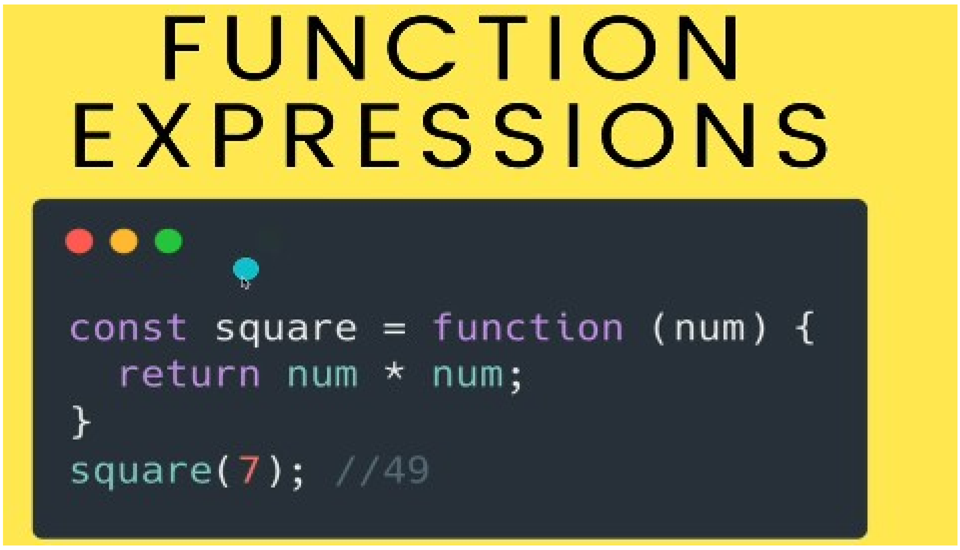
```
let radius = 8;

if(radius > 0){

    const PI = 3.14;

    let circ = 2 * PI * radius;

}

console.log(radius); //8
console.log(PI); //NOT DEFINED
console.log(circ); //NOT DEFINED
```

PI & circ are scoped to the BLOCK

Lexical scope



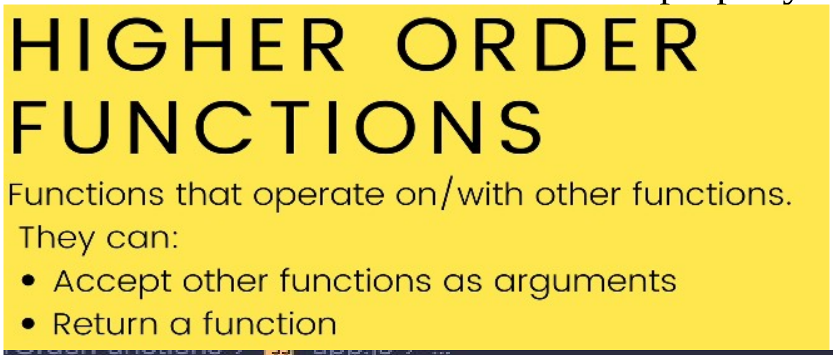
```
function bankrobbery(){
    const heroes=['black widow','spiderman','jon snow'];
    function saveUsYou(){
        function inner(){
            for(e of heroes){
                console.log("save us",e);
            }
        }
         inner();
    }
    saveUsYou();
}

bankrobbery()
---------------------------------------------------------
 func bankrobbery was called.
variable heroes, saveUsYou got define locally.

saveUsYou() called, func inner is defined.
inner() called.
printing
>>> save us blackwidow
    ...
    ...

-----------------------------------------
My question is did i get it right everything above i
said.
Also,
declare-- means there is a func or varaible with that
name.
define-- means the code to be executed when
called.
------------------------------------
my question is,
can i make this statement

even though bankrobbery is define, savesUsYou isn't
define. It gets define everytime bankrobbery is called.

That's why u can have two func with same name,where the
other is a global func.
```



FUNCTION EXPRESSIONS

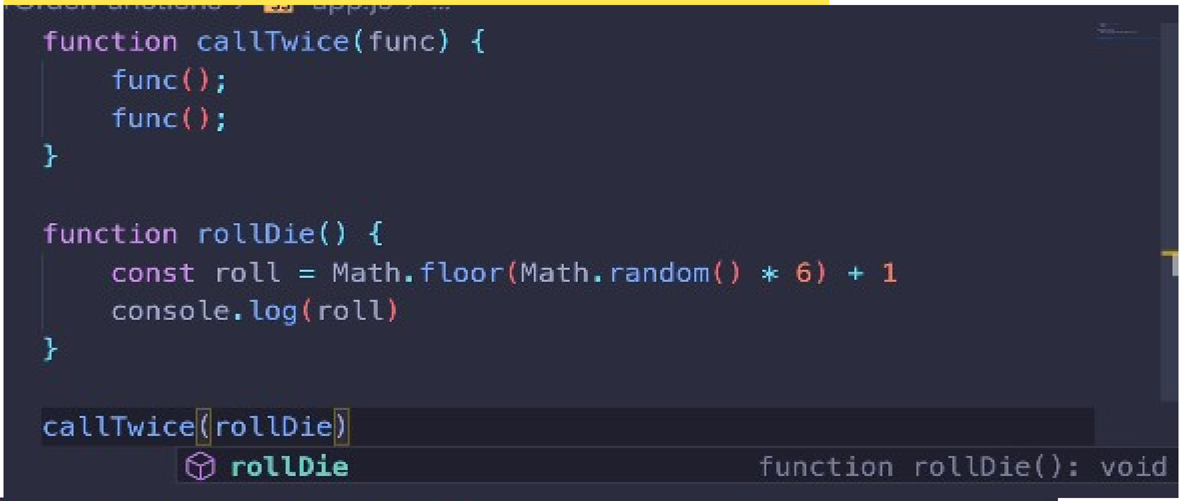
```
const square = function (num) {
  return num * num;
}
square(7); //49
```

functions are object behind the scene. So we can store them, pass them around.we can pass func as an argument or return functions as a number. We can use a function as the value in a property in a object.

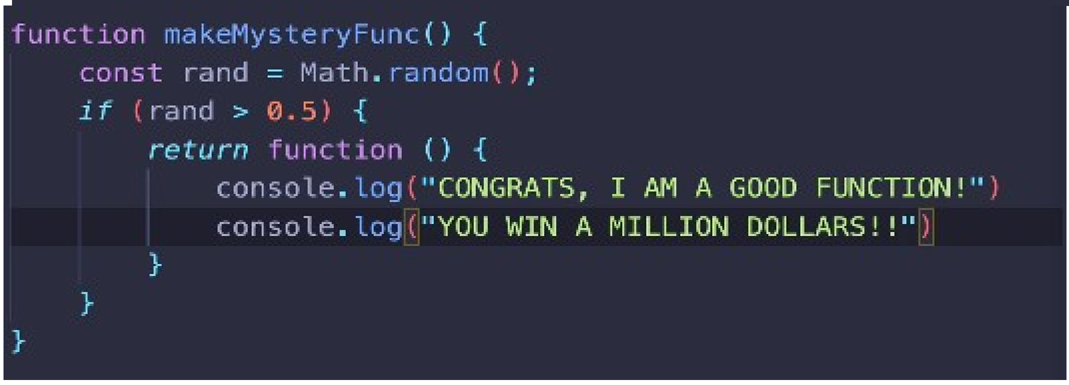


HIGHER ORDER FUNCTIONS

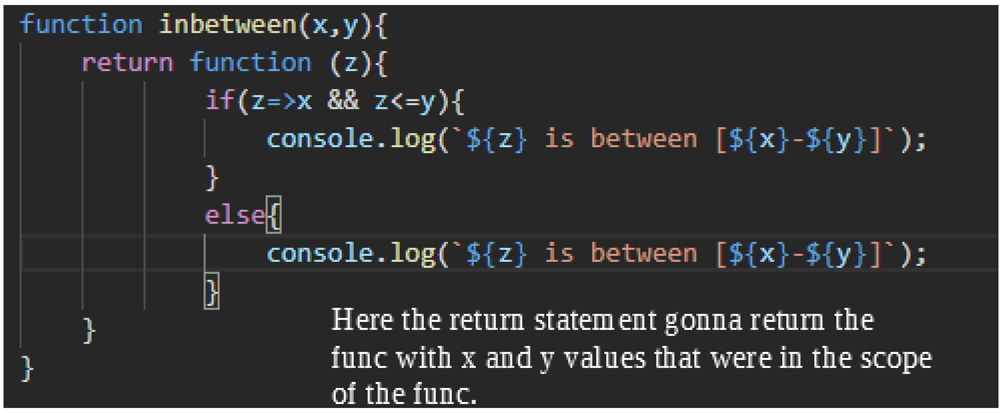
Functions that operate on/with other functions.
They can:
- Accept other functions as arguments
- Return a function



```
function callTwice(func) {
    func();
    func();
}

function rollDie() {
    const roll = Math.floor(Math.random() * 6) + 1
    console.log(roll)
}

callTwice(rollDie)
          rollDie                              function rollDie(): void
```

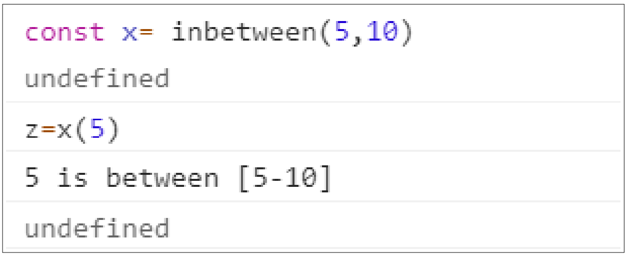


```
function makeMysteryFunc() {
    const rand = Math.random();
    if (rand > 0.5) {
        return function () {
            console.log("CONGRATS, I AM A GOOD FUNCTION!")
            console.log("YOU WIN A MILLION DOLLARS!!")
        }
    }
}
```



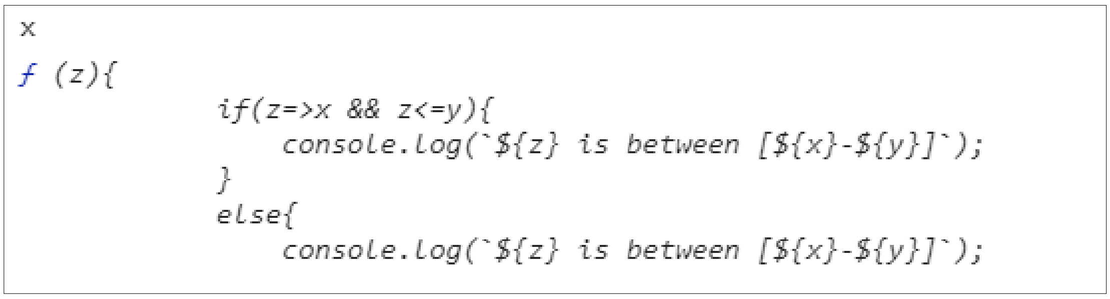
```
function inbetween(x,y){
    return function (z){
            if(z=>x && z<=y){
                console.log(`${z} is between [${x}-${y}]`);
            }
            else{
                console.log(`${z} is between [${x}-${y}]`);
            }
    }
}
```

Here the return statement gonna return the func with x and y values that were in the scope of the func.



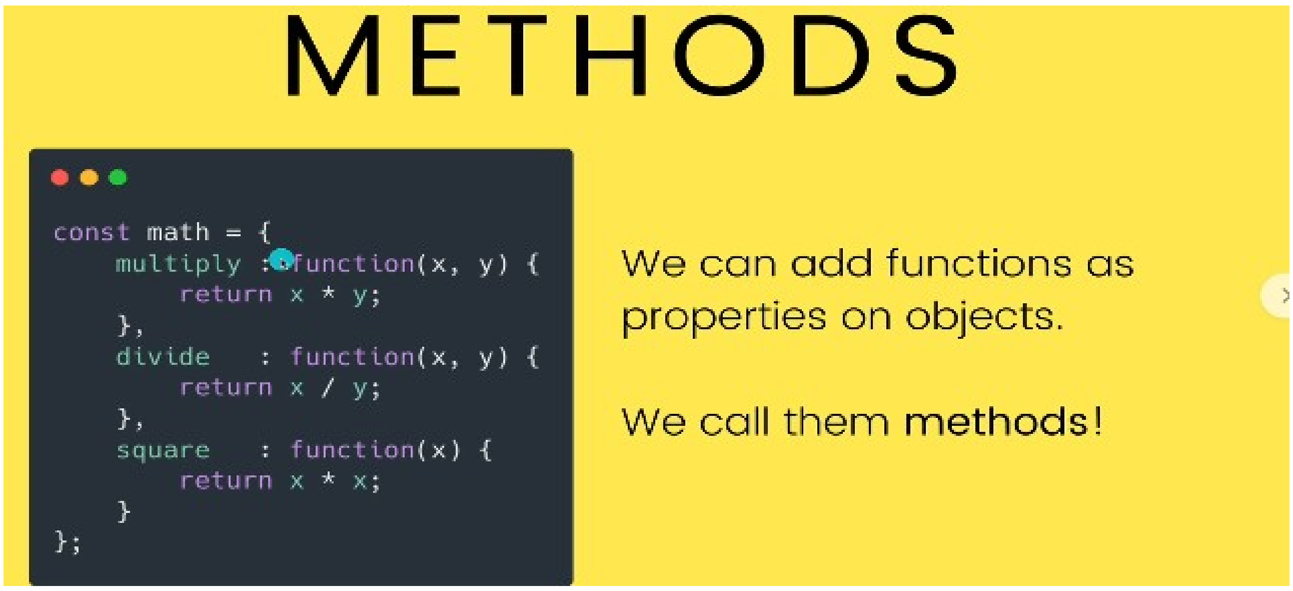
```
const x= inbetween(5,10)
undefined
z=x(5)
5 is between [5-10]
undefined
```

A factory func is a func thats gonna make a func for me.



```
x
ƒ (z){
            if(z=>x && z<=y){
                console.log(`${z} is between [${x}-${y}]`);
            }
            else{
                console.log(`${z} is between [${x}-${y}]`);
```

It maynot seem x and y are 5,10 here. They are.

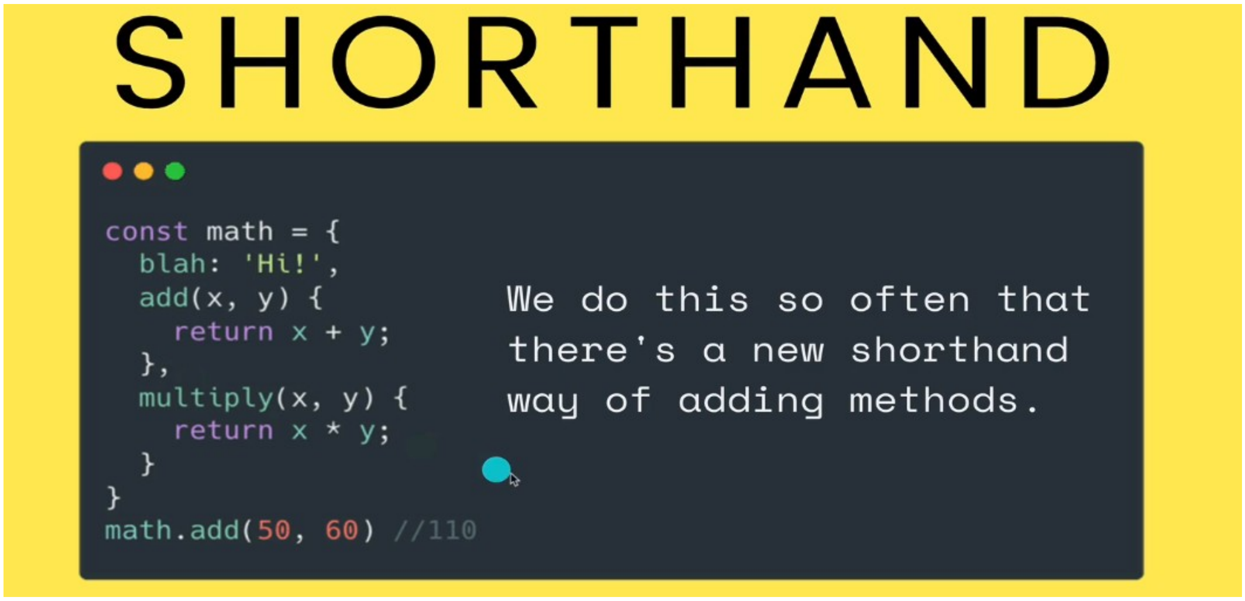


METHODS

```
const math = {
    multiply : function(x, y) {
        return x * y;
    },
    divide   : function(x, y) {
        return x / y;
    },
    square   : function(x) {
        return x * x;
    }
};
```

We can add functions as properties on objects.

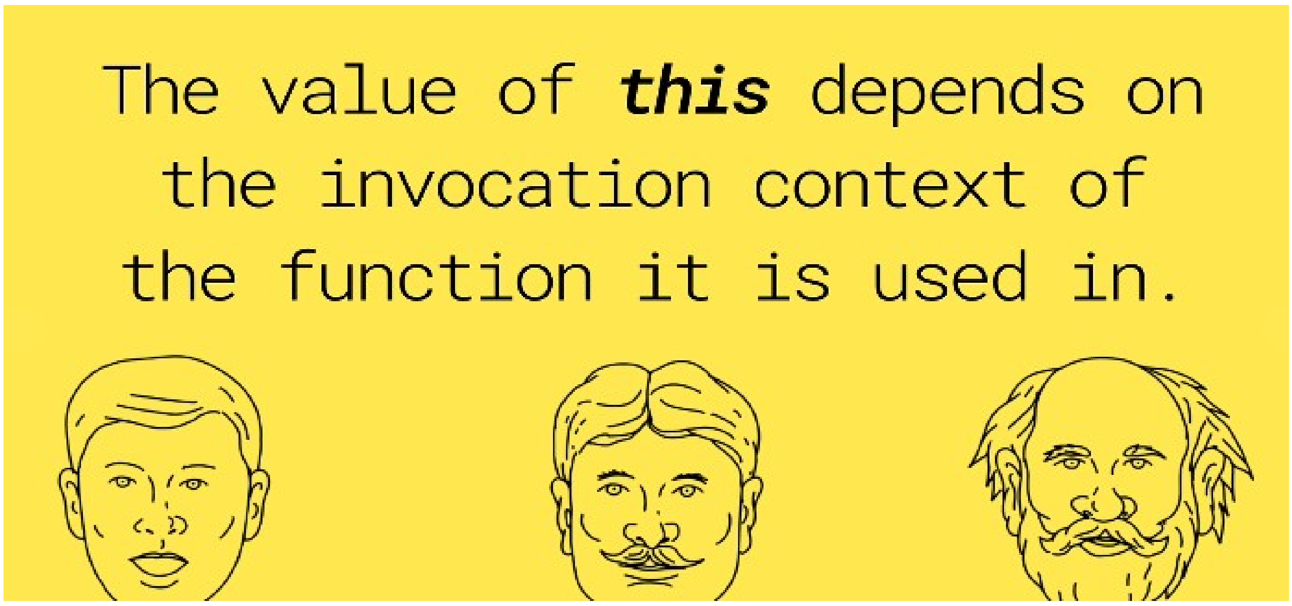
We call them **methods**!



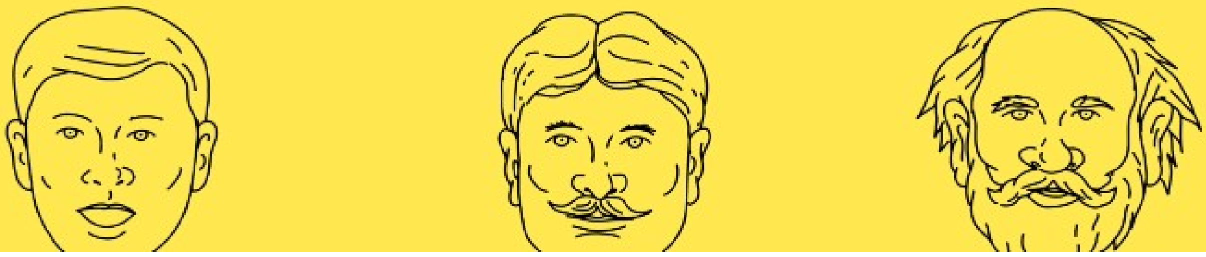
SHORTHAND

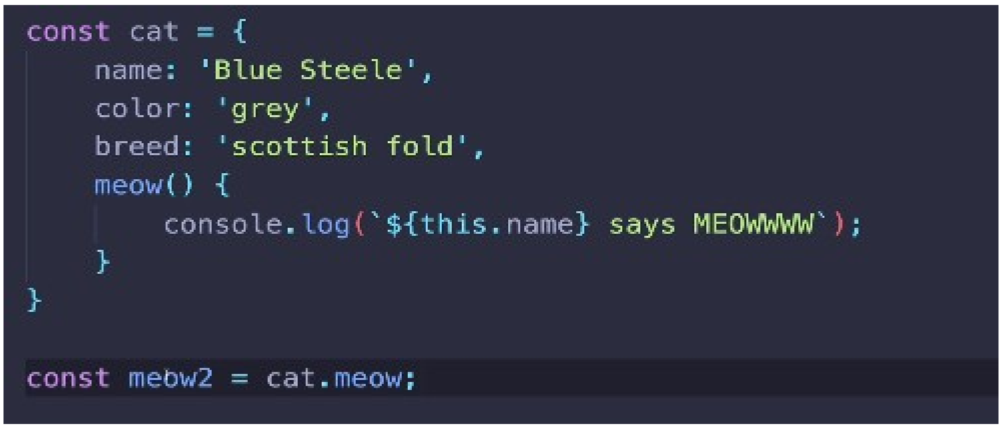
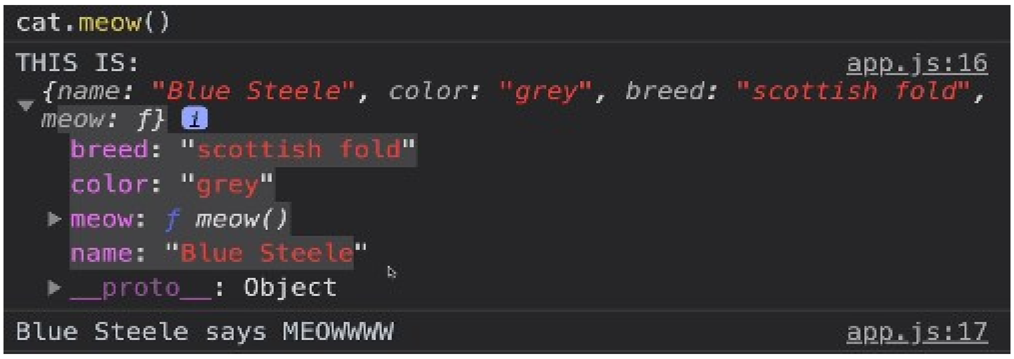
```
const math = {
  blah: 'Hi!',
  add(x, y) {
    return x + y;
  },
  multiply(x, y) {
    return x * y;
  }
}
math.add(50, 60) //110
```

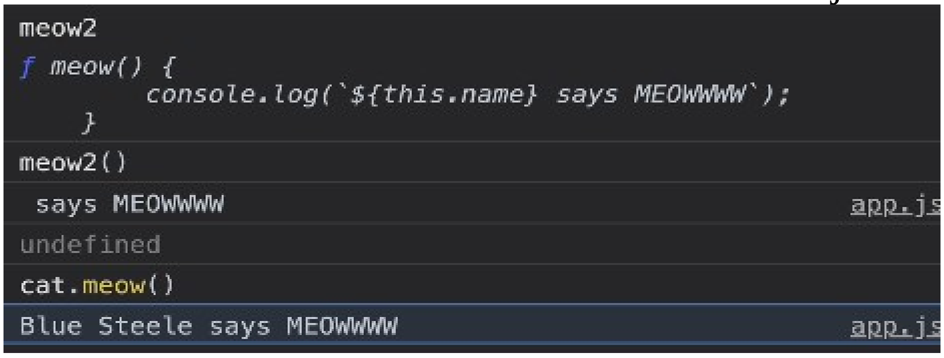
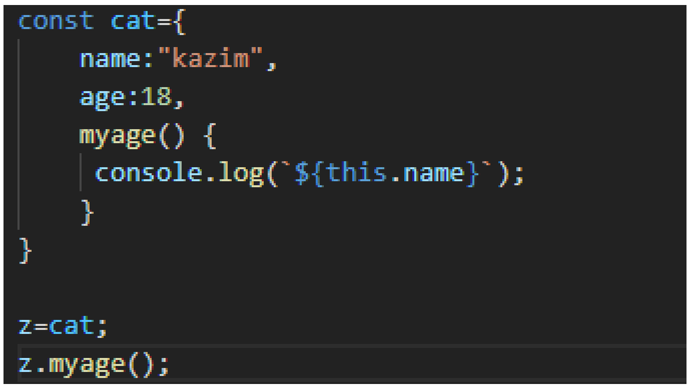
We do this so often that there's a new shorthand way of adding methods.



The value of ***this*** depends on the invocation context of the function it is used in.

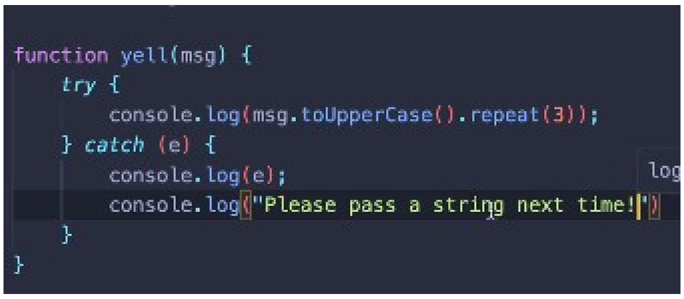



```
const cat = {
    name: 'Blue Steele',
    color: 'grey',
    breed: 'scottish fold',
    meow() {
        console.log(`${this.name} says MEOWWWW`);
    }
}

const meow2 = cat.meow;
```

```
cat.meow()
THIS IS:                                                          app.js:16
 {name: "Blue Steele", color: "grey", breed: "scottish fold",
▼meow: ƒ} i
   breed: "scottish fold"
   color: "grey"
  ▶meow: ƒ meow()
   name: "Blue Steele"
  ▶__proto__: Object
Blue Steele says MEOWWWW                                          app.js:17
```
You kinda feel meow2 is reference to the same object.

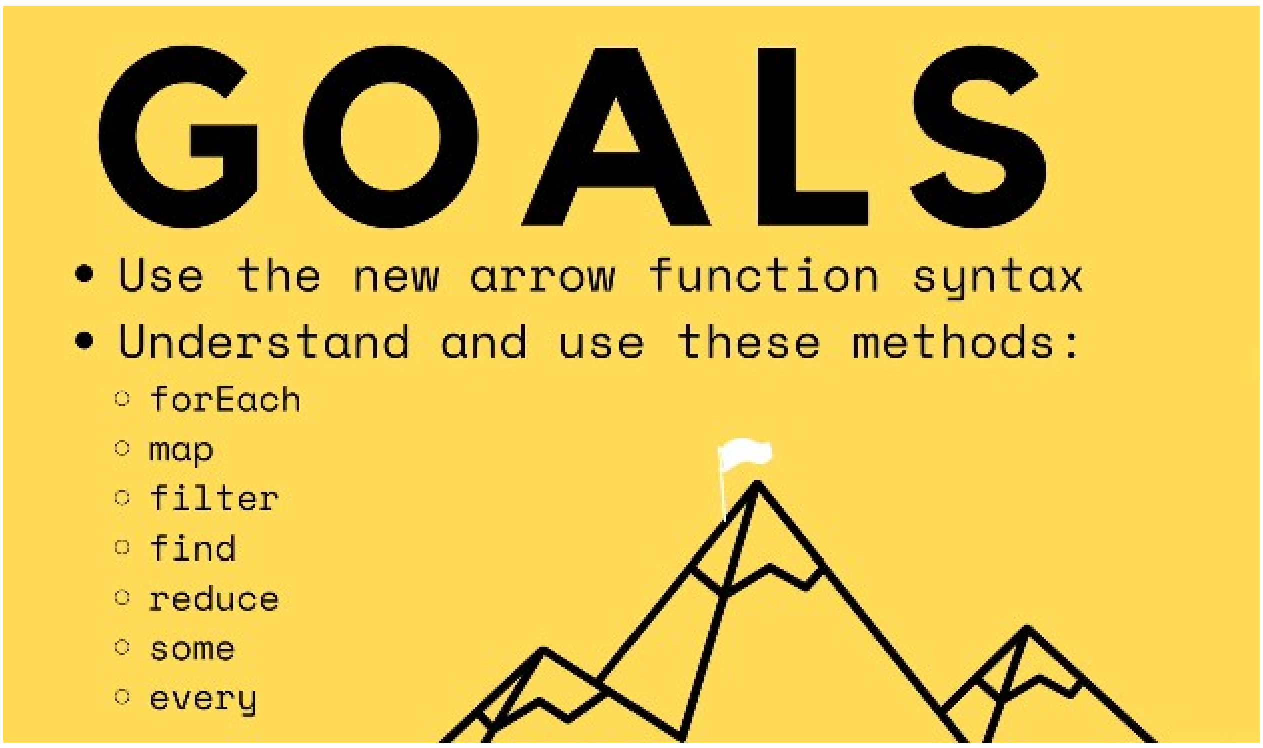

```
meow2
ƒ meow() {
        console.log(`${this.name} says MEOWWWW`);
    }
meow2()
 says MEOWWWW                                               app.js
undefined
cat.meow()
Blue Steele says MEOWWWW                                    app.js
```

```
const cat={
    name:"kazim",
    age:18,
    myage() {
     console.log(`${this.name}`);
    }
}

z=cat;
z.myage();
```
It’s different.

Why different results?
The difference is in " The value of '*this'* depends on the invocation context of the function it is used in.
When we invoke the func cat.meow() , '*this*' refers to the left of the **'.'** and
when i ran meow2(), the keyword 'this' isn't gonna refer to cat object.
It refers to window object i.e top level object or root object.
It is a pool of all objects.

Try,Catch

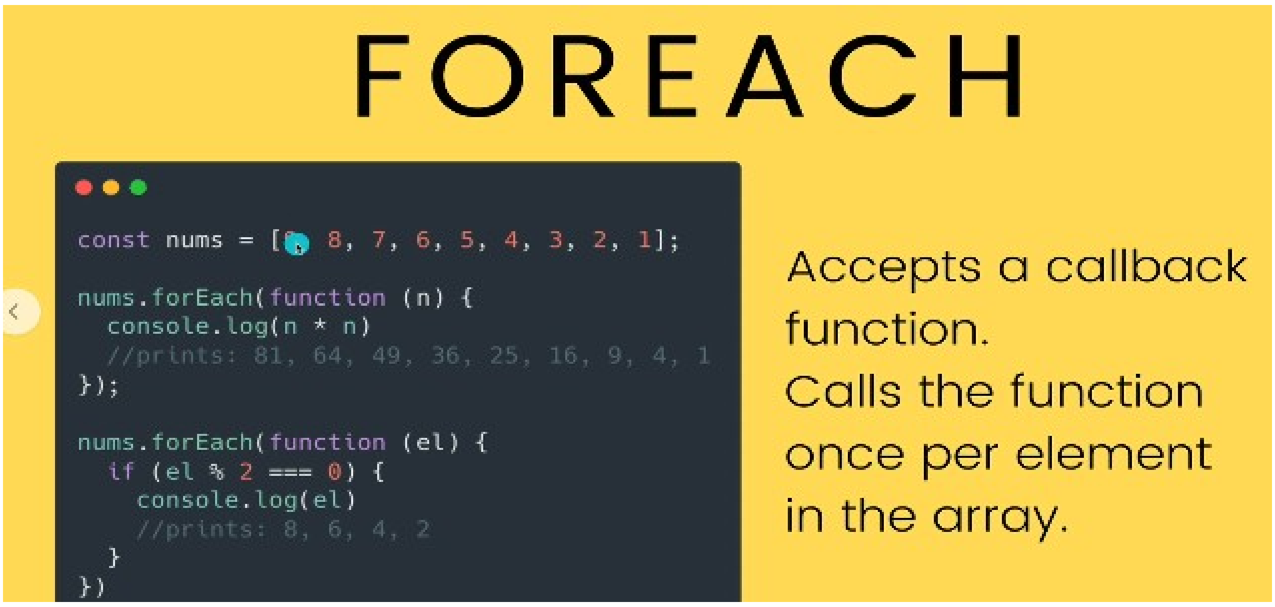


```
function yell(msg) {
    try {
        console.log(msg.toUpperCase().repeat(3));
    } catch (e) {
        console.log(e);
        console.log("Please pass a string next time!")
    }
}
```



GOALS

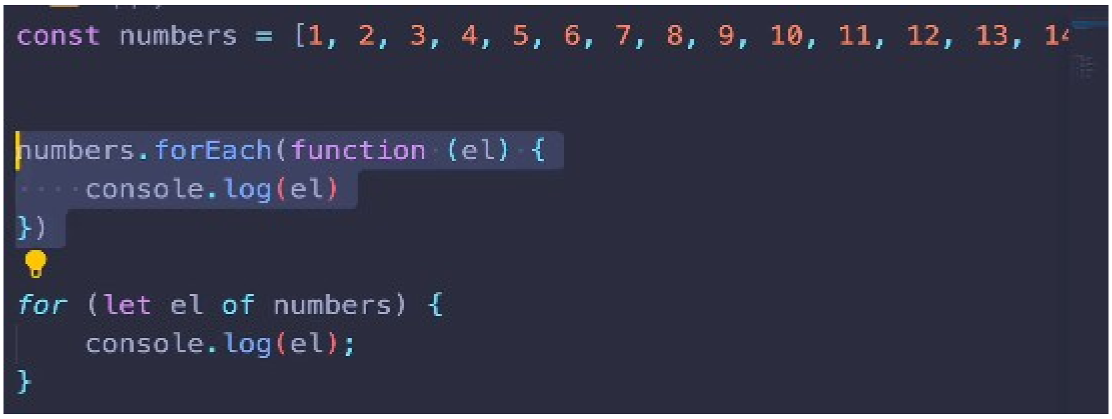
- Use the new arrow function syntax
- Understand and use these methods:
  - forEach
  - map
  - filter
  - find
  - reduce
  - some
  - every



FOREACH

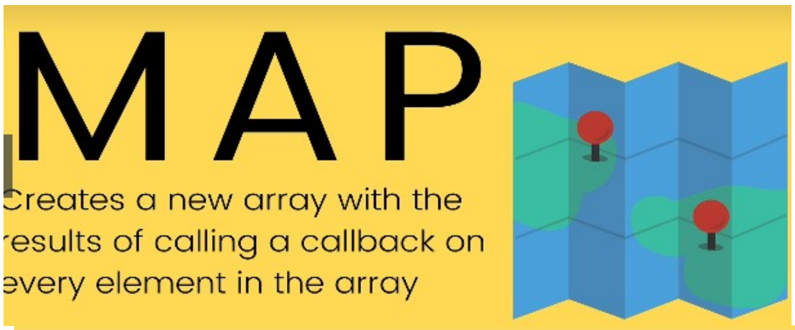
```
const nums = [9, 8, 7, 6, 5, 4, 3, 2, 1];

nums.forEach(function (n) {
  console.log(n * n)
  //prints: 81, 64, 49, 36, 25, 16, 9, 4, 1
});

nums.forEach(function (el) {
  if (el % 2 === 0) {
    console.log(el)
    //prints: 8, 6, 4, 2
  }
})
```

Accepts a callback function.
Calls the function once per element in the array.



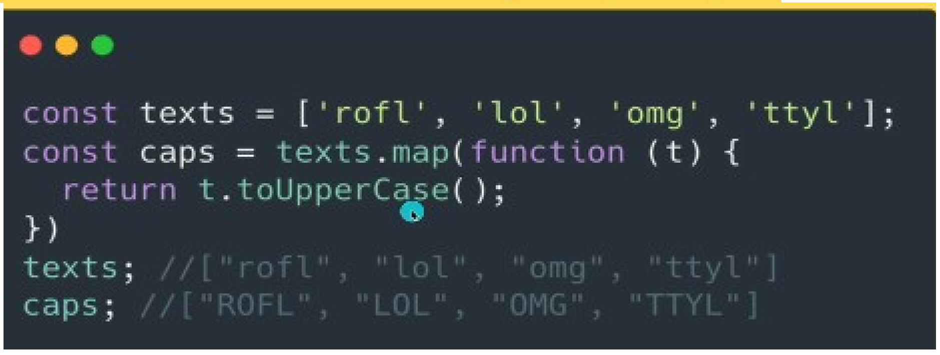
```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14


numbers.forEach(function (el) {
    console.log(el)
})

for (let el of numbers) {
    console.log(el);
}
```

for each is older way of looping.



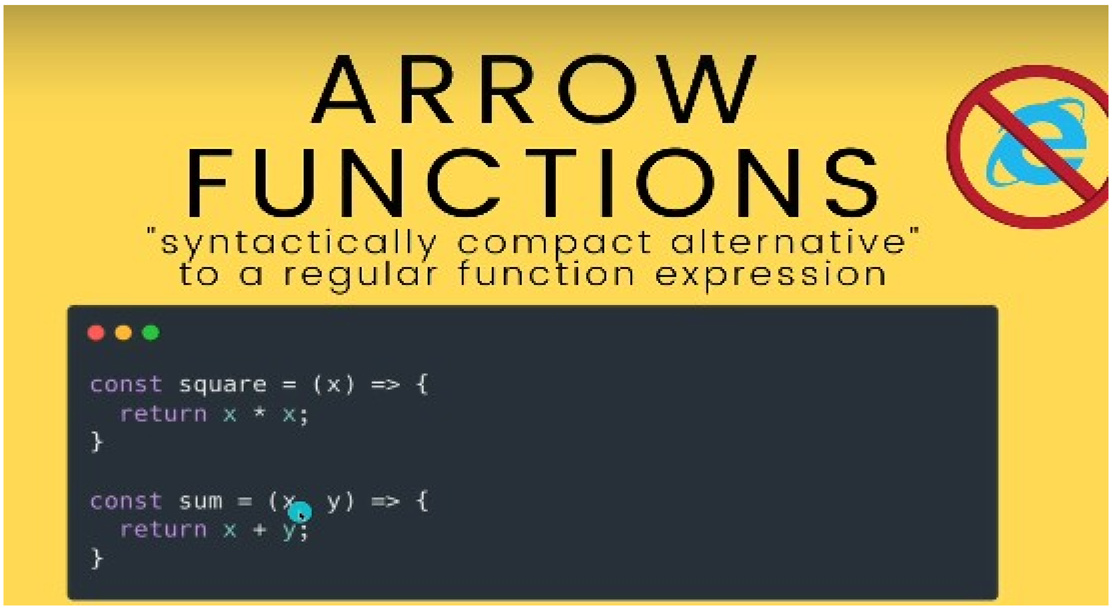
MAP

Creates a new array with the results of calling a callback on every element in the array



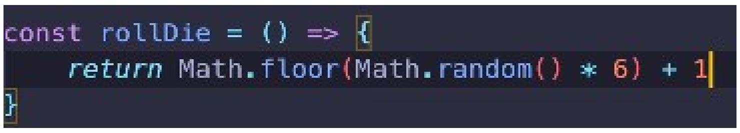
```
const texts = ['rofl', 'lol', 'omg', 'ttyl'];
const caps = texts.map(function (t) {
  return t.toUpperCase();
})
texts; //["rofl", "lol", "omg", "ttyl"]
caps; //["ROFL", "LOL", "OMG", "TTYL"]
```

Arrow functions:are newer alternative i.e more compact to write functions


ARROW FUNCTIONS
"syntactically compact alternative" to a regular function expression

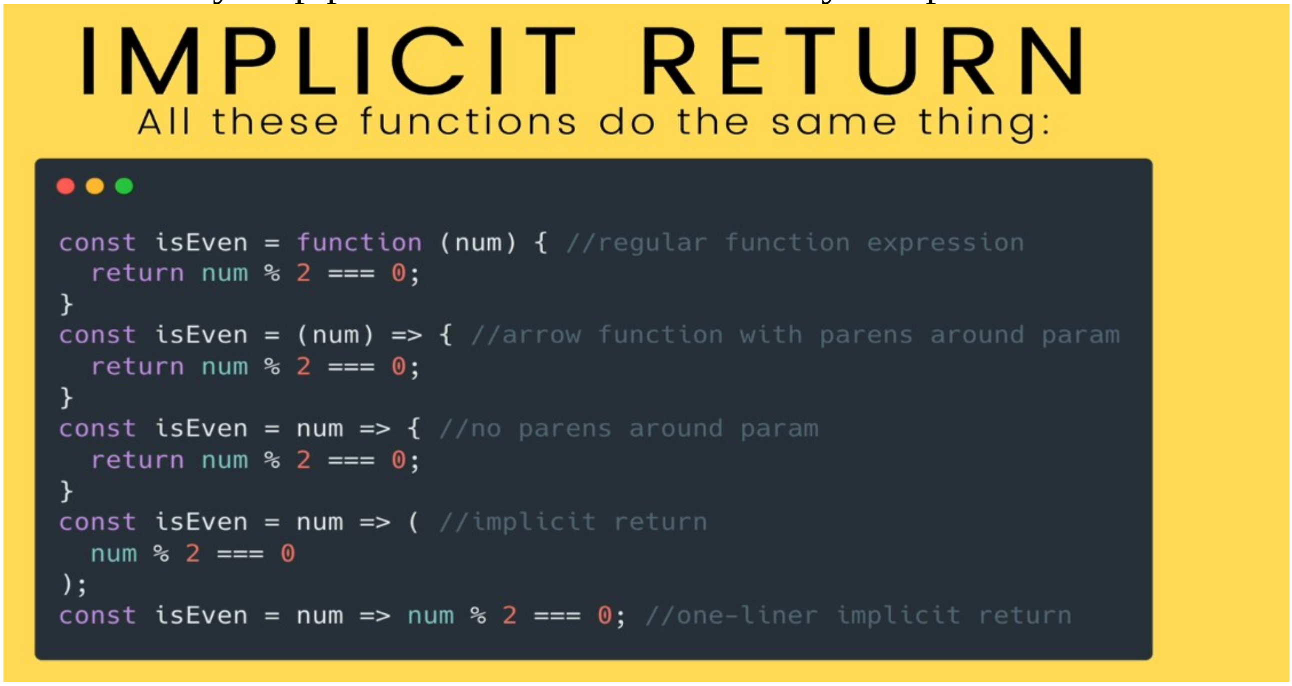
```
const square = (x) => {
  return x * x;
}

const sum = (x, y) => {
  return x + y;
}
```

If you have no parameters to input, you still have to give empty ()



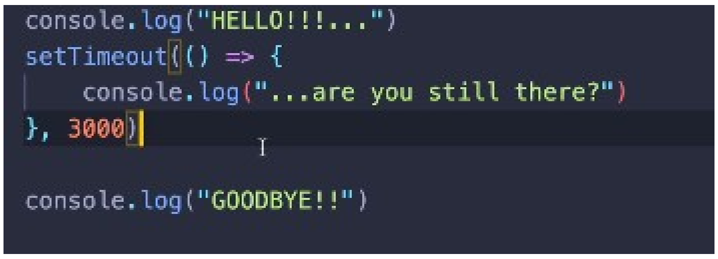
```
const rollDie = () => {
    return Math.floor(Math.random() * 6) + 1
}
```

You can only skip paranthesis when there is only one parameter.



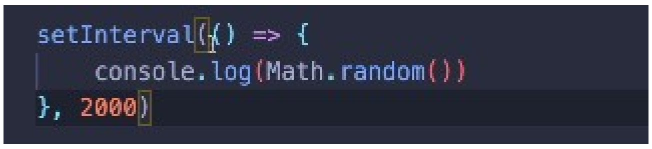
```
const isEven = function (num) { //regular function expression
  return num % 2 === 0;
}
const isEven = (num) => { //arrow function with parens around param
  return num % 2 === 0;
}
const isEven = num => { //no parens around param
  return num % 2 === 0;
}
const isEven = num => ( //implicit return
  num % 2 === 0
);
const isEven = num => num % 2 === 0; //one-liner implicit return
```

SetInterval and setTimeout



```
console.log("HELLO!!!...")
setTimeout(() => {
    console.log("...are you still there?")
}, 3000)

console.log("GOODBYE!!")
```

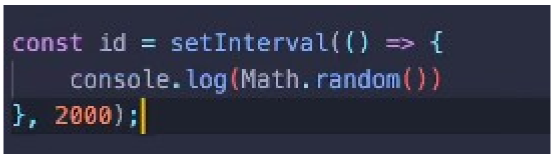
>>> Hello
GOODBYE!!
... are you still there?



```
setInterval(() => {
    console.log(Math.random())
}, 2000)
```

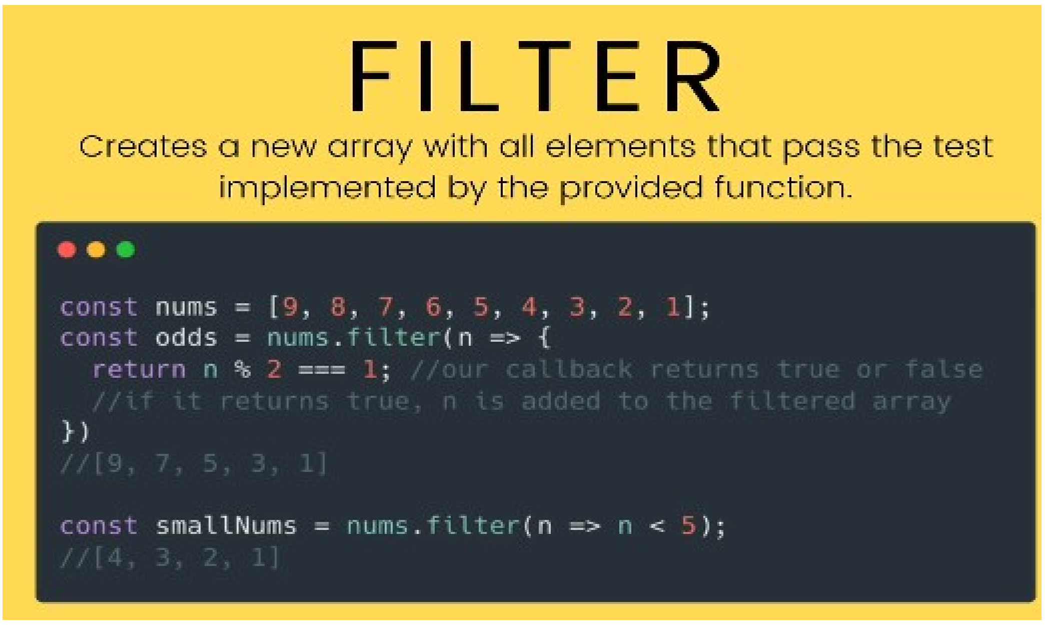
setInterval runs that func every 2s.
So
0.12132421
0.4231313
0.5323324
...
...
...

How to stop? Is by using #id and how to do that



```
const id = setInterval(() => {
    console.log(Math.random())
}, 2000);
```

i.e
clearInterval(id)

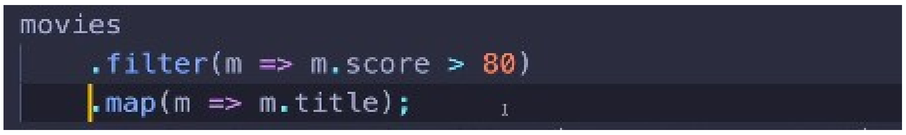


FILTER

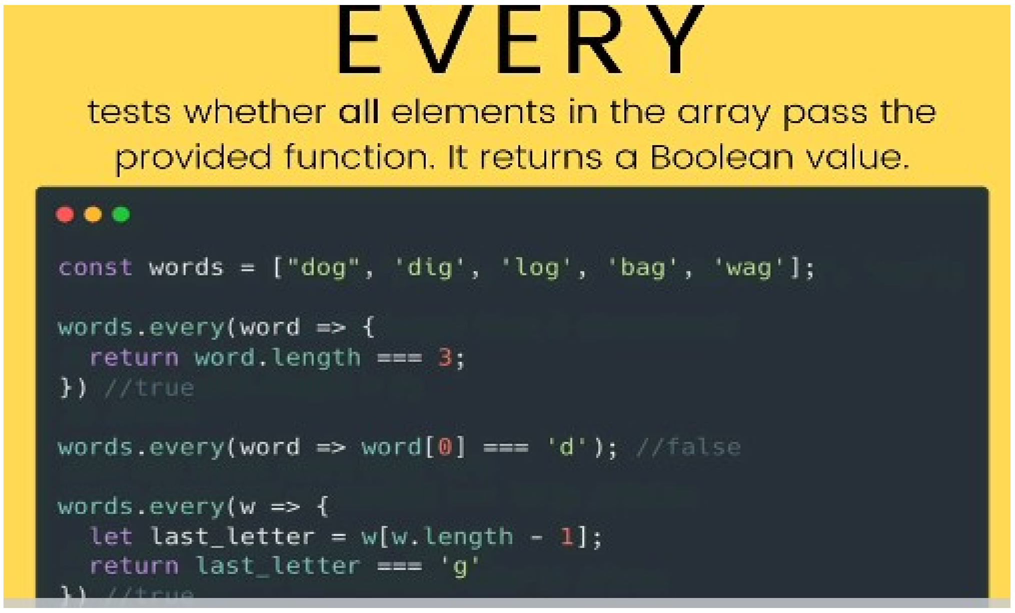
Creates a new array with all elements that pass the test implemented by the provided function.

```
const nums = [9, 8, 7, 6, 5, 4, 3, 2, 1];
const odds = nums.filter(n => {
  return n % 2 === 1; //our callback returns true or false
  //if it returns true, n is added to the filtered array
})
//[9, 7, 5, 3, 1]

const smallNums = nums.filter(n => n < 5);
//[4, 3, 2, 1]
```

You can also indent



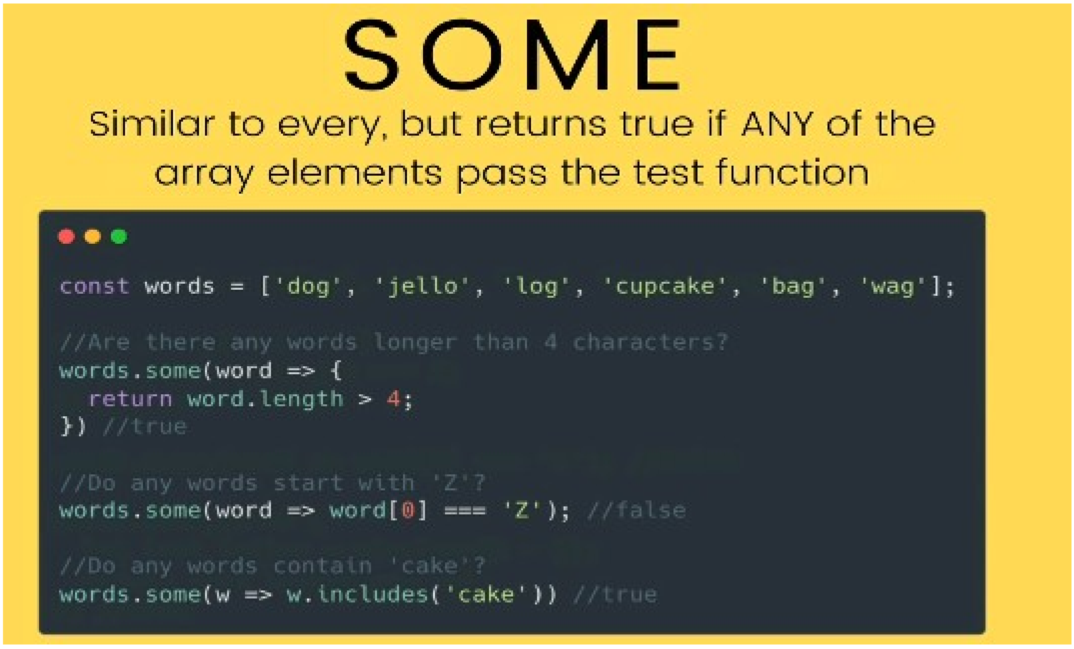
```
movies
    .filter(m => m.score > 80)
    .map(m => m.title);
```



EVERY

tests whether all elements in the array pass the provided function. It returns a Boolean value.

```
const words = ["dog", 'dig', 'log', 'bag', 'wag'];

words.every(word => {
  return word.length === 3;
}) //true

words.every(word => word[0] === 'd'); //false

words.every(w => {
  let last_letter = w[w.length - 1];
  return last_letter === 'g'
}) //true
```



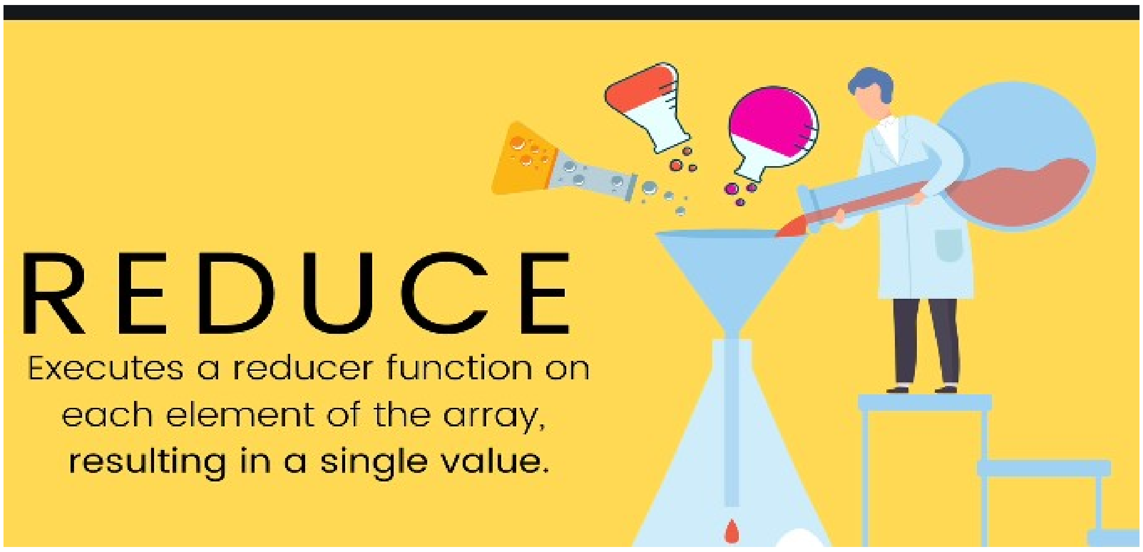
SOME

Similar to every, but returns true if ANY of the array elements pass the test function

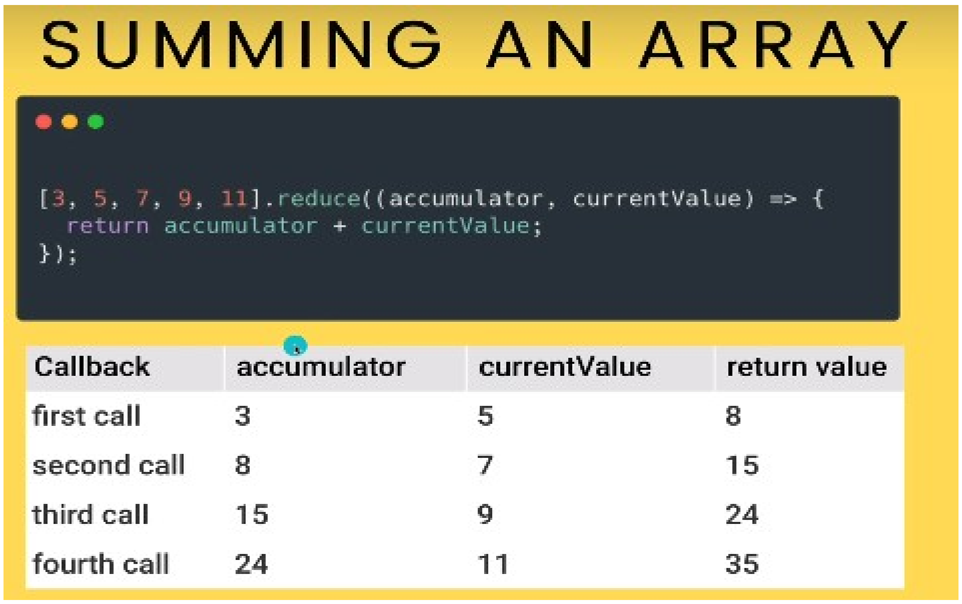
```
const words = ['dog', 'jello', 'log', 'cupcake', 'bag', 'wag'];

//Are there any words longer than 4 characters?
words.some(word => {
  return word.length > 4;
}) //true

//Do any words start with 'Z'?
words.some(word => word[0] === 'Z'); //false

//Do any words contain 'cake'?
words.some(w => w.includes('cake')) //true
```

So, some and any are like filter but with AND and OR operator.

Reduce

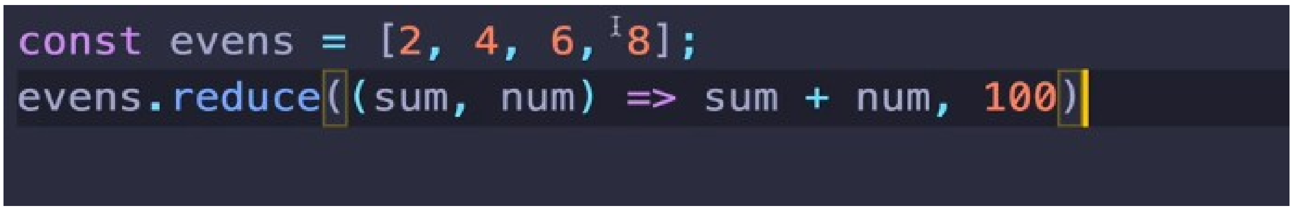

REDUCE
Executes a reducer function on each element of the array, resulting in a single value.

how it does that, is up to us.
We have to provide a Reducer func wher we have 2 parameters.


SUMMING AN ARRAY

```
[3, 5, 7, 9, 11].reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
});
```

Callback	accumulator	currentValue	return value
first call	3	5	8
second call	8	7	15
third call	15	9	24
fourth call	24	11	35



```
const evens = [2, 4, 6, 8];
evens.reduce((sum, num) => sum + num, 100)
```

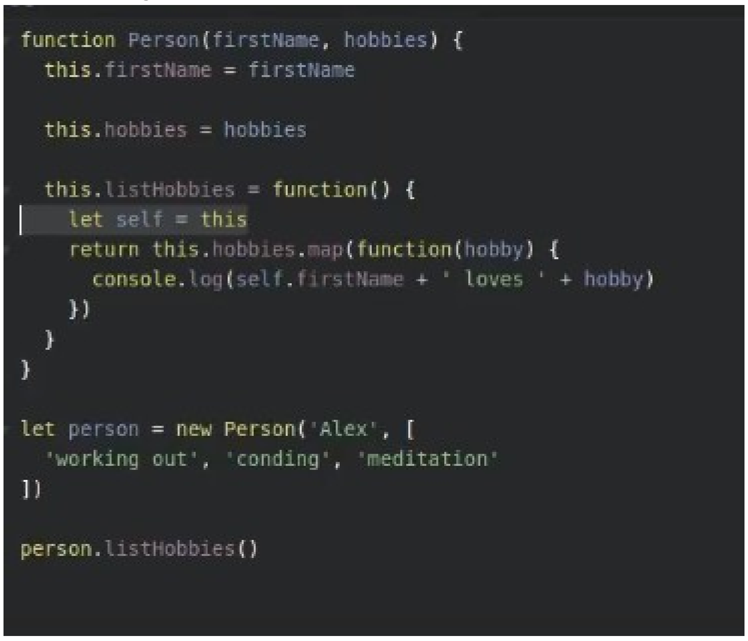
Here, we are setting the inital value of sum i.e 100.

Arrow function and this

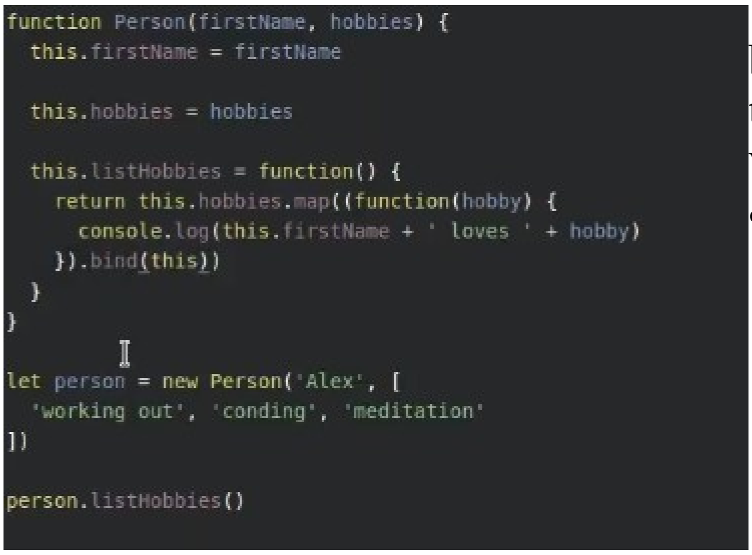
youtube video

at time 5:52,

Use arrow func with ‘this’ keyword.
They allow you to have access to this keyword that in this case refer to person object within the func.

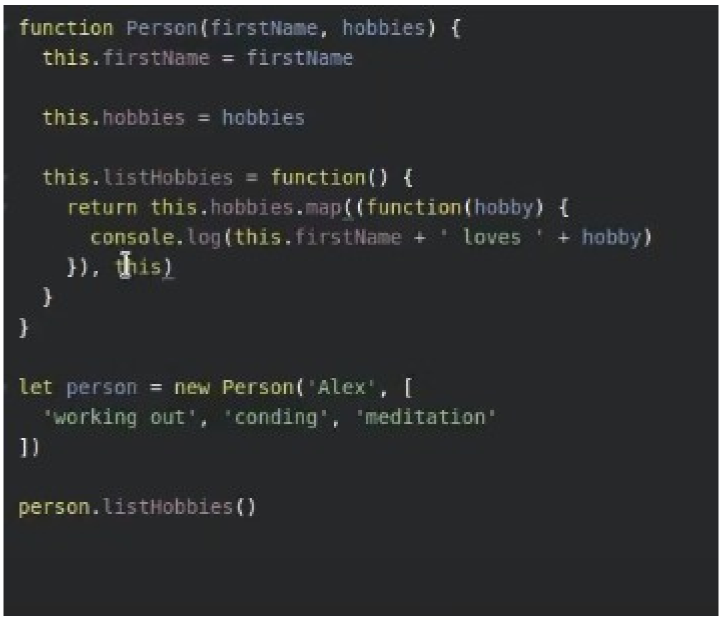


```
function Person(firstName, hobbies) {
  this.firstName = firstName

  this.hobbies = hobbies

  this.listHobbies = function() {
    let self = this
    return this.hobbies.map(function(hobby) {
      console.log(self.firstName + ' loves ' + hobby)
    })
  }
}

let person = new Person('Alex', [
  'working out', 'conding', 'meditation'
])

person.listHobbies()
```

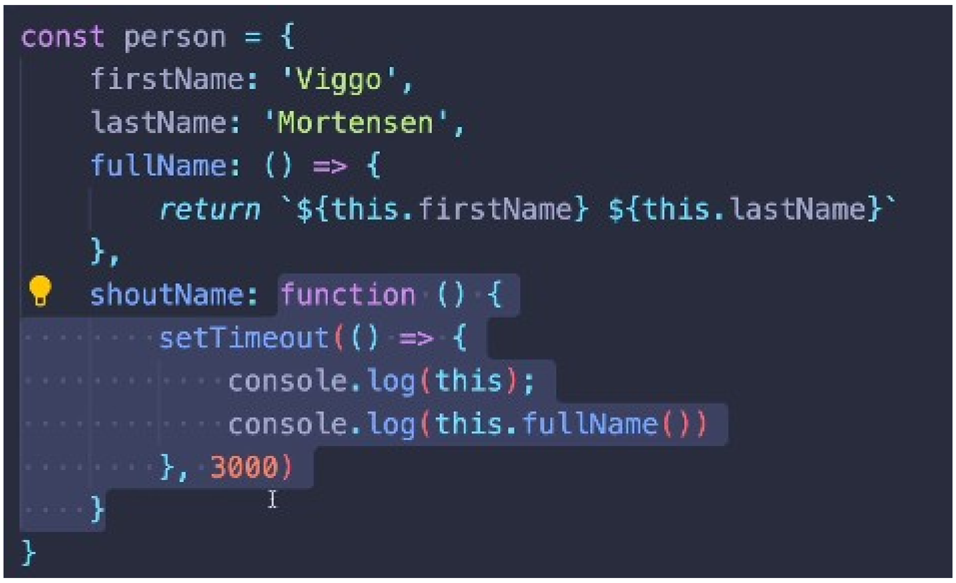


```
function Person(firstName, hobbies) {
  this.firstName = firstName

  this.hobbies = hobbies

  this.listHobbies = function() {
    return this.hobbies.map((function(hobby) {
      console.log(this.firstName + ' loves ' + hobby)
    }).bind(this))
  }
}

let person = new Person('Alex', [
  'working out', 'conding', 'meditation'
])

person.listHobbies()
```

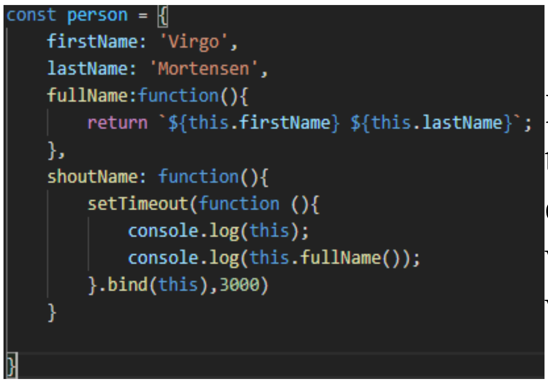
basically the bind tells the this keyword is the same this within the outer func.just like above



```
function Person(firstName, hobbies) {
  this.firstName = firstName

  this.hobbies = hobbies

  this.listHobbies = function() {
    return this.hobbies.map((function(hobby) {
      console.log(this.firstName + ' loves ' + hobby)
    }), this)
  }
}

let person = new Person('Alex', [
  'working out', 'conding', 'meditation'
])

person.listHobbies()
```



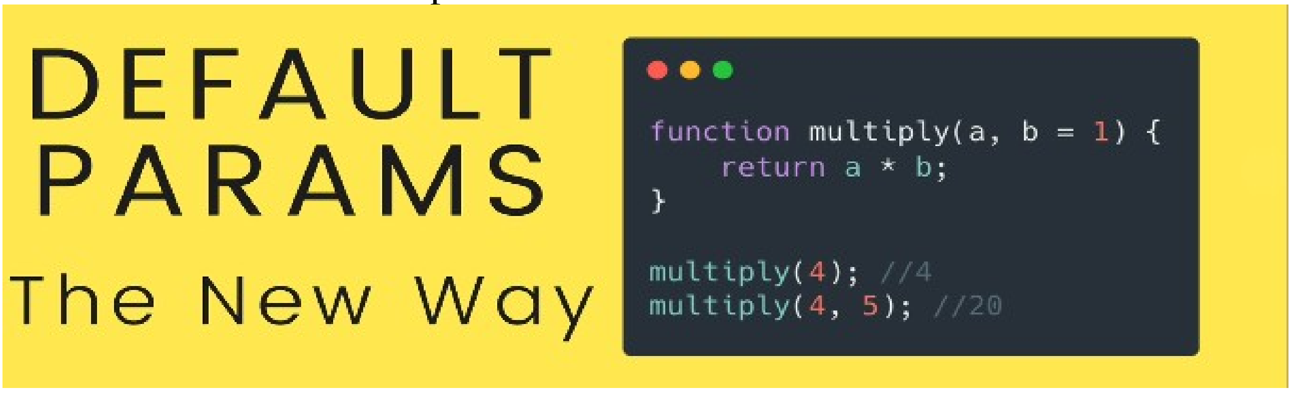
```
const person = {
    firstName: 'Viggo',
    lastName: 'Mortensen',
    fullName: () => {
        return `${this.firstName} ${this.lastName}`
    },
    shoutName: function () {
        setTimeout(() => {
            console.log(this);
            console.log(this.fullName())
        }, 3000)
    }
}
```



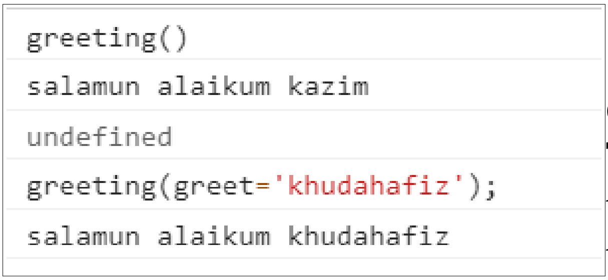
```
const person = {
    firstName: 'Virgo',
    lastName: 'Mortensen',
    fullName:function(){
        return `${this.firstName} ${this.lastName}`;
    },
    shoutName: function(){
        setTimeout(function (){
            console.log(this);
            console.log(this.fullName());
        }.bind(this),3000)
    }
}
```

Inside an arrow func, this refers to where the function was created so, considering person so at level 0 i.e window
whereas in non-arrow func, it works fine.

window.setTimeout() > function ,this doesn’t make sense cuz it looks at window level.
So using arrow func will take us out n look where the func was created. i.e at level 1. i.e window.person.firstname.

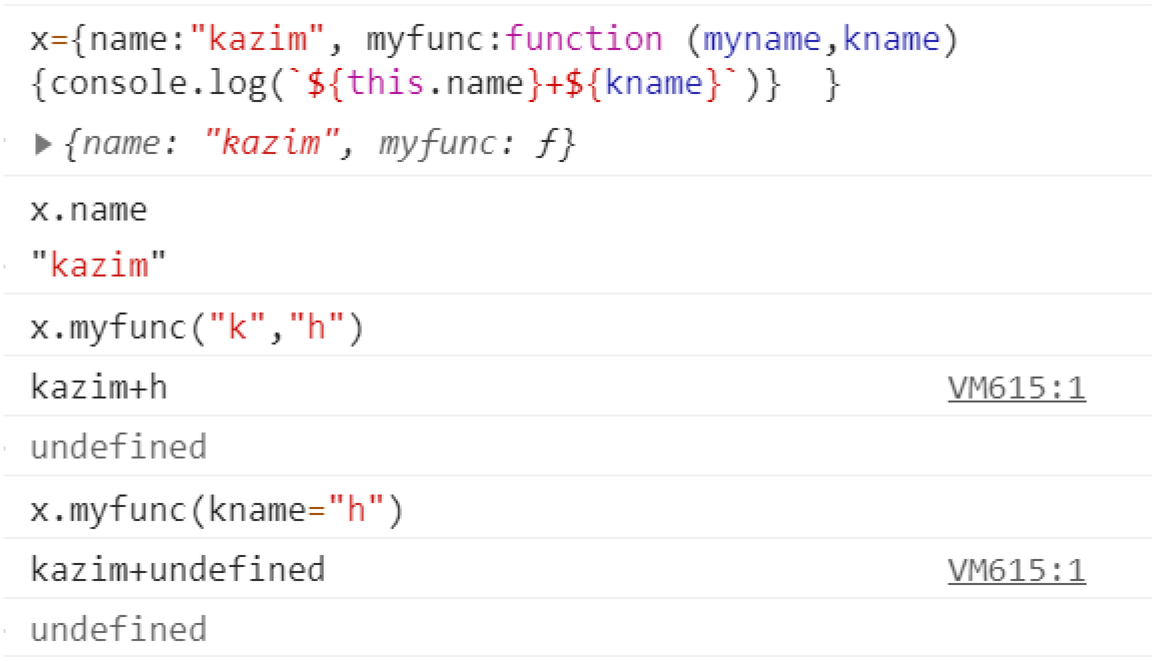


```
function multiply(a, b = 1) {
    return a * b;
}

multiply(4); //4
multiply(4, 5); //20
```

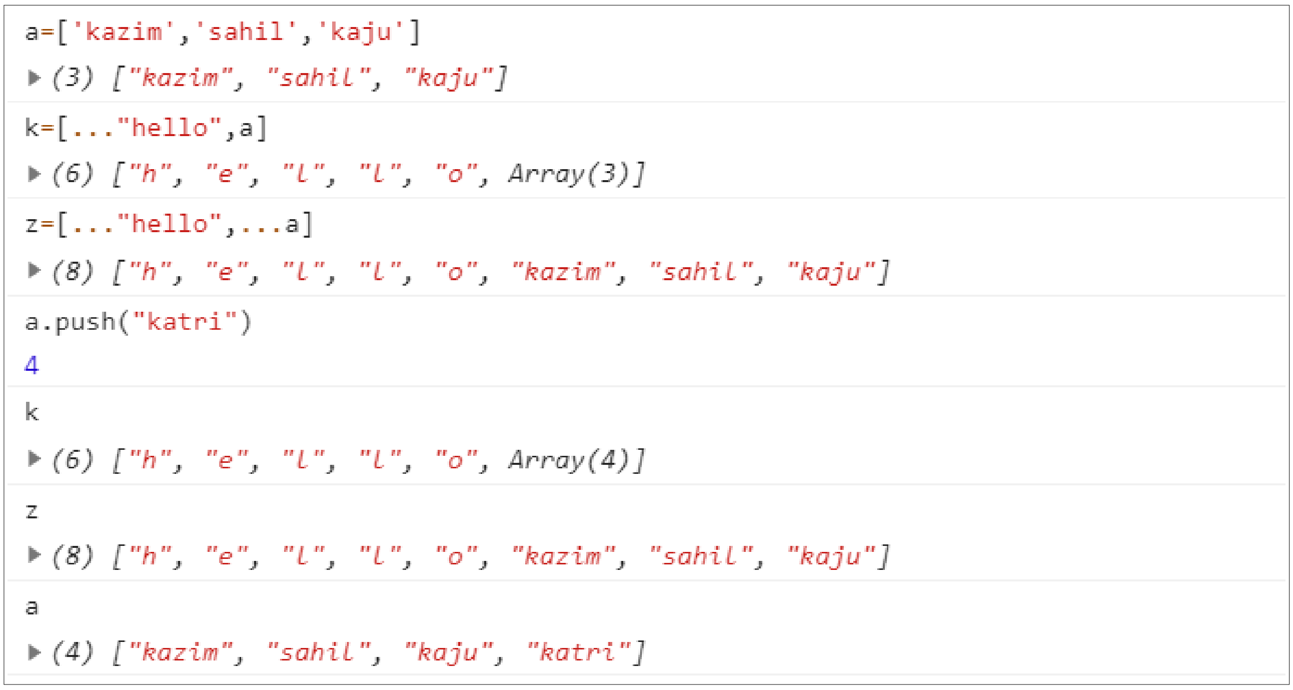


```
greeting()
salamun alaikum kazim
undefined
greeting(greet='khudahafiz');
salamun alaikum khudahafiz
```

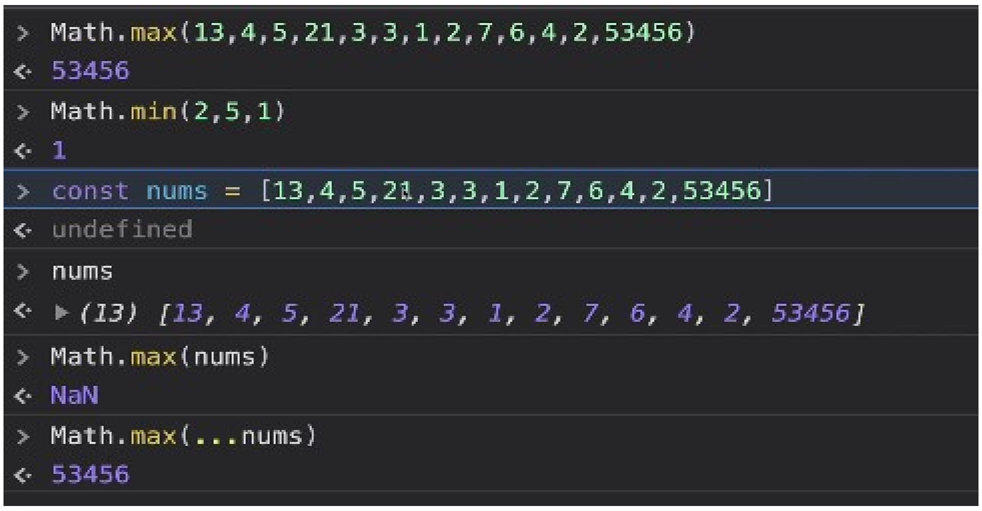
Here, i try to change ‘*greet*’ but it change the ‘*name*’.
Tip: put your tobe default paramater to the end.so if u dont pass it doesn’t get taken by other parameters.



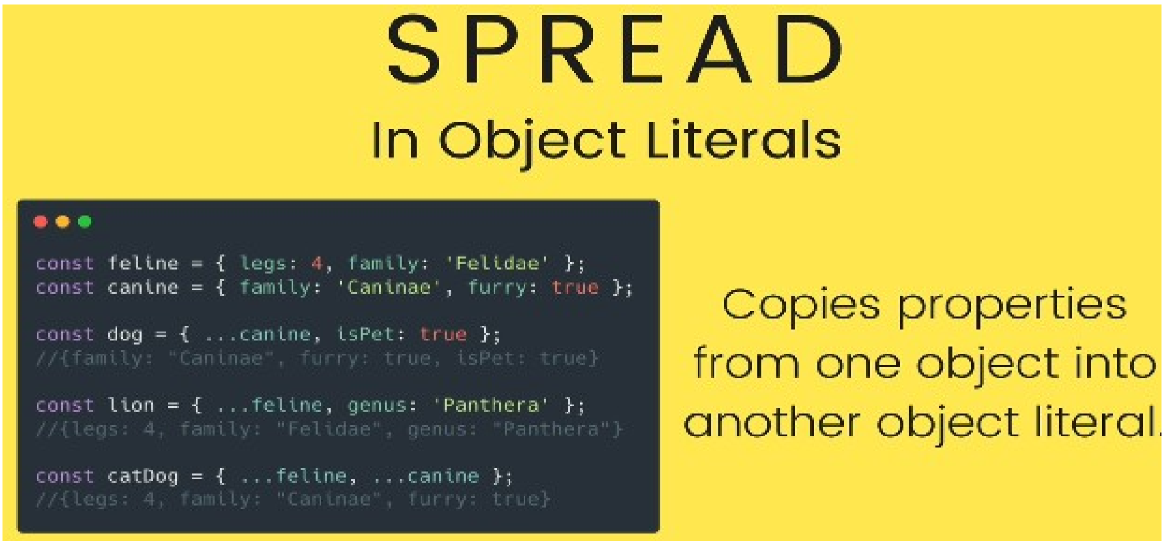
```
x={name:"kazim", myfunc:function (myname,kname)
{console.log(`${this.name}+${kname}`)}  }
▶{name: "kazim", myfunc: ƒ}
x.name
"kazim"
x.myfunc("k","h")
kazim+h                                          VM615:1
undefined
x.myfunc(kname="h")
kazim+undefined                                  VM615:1
undefined
```



```
a=['kazim','sahil','kaju']
▶(3) ["kazim", "sahil", "kaju"]
k=[..."hello",a]
▶(6) ["h", "e", "l", "l", "o", Array(3)]
z=[..."hello",...a]
▶(8) ["h", "e", "l", "l", "o", "kazim", "sahil", "kaju"]
a.push("katri")
4
k
▶(6) ["h", "e", "l", "l", "o", Array(4)]
z
▶(8) ["h", "e", "l", "l", "o", "kazim", "sahil", "kaju"]
a
▶(4) ["kazim", "sahil", "kaju", "katri"]
```



```
> Math.max(13,4,5,21,3,3,1,2,7,6,4,2,53456)
<· 53456
> Math.min(2,5,1)
<· 1
> const nums = [13,4,5,21,3,3,1,2,7,6,4,2,53456]
<· undefined
> nums
<· ▶(13) [13, 4, 5, 21, 3, 3, 1, 2, 7, 6, 4, 2, 53456]
> Math.max(nums)
<· NaN
> Math.max(...nums)
<· 53456
```



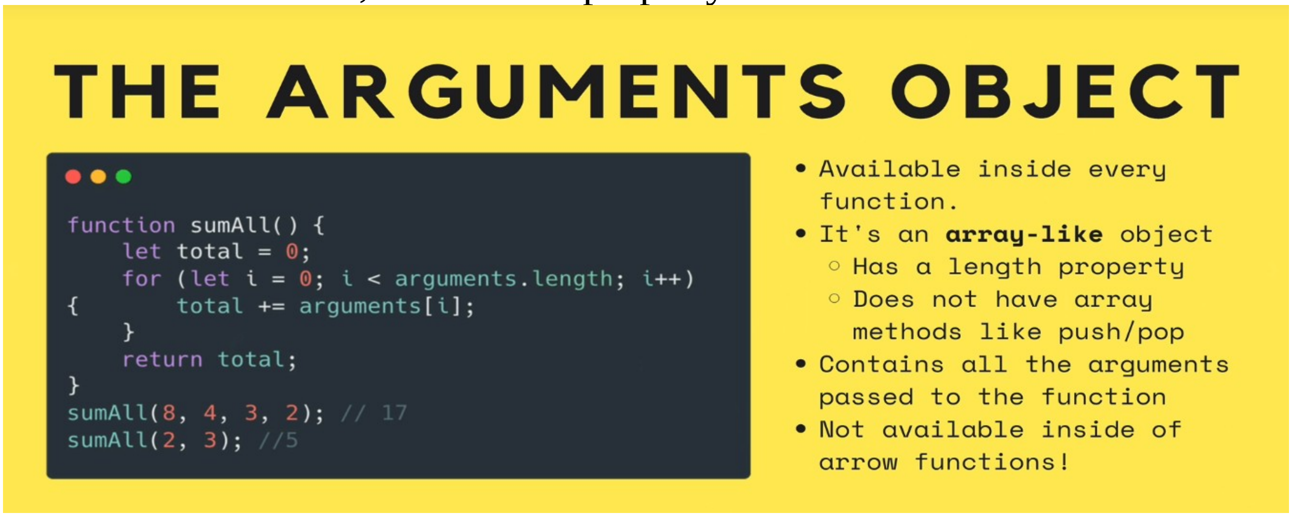
SPREAD

In Object Literals

```
const feline = { legs: 4, family: 'Felidae' };
const canine = { family: 'Caninae', furry: true };

const dog = { ...canine, isPet: true };
//{family: "Caninae", furry: true, isPet: true}

const lion = { ...feline, genus: 'Panthera' };
//{legs: 4, family: "Felidae", genus: "Panthera"}

const catDog = { ...feline, ...canine };
//{legs: 4, family: "Caninae", furry: true}
```

Copies properties from one object into another object literal.

if there is a conflict, the newest property wins.

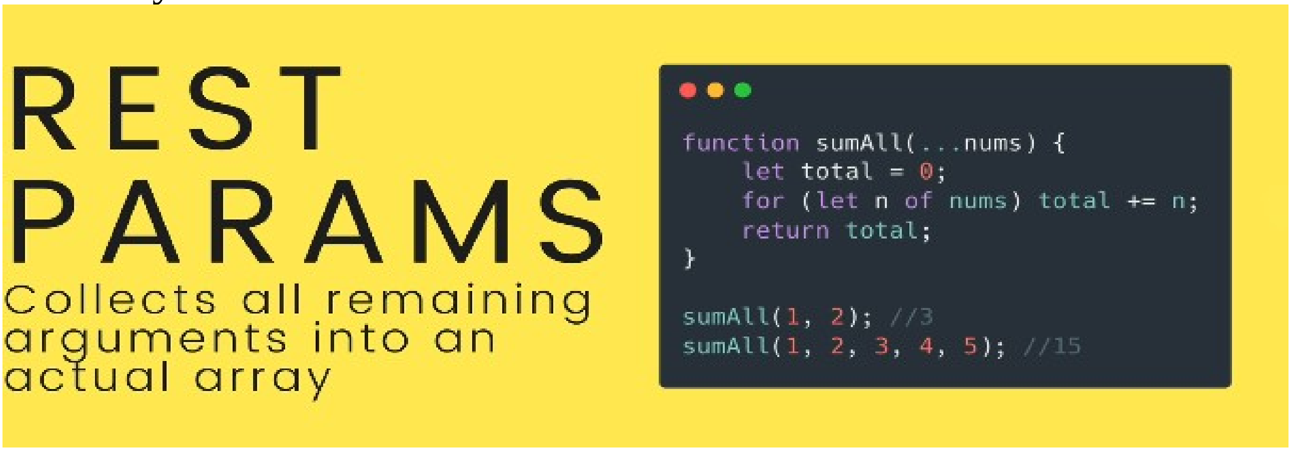


THE ARGUMENTS OBJECT

```
function sumAll() {
    let total = 0;
    for (let i = 0; i < arguments.length; i++)
{       total += arguments[i];
    }
    return total;
}
sumAll(8, 4, 3, 2); // 17
sumAll(2, 3); //5
```

- Available inside every function.
- It's an **array-like** object
  - Has a length property
  - Does not have array methods like push/pop
- Contains all the arguments passed to the function
- Not available inside of arrow functions!

In arguments object isnt an array. so you can’t use map,reduce,sort.
So what you could do is..



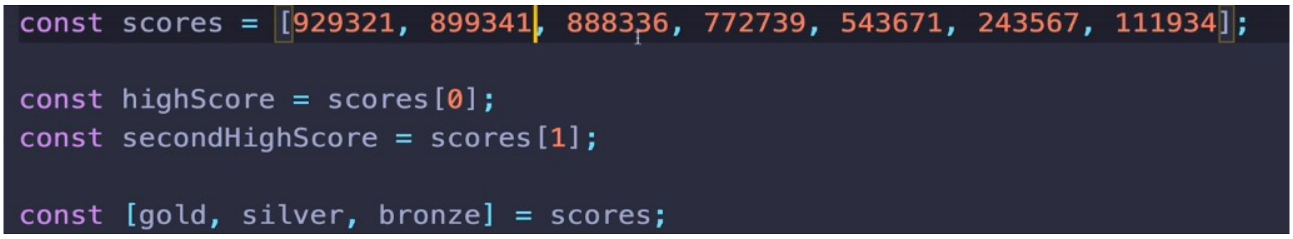
REST PARAMS

Collects all remaining arguments into an actual array

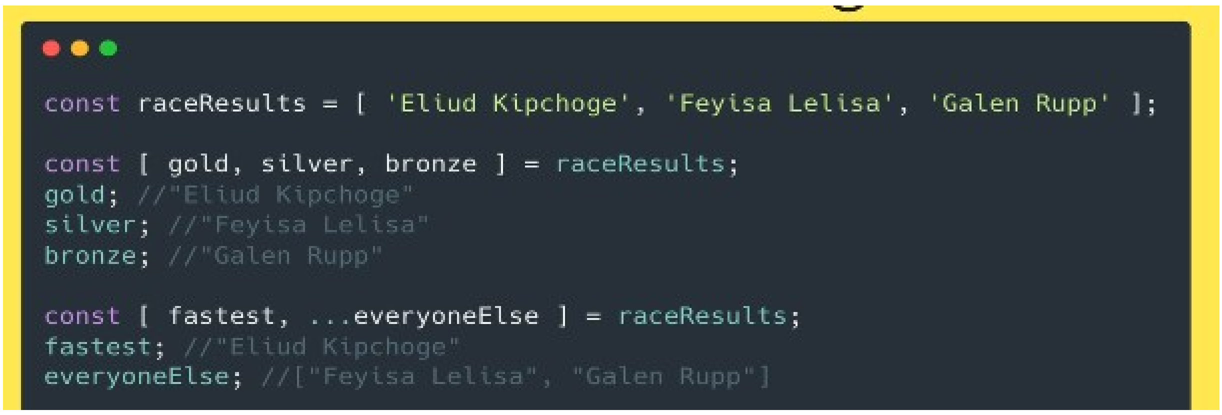
```
function sumAll(...nums) {
    let total = 0;
    for (let n of nums) total += n;
    return total;
}

sumAll(1, 2); //3
sumAll(1, 2, 3, 4, 5); //15
```

```
function x(num){
        ...
}
x(3,5)                              <!-- here num is 3 and 5 is ingored.
```

```
function x(...num){                 <!-- here all arg are considered and put in
                                                                         array
        console.log(num)
}
x(3,35,12,22)
>>> [3,35,12,22]
```



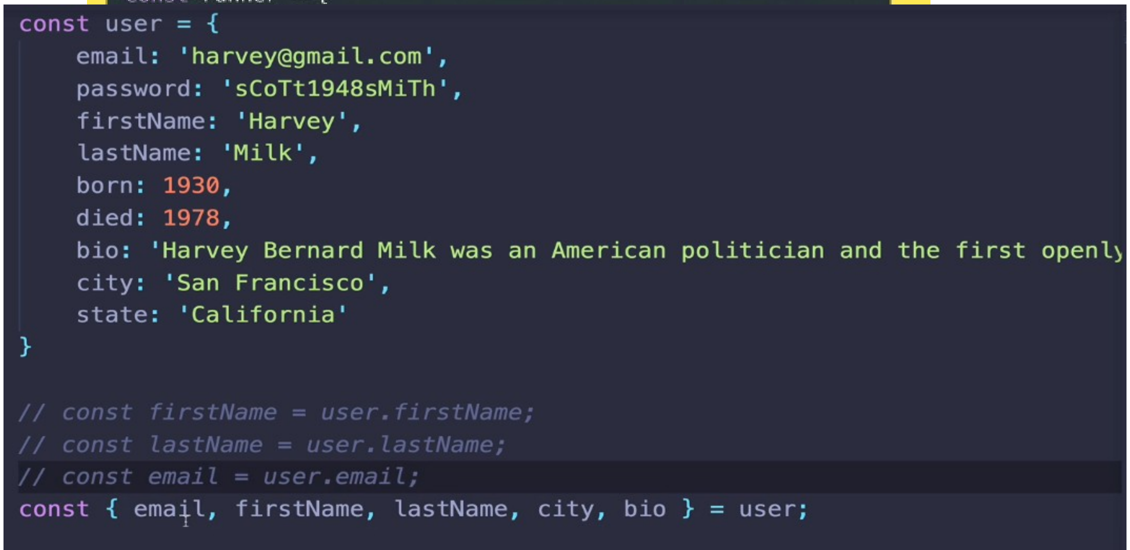
```
const scores = [929321, 899341, 888336, 772739, 543671, 243567, 111934];

const highScore = scores[0];
const secondHighScore = scores[1];

const [gold, silver, bronze] = scores;
```



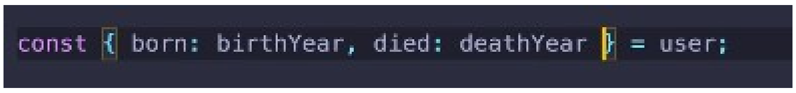
```
const raceResults = [ 'Eliud Kipchoge', 'Feyisa Lelisa', 'Galen Rupp' ];

const [ gold, silver, bronze ] = raceResults;
gold; //"Eliud Kipchoge"
silver; //"Feyisa Lelisa"
bronze; //"Galen Rupp"

const [ fastest, ...everyoneElse ] = raceResults;
fastest; //"Eliud Kipchoge"
everyoneElse; //["Feyisa Lelisa", "Galen Rupp"]
```

OBJECT
Destructuring

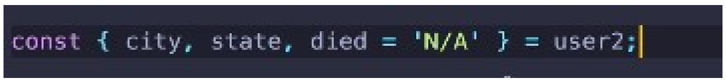
```
const runner = {
```



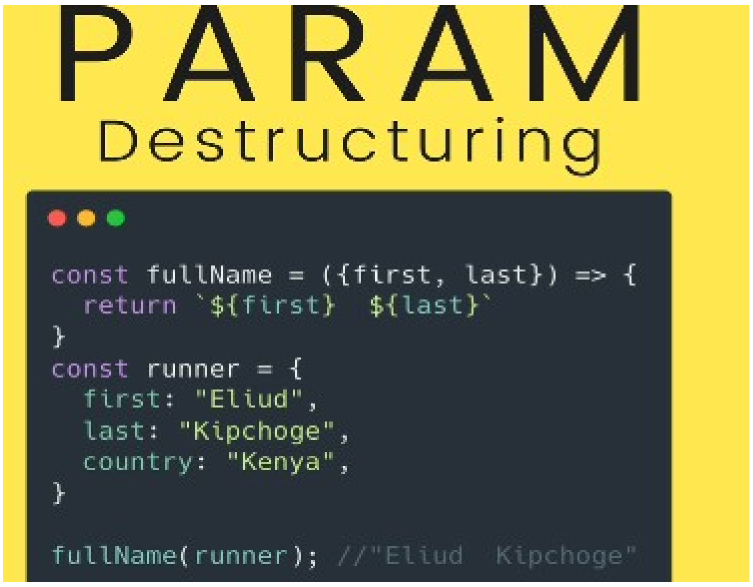
```
const user = {
    email: 'harvey@gmail.com',
    password: 'sCoTt1948sMiTh',
    firstName: 'Harvey',
    lastName: 'Milk',
    born: 1930,
    died: 1978,
    bio: 'Harvey Bernard Milk was an American politician and the first openly
    city: 'San Francisco',
    state: 'California'
}

// const firstName = user.firstName;
// const lastName = user.lastName;
// const email = user.email;
const { email, firstName, lastName, city, bio } = user;
```



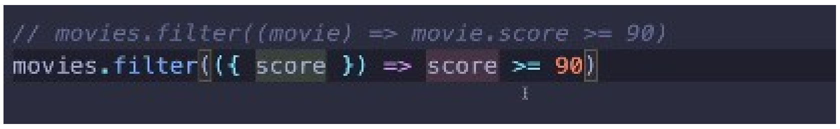
```
const { born: birthYear, died: deathYear } = user;
```



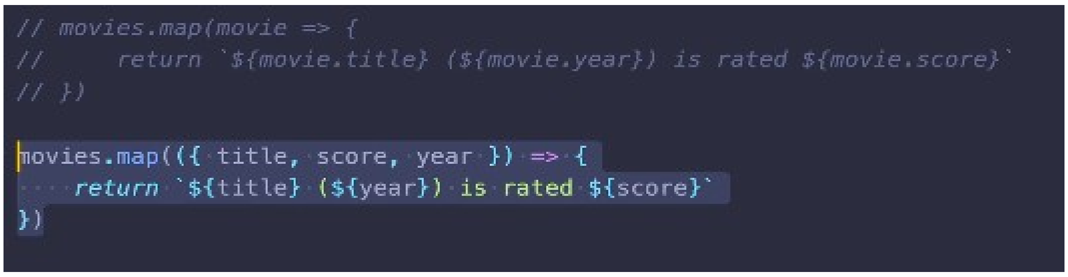
```
const { city, state, died = 'N/A' } = user2;
```



```
const fullName = ({first, last}) => {
  return `${first}  ${last}`
}
const runner = {
  first: "Eliud",
  last: "Kipchoge",
  country: "Kenya",
}

fullName(runner); //"Eliud  Kipchoge"
```

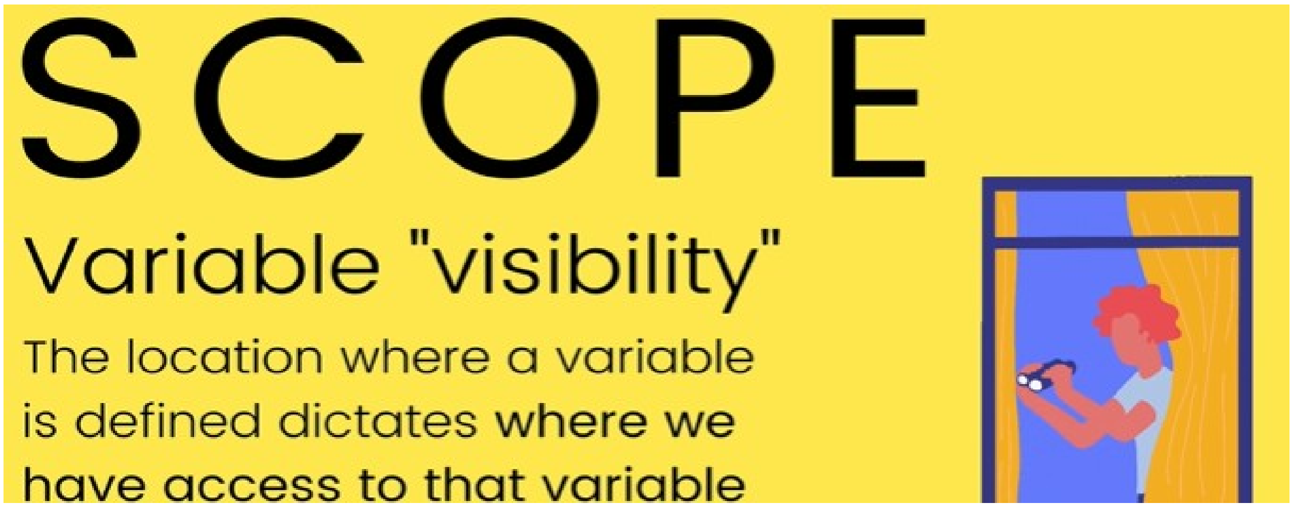


```
// movies.filter((movie) => movie.score >= 90)
movies.filter(({ score }) => score >= 90)
```



```
// movies.map(movie => {
//     return `${movie.title} (${movie.year}) is rated ${movie.score}`
// })

movies.map(({ title, score, year }) => {
    return `${title} (${year}) is rated ${score}`
})
```

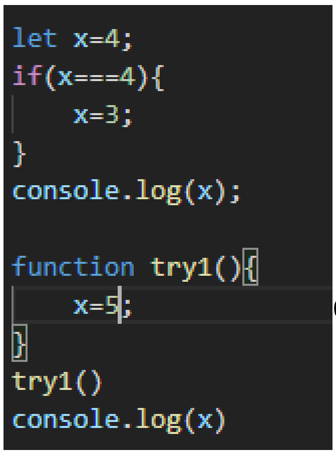
SCOPE



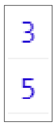
SCOPE

Variable "visibility"

The location where a variable is defined dictates where we have access to that variable

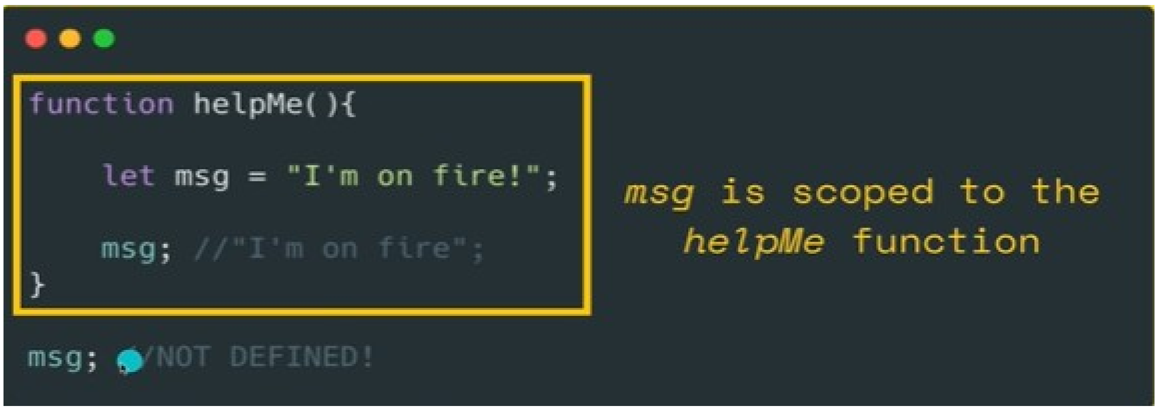


```
let x=4;
if(x===4){
    x=3;
}
console.log(x);

function try1(){
    x=5;
}
try1()
console.log(x)
```



```
3
5
```

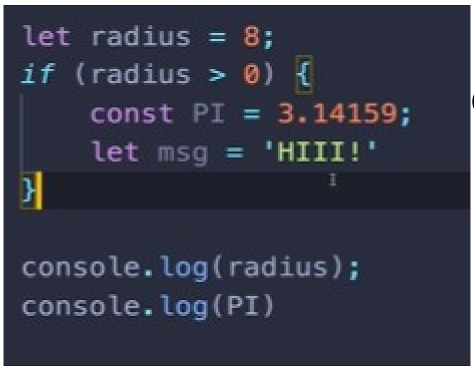
variable defined globally are mutable in any enclosing brackets.



```
function helpMe(){

    let msg = "I'm on fire!";

    msg; //"I'm on fire";
}

msg; //NOT DEFINED!
```

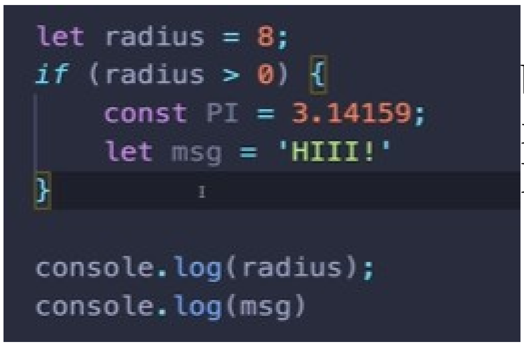
msg is scoped to the *helpMe* function

Var is scoped to the enclosing function if they are not they are part of the global scope.



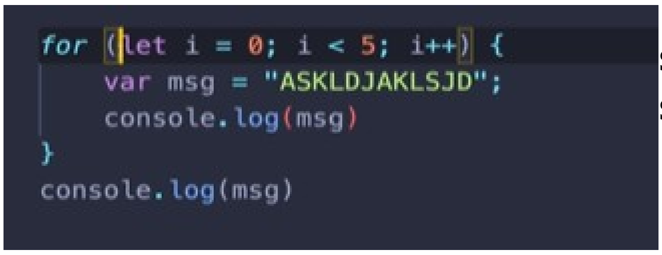
```
let radius = 8;
if (radius > 0) {
    const PI = 3.14159;
    let msg = 'HIII!'
}

console.log(radius);
console.log(PI)
```

> 8
error



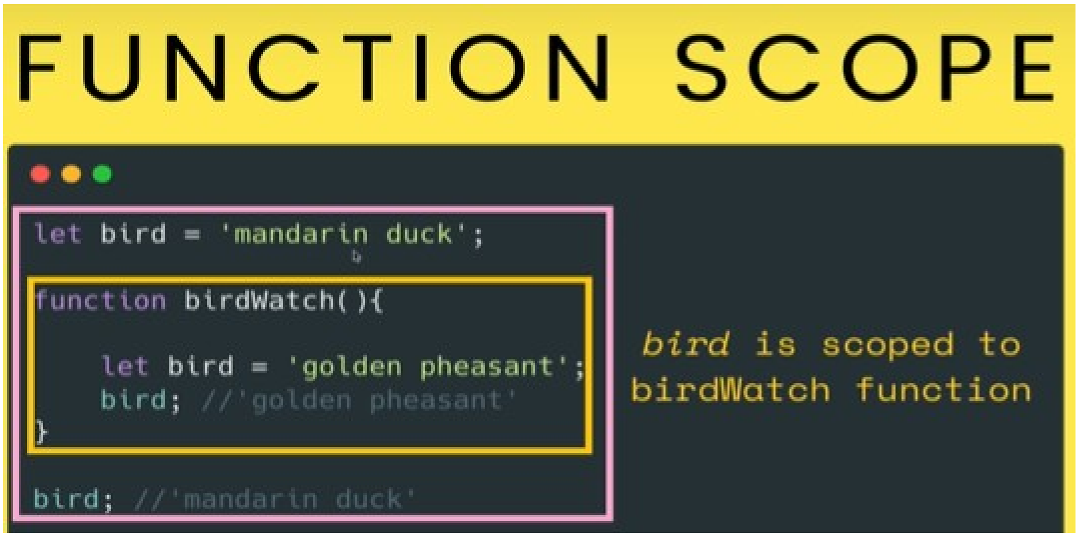
```
let radius = 8;
if (radius > 0) {
    const PI = 3.14159;
    let msg = 'HIII!'
}

console.log(radius);
console.log(msg)
```

Block refers to any time u see curly braces but except for a fucntion. Block includes conditional most commonly but also loop.



```
for (let i = 0; i < 5; i++) {
    var msg = "ASKLDJAKLSJD";
    console.log(msg)
}
console.log(msg)
```

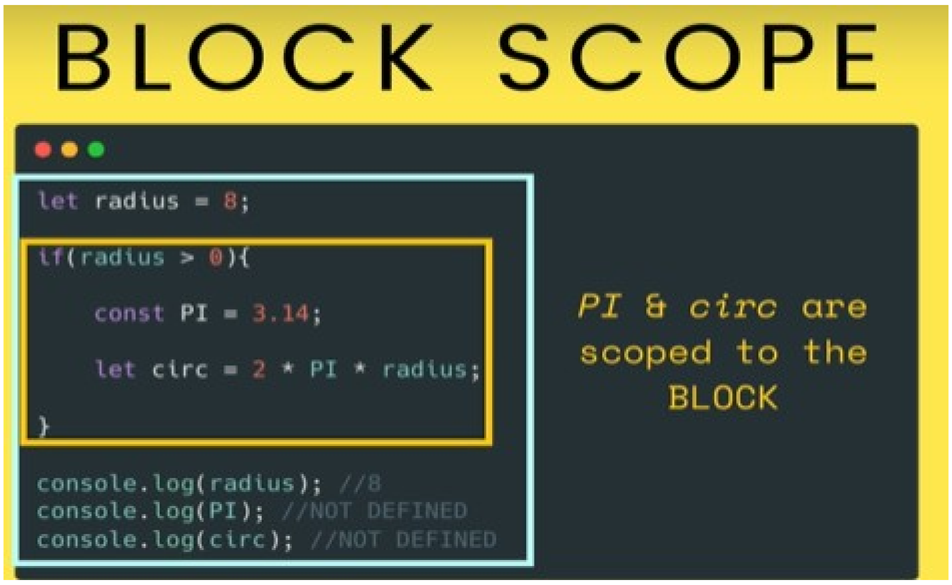
When i use var, my variables are scoped to function but they are not scoped to block.



FUNCTION SCOPE

```
let bird = 'mandarin duck';

function birdWatch(){

    let bird = 'golden pheasant';
    bird; //'golden pheasant'
}

bird; //'mandarin duck'
```

bird is scoped to birdWatch function

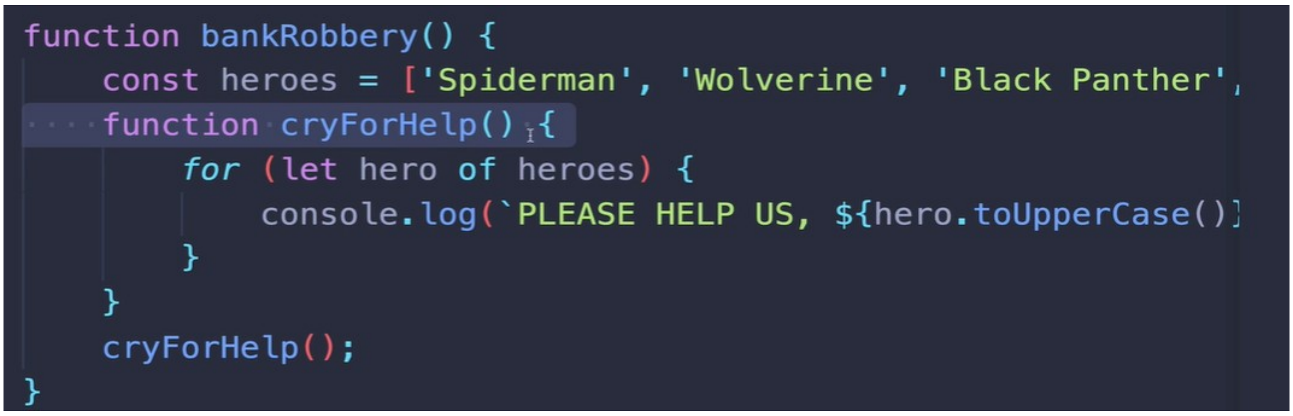


BLOCK SCOPE

```
let radius = 8;

if(radius > 0){

    const PI = 3.14;

    let circ = 2 * PI * radius;

}

console.log(radius); //8
console.log(PI); //NOT DEFINED
console.log(circ); //NOT DEFINED
```

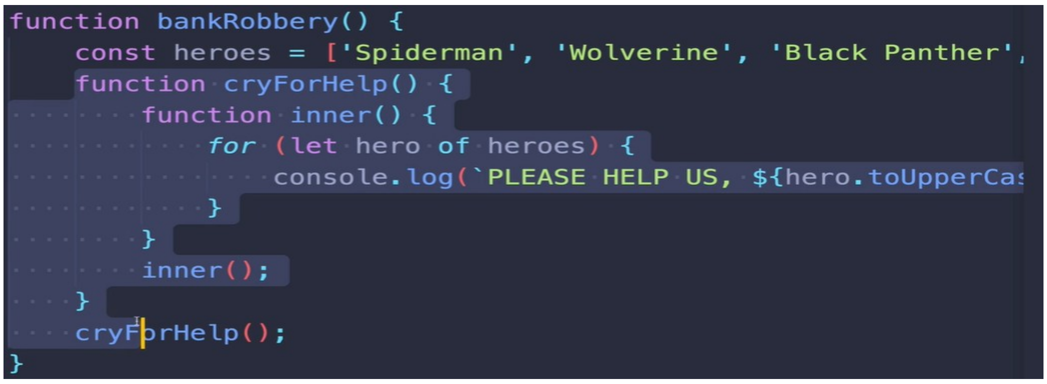
PI & circ are scoped to the BLOCK

Lexical scope



```
function bankRobbery() {
    const heroes = ['Spiderman', 'Wolverine', 'Black Panther',
    function cryForHelp() {
        for (let hero of heroes) {
            console.log(`PLEASE HELP US, ${hero.toUpperCase()}
        }
    }
    cryForHelp();
}
```

 Some inner func nested inside a parent func has access to the scope the var define in the scope of the outer func.
Above works as you imagine
but what about below e.g

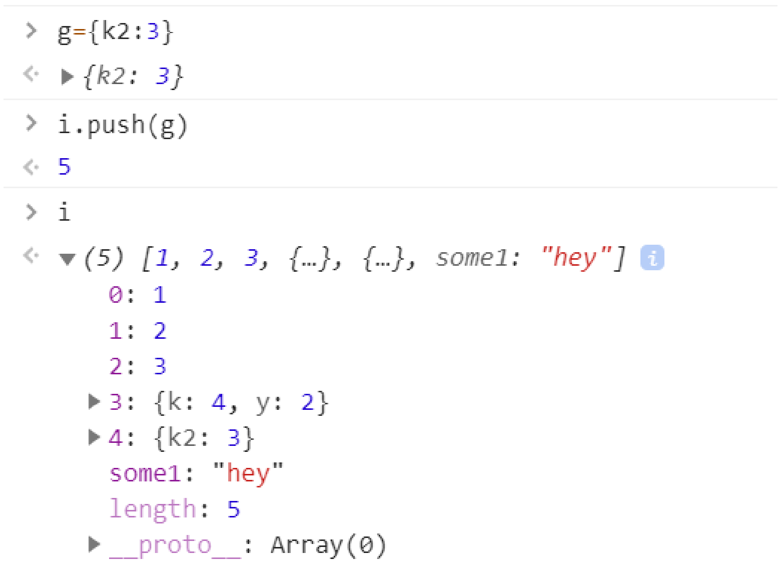


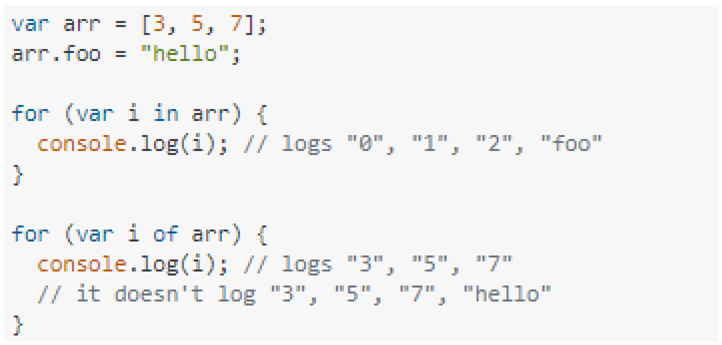
```
function bankRobbery() {
    const heroes = ['Spiderman', 'Wolverine', 'Black Panther',
    function cryForHelp() {
        function inner() {
            for (let hero of heroes) {
                console.log(`PLEASE HELP US, ${hero.toUpperCas
            }
        }
        inner();
    }
    cryForHelp();
}
```

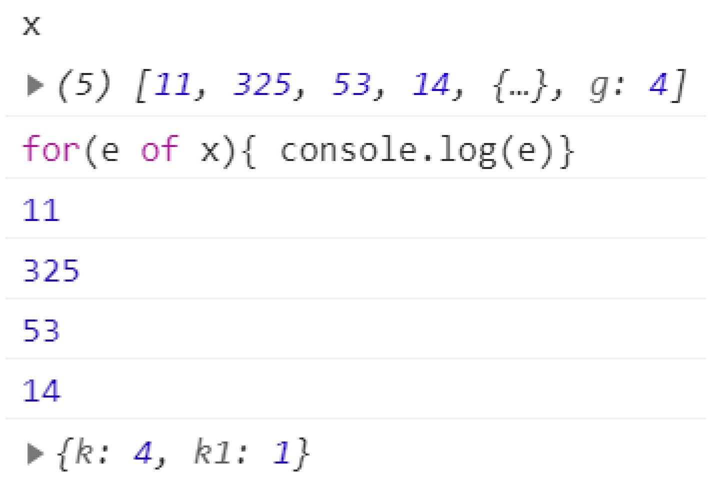
Yes, that also work as you imagine, lol.

Another weird

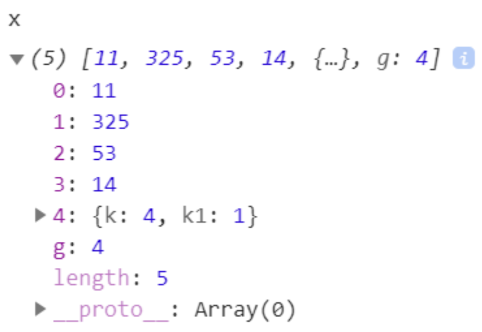
```
var i=[1,2,3]
undefined
i.some1='hey'
"hey"
i
▶(3) [1, 2, 3, some1: "hey"]
```

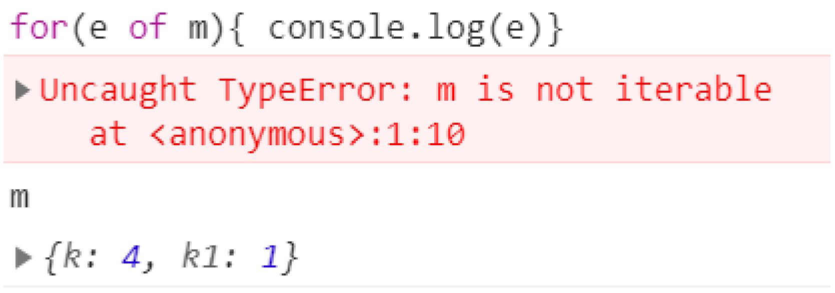

```
> g={k2:3}
<· ▶{k2: 3}
> i.push(g)
<· 5
> i
<· ▼(5) [1, 2, 3, {…}, {…}, some1: "hey"]
      0: 1
      1: 2
      2: 3
    ▶3: {k: 4, y: 2}
    ▶4: {k2: 3}
      some1: "hey"
      length: 5
    ▶__proto__: Array(0)
```


```
var arr = [3, 5, 7];
arr.foo = "hello";

for (var i in arr) {
  console.log(i); // logs "0", "1", "2", "foo"
}

for (var i of arr) {
  console.log(i); // logs "3", "5", "7"
  // it doesn't log "3", "5", "7", "hello"
}
```


```
x
▶(5) [11, 325, 53, 14, {…}, g: 4]
for(e of x){ console.log(e)}
11
325
53
14
▶{k: 4, k1: 1}
```

for of iterates over whatever have an index.


```
x
▼(5) [11, 325, 53, 14, {…}, g: 4]
    0: 11
    1: 325
    2: 53
    3: 14
  ▶4: {k: 4, k1: 1}
    g: 4
    length: 5
  ▶__proto__: Array(0)
```

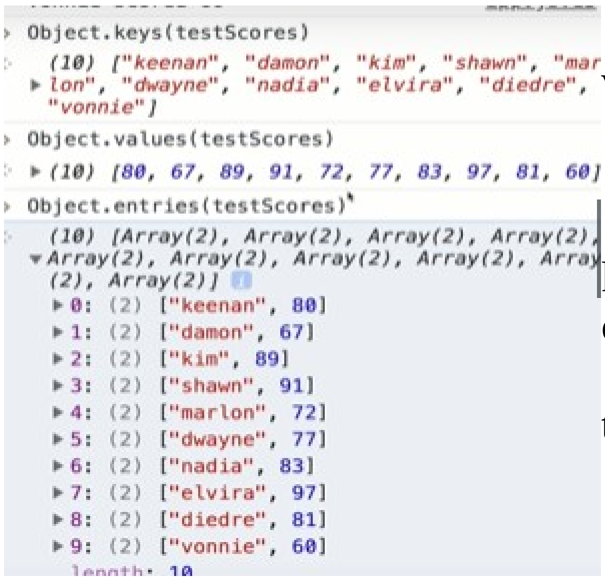


```
for(e of m){ console.log(e)}
▶Uncaught TypeError: m is not iterable
    at <anonymous>:1:10
m
▶{k: 4, k1: 1}
```

Iterating over objects

An array behind the scene is a object, a string is a object too, map and sets are objects behind the scene.

But when we talk about an actual object literal with key:value pair if we wanna iterate over them.



```
Object.keys(testScores)
  (10) ["keenan", "damon", "kim", "shawn", "mar
▶ lon", "dwayne", "nadia", "elvira", "diedre",
  "vonnie"]
Object.values(testScores)
▶ (10) [80, 67, 89, 91, 72, 77, 83, 97, 81, 60]
Object.entries(testScores)
  (10) [Array(2), Array(2), Array(2), Array(2),
▼ Array(2), Array(2), Array(2), Array(2), Array
  (2), Array(2)]
  ▶0: (2) ["keenan", 80]
  ▶1: (2) ["damon", 67]
  ▶2: (2) ["kim", 89]
  ▶3: (2) ["shawn", 91]
  ▶4: (2) ["marlon", 72]
  ▶5: (2) ["dwayne", 77]
  ▶6: (2) ["nadia", 83]
  ▶7: (2) ["elvira", 97]
  ▶8: (2) ["diedre", 81]
  ▶9: (2) ["vonnie", 60]
   length: 10
```

Entries gives a nested array of key value pairs.

Two ways:

(1)you can use for (in) to get keys and use keys to get values out of objects.

(2) you can use Object.keys() to get keys and do ur work.