JAVASCRIPT

Contents

I] Language Fundamentals

1. Difference Between Undefined and null

Q.1) What is "undefined" in JavaScript ?

At the time of declaration, data type is undefined

```
let x;
console.log(x);
console.log(typeof x);
//undefined
//undefined
```

When value is assigned, type is changed from undefined

```
let x=9;
console.log(x);
console.log(typeof x);
//9
//number

let x=null;
console.log(x);
```

Q.2) What will be the output of undefined==null & undefined===null ? Why ?

```
let x=null;
let y;
console.log(x==y);
```
// true -> undefined and null both represent nothingness, but they both are different datatypes

```
// null is a primitive data type
let x=null;
let y;
console.log(x===y);
// false
```

Q.3) Can you explicitly assign "undefined" to a variable ? (let i = undefined)

Yes we can

```
/*let x = undefined;
console.log(x);*/
let x = 10;
function a(){
  x=x+5;
}
function b(){
   console.log(x);
}
a()
b();
```

2. Function Scope vs. Block Scope

Scope is lifetime or availability of function

```
function a(){
   let x=10;
  }
  function b(){
    console.log(x);
  }
  a()
  b();
// reference error: x is not defined


let x=10;
function a(){
     x=x+5;
  }
  function b(){
    console.log(x);
  }
  a()
  b();
//10

 function test(){
            return
```

```
        {
            a:5
        }
    }
    const obj=test();
    console.log(obj);
// undefined

function test(){
        return {a:5}                    //written on same line
}
const obj=test();
console.log(obj);
// {a:5}
```

Q.1) What is hoisting in javascript ?
ES5 has function scope bcoz of hoisting
ES6 has block scope
If you declare a variable with var.. hoisting will be there

```
console.log(y);
var x;
//y is not defined


console.log(x);
var x;
//x is undefined

console.log(x);
var x=9;
//x is undefined as only declarations are hoisted not value assignments
//compiler moves all declarations at the top

Block scope - //introduced in es6
let x=9;
{
  let x=8;
  console.log(x);
}
console.log(x);
//8
//9


console.log(x);
let x;
// reference error.. cannot access x before initialization
```

Q.2) How does block scope work ?
Q.3) What is the scope of a variable ?
3. What is Automatic Semicolon Insertion(ASI)?
Q.1) Should you terminate all lines by a ";" ?

```
// yes .. good practice
console.log("Hi")
console.log("Test");
//Hi
//Test
//JS doesnot insert ; at the end of every line
//it understands operand is remaining
let a=4*
5
console.log(a);
//20
```

Q.2) Why this code is returning undefined in spite of function returning a valid object literal ?

```
function test(){
  return
  {
    a:5
  }
}
```

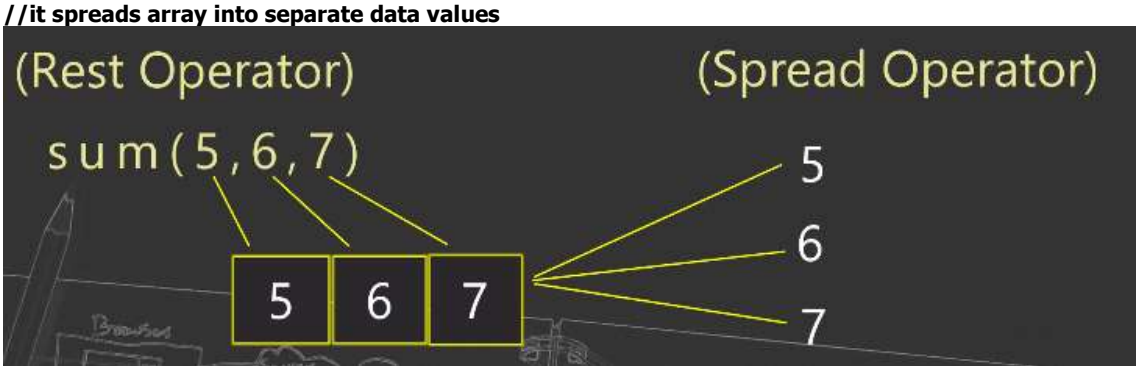
```
const obj=test();
console.log(obj);
//undefined
//internally ; is added after return statement
//return;
//{}
function test(){
   return{
      a:5
   }
}
const obj=test();
console.log(obj);
//{a:5}
```

Q.3) Can 'use strict' statement or the strict mode change the behavior of ASI ?
//No it does not mean that you have to end all lines with ;
4. Diff between rest and spread operators?
Q.1) Can we use arguments object in arrow function?
//No
//Function which takes any no of parameters.. using rest operator

```
function sum(...nums){
      console.log(nums);
}
sum(4,5);
sum(5,6,7,8);
//11
//26

function sum(...nums,a,b){
      console.log(nums);
}
sum(4,5);
sum(5,6,7,8);
// error.. rest parameter should be last
//part of ES6
//in ES5, we have an array like object called 'arguments' which does not work with arrow function so we avoid it

function sum(){
      console.log(arguments);
}
sum(4,5);
sum(5,6,7,8);
```

```
[Arguments] { '0': 4, '1': 5 }
[Arguments] { '0': 5, '1': 6, '2': 7, '3': 8 }
```

// with rest we are creating an array, arguments is object literal
//arguments does not work with arrow functions
//Whenever we use (...) with function to deal with any number of parameters, then you call it Rest operator i.e. for the rest of the parameters
Rest puts separate data values in one array, The spread does exactly opposite
//it spreads array into separate data values



Q.2) Which is the best way to create new arrays with assignment?

```
// using spread
let arr1 = [1,2,3,4,5];
let arr2 = [...arr1,6,7,8];
console.log(arr1);
//[1,2,3,4,5,6,7,8]
```

```
let arr=[1,5,89,5,900,234,1456];
console.log(Math.max(...arr));
//Math.max takes values separately not an array
console.log(Math.max(arr));
//NaN when array is directly given as argument
```

Q.3) How can you handle the "n" number of parameters passed to a function ? or create a function which can take any number of parameters and returns sum or max or any other such calculative values.

```
// rest operator
```

Q.4) Can the rest operator be placed anywhere in the function parameter list? Or

```
    function test(...a,b){
      //statements
    }
```

Is this piece of code valid? Give the reason.

```
//we cannot have rest parameter at the beginning
```

5. When do you get infinity or -infinity as output?

Q.1) How will you put a validation for positive or negative Infinity?

```
console.log(Number.NEGATIVE_INFINITY);
console.log(Number.POSITIVE_INFINITY);
//-Infinity
//infinity
//In JS numbers are stored in 64 bit format
//any number which cant be stored in 64 bit, is treated as infinity

console.log(9e4);
console.log(-9e4);
//90000
//-90000

console.log(9e400);
console.log(-9e400);
//Infinity
//-Infinty
console.log(Number.MAX_VALUE*2);
console.log(-Number.MAX_VALUE*2);
//Infinity
//-Infinty
```

//Ideally when you expect that in some calculations you might get some huge value and you want to put a validation then it is useful
If(num===Number.POSITIVE_INFINITY) & so on
// also have isFinite() method
//NaN – not a number

Q.2) What will be the output of this code?

Code:

```
console.log(1/0);
//Infinity
```

6. When do you get NaN as output?

//we normally get this error when there is a non numeric value or operation performed

Q.1) What will be the output of the below statement?

console.log(NaN==NaN); or NaN===NaN

```
//false

let a=5;
let b="value"
if(isNaN(a*b)===false){
   console.log("Is valid");
}else{
   console.log("Not valid");
}
//not valid
//a*b==NaN not good practice
//NaN==NaN → false as NaN does not match with itself.. each time it returns a unique value
//string and number cannot perform mathematical operation except + which will be concatenation
```

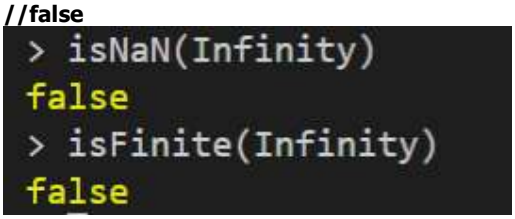
Q.2) What is the difference between isNaN() and isFinite() method?

isNaN() → checks whether number is NaN or not

isFinite() checks for NaN as well as Infinity values

```
let a=5;
let b="value"
console.log(isFinite(a*b));
console.log(isFinite(5*4));
```

```
console.log(isFinite(Number.MAX_VALUE*4));
console.log(isFinite(Infinity));
//false since a*b returns NaN
//true
//false
//false
```



7. Explain must know points of arrow function.

Q.1) Explain the syntactical features of arrow function.

```
Function Expression syntax
Const test = function(){
}
//Here we are assigning function to the object called test
//writing function as expression
Const sum =(a,b)=>a+b;
```

IIFE – Immediately invoked Function Expression

```
Function which calls itself
Used to avoid global leakages
(function(){
        Console.log("IIFE")
})();
```

Immediately invoked Arrow Function

```
(()=>{
        Console.log("IIFE")
})();
```

```
Arrow Function is function expression only
Q.2) Why "this" does not work in arrow function?
//No
//Arguments object also does not work in arrow function
//We cannot use 'new' to call arrow function
// normal function behave like class, arrow function does not have that duty
Const test=()=>{
        Console.log(this)
}
Test();

//O/P:window object → in browser
//{} → in vs code
//arrow function always refers to global window object
const test=()=>{
        Console.lo(arguments)
}
Test(1,2,3);


Q.3) Explain output of following code with reason.
const obj={
   method:()=>{
            console.log(this);
    }
}
// window object
Q.4) How can you handle arguments object like functionality in arrow function?
Using ... rest operator
Q.5) Can you write IIFE with arrow function syntax?
Yes

Arrow function cannot be called using new keyword
Const obj=new test();
//error: test is not a constructor
//we already have a class keyword in ES6
```

8. How does a closure work in Javascript?

When a function comes under another function, a closure is created.

Closure pattern remembers outer variable and also helps to access outer scope members.

Q.1) How can you access private variable or function outside the scope ?

```
function outer(){
   function inner(){
      console.log("Inner called...");
   }
}
inner();

//inner is not defined

function outer(){
   function inner(){
      console.log("Inner called...");
   }
   inner();
}
outer();

// inner called
function outer(){
   function inner(){
      console.log("Inner called...");
   }
   return inner;
}
const cl=outer();
cl();
//returning a function
//inner was private to outer

function outer(){
   function inner(){
      console.log("Inner called...");
   }
   return inner;
}
const cl=outer();
cl();

//closure pattern can remember outer variables
Main adv is that any member which is private for certain scope, can be accessed keeping it private so that the variable is away from any accidental change of value.

Accessing private members with a closure pattern assure better approach of making a variable global

Q.2) Explain the advantage of closure ?


const addCounter=()=>{
   let counter=0;
   counter++;
   return counter;
}
console.log(addCounter());
console.log(addCounter());
console.log(addCounter());
//1
//1
//1

const addCounter=()=>{
   let counter=0;
   return ()=>{
      counter++;
      return counter;
   }
}
const cl = addCounter();
console.log(cl());
console.log(cl());
console.log(cl());
```

```
//1
//2
//3
```

When we use closure, we are making private members globally available.

9. How can sum(5)(6) return 11?

Q.1) What is function currying?

It is a unique way to call inner functions where you can pass arguments partially or pass multiple arguments in a function but 1 argument at a time.

```
const sum=function(a){
   return function(b){
      return a+b;
   }
}
const cl=sum(5);
const ans=cl(6);
console.log(ans);


const sum=function(a){
   return function(b){
      return a+b;
   }
}
const ans=sum(5)(6);
console.log(ans);
//11

const PriceCalc=(price)=>{
   return (dPer)=>{
      return price*dPer;
   }
}
//getPrice();
const discountAmount = PriceCalc(300);
console.log(discountAmount(0.3)); → 90
console.log(discountAmount(0.5)); → 150
console.log(discountAmount(0.10)); → 30
```

Q.2)const multiplication=a=>b=>c=>return a*b*c

What will this statement do? Explain in detail.

Multiplication(2) →

Multiplication(2)(3) →

Multiplication(2)(3)(4) → 24

Q.3) Explain practical usage of function currying.

Currying is an incredibly useful technique of functional programming which solves various purposes like passing partial parameters or avoiding unwanted repetitions like we tried with product price where price was passed only once

10. Iterables and Iterators

Q.1) What is the purpose of the iterator ?

Specifically, an iterator is any object which implements the Iterator protocol by having a next() method that returns an object with two properties:

value
: The next value in the iteration sequence.

done
: This is true if the last value in the sequence has already been consumed. If value is present alongside done, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling next(). Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to next() should continue to return {done: true}

Array, string, map are iterators

Symbol.iterator

```
let arr = [5,2,3]
let itr = arr[Symbol.iterator]();
console.log(itr)
// Object [Array Iterator] {}

let arr = [5,2,3]
let itr = arr[Symbol.iterator]();
```

```
console.log(itr)
console.log(itr.next())
console.log(itr.next())
console.log(itr.next())
console.log(itr.next())
```

```
// Object [Array Iterator] {}
// { value: 5, done: false }
// { value: 2, done: false }
// { value: 3, done: false }
// { value: undefined, done: true } → last value undefined, done: true
```

Q.2) How do you create an iterator ?
When you want to make an object literal iterable, it should have method having key value as Symbol.iterator
The object should have next() method

```
let obj={
   a:10,
   b:15,
   [Symbol.iterator](){
     return this;
   },
    next(){

      if(obj.a<obj.b){
         return {value:obj.a++,done:false};
      }else{
         return {done:true};
      }
    }
}
for(let i of obj){
   console.log(i);
}
```

Symbol.iterator must retuen the object itself bcoz there wont be any reference otherwise

LIMITATIONS
Returning the obj reference is only possible when iterator and iterable are the same object
If there are 2 or more for...of running simultaneously then this syntax will not work
Q.3) Explain a practical use of an iterator ?
To make a object iterable which is already not
You are creating a pointer to move in the data structure which you have provided to the end user who is actually a programmer

11. Generators

Generators can help you to pause and resume the code.
Normally when you write a function, it returns a single value
You can think of generators as a kind of function which can return multiple values in phases.

Q.1) What are generator functions? Explain the syntax.
The function* is the eyword used to define a generator function
yield id an operator which passes the generator
yield also helps to receive the input and send output
Generators give a way where you can pause a process and continue from there after some time.

next() returns 2 keys:
1)value
2) Boolean(show next status)
Generators are iterable as well

```
const genFunction = function*(){
   console.log("Hello");
   yield;
   console.log("World");
   yield;
   console.log("and Galaxy");
}
const gObj = genFunction();
console.log(gObj);
gObj.next();
```

```
const genFunction = function*(){
```

```
    console.log("Hello");
    yield;
    console.log("World");
    yield;
    console.log("and Galaxy");
}
const gObj = genFunction();
for(let o of gObj){
    console.log(o);
}


const genFunction = function*(){
    console.log("Hello");
    yield "Y1";
    console.log("World");
    yield "Y2";
    console.log("and Galaxy");
}
const gObj = genFunction();
for(let o of gObj){
    console.log(o);
}



const genFunction = function*(){
    console.log("Hello");
    yield "Y1";
    console.log("World");
    yield "Y2";
    console.log("and Galaxy");
}
const gObj =[...genFunction()];
console.log(gObj);



let obj={
    start:10,
    end:20,
    *[Symbol.iterator] (){
        for(let cnt=this.start;cnt<=this.end;cnt++){
            yield cnt;
        }
    }
}
for(let i of obj){
  console.log(i);
}


function* gen(){
    console.log("Hi");

}
function* gen1(){
     const g = gen();
     g.next();
}
const g1 = gen1();
g1.next();


function* gen(){
    console.log("Hi");

}
function* gen1(){
    yield* gen();
}
const g1 = gen1();
```

```
g1.next();


function* gen(){
   try{
      yield "One";
      yield "Two";
   }finally{
      console.log("Finally")
   }
}
const gObj = gen();
console.log(gObj.next());
console.log(gObj.next());


function* gen(){
   try{
      yield "One";
      yield "Two";
   }finally{
     yield "Finally";
   }
}
const gObj = gen();
console.log(gObj.next());
console.log(gObj.return());


function* gen(){
   try{
      yield "One";
      yield "Two";
   }catch(err){
     console.log("Error:"+ err);
   }
}
const gObj = gen();
console.log(gObj.next());
console.log(gObj.throw());


function* gen(){
   let a = yield "First";
   console.log(a);
   a = yield "Second"
   console.log(a);
}
const gObj = gen();
gObj.next("A");
gObj.next("B");
gObj.next("C");
```

Q.2) Which is the right syntax function* () { } or function *(){} or function*(){}?
All are valid
Q.3) Explain all methods of generator objects.
Next() → moves the function pointer to the next line from last suspended yield
Return() → allows you to terminate the function
Throw() → helps to raise an error with the generator object
Q.4) Explain the use of "yield*"
Yield * is used to call generator function from another generator function or even to call recursive generator function
Q.5) Can you prevent return() from terminating the generator ?
Yes

```
Function* gen(){
        try{
                yield "one";
                yield "two";
        } finally{
                Yield "Finally"
```

```
        }
}

function* idMaker() {
  var index = 0;
  while (true)
    yield index++;
}

var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3



function* anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}

function* generator(i) {
  yield i;
  yield* anotherGenerator(i);
  yield i + 10;
}

var gen = generator(10);

console.log(gen.next().value); // 10
console.log(gen.next().value); // 11
console.log(gen.next().value); // 12
console.log(gen.next().value); // 13
console.log(gen.next().value); // 20
```

APPLICATIONS

1. Infinite loop – generating id

```
function* genId(){
   let id=1;
   while(true){
      yield id;
      id++;
   }
}
const getNextId = genId();

console.log(getNextId.next())
console.log(getNextId.next())
console.log(getNextId.next())

const getNextId2 = genId();
console.log(getNextId2.next())
console.log(getNextId2.next())
console.log(getNextId2.next())
```

2. Iterators

```
While(genObject.next().done!==true){

}
```

12. Memory Management and Garbage Collection

Memory

Circular reference →

```
Let teacher = new Teacher();
Let student = new student();
Teacher.student = student;
Student.teacher = teacher;
```

Memory allocation is done automatically
As long as references exist no gc will be done
If location is unreachable then that is gc ed

How does GC works in JS?
Mark and Sweep Algorithm
This algo will start from the Global object(root element) and it marks the referenced values and unreachable locations are cleared.

2. Explain Mark and Sweep algo
3. Which situation creates memory leak?
Circular reference
Accidental global variables
Closure

13. How do you handle errors in JS code?
Error, TypeError, SyntaxError
Q.1) When do you need try...catch ?
Handled at runtime

```
let a=10;
console.log(`Value of A${a}`);
letb=50;
console.log(`Value of B${b}`);
```

```
try{
   let a=10;
   console.log(`Value of A${a}`);
   letb=50;
   console.log(`Value of B${b}`);
}catch(err){
   console.log("Error Found");
}
```

```
try{
   let a=10;
   console.log(`Value of A${a}`);
   letb=50;
   console.log(`Value of B${b}`);
}catch(err){
   console.log("Error Found");
   console.log(err.name);
   console.log(err.message);
   console.log(err.stack);
}
```

Q.2) How can you generate an error ?
Usimg throw statement

```
let a=8;
let ageErr = new Error();
try{
   if(a<18){
      throw ageErr;
   }else{
      console.log("Valid age");
   }
}catch(err){
   console.log("Invalid age error");
}
```

Q.3) Can you generate SyntaxError or ReferenceError kind of errors ?
Yes

```
let a=8;
let ageErr = new Error("Invalid Age");
try{
   if(a<18){
      throw ageErr;
   }else{
      console.log("Valid age");
   }
```

```
}catch(err){
   console.log(err.message);
}
```

```
let a=8
try{
   if(a<18){
      throw new TypeError("Invalid Age");
   }else{
      console.log("Valid age");
   }
}catch(err){
   console.log(err.name)
   console.log(err.message);
}
```

Q.4) What is the purpose of the finally block ?
Eg closing sockets and all

```
let a=8;
try{
   if(a<18){
      throw new TypeError("Invalid Age");
   }else{
      console.log("Valid age");
   }
}catch(err){
   console.log(err.name)
   console.log(err.message);
}finally{
   console.log("Finally code");
}
```

Q.5) How can you refer to the name or description of error ?

Q.6) Can we have finally without a catch block as well ?
yes

II] Arrays

14. Explain Array and Traversal in Array

Array is a class and array is prototype

Q.1) What is the difference between for...in and for...of ?

ES6 onwards

```
Let arr=[‘ES5’,’ ES6’,’ ES7’,’ ES8’]
For (let elem in arr){
        Console.log(elem);
}
//O/P:0 1 2 3
//displays index

For (let elem of arr){
        Console.log(elem);
}
//ES5 ES6 ES7 ES8
//returns elements
```

//forEach cannot be stopped in between

Q.2) What will be the output of the below code ?

```
Arr.forEach((elem,index)=>{
Console.log(elem,index)
If(index==2{
        Break;
})
})
```

//illegal break statement

```
Let arr=[7,9,0]
Console.log(arr[arr.length]);
```

//undefined

Q.3) What will be the output of below statements ?

15. Add, Remove, Insert, Replace Elements in Array

Push() and unshift() → to add

pop() and shift() → to remove

Q.1) What is the difference between push() and unshift() method ?

Push() adds 1 or more elements at the end of an array & returns new length i.e. count of modified array

Unshift() adds at the beginning

Q.2) What is the difference between pop() and shift() ?

Pop() removes last element

Shift removes first element

Q.3) How can you insert an element at a given position ?

Splice() → can add, insert,replace or even remove one or more elements

Arr.splice(start,[deleteCount[,value1,...,valueN]])

```
let arr = ["1","2","3","4","5"];
console.log(arr.splice(2,1,"7","9"));
console.log(arr)
// [ '3' ]
//[ '1', '2', '7', '9', '4', '5' ]
```

Removes 1 element from index 2 and inserts 7 and 9 from index 2

```
let arr = ["1","2","3","4","5"];
console.log(arr.splice(2,0,"7","9"));
console.log(arr)
[]
[  '1', '2', '7',  '9', '3', '4',  '5']
```

Q.4) How can you remove a specific element?

```
Console.log(arr.splice(arr.length,0,”push”))
```

// adds element at the end

Q.5) What does splice() return ?

Removed elements array

// splice() changes the original array which is a bad practice

Q.6) If there is not element removed then what will splice() method return ?

```
=> let arr=["One","Two","Three","Four","Five"];
    console.log(arr.splice(2,0,"New"));
```

16. How do you perform search in an array?

Q.1)What is the difference between find() and filter() method ?

Q.2)If there is no value to return, what will findIndex() return ?

Q.3) What is the difference between indexOf() and includes() method ?

arrName.indexOf(searchElement,[startindex])
- startindex – 0 default
- indexOf() will search from beginning
- returns index no if item found
-1 if not found
If more than one found, index of first

arrName.indexOf(searchElement,[startindex])
- startindex – 0 default
- indexOf() will search from beginning
- returns index no if item found
-1 if not found
If more than one found, index of first

arrName.lastIndexOf(searchElement,[startindex])
- startindex – array length-1 default
- indexOf() will search from beginning
- returns last index no if item found
-1 if not found
If more than one found, index of last

arrName.includes(searchElement,[startindex])
- returns true or false

Case-sensitive search

Conditional Search
Let arr=[2300,4500,5600,7800,1200]
find() → returns undefined if not found

```
const rVal = arr.find((element)=>{
        return element>3000;
})
Console.log(rVal)
//4500
```

findIndex()→ returns undefined if not found

```
const rVal = arr.findIndex((element)=>{
        return element>3000;
})
Console.log(rVal)
//1
```

filter() → used to return a new array rather than single value
→ returns [] if not found

```
const rVal = arr.filter(element)=>{
        return element>3000;
})
Console.log(rVal)
// Let arr=[4500,5600,7800]
```

Q.4) How will you search multiple values in an array ?
Filter method
Q.5)What will be the output of this code ?

```
    => let arr=["One","Two","Three","Four","Five"];
       console.log(arr.lastIndexOf("Abcd"));
//-1
```

17. What is the use of map() method?

```
let arr = [2,3,6,4,5];
let newarr=arr.map(function(elem,index){
  return elem*elem;
})
console.log(arr)
console.log(newarr)
```

[2, 3, 6, 4, 5]
[4, 9, 36, 16, 25]

Let newArray = arr.map(function callback(elem[,index,[,array]]){
Return value; → which is returned to newArray
}[.thisArg]])
//thisArg → reference to callback function

Q.1) Find length of each element in a new array.

```
Let lArr = arr.map(elem=>elem.length)
```

Q.2) Find the square root of every element and store it in new array.

```
Let lArr = arr.map(elem=>Math.sqrt(elem))
```

Q.3) There is an array called products as shown here

```
=> let products=[
    {pCode:1,pName:"Apple"},
    {pCode:2,pName:"Banana"},
    {pCode:3,pName:"Grapes"},
    {pCode:4,pName:"Oranges"}
    ]
```

Get all product names (i.e pNames) in a new array.

```
Let pNames = products.map(elem=>elem.pName);
```

18. How to flatten 2D array? (REDUCE())

Aggregate or accumulative operation e.g. product,sum, average

Q.1) How will you flatten an array i.e e.g converting 2 dimensional array into single dimension ?

```
let arr = [
    [1,2],
    [4,5],
    [8,9]
  ];
  console.log(arr)
let fArr=arr.reduce((accumulator,current,index,array)=>{
  return accumulator.concat(current)
})

console.log(fArr)
```

```
[ [ 1, 2 ], [ 4, 5 ], [ 8, 9 ] ]
[ 1, 2, 4, 5, 8, 9 ]
```

Q.2) Get sum of a key field of an object literal

```
=> const employees=[
    {eNo:1001,salary:3000},
    {eNo:1002,salary:2200},
    {eNo:1003,salary:3400},
    {eNo:1004,salary:6000}
   ]
```

Then find total salary of employees.

```
let totalSalary=employees.reduce((accumulator,current,index,array)=>{
  return accumulator+(current.salary)
})
console.log(totalSalary)
//[object Object]220034006000
let totalSalary=employees.reduce((accumulator,current,index,array)=>{
  return accumulator+(current.salary)
},0)
console.log(totalSalary)
//14600S
```

Q.3) Find avg value of all elements of an array ?

```
let arr = [2,4,6,8,10];
let newarr=arr.reduce((accumulator,current,index,array)=>{
  accumulator+=current
  if(index===arr.length-1){
    return accumulator/arr.length
  }
  return accumulator
})

console.log(newarr)
```

Q.4) Find the sum or product of all elements ?

```
let sum=arr.reduce((accumulator,current,index,array)=>{
    return accumulator+current
  }
```

Q.5) What is the difference between reduce() and reduceRight() ?

reduceRight() → loops array from array.length-1 till 0th index

19. How can you sort an array?

Q.1) What will be the output in case an array has "undefined" while sorting the values ?

```
let products = ['banana',undefined,'mango','peru','chiku']
let sorted=products.sort();
console.log(sorted)
// [ 'banana', 'chiku', 'mango', 'peru', undefined ]
```

Undefined is always placed at the end of the array
Undefined values are never sorted

Q.2) How will sort an object literal ?

```
const employees=[
   {eNo:1001,salary:3000},
   {eNo:1002,salary:2200},
  {eNo:1003,salary:3400},
   {eNo:1004,salary:6000}
]

let sorted=employees.sort((a,b)=>{
   if(a.salary>b.salary) return 1;
   if(a.salary<b.salary) return -1;
   if(a.salary==b.salary) return 0;
})

console.log(sorted)
//[  { eNo: 1002, salary: 2200 },
  { eNo: 1001, salary: 3000 },
  { eNo: 1003, salary: 3400 },
  { eNo: 1004, salary: 6000 }
]
```

Q.3) How will you sort a numeric array ?

```
let price=[45,23,10,89,5]
let sortedList=price.sort()
console.log(sortedList)
// [ 10, 23, 45, 5, 89 ]
```

If you try to sort a numeric array with only sort() method then it is going to consider the unicode string value if there is no compare function given

```
let price=[45,23,10,89,5]

let sortedList=price.sort(function(a,b){
   if(a>b) return 1;
   if(a<b) return -1;
   if(a==b) return 0;

})

console.log(sortedList)
//[ 5, 10, 23, 45, 89 ] → ascending order
```

Q.4) Sort all values of array in descending order.

```
//For descending
   if(a>b) return -1;
   if(a<b) return 1;
```

20. Explain array Destructuring?

Destructuring is a concept of breaking data structure like array or object literal into data pieces or in other words individual variables.

Q.1) What is the destructuring assignment ?

```
let arr=[80,90,14,88];
let [a,b,c,d] = arr;
console.log(a,b,c,d)
//80 90 14 88
let arr=[80,90,14,88];
let [a,,c,d] = arr;
console.log(a,c,d)
//80 14 88
```

Q.2) Swap values using destructuring.

```
let a=4;
let b=5;
[a,b]=[b,a]
console.log(a,b)
//5 4
```

Q.3) What will be the output of this code ?

```
      => let [a,b,c] = [5,,7];
          console.log(a,b,c);
//5 undefined 7
```

Q.4) How will you set a default value while destructuring an array ?

```
let arr=[4,5,6]
let [a,...b] =arr;
console.log(a,b)
//4 [5 , 6]

let arr=[4]
let [a,b=0] =arr;
console.log(a,b)
//4 0
let arr=[4]
let [a,b=0] =arr;
console.log(a,b)
//4 6
```

III] String

21. String Basics [UTF - 16] - \ u- Unicode

String data type is stored in UTF-16 format
But in case if there is a special character to be displayed then Unicode is represented with 4-digit hexadecimal number. To display this character you can use
\uHHHH
Where HHHH is 4 digit hexadecimal value

1. **Explain various ways to declare a string**
   Let str=new String("Hello world")
2. How will you deal with Unicode characters?
   \u00A9 → copyright character
   \u{1F602} → smiley (long Unicode character)

22. ES6 Template Literal (String)

Let str = `template literal`

23. 'length' property and Search Methods

indexOf() and lastIndexOf() → search for char and returns the index number else -1
Normally properties or methods should be with objects, whereas string is a primitive type
JS primitive types – when executed behave like an object
It means you do have properties and methods with these primitive types as well
Search()

1. Difference between indexOf() and search() →
-index of can have start position to perform search but indexOf() cannot be used for advanced search operations like regex.
2. indexOf and lastIndexOf
3. O/P of code
Let str="This is a test"
Str.indexOf("is",5) → 5
Str.lastIndexOf("is",1) → -1

24. extraction Methods

1. String Extraction Methods
a. slice() →
str.slice(startindex[,endIndex])
goes till endIndex-1

```
console.log(str.slice(-10,-5))
```

does not change original string.. returns extracted part as a new string
b.substr() →
substr takes no of characters instead of endIndex
str.substr(10,3)
c. substring()→
same as slice but does not take negative values
returns entire string for negative values
str.substring(3,3)
//

2. Character Extraction Methods
a. charAt() → returns char at nth position
str.charAt() → if no position given.. returns first char
b. charCodeAt() → returns Unicode for char at nth position

25. Case Convention & replace() Method

1. str.ToLowerCase() →
2. str.ToUpperCase() →
3. replace() →

IV] Date and Time

26. Date and Time Basics

1) Explain different ways of creating date/time object ?
1. Date() → returns the current date and time along with the local time zone

Local Time Zone → new Date()

UTC(Universal Time Zone)

2. new Date(year,month,day,hours,minutes,seconds,milliseconds)
Const dt = new Date(2020,8,23,11,5,20,3)
2020-09-23T05:35:20.003Z
Gives September instead of august as 8 means 9th month
Const dt = new Date(2020,8)
2020-08-31T18:30:00.000Z
Considers this as 1st day of sept

Const dt = new Date(2020)
1970-01-01T00:00:02.020Z

3. new Date(milliseconds) → accepts milliseconds and that is converted into a date considering the first millisecond on 1st Jan 1970
Console.log(1599071400000)
2020-09-02T18:30:00.000Z

4. new Date(str)
New Date("2020-8-23T12:32:00Z")
If Z is not given result may vary from browser to browser
2) What will be the output of the below code ?
=> const dt = new Date(2020,08,23);
console.log(dt);

23 Sept 2020

3) Explain various formats of ISO standard followed by JavaScript ?
YYYY
MM
DD

27. Date Methods
getFullYear()
getFullMonth()
getDate()
getDay()
getUTCYear()
date.setFullYear(2021)
date.setMonth(1)
date.setDate(12)
Date.parse("2020-04-23")
Q.1) Get a Character month ?

Q.2) Find the date before 50 days of the given date.
Let dt=new Date();
Dt.setDate(dt.getDate()-50)

Q.3) What will be the output if you add 35 as date in Date() constructor.
Let dt=new Date(2020,7,35)
//Fri Sep 4 ...
//auto correction feature

28. Time Methods
getTime()
getHours()
getMinutes()
getSeconds()
getMilliSeconds()
setTime()
setHours()
setMinutes()
setSeconds()
setMilliSeconds()

Q.1) Calculate the date difference in days ?
Date.now() → returns timestamp
Let dt=new Date(2020,7,2)
Let dt1=new Date(2020,7,10)
Let n=d1-dt;

Days = (n/1000)/3600;
Q.2) How can you change hours or minutes in time ?
setHours()
setMinutes()
V] OOJS
29. What is Object Literal?
Key:value pair data structure

Q.1) Can you have dynamic keys with object literal?
Yes
```
let tV = "pCode";
let obj={
    [tV]:1001,
    pName:"Apple",
    getData(){
        console.log(obj.pCode,obj.pName)
    }
}
obj.getData()
```
Q.2) How can you add read-only properties to an object?
yes
```
const proto = Object.defineProperty({},'pCode',{
    writable:false,
    configurable:true,
    value:2001
})

const obj = Object.create(proto);
obj.pCode = 3001;
console.log(obj.pCode);
```
//3001 → if writable=true
//2001→ if writable=false
// in browser it may generate an error if in strict mode
Q.3) What is property value short hand with object literal?
ES6
```
let pCode=1001;
let pName='Apple';

let obj={
    pCode,
    pName
}
console.log(obj)
```
//no need to write pCode:pCode if both are same
Q.4) What will be the output of this code?
```
let obj={a:'First'};
let obj1 = obj;
obj1.a="Second";
console.log(obj.a);
```
→ **assignment by reference**
Explain the reason why?

//Second

Q.5) How can we create a clone or separate copy of an object literal?
Using Object.assign()
```
let obj={a:'First'};
let obj1 = Object.assign({obj});
obj1.a="Second";
console.log(obj.a);
```
30. What is 'this' object?
It contains the current context
//in browser
Console.log(this) → Window Object

//in vs code → {}

Q.1) What will be the output of this code if you run in the browser and why?
```
function test(){
  console.log(this);
}
test();
```
<ref *1> Object [global] {

```
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  …
}
```

Arrow function does not work with this object with current context
Q.2) What is the context of "this" inside arrow function? Or what will be the output of this code?

```
let obj={

   test:()=>{
     console.log(this);
   }
}
obj.test();
```

31. What is the purpose of call(), apply() and bind()?

```
function test(obj){
   this=obj
 console.log(this);
}

let obj={
   a:5
}

test(obj)
//SyntaxError: Invalid left-hand side in assignment
```

Q.1) What is the difference between call, apply and bind()?
1. call() → used to change the reference or context or value of **this** object
Functionname.call(thisArg,arg1,arg2,…)

```
function test(a,b){
 console.log(this);
}
let obj={
   a:5
}
test.call(obj,8,9)
```
//{a:5}

2. apply() → you can pass only 2 arguments – thisObject and array of parameters
// changes reference and immediately calls function
Functionname.apply(thisArg,[arg1,arg2,…])

```
function test(a,b){
 console.log(this);
}
let obj={
   a:5
}
test.apply(obj,[8,9])
```
//{a:5}

3. bind() →
// changes reference and but does not immediately call function
//it acts like a function expression
Functionname.bind(thisArg,arg1,arg2,…)

```
function test(a,b){
 console.log(this);
}
let obj={
   a:5
}
test.bind(obj,8,9)
```
//{a:5}

Q.2) What will be the output of this code? Or can you assign value to "this" object with assignment operator i.e. "="?

```
const obj={
    a:6
}
function test(){
    this=obj;
}
test();
```

// invalid left-hand sign assignment

32. Class, Class Expression and Static Members

Q.1) How can you create a class?
To create a blueprint or design or prototype
Q.2) When does constructor() called?
Every time instance of class is created

Q.3) Can we have dynamic property or method names in a class?
Yes
Q.4) What is a class expression?
Let House = class {

}
Q.5) What are static members in a class? What is the purpose of defining them as static?

```
let getData= "showData"

class House{
   static test(){
      console.log("Static method called")
      this.showData();
   }
   [getData](){
      console.log("I am showdata")
   }
}
House.test()
let obj = new House()
obj.showData()
```

obj.test() → will give an error

Q.6) How can you call another static method from a class?
Inside static method only
For creating a module which has services where we do not need to instantiate the class – so we create a class and create those members as static members

33. Inheritance, Subclassing and Extending built-in class

```
class Parent{
   constructor(){
      console.log("Parent Constructor")
   }
   pMethod(){
      console.log("Parent Method")
   }
}

class Child extends Parent {
   constructor(){
      super();
   }
   cMethod(){
      super.pMethod()
   }
}
const obj = new Child()
obj.cMethod();
```

// if you have constructor in child class then it should explicitly called constructor of parent first
//using super() --> allows you to access parent class members, methods and constructor

Q.1) How can you inherit a class?
With ES6 → extends

Q.2) What is subclassing?
Subclassing = inheritance in ES6

Q.3) What is the purpose of super keyword?
Super() → call parent constructor
Super.method() → call parent method
Super.property → zccess parent property
Q.4) How will you override a method?
Super.pMethod()
Using super, accessing parent method with same name
Q.5) How can you extend a built-in class?
extends

34. Destructuring Object Literal

It is a concept of breaking data structure like array or object literal into data pieces or in other words individual variables

Q.1) Explain a practical scenario of object literal destructuring ?

```
const obj={
   pCode:1001,
   pName:'Apple'
}
let {pCode,pName} = obj;
```

```
//used in const {name,password} = req.body
//Make sure that the keyna,e or variable name is similar
//ObjectLiteral Destructuring is often used in all programs, because you get the data from the server in a JSON format which is assigned to JS Object literal
```

Q.2) Explain the output of this code

```
=>  const { a=90, b } = {  };
      console.log(a,b);
//90 undefined
```

VI] Module – Import/Export

35. What is Module?

Q.1) What is a module ?
Provision of importing and exporting module directly which can be rescued when required
Module means dividing your code into various logical pieces and importing them when required.
Q.2) Can you import any module inside the script tag ?
Yes

```
<script>
        Import {calc} from `./calc.js'
        Calc();
</script>
// will generate CORS error
<script type='module'>
        Import {calc} from `./calc.js'
        Calc();
</script>
```

Q.3) How will you run the import and export statements on a local machine ?
As long as import and export statements running on local machine is concerned , you need to have a "server"
In frameworks like angular and react you have a development server running so this is not an issue
But when we try import and export statement with simple HTML and JS, we need some other server and to temporarily run this you can configure the chrome web server.

36. Named Export/Import

Q.1) What is a named export and import ?
1. Named export/import

```
Export function f(){
}
```

Or at the end,

```
Export {sum as total,cube}
Import {total} from `./calc'
Calc.total()
```

2. Default export/import

In modern build tools like webpack, if some imported modules are not used then those modules get removed i.e. **Tree Shaking**
Q.2) Can you avoid {} while importing a named module ?
No
Q.3) How can you import all named modules from a file ?
Import * as calc
Q.4) Is it a good practice to import all modules together ?

Q.5) Do the modules hoist ?
Yes, you do have hoisting

```
module(); // using module before import
import module from 'path'  //importing
```

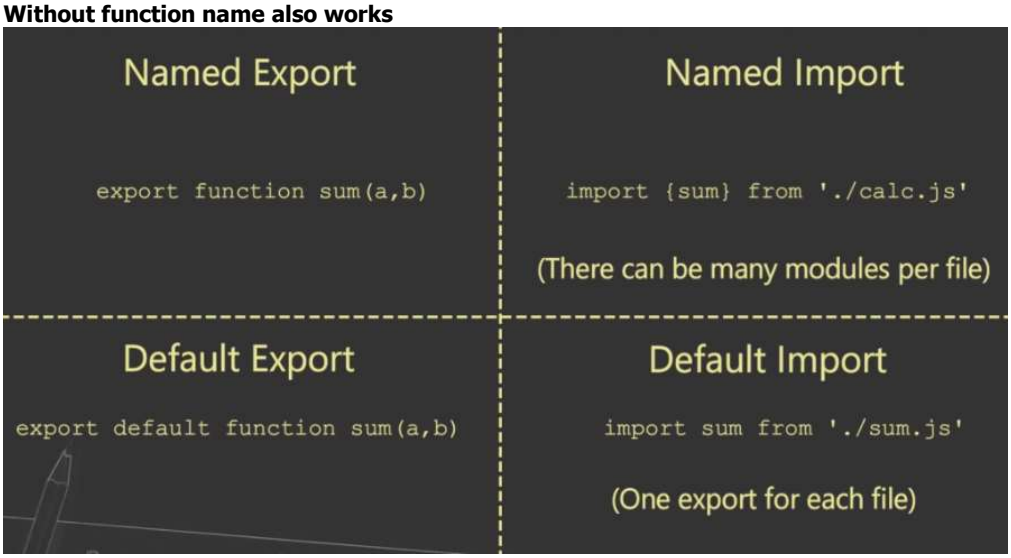
Q.6) Do you need the same name while importing a named module or you can change ?
No

37. Default Export/Import

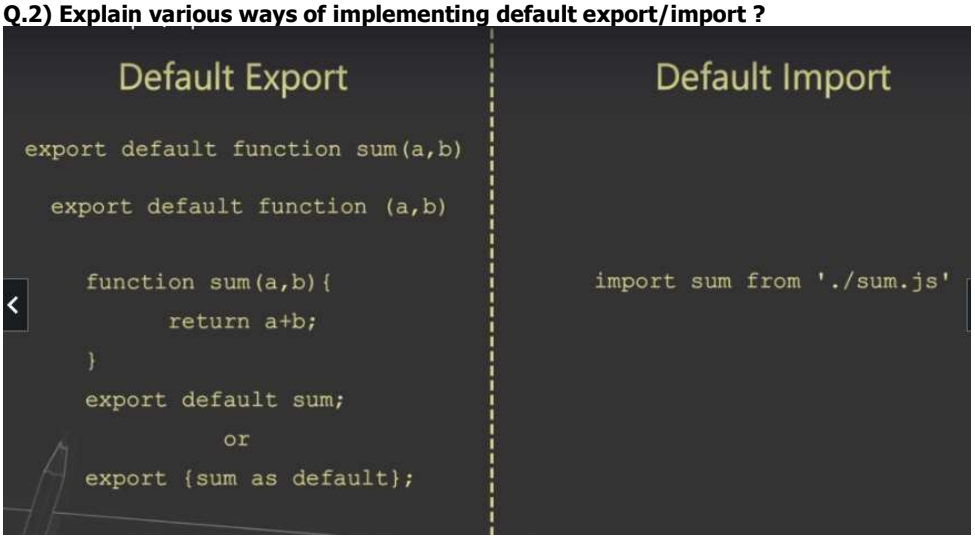
Q.1) What is default export/import & the difference between named & default export/import ?
Only one export per file for code modularity.. one file-one module

```
Export default function sum(a,b){
Return a+b;
}

Export default function (a,b){
Return a+b;
}
```

Export sum as default
Without function name also works



Q.2) Explain various ways of implementing default export/import ?

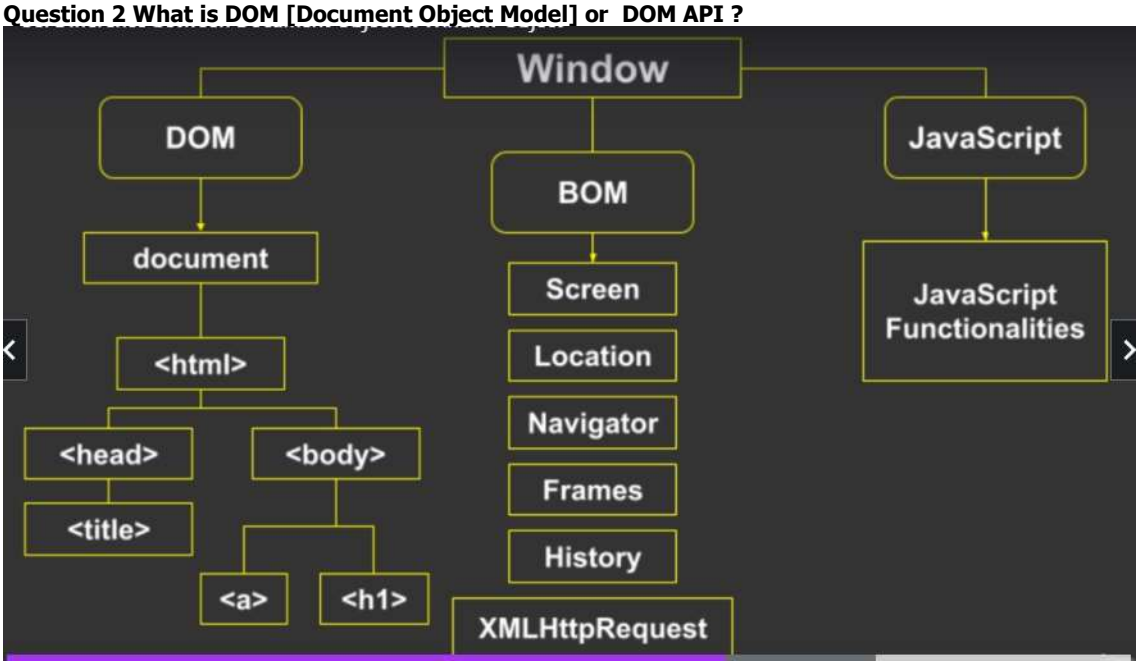


VII] DOM and Web

Creating HTML Component

38. Difference between Document Object and Window Object

Question 1 What is the difference between window and document ?
setInterval() or setTimeOut() etc are methods of window
Document is an object of window
Window.innerHeight
Window.innerWidth
Window.location
Question 2 What is DOM [Document Object Model] or DOM API ?



Question 3 What is BOM [Browser Object Model] ?

Question 4 Explain the difference between DOM and BOM ?

Question 5 Which is the global object in browser ?
window
Question 6 How will we check the innerWidth and innerHeight of window ?

Question 7 How will you get hostname or value typed in address bar ?
Using window.location

39. How to handle timer based events?

Q.1) Explain the difference between setTimeout() and setInterval() ?

setTimeOut() executes only once after a given number of intervals

setInterval() keeps on executing

```
setTimeOut(()=>{
        console.log('I will be called after 3 sec')
},3000)

Let counter=0;
Let tObj = setInterval(()=>{
        console.log('counter++)
},3000)
```

To stop

setInterval(tObj),

We need to use clearInterval(tobj)

Q.2) Why can't we write document.setTimeout() or document.setInterval() ?

Part of window not document

Q.3) How can you stop a timer ?

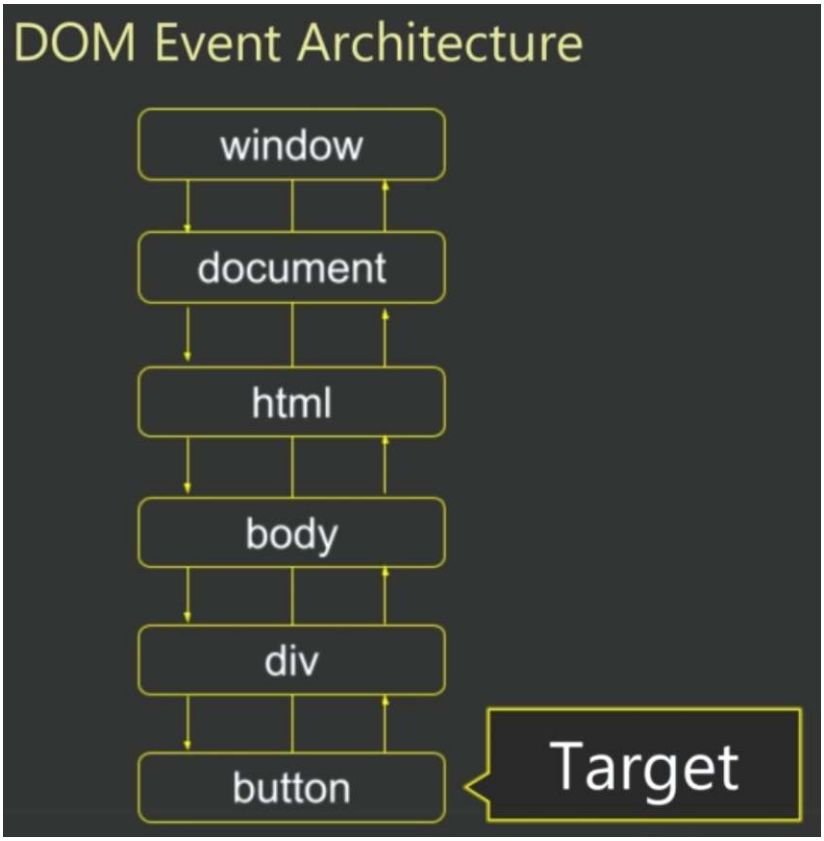
clearInterval()

Q.4) How will you create a digital clock ?

```
<!DOCTYPE html>
<html>
  <head>

  </head>
  <body>
    <h1 id="text">Waiting...</h1>
    <button onclick="tryTimer()">Start Timer</button>
    <button onclick="stopTimer()">Stop Timer</button>
    <script>
      let cnt=0;
      let tObj;
      function tryTimer(){
        tObj = setInterval(()=>{
            document.getElementById("text").innerHTML=new Date;
            cnt++;
          },1000);
        }
      function stopTimer(){
        clearInterval(tObj);
      }
    </script>
  </body>
  </html>
```

40. What is event bubbling and event capturing?

Q.1) Which is the default propagation path ?



Button has method event.target
Event.target returns the reference of element which raised the event
Div is ancestor of button

The default path of Event Propagation is bottom to top or bubbling.
In case if you want to stop this default behaviour

Event.stopPropagation();

In case if you want to change this default propagation path i.e. instead of bottom to top we make it top to bottom, then that will be called "capturing"
For capture phase, we need to use addEventListener() method to configure or bind the event
Capturing phase means from top to bottom

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      div{
        width:50%;
        height:100px;
        border: 1px solid darkslategray;
        text-align: center;
        padding:50px;
        margin: auto;
      }
    </style>
  </head>
  <body>
    <div id="div1">
      <button id="btn1" >Click Me!</button>
    </div>
    <script>
      const btn1 = document.getElementById("btn1");
      const div1 = document.getElementById("div1");
      window.onload = function(){
        btn1.addEventListener("click",btnClick,true);   // for event capturing
        div1.addEventListener("click",divClick,true)
      }
      function divClick(){
        alert("Division clicked");
      }
      function btnClick(){
```

```
            alert("Button clicked");
        }
    </script>
  </body>
</html>
```

There are few events which do not get propagated like focus events.
Bubbling or Capturing actually facilitates you with one of the most powerful event handling pattern called "Event Delegation"

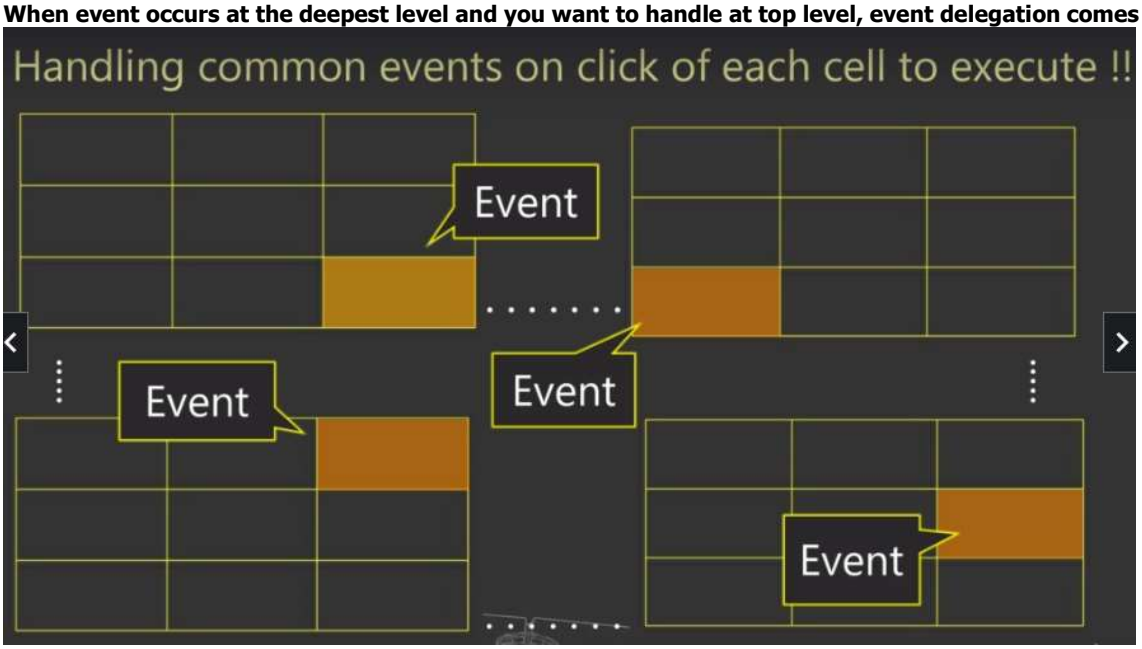
Q.2) How can you stop the event propagation ?
Event.stopPropogation()

Q.3) How can you change the default propagation path ?
addEventListener("click",btnClick,true)

Q.4) How can you get the reference of element on which event is fired ?
- event.target
- event.target.name
- event.target.value

Q.5) There can be a practical question similar to code sample and one can ask what will be the output and why ?
41. What is event delegation?
Q.1) How event bubbling or capturing is useful in the practical programming world ?
By default the events bubble up, it means that when an event is triggered on an element, the same event is also triggered for that element's ancestors.

When event occurs at the deepest level and you want to handle at top level, event delegation comes into play.

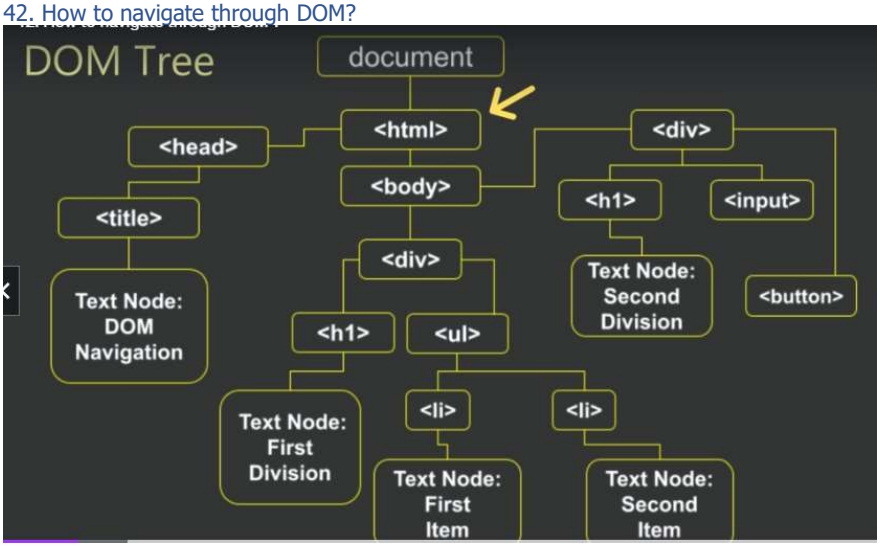



```
<!DOCTYPE html>
<html>
  <head>
    <style>
      table{
        width:50%;

        border: 1px solid darkslategray;
        text-align: center;
        padding:5px;
        margin: auto;
      }
      td{
        border: 1px solid darkgray;
        padding:10px;
      }

    </style>
  </head>
```

```
  <body>
    <table onclick="mainClick()">
      <tr>
        <th>Code</th>
        <th>Product</th>
        <th>Price</th>
      </tr>
      <tr>
        <td>1001</td>
        <td>Apple</td>
        <td>$20</td>
      </tr>
      <tr>
        <td>1002</td>
        <td>Orange</td>
        <td>$10</td>
      </tr>
      <tr>
        <td>1003</td>
        <td>Banana</td>
        <td>$5</td>
      </tr>
        </table>
    <script>
      function mainClick(){
        console.log(event.target);
        console.log(event.target.tagName);
      }
    </script>
  </body>
</html>
```

Q.2) Which property can be used to check the element type on which the event is triggered ?
Event.target.tagName
Q.3) How can you get value of the element which raised the event ?
Event.target.innerHTML
Or
Event.target.value

42. How to navigate through DOM?



Q.1) What is a node in DOM tree ? Explain different types of nodes ?

Constant	Value	Description
ELEMENT_NODE	1	An element node such as <h1> or .
TEXT_NODE	3	The actual text of element.
COMMENT_NODE	8	A comment node i.e. <!-- some comment -->
DOCUMENT_NODE	9	A document node i.e. the parent of <html> element.
DOCUMENT_TYPE_NODE	10	A document type node

console.log(document.documentElement) refers to topmost node
console.log(document.head)
console.log(document.body)
console.log(document.body.childNodes)
console.log(document.body.childNodes.length) → includes text element for every whitespace
if we want only html nodes then
console.log(document.body.children) → excludes text element, prints all tags only

```
<head>
    <script>
        Console.log(document.body);
    </script>
</head>
```

This returns null as no body declared at this point

```
const div1=document.getElementById("div1");
console.log(div1.firstChild)       //gives first child in div1 can include whitespace.. texxt
console.log(div1.firstElementChild) // gives first HTML element in div1.. excludes whitespace
console.log(div1.firstChild)       //gives last child in div1 can include whitespace.. texxt
console.log(div1.firstElementChild) // gives last HTML element.. excludes whitespace

// div1.firstChild.nodeName
// div1.firstChild.nodeType
// div1.firstChild.nodeValue
```

Q.2) How can you access next same level element for a given node element ?
Div1.nextSibling // includes whitespace
Div1.nextElementSibling
Div1.previousSibling // includes whitespace
Div1.previousElementSibling
Q.3) What is the difference between childeNodes and children property ?
childeNodes → All elements including whitespace
children → Only regular html elements
Q.4) How can you check whether an element has child nodes or not ?
- childNodes.length
- hasChildNodes()
Q.5) What is the purpose of DOM navigation ?
Have an access to all nodes in a page programmatically, to remove, add or modify the content
Q.6) What will be the output of below given statement - element.lastChild.nextSibling ?
null

43. getElementBy id,tag,class,name

Q.1) Can you use document.getElementById() & element.getElementById() ?
document.getElementById() → gets reference of particular element
only works with DOM
element.getElementById() → NO
Q.2) What does the getElementsByTagName() return ?
Returns array of reference of all elements having tag
Q.3) What is the difference between getElementsByClassName() & getElementsByName() ?
getElementsByClassName → Returns array of reference of all elements having class ClassName
getElementsByName → Returns array of reference of all elements having name Name

44. querySelector() and querySelectorAll()

querySelector methods search elements using CSS selectors like id, name , tag etc

Q.1) Explain the difference between querySelector() & querySelectorAll().
querySelector() → returns the first element that matches specified CSS selector inside the method given as an argument

- **It searches in the entire document**

querySelectorAll() → returns all the elements

```
<!DOCTYPE html>
<html>
```

```
  <body>
    <div>
      <p>First</p>
      <p>Second</p>
      <p class="a">Third</p>
      <h3 class="a">H31</h3>
      <h3>H32</h3>
   </div>
   <div>
     <p>Next division</p>
   </div>
  <button onclick="btnClicked()">Change Content</button>
   </body>
   <script>
     function btnClicked(){
       const elements = document.querySelector(".a");

       console.log(elements);
     }
   </script>
</html>
```

Q.2) How can you search elements using pseudo classes?

```
const elements=document.querySelector(":hover");

const elements=document.querySelectorAll(":hover");
```

All the elements which are having pseudo class :hover will be returned into the elements variable

Q.3) Between getElementBy methods and querySelector methods which is better to use?

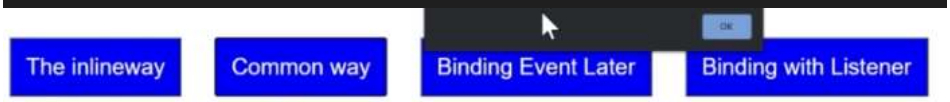
getElementBy* is perfect to search a specific element, works on all browsers

querySelector is useful for classname, selectors etc, works on IE-8 or above

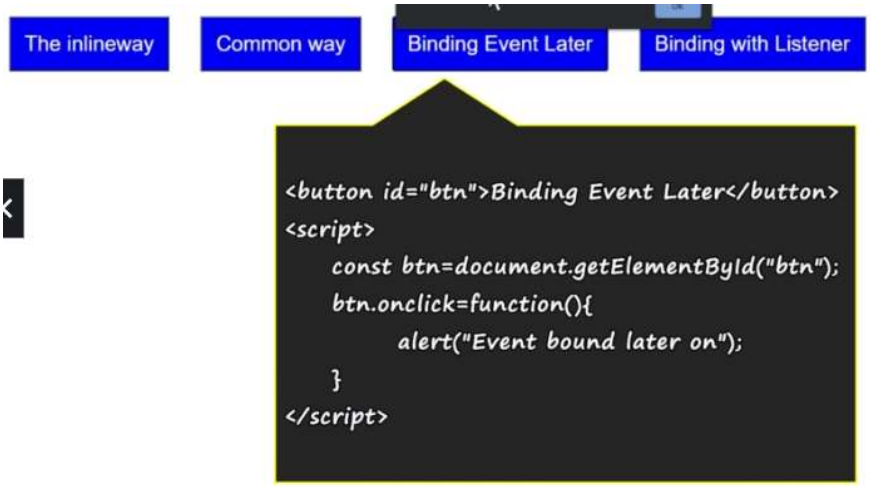
45. Event Basics

Q.1) Explain various ways to configure an event with HTML elements.

```
<button onclick="alert('Not an idea way')">The inlineway</button>
<button onclick="showMsg()">Common way</button>
<button id="btn">Binding Event Later</button>
<button id="btn1">Binding with Listener</button>
```




```
<button onclick="showMsg()">Common way</button>
<script>
    function showMsg(){
        alert("The common way");
    }
</script>
```



Binding With Listener

```
const btn = document.getElementById("btn");
btn.onclick=function(){
    alert("Event bound later on");
}
const btn1 = document.getElementById("btn1");
btn1.addEventListene
function btn1Clicked
    alert("Add even")
}
```

alert(message?: any): void

Q.2) Explain syntax of addEventListener() method.

```
element.addEventListener(event,function[,useCapture])
```

Q.3) What is an event object ?

Event object is the parent of all event objects like MouseEvent, KeyboardEvent, FocusEvent etc

46. Mouse Event

oncontextmenu event can be used to handle Right Click

```
<div id="divE"
onclick="eHandler('Click')"
oncontextmenu="eHandler('Right Click')"
ondblclick="eHandler('Double Click')">Click Events</div>

<script>
function eHandler(msg){

document.getElementById("divE").innerHTML = msg;

}
```

Q.1) How will you disable right click on an element?

Return false in context menu

```
<div id="divE"
onclick="eHandler('Click')"
oncontextmenu="return false"
ondblclick="eHandler('Double Click')">Click Events</div>


<script>
function eHandler(msg){



document.getElementById("divE").innerHTML = msg;

}
```

Q.2) How can you check whether a special key is pressed or not while an event is generated?
Q.3) How do you check mouse button value while an event is generated?

```
function eHandler(){
    let msg='';
    if(event.button===0){
        msg="Left Button Pressed";
    }else if(event.button===1){
        msg="Middle Button"
    }else if(event.button===2){
        msg="Right Button";
    }

    document.getElementById("divE").innerHTML = msg;

}
```

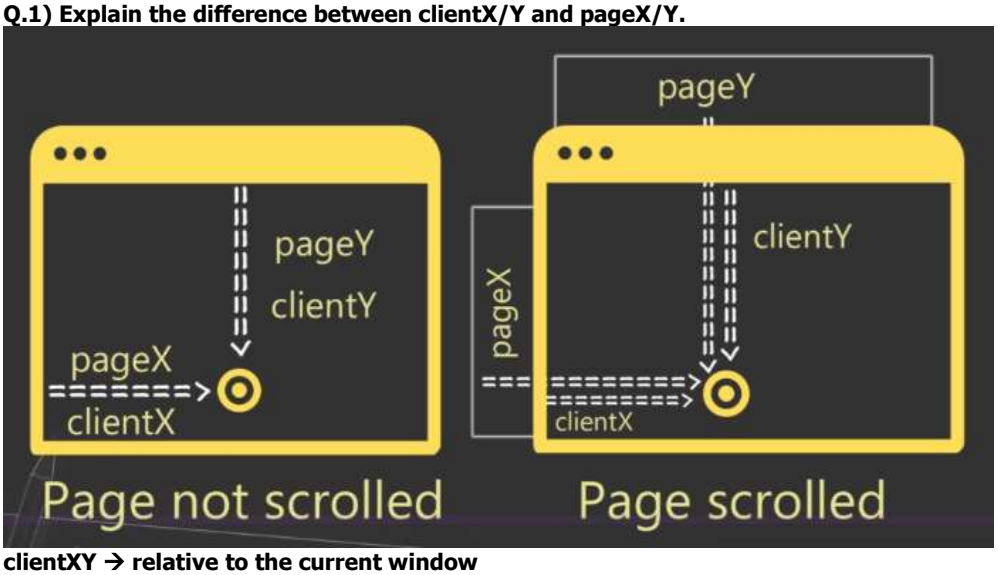
Q.4) For ctrl key and cmd on mac - how will you right the condition to check the key press.
If(event.altKey){}
If(event.ctrlKey){}
If(event.shiftKey){}
If(event.altKey && event.ctrlKey){}
If(event.metaKey){} // for mac machine
If(event.metaKey && event.ctrlkey){} // for both mac and windows os

47. ClientX/Y vs PageX/Y vs ScreenX/Y – Coordinates

```
<!DOCTYPE html>
<html>
  <head>
  <style>
    div{
        width: 300px;
        height: 90px;
        display: inline-block;
        padding:20px;
        background-color: blue;
        color: white;
        font-size: 30px;
        text-align: center;
        margin-top: 20px;
    }
    #main{
        height: 500px;
        width: 300px;
        background-color: cornflowerblue;
    }
  </style>
  </head>
  <body onmousemove="eHandler()">
    <div id="divE">X Y Display</div><br/>
    <div id="main"></div>
  <script>
   function eHandler(){
    let msg='';
```

```
    msg="client-X:" + event.clientX + ",Y:" + event.clientY;
    msg=msg + "\npage-X:" + event.pageX + ",Y:" + event.pageY;
    msg=msg + "\nscreen-X:" + event.screenX + ",Y:" + event.screenY;
    document.getElementById("divE").innerHTML = msg;
  }
  </script>
  </body>
</html>
```

Q.1) Explain the difference between clientX/Y and pageX/Y.



clientXY → relative to the current window

pageXY → relative to the document/page

scrrenXY entire screen size of device

Q.2) Explain the difference between clientX/Y and screenX/Y.

48. Keyboard Events

Kedown
Keypress → doen't give any key values
Keup
When you press the key, it is keydown event and it has a property called event.repeat which is set to true when an event us getting repeated

Q.1) How can you check if control and Z is pressed or not ?

If(event.ctrlKey && event.code===KeyZ){
}

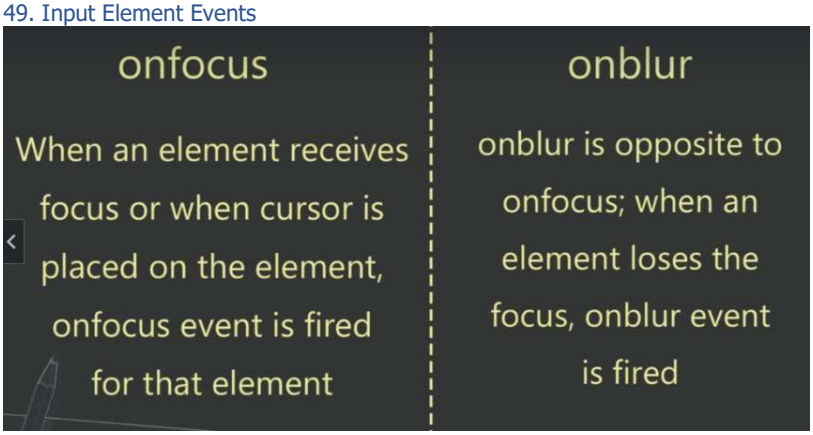
Q.2) What does event.code return ?

If 4 pressed → Digit4
If g pressed → KeyG

Q.3) Explain the difference between event.code and event.key ?

Key returns just value

49. Input Element Events



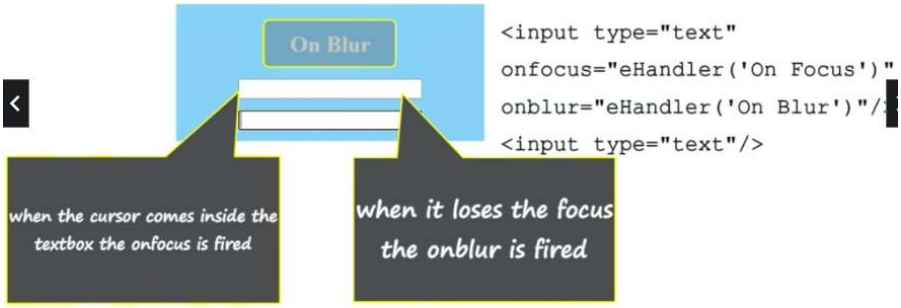
Q.1) What is the difference between onblur and onchange event?

onchange event is generally configured with drop down lists i.e.,

```
<select>
<option>Text</option>
</select>
```

But it can be attached with textbox, checkbox or radio buttons

For textbox the change event is fired when it loses the focus & also when there is a change in the content, which is very useful in case of validations




```
<input type="text"
onfocus="eHandler('On Focus')"
onblur="eHandler('On Blur')"/>
<input type="text"/>
```


```
<input type="text"
onfocus="eHandler('On Focus')"
onchange="eHandler('On Change')"/>
<input type="text"/>
```

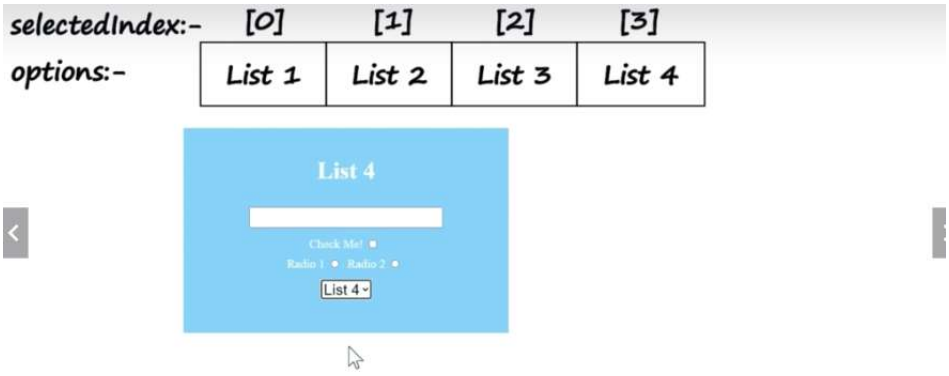
onchange is fired when the focus is lost
and at the same time the content is also changed

sn

onblur also gets fired on losing the focus; same as onchange but

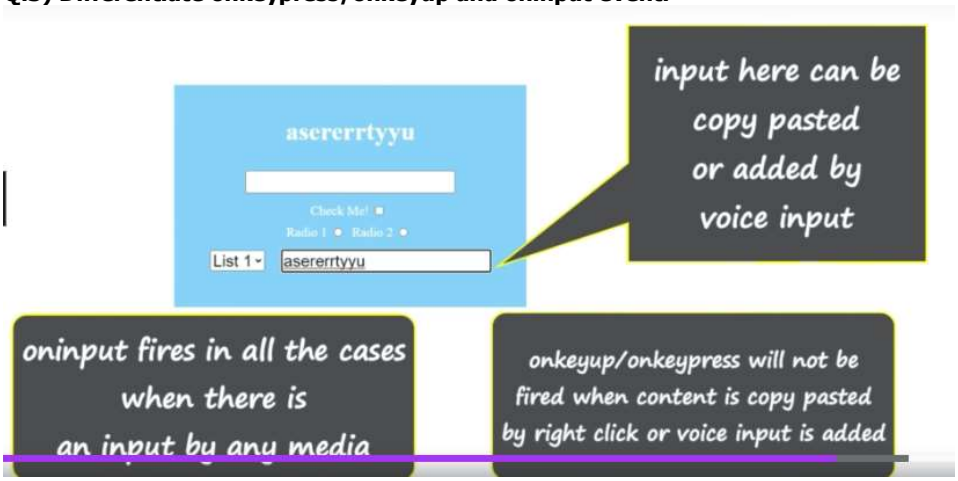
onchange is fired when there is also some changes done in the element.

Q.2) How will you handle events of checkbox and radio buttons?




```
<select onchange="eHandler(this.options[this.selectedIndex].text)">
```

Q.3) Differentiate onkeypress/onkeyup and oninput event.



Q.4) Which event you prefer with select [i.e. drop down list]?
onchange
Q.5) How will you get the text value of selected item from a select element i.e. a drop down list?

```
<select onchange="eHandler(this.options[this.selectedIndex].text)">
   <option value="1">List 1</option>
   <option value="2">List 2</option>
   <option value="3">List 3</option>
   <option value="4">List 4</option>
</select>
```

50. oncut – oncopy – onpaste events

Q.1) Explain about cut,copy and paste events

Q.2) How can you stop an event of cut, copy or paste for an element ?
Oncut = "return false"
Oncut =event.preventDefault()
You can cancel only those events which are cancellable

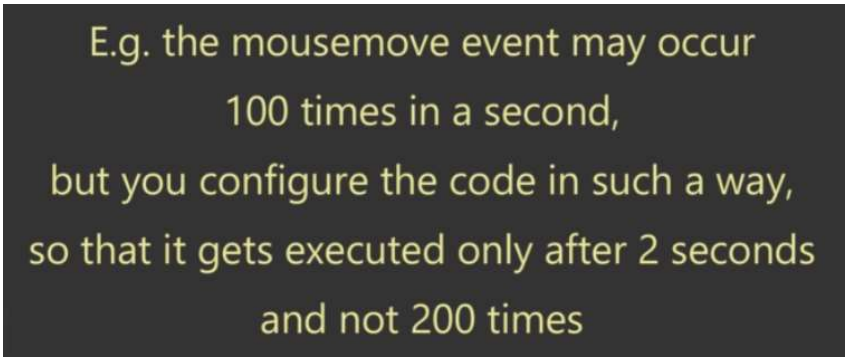
VIII] Debounce and Throttle

51. Introduction to Debounce and Throttling

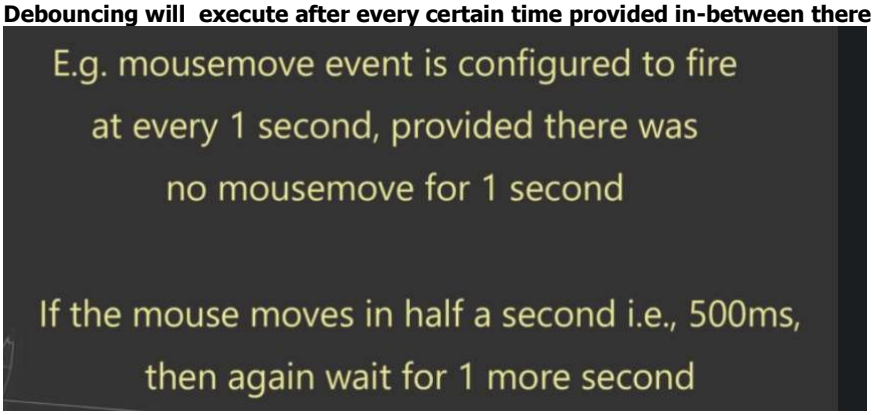
It is a concept to improve the event handling and application performance.
They help you to control /improvise performance while events are executed.

52. Debouncing vs Throttling – The concept

Throttling is about firing event after certain time regardless of when and how many times they are actually fired



Debouncing will execute after every certain time provided in-between there was no event call



53. Throttling – Implementation

```
function throttleCount(){
    if(fireThrottle==true){
        tCnt++;
        th2.innerHTML = "Throttle Count:" + tCnt;
        fireThrottle=false;
        setTimeout(function(){
            fireThrottle=true;
        },500);
    }
}
```

54. Debouncing – Implementation

Debounce also executes after a certain period of time but there is one condition:
The waiting period is reset everytime when you move the mouse.
Let dTimer;

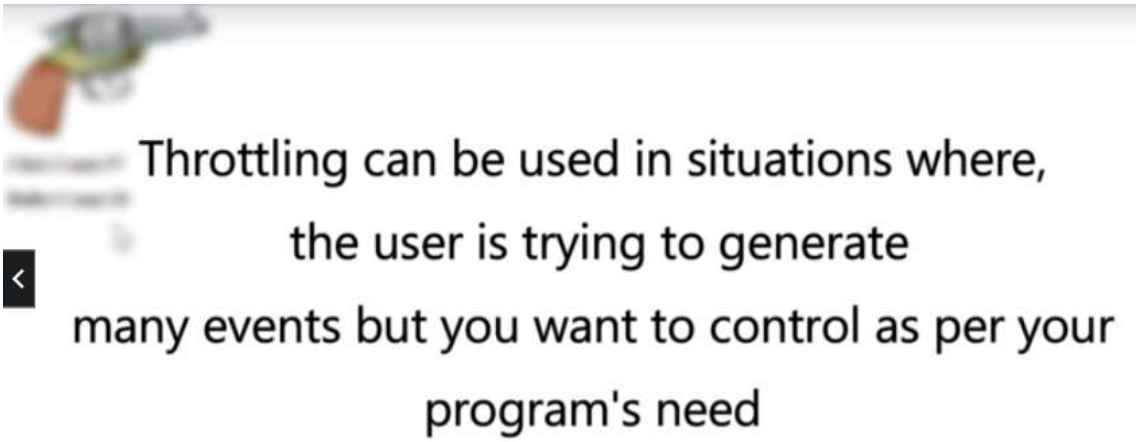
```
function de (method) clearTimeout(handle?: number): void
    window.clearTimeout(dTimer);
    dTimer = setTimeout(function(){
        dCnt++;
        dh2.innerHTML = "Debounce Count:" + dCnt;

    },500);
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <style>
    div{
      width: 500px;
      height: 500px;
      background-color:cornflowerblue;
      border: 1px solid rgb(0, 26, 255);
      margin: auto;
      text-align: center;
      align-items: center;
    }
    h2{
      color: white;
```

```
        vertical-align: middle;
        font-size: 40px;
        font-family:monospace
      }
      body{
        margin: 100px;
      }
  </style>
</head>
<body>
   <div onmousemove="showCounts()">
      <h2 id="nCnt">Normal Count: 0</h2>
      <h2 id="tCnt">Throttle Count: 0</h2>
      <h2 id="dCnt">Debounce Count: 0</h2>
   </div>
   <script>
      let nCnt=0;
      let tCnt=0;
      let dCnt=0;
      let nh2 = document.getElementById("nCnt");
      let th2 = document.getElementById("tCnt");
      let dh2 = document.getElementById("dCnt");
      let fireThrottle=true;
      let dTimer;
      function showCounts(){
         normalCount();
         throttleCount();
         debounceCount();
      }
      function normalCount(){
         nCnt++;
         nh2.innerHTML = "Normal Count: " + nCnt;
      }
      function throttleCount(){
         if(fireThrottle===true){
            tCnt++;
            th2.innerHTML = "Throttle Count: " + tCnt;
            fireThrottle=false;
            setTimeout(function(){
               fireThrottle=true;
            },200)

         }
      }
      function debounceCount(){

         window.clearInterval(dTimer);
          dTimer =  window.setTimeout(function(){

            dCnt++;
            dh2.innerHTML = "Debounce Count: " + dCnt;
          },200);

      }
   </script>
</body>
</html>
```

55. Use case – Throttling

Useful when user performs continuous actions like scrolling and resizing
No of clicks and no of bullet fired in a game



56. Use case – Debouncing
For search like functionality or auto complete feature while typing esp when the search is making API calls on the server

IX] Asynchronous JS
57. What is callback function?

Used in asynchronous process
Replaced now a days with async/await and promises

```
let data;
function fetchData(){
  setTimeout(()=>{
    data={pCode:1001,pName:'Orange'};
  },2000)
}
function displayData(){
   console.log(data);
   console.log("Program ends here....");
}
console.log("Program starts here....");
fetchData();
displayData();
```

Program starts here....
undefined
Program ends here....

```
let data;
function fetchData(cb){
  setTimeout(()=>{
    data={pCode:1001,pName:'Orange'};
    cb();
  },2000)
}

console.log("Program starts here....");
fetchData(function(){
   console.log(data);
   console.log("Program ends here....");
});
```

////////

Q.1)What is an Asynchronous process ?
Q.2)What is the meaning of callback hell or pyramid of doom ?

```
function cSqr(n,cb){
  setTimeout(()=>{
    cb(n*n)
  },1000)
}
cSqr(2,function(res){
   console.log(res);
```

```
    cSqr(res,function(res1){
      console.log(res1);
      cSqr(res1,function(res2){
        console.log(res2);
        cSqr(res2,function(res3){
          console.log(res3);
        })
      })
    })
})
```

It is difficult to handle as there are multiple API calls and callback functions.

58. What is a Promise?

```
let pr = new Promise((resolve,reject)=>{
})
console.log(pr)
//Promise{<pending>} → default

let pr = new Promise((resolve,reject)=>{
   resolve();
})
console.log(pr)
//Promise{undefined} → resolve is not passing any state

let pr = new Promise((resolve,reject)=>{
   resolve("done");
})
console.log(pr)
//Promise{'done'} → resolve is not passing any state

let pr = new Promise((resolve,reject)=>{
   reject();
})
console.log(pr)
//Promise { <rejected> undefined }
//(node:5292) UnhandledPromiseRejectionWarning: undefined

let pr = new Promise((resolve,reject)=>{
   reject("error");
})
console.log(pr)
//Promise { <rejected> 'error' }
//(node:22516) UnhandledPromiseRejectionWarning: error
```

Q.1) How will you configure a process which should be executed at the end of promise resolve or reject?

```
const callBind = require("call-bind");

// Promisification/Promisifying/Promisify
const cSqr=(n)=>{
   // let pr =
   return new Promise((resolve,reject)=>{
      setTimeout(() => {
         resolve(n*n)
      }, 1000);
      })
   // return pr;
}
cSqr(2).then((res)=>{
   console.log(res)
   return res;
}).then((res)=>{
   console.log(res*res)
}).catch((err)=>{
   console.log(error)
}).finally(()=>{
   console.log('END OF PROMISE')
})
```

Q.2) What is chaining the promise? Explain the syntax.

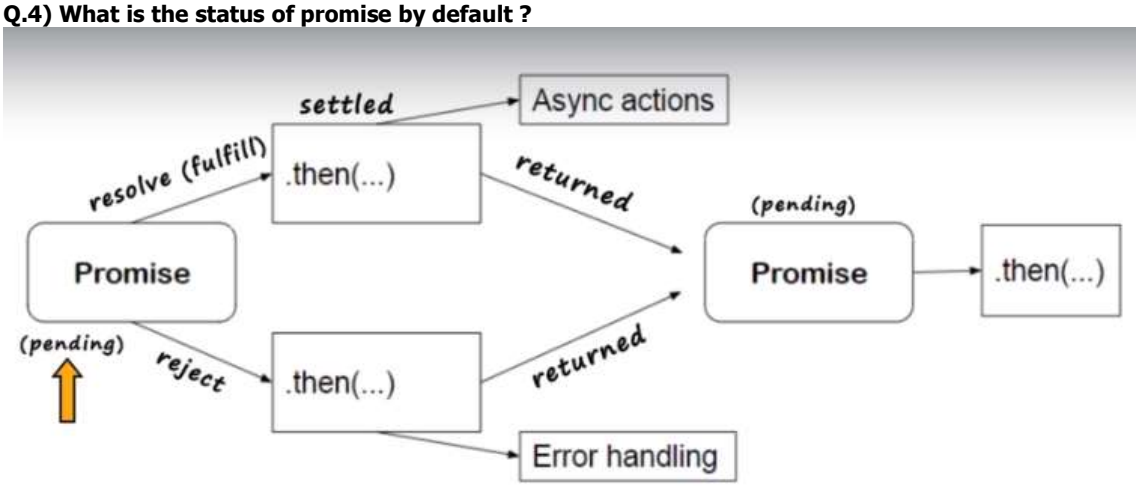
Every then should return a promise when you are chaining the promise with multiple processes

Q.3) What will be the output of the given code?

```
        const pr = new Promise((resolve,reject)=>{
        resolve("Over");
```

```
        setTimeout(()=>{
        resolve("Again");
        },2000)
    })

        pr.then((msg)=>{
        console.log(msg);
        return pr;
    }).then((msg)=>{
        console.log(msg);
    })
// Over
//Over
// once you have one resolve, no more resolves are processed
```

Q.4) What is the status of promise by default ?



Q.5) What will be the status of promise when resolved or rejected ?

59. Explain Promise.all() vs Promise.allSettled() vs Promise.race()

```
Promise.all([p1,p2,p3,...,pn]).then(values)=>{
    //statements
}
```

allSettled() method waits for all promises regardless of theie state and returns Promise at the end
Promises.allSettled()... →

```
[
  { status: 'fulfilled', value: 'Promise1' },
  { status: 'fulfilled', value: 'Promise2' },
  { status: 'rejected', reason: 'Promise3' }
]
```

Race() method returns a promise as soon as any of the promise returns the state from the iterable list provided. Kind of race among promises

Q.1) What will be the output if the iterable is not a promise inside all() method arguments,e.g as shown below where "p2" is not a promise ?

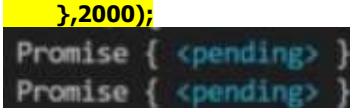
```
    const p1 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve("Promise1");
    },2000)
})
    const p2 = "Normal string";
    const p3 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve("Promise3");
    },2000)
})
    Promise.all([p1, p2,p3]).then((prMsgs) => {
    console.log(prMsgs);
}).catch((err)=>{
```

```
    console.log(err);
})
```
//Promise1 Normal String Promise3

Q.2) How will you handle rejected multiple promises in the most efficient way ?
Promise.allSettled()
Q.3) Explain the purpose of Promise.race() method.

Q.4) What will be the output of empty iterable if given as a parameter inside the race() method as shown in this code ?

```
const obj = Promise.race([]);
console.log(obj);
setTimeout(()=>{
        console.log(obj);
},2000);
```

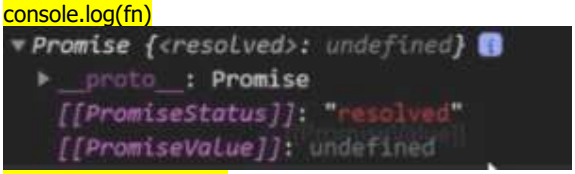


//pending forever

60. Explain functionality of async/await?

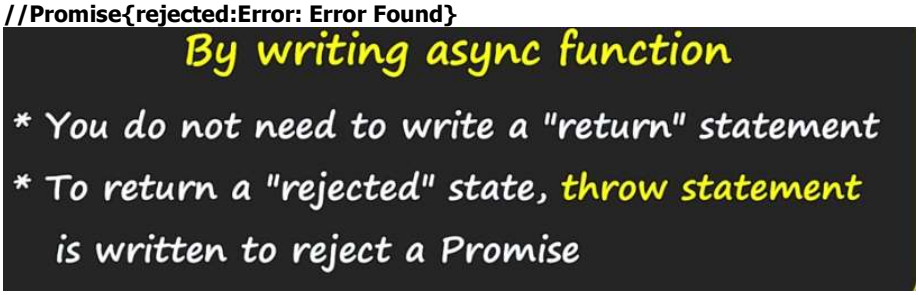
```
function test(){
}
const fn=test();
console.log(fn)
```
// undefined

```
Async  function test(){
}
const fn=test();
console.log(fn)
```

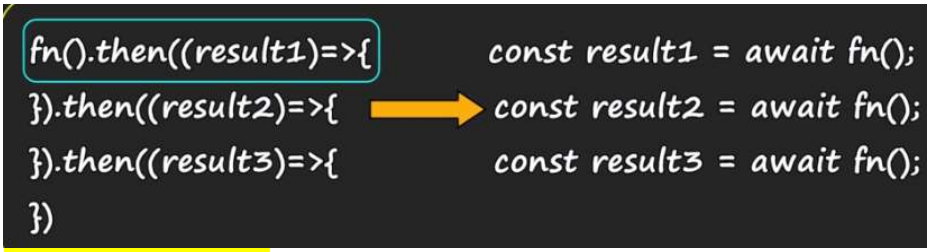



```
Async  function test(){
        Return "done"
}
const fn=test();
console.log(fn)
```
//Promise{'done'}

```
Async  function test(){
Throw new error("Error found")
}
const fn=test();
console.log(fn)
```
//Promise{rejected:Error: Error Found}




```
Const test = sync()=>{
        Return 0;
}
```




```
let res = await cSqr(2);
```

```
console.log(res);
const cSqr=(a)=>{
   return new Promise((resolve,reject)=>{
       setTimeout(()=>{
          resolve(a*a);
       },2000)
    });
 }
async function showRes(){
   let res = await cSqr(2);
   console.log(res);
}
showRes();
```

Q.1) Can you write await without async ?
NO
Q.2) Do you need to return a promise from an async function ?
No. Function having async keyword returns promise always by default
Q.3) Can you have an async IIFE ?
Any function can be async IIFE function
Q.4) Which is the best way to handle errors with await ?
Try catch
Q.5) Explain the syntax of Promise.all() with async/await ?

61. AJAX and XMLHTTPRequest(XHR)

XHR Object
Methods → Open() and send(),
-abort()
Events →
Load
Onprogress – fired continuously when a request receives data
Onerror – fired when there is an error encountered while requesting
Onreadystatechange –

Property → response, status

Q.1) What is AJAX?
Fetch API or XHR Object is used to make an ajax call
Q.2) How will you set a timeout with an XHR request?
Xhr.timeout=2000
Request.ontimeout=function(){
}
Q.3) How can you set the data format for response?
Xhr.responseType = "" → text only
Xhr.responseType = 'text'

Xhr.responseType = "blob" → binary
Xhr.responseType = "document" → html doc
Xhr.responseType = "json"

Q.4) Explain "readyState" property of XHR.
To show progressbar – readystate=3
Q.5) Explain the "onreadystatechange" event.
Fired when there is a new value in the readyState property
Q.6) How can you force the XHR request to stop the execution?
Xhr.abort()
Q.7) When is the "load" method called?
Called when the call to the server is completely successful
It is executed when the AJAX call is successfully completed
Q.8) How will you set HTTP headers? Explain the sequence.

```
async function fetchData(){
let data;
const response = await fetch('https://jsonplaceholder.typicode.com/users');
console.log(response);
if(response.ok){
   data = await response.json(); //will do the parsing of response to json
}
console.log(data);
}
fetchData();
```

```
request.open('GET','https://jsonplaceholder.typicode.com/comments
XMLHttpRequest.setRequestHeader(header,value);
request.send();
```

After open() set the headers

```
let todo={
   userId: 101,
   id: 1,
   title: "New title",
   completed: false
}
const response = fetch('https://jsonplaceholder.typicode.com/todos',{
  method:'POST',
  headers:{
     'Content-Type':'application/json;charset=utf-8'
  },
  body:JSON.stringify(todo)
}).then(res=>console.log(res));
```

Q.9) Explain different ways of getting response header values?

62. The fetch API

Const responsePromise = fetch(url[,optional object])
url – resource from where to get the data
by default – get request
Optional object – contains keys which are information set with HTTP call
Q.1) Explain the fetch API functionality

```
fetch('https://jsonplaceholder.typicode.com/users')
      .then(response=>response.json())
      .then(data=>console.log(data));
```

- fetch API helps in making an AJAX call
-add data – perform CRUD operations
-Downloading/Uploading files – dealing with images etc.
Q.2) How will you make a DELETE request with fetch API?
Method:'DELETE'

63. Async Iterators and Generators

Q.1) What is Symbol.asyncIterator?
Symbol.asyncIterator is a way to implement with asynchronous process

```
let obj={
  a:10,
  b:20,
  [Symbol.asyncIterator](){
```

```
return {
   async next(){
     await new Promise(resolve=>setTimeout(resolve,1000));
      if(obj.a<obj.b){
          return {value:obj.a++,done:false};
      }else{
          return {done:true};
      }
   }
}

}
};

(async ()=>{
  for await (let i of obj){
    console.log(i);
    }
})();
```

Q.2) Can we use spread operator with async iterator e.g. [...obj]?
No.
It is not possible to use the spread bcoz spread is going to return an array.
The array is returned when the values are retrieved
Whereas in async process, data us retrieved in small chunks and then it gathers
Q.3) How will you handle the async iterator using generator function syntax?

```
let obj={
  start:10,
  end:15,
  async *[Symbol.asyncIterator](){
      for(let cnt=this.start;cnt<=this.end;cnt++){
        await  new Promise(resolve=>setTimeout(resolve,1000));
            yield cnt;
      }
}
};

(async ()=>{
  for await(let i of obj){
    console.log(i);
    }
})();
```

X] Map, Set, WeakMap and WeakSet(ES6 Data Structure)

64. Map

```
let product = new Map();
product.set('pCode','1001')
       .set(1,'Apple')
       .set(true,'Available');

console.log(product)
```

Set method returns the map
Values() – loop through all values
Keys() - loop through all keys
Entries() - loop through all keys and values

```
for(let [k,v] of product.entries()){
    console.log(e);
}
```

Object => entries() can be used to convert map into simple object

```
let obj={
    pCode:1001,
    pName:'Orange',
    price:56
}
console.log(Object.entries(obj));
```

Map.has(key) → checks whether the given key is existing in the map or not
Product.has('pCode')

Q.1) What are the advantages of using Map over objects?
Map is iterable

Map	*Object*
Map's keys can also be values - they can be of any datatype	Object's have keys as strings - value to value mapping is possible

Map	*Object*
Keys are ordered - the insertion & retrieval is always in the same order	Keys are not ordered

Map	*Object*
Map has size property which returns number of key/values pairs	Doesn't have any direct way - you need to code to get the count

Maps are iterable	Objects are not iterable

Quite optimized for data manipulation	Not optimized for data manipulation

Q.2) How will you iterate through Map ?
Map.forEach()

```
Product.forEach((v,k,m)=>{
        Console.log(`${v}-${k}`)
})
```

Q.3) Can you assign key/values with this syntax?

map[`key`]=value
Explain the reason for the answer.
Type not maintained.. better to use set method
Keys will become strings only

Q.4) How can you convert simple objects to Map?

```
Let obj = {
        pCode:1001,
        pName:'orange',
        price:55
}
Let product = new map(Object.entries(obj))
```

Q.5) How can you convert a Map into simple objects?

```
Let obj = Object.fromEntries(product.entries())
```

Q.6) How will you remove an item or all items from Map?
Delete(key) → removes values by key
Clear() → wipes out everything from the map

65. Set

Q.1) What is the difference between Map and Set ?
Set does not have keys.
It has only values and that too unique values.
So if you add the same or duplicate value, it is ignored.
It means you will have unique or non repeating values only
Adding values in set →

```
Let names = new Set()
Names.add(1)
Names.add(2)
Names.add(3)
Names.add(1)
Console.log(names)
Console.log(names.size)
//Set(1,2,3)
//3
```

Set is also iterable, so you can put a for…of directly on the Set collection. Or forEach()

```
For(let v of names){
        Console.log(v)
}
For(let v of names.values()){
        Console.log(v)
}
For(let v of names.values()){
        Console.log(v)
}
```

All have same output

```
//1 2 3
For(let v of names.entries()){
        Console.log(v)
}
```


```
Names.forEach((v1,v2,s)=>{

})
```

Q.2) How will you remove specific values from Set ?
Delete() and clear()
Q.3) What will entries() method with Set will return ?
66. WeakMap() and WeakSet()
Q.1) Explain difference between Map and WeakMap ?
-Map and Set hold the keys in a strong way, whereas
WeakMap and WeakSet hold the keys weakly.
It means if the keys are garbage collected, so do the values
Values are removed on GC if there is no direct reference of the object
- Keys in WeakMap are only objects, you cannot have primitive keys like string, number etc.\
Bcoz of weak keys, WeakMap does not have iterable methods like keys(), values() or entries()

- Weak Collections do not hold values when the garbage is collected – this is very handy with the DOM, where DOM nodes are removed the garbage collected locations will be free as they are not held strongly
Q.2) Explain difference between Set and WeakSet ?

These weak collections are also used, when you want to make sure that there is no memory leakages

XI] Debugging Techniques

Run-time and design-time errors?
Run-time errors – bugs
- often faced by end -user

Design-time errors – syntax errors
- faced by developers

Google chrome plugins for debug

67. Introduction to debugging & Developer Tool

68. Understanding Developer tool

1. How do you make web page responsive?
Mediaquery, flex, bootstrap etc

2. What is a Breakpoint?
To break or pause the execution of code for debugging

69. Debugging Pane : Watch, Call Stack & Scope

Watch → you can check values and all other info provided by it for the current and additional variables, along with inspecting the changes as the program continues to run

Callstack → data structure that provides info about the current subroutine or function of the program beaing debugged
- it displays stack of all the functions called and currently executing
- using callstack when debugging is performed, it is termed as stack tracing

2. What is a stacktrace?

A stack trace works on the call stack which shows a stack of functions and subroutines currently under execution
It is a list of subroutine calls that are performed at the time of execution
Call stack us used to analyze the contents of each subroutine and how they run

70. Debugging Pane : Code navigation

Step into the next function call option jumps inside the next function related to the event

71. Event Listener Breakpoints

Line Breakpoints → used when exact region of code to debug is known,
Event listener breakpoints →
Conditional Breakpoints →

1. How does programmatic breakpoints work?
Debugger
2. Conditional breakpoints

72. Conditional & Programmatic breakpoints

73. What are DOM breakpoints?

R-click→inspect→break on→
1. Subtree Modification breakpoint → triggered when there are changes like adding, moving or removing the child nodes i.e. modification in the subtree
2. Attribute Modification → debugger pauses when there are changes in the attributes of the node or elements
3. Node Removal Breakpoint

74. How does XHR/Fetch breakpoints work?

To break execution when XHR URL request or a fetch API that the page requests has a specific string pattern, XHR/fetch breakpoints are used.
It fetches URL and pauses when the debugger runs into the specified string pattern.

75. Exception Breakpoints

To pause the execution on a line that is throwing either caught or uncaught exception.
1. how to debug the code with exceptions?

76. The "console" Object Methods

Console object provides access to the developer tool debugger console from onside the code
Part of window object
Console.log
Console.info → outputs the info log messages to the console
Console.error
Console.table
Console.time
Console.warn
Console.clear
Console.debug
Console.time

77. console.time & related methods

Console.time() sets a timer and takes a specific name or a label for that timer as a parameter

```
Console.time('timer1')
...
Console.timeEnd('timer1')
```

Used to check the response time of your code and manage it
Console.timeLog('timer1') → provides current value of the timer, set by using Console.time() method

XI] Coding Exercises

78. Is given value an array or not?

```
let arr = [4,5,6]
console.log(typeof arr)         // object
console.log(Object.prototype.toString.call(arr))  // [object array]

if(toString.call(arr)==="[object Array]"){
    console.log(true)
} else {
    console.log(false)
}

// toString.call is useful when you need to deal with more detailed type of values
```

79. Remove duplicate values from Array

```
let arr = [4,5,6,4,7,8,7]

let arr1=arr.filter((v,i)=>{
    console.log(` ${v} - ${arr.indexOf(v)} - ${i}`)
    return arr.indexOf(v)===i
})

console.log(arr1)

//ES6 way
let arr2=[... new Set(arr)]
console.log(arr2)
```

80. Remove null, undefined, 0, NaN and '' from array?

```
let arr = [false,0,NaN,6,undefined,90,'','Hi']

//Truthy expression and Falsy Expression
// for false,0,NaN,undefined, or '' the JS returns falsy
let narr =arr.filter((v,i)=>{
    return(v)
})
//O/P:
// false
// 0
// NaN
// 6
// undefined
// 90
//
// Hi
console.log(narr)
//[ 6, 90, 'Hi' ]
// To remove elements like false, NaN,0, etc from an array, you can use the truthy and falsy technique
```

81. Finding Factorial

82. Prime Number

83. Vowel and Consonant - Algorithm

84. Array intersection and union – ES6 way

```
let arr1 = [4,5,6,7,9,0]
let arr2 = [4,5,12,34,66,77,88]

let intersection=arr1.filter((v)=>{
    return arr2.includes(v)
})
console.log(intersection)
let union=[...new Set([...arr1,...arr2])]
console.log(union)
```

CODING OUTPUT QUESTIONS

Question 1: (Strings, Numbers, Boolean)

```
var num = 8;
var num = 10;
console.log(num);
```

Answer — 10

Explanation — With the var keyword, you can declare multiple variables with the same name. The variable will then hold the latest value. You cannot do this with let or const since they're block-scoped.

Question 2:

```
function sayHi() {
  console.log(name);
  console.log(age);
  var name = 'Ayush';
  let age = 21;
}

sayHi();
```

Answer — undefined and ReferenceError
Explanation — Within the function, we first declare the name variable with the var keyword. This means that the variable gets hoisted (memory space is set up during the creation phase) with the default value of undefined, until we actually get to the line where we define the variable. We haven't defined the variable yet on the line where we try to log the name variable, so it still holds the value of undefined.

Variables with the let keyword (and const) are hoisted, but unlike var, don't get initialized. They are not accessible before the line we declare (initialize) them. This is called the **"temporal dead zone"**. When we try to access the variables before they are declared, JavaScript throws a ReferenceError.

Question 3:

```
function getAge() {
  'use strict';
  age = 21;
  console.log(age);
}

getAge();
```

Answer — ReferenceError
Explanation —With "use strict", you can make sure that you don't accidentally declare global variables. We never declared the variable age, and since we use "use strict", it will throw a reference error. If we didn't use "use strict", it would have worked, since the property age would have gotten added to the global object.

Question 4:

```
+true;
!'Ayush';
```

Answer — 1 and false
Explanation —The unary plus tries to convert an operand to a number. true is 1, and false is 0.

The string 'Ayush' is a truthy value. What we're actually asking, is "is this truthy value falsy?". This returns false.

Question 5:

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

Answer — 0 2 2.
Explanation —The postfix unary operator ++:

Returns the value (this returns 0).
Increments the value (number is now 1).
The prefix unary operator ++:

Increments the value (number is now 2).
Returns the value (this returns 2).
This returns 0 2 2.

Question 6:

```
function sum(a, b) {
  return a + b;
}

sum(1, '2');
```

Answer — "12"
Explanation —JavaScript is a dynamically typed language: we don’t specify what types of certain variables are. Values can automatically be converted into another type without you knowing, which is called implicit type coercion. Coercion is converting from one type into another.

In this example, JavaScript converts the number 1 into a string, in order for the function to make sense and return a value. During the addition of a numeric type (1) and a string type ('2'), the number is treated as a string. We can concatenate strings like "Hello" + "World", so what's happening here is "1" + "2" which returns "12".

Question 7:

```
String.prototype.giveAyushPizza = () => {
```

```
  return 'Just give Ayush pizza already!';
};

const name = 'Ayush';

name.giveAyushPizza();
```

Answer — "Just give Ayush pizza already!"
Explanation —String is a built-in constructor, which we can add properties to. I just added a method to its prototype. Primitive strings are automatically converted into a string object, generated by the string prototype function. So, all strings (string objects) have access to that method!

Question 8:

```
for (let i = 1; i < 5; i++) {
  if (i === 3) continue;
  console.log(i);
}
```

Answer — 1 2 4
Explanation —The continue statement skips an iteration if a certain condition returns true.

Question 9:

```
function sayHi() {
  return (() => 0)();
}

console.log(typeof sayHi());
```

Answer — "number"
Explanation —The sayHi function returns the returned value of the immediately invoked function expression (IIFE). This function returned 0, which is type "number".

FYI: there are only 7 built-in types: null, undefined, boolean, number, string, object, and symbol. "function" is not a type, since functions are objects, it's of type "object".

Question 10:

```
console.log(typeof typeof 1);
```

Answer —"string"
Explanation —typeof 1 returns "number". And typeof "number" returns "string".

Question 11:

```
!!null;
!!'';
!!1;
```

Answer — false false true
Explanation —null is falsy. !null returns true. !true returns false.

"" is falsy. !"" returns true. !true returns false.

1 is truthy. !1 returns false. !false returns true.

Question 12:

```
[...'Ayush'];
```

Answer — ["A", "y", "u", "s", "h"]
Explanation —A string is an iterable. The spread operator maps every character of an iterable to one element.

Question 13:

```
console.log(3 + 4 + '5');
```

Answer — "75"
Explanation — Operator associativity is the order in which the compiler evaluates the expressions, either left-to-right or right-to-left. This only happens if all operators have the same precedence. We only have one type of operator: +. For addition, the associativity is left-to-right.

3 + 4 gets evaluated first. This results in the number 7.

7 + '5' results in "75" because of **coercion**. JavaScript converts the number 7 into a string. We can concatenate two strings using the +operator. "7" + "5" results in "75".

Question 14:

```
var a = 10;
var b = a;
b = 20;
console.log(a);
console.log(b);
var a = 'Ayush';
```

```
var b = a;
b = 'Verma';
console.log(a);
console.log(b);
```
Answer —
1. 10 and 20
2. "Ayush" and "Verma"
Explanation —The value assigned to the variable of primitive data type is tightly coupled. That means, whenever you create a copy of a variable of primitive data type, the value is copied to a new memory location to which the new variable is pointing to. When you make a copy, it will be a real copy.

Question 15:
```
function sum(){
  return arguments.reduce((a, b) => a + b);
}

console.log(sum(1,2,3)); (1)
function sum(...arguments){
  return arguments.reduce((a, b) => a + b);
}

console.log(sum(1,2,3)); (2)
```
Answer —
1. Error will be thrown.
2. 6
Explanation —
1. Arguments are not fully functional array, they have only one method length. Other methods cannot be used on them.
2. ... rest operator creates an array of all functions parameters. We then use this to return the sum of them.

Question 16:
```
console.log(1 == '1');
console.log(false == '0');
console.log(true == '1');
console.log('1' == '01');
console.log(10 == 5 + 5);
```
Answer — true true true false true.
Explanation —'1' == '01' as we are comparing two strings here they are different but all other equal.

Question 17:
```
console.log('1' - - '1'); (1)
console.log('1' + - '1'); (2)
```
Answer —
1. 2
2. "1–1"
Explanation —
1. With type coercion string is converted to number and are treated as 1 - -1 = 2.
2.+ operator is used for concatenation of strings in javascript, so it is evaluated as '1' + '-1' = 1-1.

Question 18:
```
let lang = 'javascript';
(function(){
  let lang = 'java';
})();

console.log(lang); (1)
(function(){
  var lang2 = 'java';
})();

console.log(lang2); (2)
```
Answer —
1. "javascript"
2. Error will be thrown.
Explanation —
1. Variables defined with let are blocked scope and are not added to the global object.
2. Variables declared with var keyword are function scoped, so wrapping the function inside a closure will restrict it from being accessed outside that is why it throws error

Question 19:
```
(function(){
  console.log(typeof this);
}).call(10);
```

Answer — object
Explanation — call invokes the function with new this which in this case is 10 which is basically a constructor of Number and Number is object in javascript.

Question 20:
console.log("ayushv.medium.com/" instanceof String); (1)
const s = new String('ayushv.medium.com/');
console.log(s instanceof String); (2)
Answer —
1. false
2. true
Explanation — Only strings defined with String() constructor are instance of it.

Question 21: (Objects, Arrays)
const obj = { a: 'one', b: 'two', a: 'three' };
console.log(obj);
Answer — { a: "three", b: "two"}"
Explanation —If you have two keys with the same name, the key will be replaced. It will still be in its first position, but with the last specified value.

Question 22:
let c = { greeting: 'Hey!' };
let d;
d = c;
c.greeting = 'Hello';
console.log(d.greeting);
Answer — Hello
Explanation — In JavaScript, all objects interact by reference when setting them equal to each other.

First, a variable c holds a value to an object. Later, we assign d with the same reference that c has to the object. When you change one object, you change all of them.

Question 23:
let a = 3;
let b = new Number(3);
let c = 3;
console.log(a == b);
console.log(a === b);
console.log(b === c);
Answer — true false false
Explanation —new Number() is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an object.

When we use the == operator, it only checks whether it has the same value. They both have the value of 3, so it returns true.

However, when we use the === operator, both value and type should be the same. It's not: new Number() is not a number, it's an object. Both return false.

Question 24:
function getAge(...args) {
** console.log(typeof args);**
}
getAge(21);
Answer — "object"
Explanation —The rest parameter (...args) lets us "collect" all remaining arguments into an array. An array is an object, so typeof args returns "object".
Question 25:
let greeting;
greetign = {}; // Typo!
console.log(greetign);
Answer — {}
Explanation —It logs the object, because we just created an empty object on the global object! When we mistyped greeting as greetign, the JS interpreter actually saw this as global.greetign = {} (or window.greetign = {} in a browser).

In order to avoid this, we can use "use strict". This makes sure that you have declared a variable before setting it equal to anything.

Question 26:
function checkAge(data) {
** if (data === { age: 18 }) {**
**   console.log('You are an adult!');**

```
  } else if (data == { age: 18 }) {
    console.log('You are still an adult.');
  } else {
    console.log(`Hmm.. You don't have an age I guess`);
  }
}

checkAge({ age: 18 });
```

Answer — Hmm.. You don't have an age I guess
Explanation —When testing equality, primitives are compared by their value, while objects are compared by their reference. JavaScript checks if the objects have a reference to the same location in memory.

The two objects that we are comparing don't have that: the object we passed as a parameter refers to a different location in memory than the object we used in order to check equality.

This is why both { age: 18 } === { age: 18 } and { age: 18 } == { age: 18 } return false.

Question 27:

```
const obj = { 1: 'a', 2: 'b', 3: 'c' };
const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty('1');
obj.hasOwnProperty(1);
set.has('1');
set.has(1);
```

Answer — true true false true
Explanation —All object keys (excluding Symbols) are strings under the hood, even if you don't type it yourself as a string. This is why obj.hasOwnProperty('1') also returns true.

It doesn't work that way for a set. There is no '1' in our set: set.has('1') returns false. It has the numeric type 1, set.has(1) returns true.

Question 28:

```
const a = {};
const b = { key: 'b' };
const c = { key: 'c' };

a[b] = 123;
a[c] = 456;

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

Answer —456
Explanation —Object keys are automatically converted into strings. We are trying to set an object as a key to object a, with the value of 123.

However, when we stringify an object, it becomes "[object Object]". So what we are saying here, is that a["[object Object]"] = 123. Then, we can try to do the same again. c is another object that we are implicitly stringifying. So then, a["[object Object]"] = 456.

Then, we log a[b], which is actually a["[object Object]"]. We just set that to 456, so it returns 456.

Question 29:

```
const numbers = [1, 2, 3];
numbers[10] = 11;
console.log(numbers);
```

Answer — [1, 2, 3, 7 x empty, 11]
Explanation —When you set a value to an element in an array that exceeds the length of the array, JavaScript creates something called "empty slots". These actually have the value of undefined, but you will see something like:

[1, 2, 3, 7 x empty, 11]

depending on where you run it (it's different for every browser, node, etc.).

Question 30:

```
let person = { name: 'Ayush' };
const members = [person];
person = null;
console.log(members);
```

Answer — [{ name: "Ayush" }]
Explanation —We are only modifying the value of the person variable, and not the first element in the array, since that element has a different (copied) reference to the object. The first element in members still holds its reference to the original object. When we log the members array, the first element still holds the value of the object, which gets logged.

Question 31:

```
const person = {
  name: 'Ayush',
  age: 21,
};

for (const item in person) {
  console.log(item);
}
```

Answer — "name", "age"

Explanation —With a for-in loop, we can iterate through object keys, in this case name and age. Under the hood, object keys are strings (if they're not a Symbol). On every loop, we set the value of item equal to the current key it's iterating over. First, item is equal to name, and gets logged. Then, item is equal to age, which gets logged.

Question 32:

```
[1, 2, 3].map(num => {
  if (typeof num === 'number') return;
  return num * 2;
});
```

Answer — [undefined, undefined, undefined]

Explanation —When mapping over the array, the value of num is equal to the element it's currently looping over. In this case, the elements are numbers, so the condition of the if statement typeof num === "number" returns true. The map function creates a new array and inserts the values returned from the function.

However, we don't return a value. When we don't return a value from the function, the function returns undefined. For every element in the array, the function block gets called, so for each element, we return undefined.

Question 33:

```
var obj = {a:1};
var secondObj = obj;
secondObj.a = 2;
console.log(obj);
console.log(secondObj);

var obj = {a:1};
var secondObj = obj;
secondObj = {a:2};
console.log(obj);
console.log(secondObj);
```

Answer —

1. { a:2 }and { a:2 }
2. { a:1 }and { a:2 }

Explanation —

1. If the object property is changed, then the new object is pointing to the same memory address, so the original object property will also change. (call by reference)
2. If the object is reassigned with a new object then it is allocated to a new memory location, i.e it will be a real copy (call by value).

Question 34:

```
const arrTest = [10, 20, 30, 40, 50][1, 3];
console.log(arrTest);
```

Answer — 40

Explanation —The last element from the second array is used as the index to get the value from first array like arrTest[3].

Question 35:

```
console.log([] + []);            (1)
console.log([1] + []);           (2)
console.log([1] + "abc");        (3)
console.log([1, 2, 3] + [1, 3, 4]); (4)
```

Answer —

1. ""
2. "1"
3. "1abc"
2. "1,2,31,3,4"

Explanation —

1. An empty array is while printing in console.log is treated as Array.toString(), so it prints an empty string.
2. An empty array when printed in console.log is treated as Array.toString() and so it is basically "1" + "" = "".
3. "1" + "abc" = "1abc".
4."1, 2, 3" + "1, 3, 4" = "1,2,31,3,4".

Question 36:

```
const ans1 = NaN === NaN;
const ans2 = Object.is(NaN, NaN);
```
console.log(ans1, ans2);
Answer — false true
Explanation —NaN is a unique value so it fails in equality check, but it is the same object so Object.is returns true.

Question 37:
```
var a = 3;
var b = {
  a: 9,
  b: ++a
};
console.log(a + b.a + ++b.b);
```
Answer — 18
Explanation — Prefix operator increments the number and then returns it. So the following expression will be evaluated as 4 + 9 + 5 = 18.

Question 38:
```
const arr = [1, 2, undefined, NaN, null, false, true, "", 'abc', 3];
console.log(arr.filter(Boolean)); (1)
const arr = [1, 2, undefined, NaN, null, false, true, "", 'abc', 3];
console.log(arr.filter(!Boolean)); (2)
```
Answer —
1. [1, 2, true, "abc", 3].
2. It will throw an error.
Explanation —
1. Array.filter() returns the array which matches the condition. As we have passed Boolean it returned all the truthy value.
2. As Array.filter() accepts a function, !Boolean returns false which is not a function so it throws an error Uncaught TypeError: false is not a function.

Question 39:
```
const person = {
  name: 'Ayush Verma',
  .25e2: 25
};

console.log(person[25]);
console.log(person[.25e2]);
console.log(person['.25e2']);
```
Answer — 25 25 undefined
Explanation —While assign the key the object evaluates the numerical expression so it becomes person[.25e2] = person[25]. Thus while accessing when we use 25 and .25e2 it returns the value but for '.25e2' is undefined.

Question 40:
```
console.log(new Array(3).toString());
```
Answer — ",,"
Explanation —Array.toString() creates string of the array with comma separated values.

Question 41: (setTimeout and "this" keyword)
```
const foo = () => console.log('First');
const bar = () => setTimeout(() => console.log('Second'));
const baz = () => console.log('Third');
bar();
foo();
baz();
```
Answer — First Third Second
Explanation —We have a setTimeout function and invoked it first. Yet, it was logged last.

This is because, in browsers, we don't just have the runtime engine, we also have something called a WebAPI. The WebAPI gives us the setTimeout function to start with and for example the DOM. The WebAPI can't just add stuff to the stack whenever it's ready. Instead, it pushes the callback function to something called the queue. An event loop looks at the stack and task queue. If the stack is empty, it takes the first thing on the queue and pushes it onto the stack.

Question 42:
```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```
Answer — 3 3 3 and 0 1 2

Explanation — Because of the event queue in JavaScript, the setTimeout callback function is called after the loop has been executed. Since the variable i in the first loop was declared using the var keyword, this value was global. During the loop, we incremented the value of i by 1 each time, using the unary operator ++. By the time the setTimeout callback function was invoked, i was equal to 3 in the first example.
In the second loop, the variable i was declared using the let keyword: variables declared with the let (and const) keyword are block-scoped (a block is anything between { }). During each iteration, i will have a new value, and each value is scoped inside the loop.

Question 43:

```
let obj = {
   x: 2,
   getX: function() {
      setTimeout(() => console.log('a'), 0);
      new Promise( res =>  res(1)).then(v => console.log(v));
      setTimeout(() => console.log('b'), 0);
   }
}
obj.getX();
```

Answer — 1 a b
Explanation —When a macrotask is completed, all other microtasks are executed in turn first, and then the next macrotask is executed.

Mircotasks include: MutationObserver, Promise.then() and Promise.catch(), other techniques based on Promise such as the fetch API, V8 garbage collection process, process.nextTick() in node environment.

Marcotasks include initial script, setTimeout, setInterval, setImmediate, I/O, UI rendering.

An immediately resolved promise is processed faster than an immediate timer because of the event loop priorities dequeuing jobs from the job queue (which stores the fulfilled promises' callbacks) over the tasks from the task queue (which stores timed out setTimeout() callbacks).

Question 44:

```
const shape = {
  radius: 10,
  diameter() {
    return this.radius * 2;
  },
  perimeter: () => 2 * Math.PI * this.radius,
};
console.log(shape.diameter());
console.log(shape.perimeter());
```

Answer — 20 and NaN
Explanation —Note that the value of diameter is a regular function, whereas the value of perimeter is an arrow function.

With arrow functions, the this keyword refers to its current surrounding scope, unlike regular functions! This means that when we call perimeter, it doesn't refer to the shape object, but to its surrounding scope (window for example).

There is no value radius on that object, which returns NaN.

Question 45:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const member = new Person('Ayush', 'Verma');
Person.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};
console.log(member.getFullName());
```

Answer — TypeError
Explanation —In JavaScript, functions are objects, and therefore, the method getFullName gets added to the constructor function object itself. For that reason, we can call Person.getFullName(), but member.getFullName throws a TypeError.

If you want a method to be available to all object instances, you have to add it to the prototype property:

```
Person.prototype.getFullName = function() {
  return `${this.firstName} ${this.lastName}`;
};
```

Question 46:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
```

```
  this.lastName = lastName;
}

const ayush = new Person('Ayush', 'Verma');
const sarah = Person('Sarah', 'Smith');

console.log(ayush);
console.log(sarah);
```

Answer — Person {firstName: "Ayush", lastName: "Verma"} and undefined
Explanation —For sarah, we didn't use the new keyword. When using new, this refers to the new empty object we create. However, if you don't add new, this refers to the global object!

We said that this.firstName equals "Sarah" and this.lastName equals "Smith". What we actually did, is defining global.firstName = 'Sarah' and global.lastName = 'Smith'. sarah itself is left undefined, since we don't return a value from the Person function.

Question 47:

```
const person = { name: 'Ayush' };

function sayHi(age) {
  return `${this.name} is ${age}`;
}

console.log(sayHi.call(person, 21));
console.log(sayHi.bind(person, 21));
```

Answer — Ayush is 21 function
Explanation —With both, we can pass the object to which we want the this keyword to refer. However, .call is also executed immediately!

.bind. returns a copy of the function, but with a bound context! It is not executed immediately.

Question 48:

```
let obj = {
    x: 2,
    getX: function() {
        console.log(this.x);
    }
}
obj.getX(); (1)
let x = 5;
let obj = {
    x: 2,
    getX:() => {
        console.log(this.x)
    }
}
obj.getX(); (2)
let x = 5;
let obj = {
    x: 2,
    getX: function(){
        let x = 10;
        console.log(this.x);
    }
}
let y = obj.getX;
y(); (3)
```

Answer —
1) 2
2) 5
3) 5
Explanation — First case is a regular function, the this keyword is bound to different values based on the context in which the function is called. Here obj is calling the function to this will point to current obj.

The second case is an arrow function, it will use the value of this in their lexical scope i.e value of x in surrounding scope. Here surrounding is the global scope or window object and "x" is also present. If "x" is not present then it is undefined.

In the third case, "y" is assigned a value of obj.getX,and "y" is in the global scope or window object. Hence "this" will point to global scope i.e. 5.

Question 49:

```
let a = 10, b = 20;
setTimeout(function () {
```

```
  console.log('Ayush');
  a++;
  b++;
  console.log(a + b);
});
console.log(a + b);
```

Answer — 30 “Ayush” 32.
Explanation —Settimeout pushes the function into BOM stack in event loop or it is executed after everything is executed in main function. So the results are printed after the console.log of main function.

Question 50:

```
function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
  }
    b();
}

var site = 'Wikipedia';
a(); (1)
function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
  }

  b();
}

var site = 'Wikipedia';
new a(); (2)
function a() {
  this.site = 'Ayush';

  function b(){
    console.log(this.site);
  }
  b();
}

let site = 'Wikipedia';
new a(); (3)
```

Answer —
1. 'Ayush'.
2. 'Wikipedia'
3. undefined

Explanation —
1. When a function with normal syntax is executed the value of this of the nearest parent will be used if not set at execution time, so in this case it is window, we are then updating setting the property site in the window object and accessing it inside so it is 'Ayush'.
2. When a function is invoked as a constructor with new keyword then the value of this will be the new object which will be created at execution time so currently, it is { site: 'Ayush' }.
But when a function with normal syntax is executed, the value of this will default to global scope if it is not assigned at the execution time so it is window for b() and var site = 'Wikipedia'; adds the value to the global object, hence when it is accessed inside b() it prints 'Wikipedia'.
3. Variables defined with let are not added to the global scope. Hence, undefined.

https://github.com/lydiahallie/javascript-questions

40. What's the output?

```
[[0, 1], [2, 3]].reduce(
  (acc, cur) => {
    return acc.concat(cur);
  },
  [1, 2],
);
```

A: [0, 1, 2, 3, 1, 2]
B: [6, 1, 2]
C: [1, 2, 0, 1, 2, 3]
D: [1, 2, 6]

[1, 2] is our initial value. This is the value we start with, and the value of the very first acc. During the first round, acc is [1,2], and cur is [0, 1]. We concatenate them, which results in [1, 2, 0, 1].

Then, [1, 2, 0, 1] is acc and [2, 3] is cur. We concatenate them, and get [1, 2, 0, 1, 2, 3]

44. What's the output?

```
function* generator(i) {
  yield i;
  yield i * 2;
}

const gen = generator(10);

console.log(gen.next().value);
console.log(gen.next().value);
```

A: [0, 10], [10, 20]
B: 20, 20
C: 10, 20
D: 0, 10 and 10, 20

Answer: C
Regular functions cannot be stopped mid-way after invocation. However, a generator function can be "stopped" midway, and later continue from where it stopped. Every time a generator function encounters a yield keyword, the function yields the value specified after it. Note that the generator function in that case doesn't return the value, it yields the value.

First, we initialize the generator function with i equal to 10. We invoke the generator function using the next() method. The first time we invoke the generator function, i is equal to 10. It encounters the first yield keyword: it yields the value of i. The generator is now "paused", and 10 gets logged.

Then, we invoke the function again with the next() method. It starts to continue where it stopped previously, still with i equal to 10. Now, it encounters the next yield keyword, and yields i * 2. i is equal to 10, so it returns 10 * 2, which is 20. This results in 10, 20.

45. What does this return?

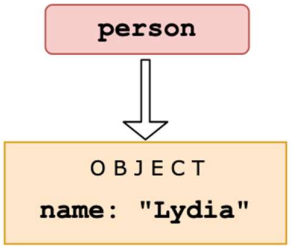
```
const firstPromise = new Promise((res, rej) => {
  setTimeout(res, 500, 'one');
});

const secondPromise = new Promise((res, rej) => {
  setTimeout(res, 100, 'two');
});

Promise.race([firstPromise, secondPromise]).then(res => console.log(res));
```

- A: "one"
- B: "two"
- C: "two" "one"
- D: "one" "two"

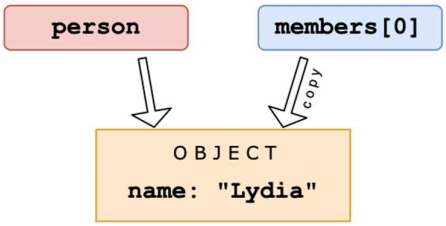
Answer: B
When we pass multiple promises to the Promise.race method, it resolves/rejects the *first* promise that resolves/rejects. To the setTimeout method, we pass a timer: 500ms for the first promise (firstPromise), and 100ms for the second promise (secondPromise). This means that the secondPromise resolves first with the value of 'two'. res now holds the value of 'two', which gets logged.

46. What's the output?
```
let person = { name: 'Lydia' };
const members = [person];
person = null;

console.log(members);
```
- A: null
- B: [null]
- C: [{}]
- D: [{ name: "Lydia" }]

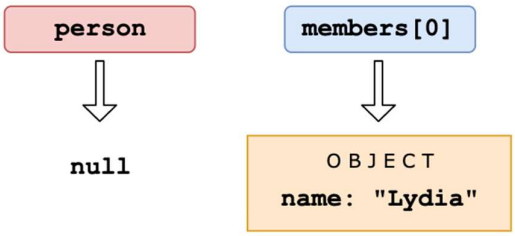
Answer
Answer: D
First, we declare a variable person with the value of an object that has a name property.



Then, we declare a variable called members. We set the first element of that array equal to the value of the person variable. Objects interact by *reference* when setting them equal to each other. When you assign a reference from one variable to another, you make a *copy* of that reference. (note that they don't have the *same* reference!)



Then, we set the variable person equal to null.



We are only modifying the value of the person variable, and not the first element in the array, since that element has a different (copied) reference to the object. The first element in members still holds its reference to the original object. When we log the members array, the first element still holds the value of the object, which gets logged.
47. What's the output?
```
const person = {
  name: 'Lydia',
  age: 21,
};

for (const item in person) {
  console.log(item);
}
```
A: { name: "Lydia" }, { age: 21 }
B: "name", "age"
C: "Lydia", 21
D: ["name", "Lydia"], ["age", 21]
Answer: B
With a for-in loop, we can iterate through object keys, in this case name and age. Under the hood, object keys are strings (if they're not a Symbol). On every loop, we set the value of item equal to the current key it's iterating over. First, item is equal to name, and gets logged. Then, item is equal to age, which gets logged.

49. What's the value of num?
```
const num = parseInt('7*6', 10);
```
A: 42

B: "42"
C: 7
D: NaN
Answer: C
Only the first numbers in the string is returned. Based on the *radix* (the second argument in order to specify what type of number we want to parse it to: base 10, hexadecimal, octal, binary, etc.), the parseInt checks whether the characters in the string are valid. Once it encounters a character that isn't a valid number in the radix, it stops parsing and ignores the following characters.
* is not a valid number. It only parses "7" into the decimal 7. num now holds the value of 7.
50. What's the output?
```
[1, 2, 3].map(num => {
  if (typeof num === 'number') return;
  return num * 2;
});
```
A: []
B: [null, null, null]
C: [undefined, undefined, undefined]
D: [3 x empty]
Answer: C
When mapping over the array, the value of num is equal to the element it’s currently looping over. In this case, the elements are numbers, so the condition of the if statement typeof num === "number" returns true. The map function creates a new array and inserts the values returned from the function.
However, we don’t return a value. When we don’t return a value from the function, the function returns undefined. For every element in the array, the function block gets called, so for each element we return undefined.
51. What's the output?
```
function getInfo(member, year) {
  member.name = 'Lydia';
  year = '1998';
}

const person = { name: 'Sarah' };
const birthYear = '1997';

getInfo(person, birthYear);

console.log(person, birthYear);
```
A: { name: "Lydia" }, "1997"
B: { name: "Sarah" }, "1998"
C: { name: "Lydia" }, "1998"
D: { name: "Sarah" }, "1997"
52. What's the output?
```
function greeting() {
  throw 'Hello world!';
}

function sayHi() {
  try {
    const data = greeting();
    console.log('It worked!', data);
  } catch (e) {
    console.log('Oh no an error:', e);
  }
}

sayHi();
```
A: It worked! Hello world!
B: Oh no an error: undefined
C: SyntaxError: can only throw Error objects
D: Oh no an error: Hello world!
53. What's the output?
```
function Car() {
  this.make = 'Lamborghini';
  return { make: 'Maserati' };
}

const myCar = new Car();
console.log(myCar.make);
```
A: "Lamborghini"
B: "Maserati"
C: ReferenceError
D: TypeError
Answer: B

When you return a property, the value of the property is equal to the *returned* value, not the value set in the constructor function. We return the string "Maserati", so myCar.make is equal to "Maserati".

54. What's the output?

```
(() => {
  let x = (y = 10);
})();

console.log(typeof x);
console.log(typeof y);
```

A: "undefined", "number"
B: "number", "number"
C: "object", "number"
D: "number", "undefined"

Answer: A

let x = (y = 10); is actually shorthand for:

```
y = 10;
let x = y;
```

When we set y equal to 10, we actually add a property y to the global object (window in browser, global in Node). In a browser, window.y is now equal to 10.

Then, we declare a variable x with the value of y, which is 10. Variables declared with the let keyword are *block scoped*, they are only defined within the block they're declared in; the immediately invoked function expression (IIFE) in this case. When we use the typeof operator, the operand x is not defined: we are trying to access x outside of the block it's declared in. This means that x is not defined. Values who haven't been assigned a value or declared are of type "undefined". console.log(typeof x) returns "undefined".

However, we created a global variable y when setting y equal to 10. This value is accessible anywhere in our code. y is defined, and holds a value of type "number". console.log(typeof y) returns "number"

55. What's the output?

```
class Dog {
  constructor(name) {
    this.name = name;
  }
}

Dog.prototype.bark = function() {
  console.log(`Woof I am ${this.name}`);
};

const pet = new Dog('Mara');

pet.bark();

delete Dog.prototype.bark;

pet.bark();
```

A: "Woof I am Mara", TypeError
B: "Woof I am Mara", "Woof I am Mara"
C: "Woof I am Mara", undefined
D: TypeError, TypeError

Answer: A

We can delete properties from objects using the delete keyword, also on the prototype. By deleting a property on the prototype, it is not available anymore in the prototype chain. In this case, the bark function is not available anymore on the prototype after delete Dog.prototype.bark, yet we still try to access it.

When we try to invoke something that is not a function, a TypeError is thrown. In this case TypeError: pet.bark is not a function, since pet.bark is undefined.

56. What's the output?

```
const set = new Set([1, 1, 2, 3, 4]);

console.log(set);
```

A: [1, 1, 2, 3, 4]
B: [1, 2, 3, 4]
C: {1, 1, 2, 3, 4}
D: {1, 2, 3, 4}

57. What's the output?

```
// counter.js
let counter = 10;
export default counter;
// index.js
import myCounter from './counter';

myCounter += 1;

console.log(myCounter);
```

A: 10
B: 11
C: Error
D: NaN
Answer: C
An imported module is *read-only*: you cannot modify the imported module. Only the module that exports them can change its value.
When we try to increment the value of myCounter, it throws an error: myCounter is read-only and cannot be modified.
58. What's the output?
```
const name = 'Lydia';
age = 21;

console.log(delete name);
console.log(delete age);
```
A: false, true
B: "Lydia", 21
C: true, true
D: undefined, undefined
Answer

---

59. What's the output?
```
const numbers = [1, 2, 3, 4, 5];
const [y] = numbers;

console.log(y);
```
A: [[1, 2, 3, 4, 5]]
B: [1, 2, 3, 4, 5]
C: 1
D: [1]
Answer

---

60. What's the output?
```
const user = { name: 'Lydia', age: 21 };
const admin = { admin: true, ...user };

console.log(admin);
```
A: { admin: true, user: { name: "Lydia", age: 21 } }
B: { admin: true, name: "Lydia", age: 21 }
C: { admin: true, user: ["Lydia", 21] }
D: { admin: true }
61. What's the output?
```
const person = { name: 'Lydia' };

Object.defineProperty(person, 'age', { value: 21 });

console.log(person);
console.log(Object.keys(person));
```
A: { name: "Lydia", age: 21 }, ["name", "age"]
B: { name: "Lydia", age: 21 }, ["name"]
C: { name: "Lydia"}, ["name", "age"]
D: { name: "Lydia"}, ["age"]
Answer: B
With the defineProperty method, we can add new properties to an object, or modify existing ones. When we add a property to an object using the defineProperty method, they are by default *not enumerable*. The Object.keys method returns all *enumerable* property names from an object, in this case only "name".
Properties added using the defineProperty method are immutable by default. You can override this behavior using the writable, configurable and enumerable properties. This way, the defineProperty method gives you a lot more control over the properties you're adding to an object.
62. What's the output?
```
const settings = {
  username: 'lydiahallie',
  level: 19,
  health: 90,
};

const data = JSON.stringify(settings, ['level', 'health']);
console.log(data);
```
A: "{"level":19, "health":90}"
B: "{"username": "lydiahallie"}"
C: "["level", "health"]"
D: "{"username": "lydiahallie", "level":19, "health":90}"
Answer: A

The second argument of JSON.stringify is the *replacer*. The replacer can either be a function or an array, and lets you control what and how the values should be stringified.
If the replacer is an *array*, only the property names included in the array will be added to the JSON string. In this case, only the properties with the names "level" and "health" are included, "username" is excluded. data is now equal to "{"level":19, "health":90}".
If the replacer is a *function*, this function gets called on every property in the object you're stringifying. The value returned from this function will be the value of the property when it's added to the JSON string. If the value is undefined, this property is excluded from the JSON string.
63. What's the output?

```
let num = 10;

const increaseNumber = () => num++;
const increasePassedNumber = number => number++;

const num1 = increaseNumber();
const num2 = increasePassedNumber(num1);

console.log(num1);
console.log(num2);
```

A: 10, 10
B: 10, 11
C: 11, 11
D: 11, 12
64. What's the output?

```
const value = { number: 10 };

const multiply = (x = { ...value }) => {
  console.log((x.number *= 2));
};

multiply();
multiply();
multiply(value);
multiply(value);
```

A: 20, 40, 80, 160
B: 20, 40, 20, 40
C: 20, 20, 20, 40
D: NaN, NaN, 20, 40

Answer: C
In ES6, we can initialize parameters with a default value. The value of the parameter will be the default value, if no other value has been passed to the function, or if the value of the parameter is "undefined". In this case, we spread the properties of the value object into a new object, so x has the default value of { number: 10 }.
The default argument is evaluated at *call time*! Every time we call the function, a *new* object is created. We invoke the multiply function the first two times without passing a value: x has the default value of { number: 10 }. We then log the multiplied value of that number, which is 20.
The third time we invoke multiply, we do pass an argument: the object called value. The *= operator is actually shorthand for x.number = x.number * 2: we modify the value of x.number, and log the multiplied value 20.
The fourth time, we pass the value object again. x.number was previously modified to 20, so x.number *= 2 logs 40.

65. What's the output?

```
[1, 2, 3, 4].reduce((x, y) => console.log(x, y));
```

A: 1 2 and 3 3 and 6 4
B: 1 2 and 2 3 and 3 4
C: 1 undefined and 2 undefined and 3 undefined and 4 undefined
D: 1 2 and undefined 3 and undefined 4
Answer: D
The first argument that the reduce method receives is the *accumulator*, x in this case. The second argument is the *current value*, y. With the reduce method, we execute a callback function on every element in the array, which could ultimately result in one single value.
In this example, we are not returning any values, we are simply logging the values of the accumulator and the current value.
The value of the accumulator is equal to the previously returned value of the callback function. If you don't pass the optional initialValue argument to the reduce method, the accumulator is equal to the first element on the first call.
On the first call, the accumulator (x) is 1, and the current value (y) is 2. We don't return from the callback function, we log the accumulator and current value: 1 and 2 get logged.
If you don't return a value from a function, it returns undefined. On the next call, the accumulator is undefined, and the current value is 3. undefined and 3 get logged.
On the fourth call, we again don't return from the callback function. The accumulator is again undefined, and the current value is 4. undefined and 4 get logged.
66. With which constructor can we successfully extend the Dog class?

```
class Dog {
```

```
  constructor(name) {
    this.name = name;
  }
};

class Labrador extends Dog {
  // 1
  constructor(name, size) {
    this.size = size;
  }
  // 2
  constructor(name, size) {
    super(name);
    this.size = size;
  }
  // 3
  constructor(size) {
    super(name);
    this.size = size;
  }
  // 4
  constructor(name, size) {
    this.name = name;
    this.size = size;
  }

};
```

A: 1
B: 2
C: 3
D: 4
Answer

---

67. What's the output?

```
// index.js
console.log('running index.js');
import { sum } from './sum.js';
console.log(sum(1, 2));

// sum.js
console.log('running sum.js');
export const sum = (a, b) => a + b;
```

A: running index.js, running sum.js, 3
B: running sum.js, running index.js, 3
C: running sum.js, 3, running index.js
D: running index.js, undefined, running sum.js
Answer: B
With the import keyword, all imported modules are *pre-parsed*. This means that the imported modules get run *first*, the code in the file which imports the module gets executed *after*.
This is a difference between require() in CommonJS and import! With require(), you can load dependencies on demand while the code is being run. If we would have used require instead of import, running index.js, running sum.js, 3 would have been logged to the console.
68. What's the output?

```
console.log(Number(2) === Number(2));
console.log(Boolean(false) === Boolean(false));
console.log(Symbol('foo') === Symbol('foo'));
```

A: true, true, false
B: false, true, false
C: true, false, true
D: true, true, true
Answer: A
Every Symbol is entirely unique. The purpose of the argument passed to the Symbol is to give the Symbol a description. The value of the Symbol is not dependent on the passed argument. As we test equality, we are creating two entirely new symbols: the first Symbol('foo'), and the second Symbol('foo'). These two values are unique and not equal to each other, Symbol('foo') === Symbol('foo') returns false.
69. What's the output?

```
const name = 'Lydia Hallie';
console.log(name.padStart(13));
console.log(name.padStart(2));
```

A: "Lydia Hallie", "Lydia Hallie"
B: " Lydia Hallie", " Lydia Hallie" ("[13x whitespace]Lydia Hallie", "[2x whitespace]Lydia Hallie")
C: " Lydia Hallie", "Lydia Hallie" ("[1x whitespace]Lydia Hallie", "Lydia Hallie")

D: "Lydia Hallie", "Lyd",
Answer: C
With the padStart method, we can add padding to the beginning of a string. The value passed to this method is the *total* length of the string together with the padding. The string "Lydia Hallie" has a length of 12. name.padStart(13) inserts 1 space at the start of the string, because 12 + 1 is 13.
If the argument passed to the padStart method is smaller than the length of the array, no padding will be added.
70. What's the output?
console.log('🥑' + '💻');
A: "🥑💻"
B: 257548
C: A string containing their code points
D: Error
Answer: A
With the + operator, you can concatenate strings. In this case, we are concatenating the string "🥑" with the string "💻", resulting in "🥑💻"
71. How can we log the values that are commented out after the console.log statement?

```
function* startGame() {
  const answer = yield 'Do you love JavaScript?';
  if (answer !== 'Yes') {
    return "Oh wow... Guess we're gone here";
  }
  return 'JavaScript loves you back ❤️';
}

const game = startGame();
console.log(/* 1 */); // Do you love JavaScript?
console.log(/* 2 */); // JavaScript loves you back ❤️
```

A: game.next("Yes").value and game.next().value
B: game.next.value("Yes") and game.next.value()
C: game.next().value and game.next("Yes").value
D: game.next.value() and game.next.value("Yes")
Answer: C
A generator function "pauses" its execution when it sees the yield keyword. First, we have to let the function yield the string "Do you love JavaScript?", which can be done by calling game.next().value.
Every line is executed, until it finds the first yield keyword. There is a yield keyword on the first line within the function: the execution stops with the first yield! *This means that the variable answer is not defined yet!*
When we call game.next("Yes").value, the previous yield is replaced with the value of the parameters passed to the next() function, "Yes" in this case. The value of the variable answer is now equal to "Yes". The condition of the if-statement returns false, and JavaScript loves you back ❤️ gets logged.
72. What's the output?
console.log(String.raw`Hello\nworld`);
A: Hello world!
B: Hello
world
C: Hello\nworld
D: Hello\n
world
Answer: C
String.raw returns a string where the escapes (\n, \v, \t etc.) are ignored! Backslashes can be an issue since you could end up with something like:
const path = `C:\Documents\Projects\table.html`
Which would result in:
"C:DocumentsProjects able.html"
With String.raw, it would simply ignore the escape and print:
C:\Documents\Projects\table.html
In this case, the string is Hello\nworld, which gets logged.
73. What's the output?

```
async function getData() {
  return await Promise.resolve('I made it!');
}

const data = getData();
console.log(data);
```

A: "I made it!"
B: Promise {<resolved>: "I made it!"}
C: Promise {<pending>}
D: undefined
Answer: C
An async function always returns a promise. The await still has to wait for the promise to resolve: a pending promise gets returned when we call getData() in order to set data equal to it.
If we wanted to get access to the resolved value "I made it", we could have used the .then() method on data:

```
data.then(res => console.log(res))
```
This would've logged "I made it!"
74. What's the output?
```
function addToList(item, list) {
  return list.push(item);
}

const result = addToList('apple', ['banana']);
console.log(result);
```
A: ['apple', 'banana']
B: 2
C: true
D: undefined
Answer: B
The .push() method returns the *length* of the new array! Previously, the array contained one element (the string "banana") and had a length of 1. After adding the string "apple" to the array, the array contains two elements, and has a length of 2. This gets returned from the addToList function.
The push method modifies the original array. If you wanted to return the *array* from the function rather than the *length of the array*, you should have returned list after pushing item to it.
75. What's the output?
```
const box = { x: 10, y: 20 };

Object.freeze(box);

const shape = box;
shape.x = 100;

console.log(shape);
```
A: { x: 100, y: 20 }
B: { x: 10, y: 20 }
C: { x: 100 }
D: ReferenceError
Answer

---

76. What's the output?
```
const { name: myName } = { name: 'Lydia' };

console.log(name);
```
A: "Lydia"
B: "myName"
C: undefined
D: ReferenceError
Answer

---

77. Is this a pure function?
```
function sum(a, b) {
  return a + b;
}
```
A: Yes
B: No
Answer

---

78. What is the output?
```
const add = () => {
  const cache = {};
  return num => {
    if (num in cache) {
      return `From cache! ${cache[num]}`;
    } else {
      const result = num + 10;
      cache[num] = result;
      return `Calculated! ${result}`;
    }
  };
};

const addFunction = add();
console.log(addFunction(10));
console.log(addFunction(10));
console.log(addFunction(5 * 2));
```
A: Calculated! 20 Calculated! 20 Calculated! 20
B: Calculated! 20 From cache! 20 Calculated! 20

C: Calculated! 20 From cache! 20 From cache! 20
D: Calculated! 20 From cache! 20 Error
Answer

---

79. What is the output?
const myLifeSummedUp = ['🍷', '💻', '🍷', '🍫'];

```
for (let item in myLifeSummedUp) {
  console.log(item);
}

for (let item of myLifeSummedUp) {
  console.log(item);
}
```

A: 0 1 2 3 and "🍷" "💻" "🍷" "🍫"
B: "🍷" "💻" "🍷" "🍫" and "🍷" "💻" "🍷" "🍫"
C: "🍷" "💻" "🍷" "🍫" and 0 1 2 3
D: 0 1 2 3 and {0: "🍷", 1: "💻", 2: "🍷", 3: "🍫"}
Answer

---

80. What is the output?
const list = [1 + 2, 1 * 2, 1 / 2];
console.log(list);
A: ["1 + 2", "1 * 2", "1 / 2"]
B: ["12", 2, 0.5]
C: [3, 2, 0.5]
D: [1, 1, 1]
Answer

---

81. What is the output?

```
function sayHi(name) {
  return `Hi there, ${name}`;
}

console.log(sayHi());
```

A: Hi there,
B: Hi there, undefined
C: Hi there, null
D: ReferenceError
Answer

---

82. What is the output?
var status = '😎';

```
setTimeout(() => {
  const status = '😍';

  const data = {
    status: '🥑',
    getStatus() {
      return this.status;
    },
  };

  console.log(data.getStatus());
  console.log(data.getStatus.call(this));
}, 0);
```

A: "🥑" and "😍"
B: "🥑" and "😎"
C: "😍" and "😎"
D: "😎" and "😎"
Answer

---

83. What is the output?

```
const person = {
  name: 'Lydia',
  age: 21,
};

let city = person.city;
city = 'Amsterdam';
```

```
console.log(person);
```
A: { name: "Lydia", age: 21 }
B: { name: "Lydia", age: 21, city: "Amsterdam" }
C: { name: "Lydia", age: 21, city: undefined }
D: "Amsterdam"
Answer

---

84. What is the output?
```
function checkAge(age) {
  if (age < 18) {
    const message = "Sorry, you're too young.";
  } else {
    const message = "Yay! You're old enough!";
  }

  return message;
}

console.log(checkAge(21));
```
A: "Sorry, you're too young."
B: "Yay! You're old enough!"
C: ReferenceError
D: undefined
Answer

---

85. What kind of information would get logged?
```
fetch('https://www.website.com/api/user/1')
  .then(res => res.json())
  .then(res => console.log(res));
```
A: The result of the fetch method.
B: The result of the second invocation of the fetch method.
C: The result of the callback in the previous .then().
D: It would always be undefined.
Answer

---

86. Which option is a way to set hasName equal to true, provided you cannot pass true as an argument?
```
function getName(name) {
  const hasName = //
}
```
A: !!name
B: name
C: new Boolean(name)
D: name.length
Answer

---

87. What's the output?
```
console.log('I want pizza'[0]);
```
A: """
B: "I"
C: SyntaxError
D: undefined
Answer

---

88. What's the output?
```
function sum(num1, num2 = num1) {
  console.log(num1 + num2);
}

sum(10);
```
A: NaN
B: 20
C: ReferenceError
D: undefined
Answer

---

89. What's the output?
```
// module.js
export default () => 'Hello world';
export const name = 'Lydia';

// index.js
```

```
import * as data from './module';

console.log(data);
```
A: { default: function default(), name: "Lydia" }
B: { default: function default() }
C: { default: "Hello world", name: "Lydia" }
D: Global object of module.js
Answer

---

90. What's the output?
```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const member = new Person('John');
console.log(typeof member);
```
A: "class"
B: "function"
C: "object"
D: "string"
Answer

---

91. What's the output?
```
let newList = [1, 2, 3].push(4);

console.log(newList.push(5));
```
A: [1, 2, 3, 4, 5]
B: [1, 2, 3, 5]
C: [1, 2, 3, 4]
D: Error
Answer

---

92. What's the output?
```
function giveLydiaPizza() {
  return 'Here is pizza!';
}

const giveLydiaChocolate = () =>
  "Here's chocolate... now go hit the gym already.";

console.log(giveLydiaPizza.prototype);
console.log(giveLydiaChocolate.prototype);
```
A: { constructor: ...} { constructor: ...}
B: {} { constructor: ...}
C: { constructor: ...} {}
D: { constructor: ...} undefined
Answer

---

93. What's the output?
```
const person = {
  name: 'Lydia',
  age: 21,
};

for (const [x, y] of Object.entries(person)) {
  console.log(x, y);
}
```
A: name Lydia and age 21
B: ["name", "Lydia"] and ["age", 21]
C: ["name", "age"] and undefined
D: Error
Answer

---

94. What's the output?
```
function getItems(fruitList, ...args, favoriteFruit) {
  return [...fruitList, ...args, favoriteFruit]
}

getItems(["banana", "apple"], "pear", "orange")
```
A: ["banana", "apple", "pear", "orange"]

B: [["banana", "apple"], "pear", "orange"]
C: ["banana", "apple", ["pear"], "orange"]
D: SyntaxError
Answer

---

95. What's the output?
```
function nums(a, b) {
  if (a > b) console.log('a is bigger');
  else console.log('b is bigger');
  return
  a + b;
}

console.log(nums(4, 2));
console.log(nums(1, 2));
```
A: a is bigger, 6 and b is bigger, 3
B: a is bigger, undefined and b is bigger, undefined
C: undefined and undefined
D: SyntaxError
Answer

---

96. What's the output?
```
class Person {
  constructor() {
    this.name = 'Lydia';
  }
}

Person = class AnotherPerson {
  constructor() {
    this.name = 'Sarah';
  }
};

const member = new Person();
console.log(member.name);
```
A: "Lydia"
B: "Sarah"
C: Error: cannot redeclare Person
D: SyntaxError
Answer

---

97. What's the output?
```
const info = {
  [Symbol('a')]: 'b',
};

console.log(info);
console.log(Object.keys(info));
```
A: {Symbol('a'): 'b'} and ["{Symbol('a')"]
B: {} and []
C: { a: "b" } and ["a"]
D: {Symbol('a'): 'b'} and []
Answer

---

98. What's the output?
```
const getList = ([x, ...y]) => [x, y]
const getUser = user => { name: user.name, age: user.age }

const list = [1, 2, 3, 4]
const user = { name: "Lydia", age: 21 }

console.log(getList(list))
console.log(getUser(user))
```
A: [1, [2, 3, 4]] and SyntaxError
B: [1, [2, 3, 4]] and { name: "Lydia", age: 21 }
C: [1, 2, 3, 4] and { name: "Lydia", age: 21 }
D: Error and { name: "Lydia", age: 21 }
Answer

---

99. What's the output?
```
const name = 'Lydia';
```

```
console.log(name());
```
A: SyntaxError
B: ReferenceError
C: TypeError
D: undefined
Answer

---

100. What's the value of output?
```
// 🎉✨ This is my 100th question! ✨🎉

const output = `${[] && 'Im'}possible!
You should${'' && `n't`} see a therapist after so much JavaScript lol`;
```
A: possible! You should see a therapist after so much JavaScript lol
B: Impossible! You should see a therapist after so much JavaScript lol
C: possible! You shouldn't see a therapist after so much JavaScript lol
D: Impossible! You shouldn't see a therapist after so much JavaScript lol
Answer

---

101. What's the value of output?
```
const one = false || {} || null;
const two = null || false || '';
const three = [] || 0 || true;

console.log(one, two, three);
```
A: false null []
B: null "" true
C: {} "" []
D: null null true
Answer

---

102. What's the value of output?
```
const myPromise = () => Promise.resolve('I have resolved!');

function firstFunction() {
  myPromise().then(res => console.log(res));
  console.log('second');
}

async function secondFunction() {
  console.log(await myPromise());
  console.log('second');
}

firstFunction();
secondFunction();
```
A: I have resolved!, second and I have resolved!, second
B: second, I have resolved! and second, I have resolved!
C: I have resolved!, second and second, I have resolved!
D: second, I have resolved! and I have resolved!, second
Answer

---

103. What's the value of output?
```
const set = new Set();

set.add(1);
set.add('Lydia');
set.add({ name: 'Lydia' });

for (let item of set) {
  console.log(item + 2);
}
```
A: 3, NaN, NaN
B: 3, 7, NaN
C: 3, Lydia2, [object Object]2
D: "12", Lydia2, [object Object]2
Answer

---

104. What's its value?
```
Promise.resolve(5);
```
A: 5
B: Promise {<pending>: 5}

C: Promise {<fulfilled>: 5}
D: Error
Answer

---

105. What's its value?
```
function compareMembers(person1, person2 = person) {
  if (person1 !== person2) {
    console.log('Not the same!');
  } else {
    console.log('They are the same!');
  }
}

const person = { name: 'Lydia' };

compareMembers(person);
```
A: Not the same!
B: They are the same!
C: ReferenceError
D: SyntaxError
Answer

---

106. What's its value?
```
const colorConfig = {
  red: true,
  blue: false,
  green: true,
  black: true,
  yellow: false,
};

const colors = ['pink', 'red', 'blue'];

console.log(colorConfig.colors[1]);
```
A: true
B: false
C: undefined
D: TypeError
Answer

---

107. What's its value?
```
console.log('❤️' === '❤️');
```
A: true
B: false
Answer

---

108. Which of these methods modifies the original array?
```
const emojis = ['✨', '🥑', '😍'];

emojis.map(x => x + '✨');
emojis.filter(x => x !== '🥑');
emojis.find(x => x !== '🥑');
emojis.reduce((acc, cur) => acc + '✨');
emojis.slice(1, 2, '✨');
emojis.splice(1, 2, '✨');
```
A: All of them
B: map reduce slice splice
C: map slice splice
D: splice
Answer

---

109. What's the output?
```
const food = ['🍕', '🍫', '🥑', '🍔'];
const info = { favoriteFood: food[0] };

info.favoriteFood = '🍝';

console.log(food);
```
A: ['🍕', '🍫', '🥑', '🍔']
B: ['🍝', '🍫', '🥑', '🍔']
C: ['🍝', '🍕', '🍫', '🥑', '🍔']

D: ReferenceError
Answer

---

110. What does this method do?
JSON.parse();
A: Parses JSON to a JavaScript value
B: Parses a JavaScript object to JSON
C: Parses any JavaScript value to JSON
D: Parses JSON to a JavaScript object only
Answer

---

111. What's the output?
```
let name = 'Lydia';

function getName() {
  console.log(name);
  let name = 'Sarah';
}

getName();
```
A: Lydia
B: Sarah
C: undefined
D: ReferenceError
Answer

---

112. What's the output?
```
function* generatorOne() {
  yield ['a', 'b', 'c'];
}

function* generatorTwo() {
  yield* ['a', 'b', 'c'];
}

const one = generatorOne();
const two = generatorTwo();

console.log(one.next().value);
console.log(two.next().value);
```
A: a and a
B: a and undefined
C: ['a', 'b', 'c'] and a
D: a and ['a', 'b', 'c']
Answer

---

113. What's the output?
```
console.log(`${(x => x)('I love')} to program`);
```
A: I love to program
B: undefined to program
C: ${(x => x)('I love') to program
D: TypeError
Answer

---

114. What will happen?
```
let config = {
  alert: setInterval(() => {
    console.log('Alert!');
  }, 1000),
};

config = null;
```
A: The setInterval callback won't be invoked
B: The setInterval callback gets invoked once
C: The setInterval callback will still be called every second
D: We never invoked config.alert(), config is null
Answer

---

115. Which method(s) will return the value 'Hello world!'?
```
const myMap = new Map();
const myFunc = () => 'greeting';
```

```
myMap.set(myFunc, 'Hello world!');

//1
myMap.get('greeting');
//2
myMap.get(myFunc);
//3
myMap.get(() => 'greeting');
```

A: 1
B: 2
C: 2 and 3
D: All of them
Answer

---

116. What's the output?

```
const person = {
  name: 'Lydia',
  age: 21,
};

const changeAge = (x = { ...person }) => (x.age += 1);
const changeAgeAndName = (x = { ...person }) => {
  x.age += 1;
  x.name = 'Sarah';
};

changeAge(person);
changeAgeAndName();

console.log(person);
```

A: {name: "Sarah", age: 22}
B: {name: "Sarah", age: 23}
C: {name: "Lydia", age: 22}
D: {name: "Lydia", age: 23}
Answer

---

117. Which of the following options will return 6?

```
function sumValues(x, y, z) {
  return x + y + z;
}
```

A: sumValues([...1, 2, 3])
B: sumValues([...[1, 2, 3]])
C: sumValues(...[1, 2, 3])
D: sumValues([1, 2, 3])
Answer

---

118. What's the output?

```
let num = 1;
const list = ['🥳', '🤠', '😍', '🤪'];

console.log(list[(num += 1)]);
```

A: 🤠
B: 😍
C: SyntaxError
D: ReferenceError
Answer

---

119. What's the output?

```
const person = {
  firstName: 'Lydia',
  lastName: 'Hallie',
  pet: {
    name: 'Mara',
    breed: 'Dutch Tulip Hound',
  },
  getFullName() {
    return `${this.firstName} ${this.lastName}`;
  },
};

console.log(person.pet?.name);
console.log(person.pet?.family?.name);
```

```
console.log(person.getFullName?.());
console.log(member.getLastName?.());
```
A: undefined undefined undefined undefined
B: Mara undefined Lydia Hallie ReferenceError
C: Mara null Lydia Hallie null
D: null ReferenceError null ReferenceError
Answer

---

120. What's the output?
```
const groceries = ['banana', 'apple', 'peanuts'];

if (groceries.indexOf('banana')) {
  console.log('We have to buy bananas!');
} else {
  console.log(`We don't have to buy bananas!`);
}
```
A: We have to buy bananas!
B: We don't have to buy bananas
C: undefined
D: 1
Answer

---

121. What's the output?
```
const config = {
  languages: [],
  set language(lang) {
    return this.languages.push(lang);
  },
};

console.log(config.language);
```
A: function language(lang) { this.languages.push(lang }
B: 0
C: []
D: undefined
Answer

---

122. What's the output?
```
const name = 'Lydia Hallie';

console.log(!typeof name === 'object');
console.log(!typeof name === 'string');
```
A: false true
B: true false
C: false false
D: true true
Answer

---

123. What's the output?
```
const add = x => y => z => {
  console.log(x, y, z);
  return x + y + z;
};

add(4)(5)(6);
```
A: 4 5 6
B: 6 5 4
C: 4 function function
D: undefined undefined 6
Answer

---

124. What's the output?
```
async function* range(start, end) {
  for (let i = start; i <= end; i++) {
    yield Promise.resolve(i);
  }
}

(async () => {
  const gen = range(1, 3);
  for await (const item of gen) {
    console.log(item);
```

```
  }
})();
```
A: Promise {1} Promise {2} Promise {3}
B: Promise {<pending>} Promise {<pending>} Promise {<pending>}
C: 1 2 3
D: undefined undefined undefined
Answer

---

125. What's the output?
```
const myFunc = ({ x, y, z }) => {
  console.log(x, y, z);
};

myFunc(1, 2, 3);
```
A: 1 2 3
B: {1: 1} {2: 2} {3: 3}
C: { 1: undefined } undefined undefined
D: undefined undefined undefined
Answer

---

126. What's the output?
```
function getFine(speed, amount) {
  const formattedSpeed = new Intl.NumberFormat('en-US', {
    style: 'unit',
    unit: 'mile-per-hour'
  }).format(speed);

  const formattedAmount = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD'
  }).format(amount);

  return `The driver drove ${formattedSpeed} and has to pay ${formattedAmount}`;
}

console.log(getFine(130, 300))
```
A: The driver drove 130 and has to pay 300
B: The driver drove 130 mph and has to pay $300.00
C: The driver drove undefined and has to pay undefined
D: The driver drove 130.00 and has to pay 300.00
Answer

---

127. What's the output?
```
const spookyItems = ['👻', '🎃', '🕸'];
({ item: spookyItems[3] } = { item: '💀' });

console.log(spookyItems);
```
A: ["👻", "🎃", "🕸"]
B: ["👻", "🎃", "🕸", "💀"]
C: ["👻", "🎃", "🕸", { item: "💀" }]
D: ["👻", "🎃", "🕸", "[object Object]"]
Answer

---

128. What's the output?
```
const name = 'Lydia Hallie';
const age = 21;

console.log(Number.isNaN(name));
console.log(Number.isNaN(age));

console.log(isNaN(name));
console.log(isNaN(age));
```
A: true false true false
B: true false false false
C: false false true false
D: false true false true
Answer

---

129. What's the output?
```
const randomValue = 21;
```

```
function getInfo() {
  console.log(typeof randomValue);
  const randomValue = 'Lydia Hallie';
}

getInfo();
```
A: "number"
B: "string"
C: undefined
D: ReferenceError
Answer

---

130. What's the output?
```
const myPromise = Promise.resolve('Woah some cool data');

(async () => {
  try {
    console.log(await myPromise);
  } catch {
    throw new Error(`Oops didn't work`);
  } finally {
    console.log('Oh finally!');
  }
})();
```
A: Woah some cool data
B: Oh finally!
C: Woah some cool data Oh finally!
D: Oops didn't work Oh finally!
Answer

---

131. What's the output?
```
const emojis = ['🥑', ['✨', '✨', ['🍕', '🍕']]];

console.log(emojis.flat(1));
```
A: ['🥑', ['✨', '✨', ['🍕', '🍕']]]
B: ['🥑', '✨', '✨', ['🍕', '🍕']]
C: ['🥑', ['✨', '✨', '🍕', '🍕']]
D: ['🥑', '✨', '✨', '🍕', '🍕']
Answer

---

132. What's the output?
```
class Counter {
  constructor() {
    this.count = 0;
  }

  increment() {
    this.count++;
  }
}

const counterOne = new Counter();
counterOne.increment();
counterOne.increment();

const counterTwo = counterOne;
counterTwo.increment();

console.log(counterOne.count);
```
A: 0
B: 1
C: 2
D: 3
Answer

---

133. What's the output?
```
const myPromise = Promise.resolve(Promise.resolve('Promise'));

function funcOne() {
  setTimeout(() => console.log('Timeout 1!'), 0);
  myPromise.then(res => res).then(res => console.log(`${res} 1!`));
  console.log('Last line 1!');
```

```
}

async function funcTwo() {
  const res = await myPromise;
  console.log(`${res} 2!`)
  setTimeout(() => console.log('Timeout 2!'), 0);
  console.log('Last line 2!');
}

funcOne();
funcTwo();
```

A: Promise 1! Last line 1! Promise 2! Last line 2! Timeout 1! Timeout 2!
B: Last line 1! Timeout 1! Promise 1! Last line 2! Promise2! Timeout 2!
C: Last line 1! Promise 2! Last line 2! Promise 1! Timeout 1! Timeout 2!
D: Timeout 1! Promise 1! Last line 1! Promise 2! Timeout 2! Last line 2!
Answer

---

134. How can we invoke sum in sum.js from index.js?

```
// sum.js
export default function sum(x) {
  return x + x;
}

// index.js
import * as sum from './sum';
```

A: sum(4)
B: sum.sum(4)
C: sum.default(4)
D: Default aren't imported with *, only named exports
Answer

---

135. What's the output?

```
const handler = {
  set: () => console.log('Added a new property!'),
  get: () => console.log('Accessed a property!'),
};

const person = new Proxy({}, handler);

person.name = 'Lydia';
person.name;
```

A: Added a new property!
B: Accessed a property!
C: Added a new property! Accessed a property!
D: Nothing gets logged
Answer

---

136. Which of the following will modify the person object?

```
const person = { name: 'Lydia Hallie' };

Object.seal(person);
```

A: person.name = "Evan Bacon"
B: person.age = 21
C: delete person.name
D: Object.assign(person, { age: 21 })
Answer

---

137. Which of the following will modify the person object?

```
const person = {
  name: 'Lydia Hallie',
  address: {
    street: '100 Main St',
  },
};

Object.freeze(person);
```

A: person.name = "Evan Bacon"
B: delete person.address
C: person.address.street = "101 Main St"
D: person.pet = { name: "Mara" }
Answer

---

138. What's the output?
```
const add = x => x + x;

function myFunc(num = 2, value = add(num)) {
  console.log(num, value);
}

myFunc();
myFunc(3);
```
A: 2 4 and 3 6
B: 2 NaN and 3 NaN
C: 2 Error and 3 6
D: 2 4 and 3 Error
Answer

---

139. What's the output?
```
class Counter {
  #number = 10

  increment() {
    this.#number++
  }

  getNum() {
    return this.#number
  }
}

const counter = new Counter()
counter.increment()

console.log(counter.#number)
```
A: 10
B: 11
C: undefined
D: SyntaxError
Answer

---

140. What's missing?
```
const teams = [
  { name: 'Team 1', members: ['Paul', 'Lisa'] },
  { name: 'Team 2', members: ['Laura', 'Tim'] },
];

function* getMembers(members) {
  for (let i = 0; i < members.length; i++) {
    yield members[i];
  }
}

function* getTeams(teams) {
  for (let i = 0; i < teams.length; i++) {
    // ✨ SOMETHING IS MISSING HERE ✨
  }
}

const obj = getTeams(teams);
obj.next(); // { value: "Paul", done: false }
obj.next(); // { value: "Lisa", done: false }
```
A: yield getMembers(teams[i].members)
B: yield* getMembers(teams[i].members)
C: return getMembers(teams[i].members)
D: return yield getMembers(teams[i].members)
Answer

---

141. What's the output?
```
const person = {
  name: 'Lydia Hallie',
  hobbies: ['coding'],
};

function addHobby(hobby, hobbies = person.hobbies) {
```

```
  hobbies.push(hobby);
  return hobbies;
}

addHobby('running', []);
addHobby('dancing');
addHobby('baking', person.hobbies);

console.log(person.hobbies);
```

A: ["coding"]
B: ["coding", "dancing"]
C: ["coding", "dancing", "baking"]
D: ["coding", "running", "dancing", "baking"]
Answer

---

142. What's the output?

```
class Bird {
  constructor() {
    console.log("I'm a bird. 🦢");
  }
}

class Flamingo extends Bird {
  constructor() {
    console.log("I'm pink. 🌸");
    super();
  }
}

const pet = new Flamingo();
```

A: I'm pink. 🌸
B: I'm pink. 🌸 I'm a bird. 🦢
C: I'm a bird. 🦢 I'm pink. 🌸
D: Nothing, we didn't call any method
Answer

---

143. Which of the options result(s) in an error?

```
const emojis = ['🎄', '🎅', '🎁', '⭐'];

/* 1 */ emojis.push('🦌');
/* 2 */ emojis.splice(0, 2);
/* 3 */ emojis = [...emojis, '🥂'];
/* 4 */ emojis.length = 0;
```

A: 1
B: 1 and 2
C: 3 and 4
D: 3
Answer

---

144. What do we need to add to the person object to get ["Lydia Hallie", 21] as the output of [...person]?

```
const person = {
  name: "Lydia Hallie",
  age: 21
}

[...person] // ["Lydia Hallie", 21]
```

A: Nothing, object are iterable by default
B: *[Symbol.iterator]() { for (let x in this) yield* this[x] }
C: *[Symbol.iterator]() { yield* Object.values(this) }
D: *[Symbol.iterator]() { for (let x in this) yield this }
Answer

---

145. What's the output?

```
let count = 0;
const nums = [0, 1, 2, 3];

nums.forEach(num => {
    if (num) count += 1
})

console.log(count)
```

A: 1
B: 2
C: 3
D: 4
Answer

---

146. What's the output?
```
function getFruit(fruits) {
	console.log(fruits?.[1]?.[1])
}

getFruit([['🍊', '🍌'], ['🍍']])
getFruit()
getFruit([['🍍'], ['🍊', '🍌']])
```
A: null, undefined, 🍌
B: [], null, 🍌
C: [], [], 🍌
D: undefined, undefined, 🍌
Answer

---

147. What's the output?
```
class Calc {
	constructor() {
		this.count = 0
	}

	increase() {
		this.count ++
	}
}

const calc = new Calc()
new Calc().increase()

console.log(calc.count)
```
A: 0
B: 1
C: undefined
D: ReferenceError
Answer

---

148. What's the output?
```
const user = {
	email: "e@mail.com",
	password: "12345"
}

const updateUser = ({ email, password }) => {
	if (email) {
		Object.assign(user, { email })
	}

	if (password) {
		user.password = password
	}

	return user
}

const updatedUser = updateUser({ email: "new@email.com" })

console.log(updatedUser === user)
```
A: false
B: true
C: TypeError
D: ReferenceError
Answer

---

149. What's the output?
```
const fruit = ['🍌', '🍊', '🍎']
```

```
fruit.slice(0, 1)
fruit.splice(0, 1)
fruit.unshift('🍇')

console.log(fruit)
```

A: ['🍌', '🍊', '🍎']
B: ['🍊', '🍎']
C: ['🍇', '🍊', '🍎']
D: ['🍇', '🍌', '🍊', '🍎']
Answer

---

150. What's the output?

```
const animals = {};
let dog = { emoji: '🐶' }
let cat = { emoji: '🐈' }

animals[dog] = { ...dog, name: "Mara" }
animals[cat] = { ...cat, name: "Sara" }

console.log(animals[dog])
```

A: { emoji: "🐶", name: "Mara" }
B: { emoji: "🐈", name: "Sara" }
C: undefined
D: ReferenceError
Answer

---

151. What's the output?

```
const user = {
	email: "my@email.com",
	updateEmail: email => {
		this.email = email
	}
}

user.updateEmail("new@email.com")
console.log(user.email)
```

A: my@email.com
B: new@email.com
C: undefined
D: ReferenceError
Answer

---

152. What's the output?

```
const promise1 = Promise.resolve('First')
const promise2 = Promise.resolve('Second')
const promise3 = Promise.reject('Third')
const promise4 = Promise.resolve('Fourth')

const runPromises = async () => {
	const res1 = await Promise.all([promise1, promise2])
	const res2  = await Promise.all([promise3, promise4])
	return [res1, res2]
}

runPromises()
	.then(res => console.log(res))
	.catch(err => console.log(err))
```

A: [['First', 'Second'], ['Fourth']]
B: [['First', 'Second'], ['Third', 'Fourth']]
C: [['First', 'Second']]
D: 'Third'
Answer

---

153. What should the value of method be to log { name: "Lydia", age: 22 }?

```
const keys = ["name", "age"]
const values = ["Lydia", 22]

const method = /* ?? */
Object[method](keys.map((_, i) => {
	return [keys[i], values[i]]
})) // { name: "Lydia", age: 22 }
```

A: entries
B: values
C: fromEntries
D: forEach
Answer

---

154. What's the output?

```
const createMember = ({ email, address = {}}) => {
	const validEmail = /.+\@.+\..+/.test(email)
	if (!validEmail) throw new Error("Valid email pls")

	return {
		email,
		address: address ? address : null
	}
}

const member = createMember({ email: "my@email.com" })
console.log(member)
```

A: { email: "my@email.com", address: null }
B: { email: "my@email.com" }
C: { email: "my@email.com", address: {} }
D: { email: "my@email.com", address: undefined }

155. What's the output?

```
let randomValue = { name: "Lydia" }
randomValue = 23

if (!typeof randomValue === "string") {
	console.log("It's not a string!")
} else {
	console.log("Yay it's a string!")
}
```

A: It's not a string!
B: Yay it's a string!
C: TypeError
D: undefined
Answer: B
The condition within the if statement checks whether the value of !typeof randomValue is equal to "string". The ! operator converts the value to a boolean value. If the value is truthy, the returned value will be false, if the value is falsy, the returned value will be true. In this case, the returned value of typeof randomValue is the truthy value "number", meaning that the value of !typeof randomValue is the boolean value false.
!typeof randomValue === "string" always returns false, since we're actually checking false === "string". Since the condition returned false, the code block of the else statement gets run, and Yay it's a string! gets logged.

MISCELLANEOUS TOPICS

The JavaScript Event Loop: Explained

The event loop is the secret behind **JavaScript's asynchronous programming**. JS executes all operations on a single thread, but using a few smart data structures, gives us the illusion of multi-threading. The asynchronous behavior is not part of the JavaScript language itself, rather it is built on top of the core JavaScript language in the browser (or the programming environment) and accessed through the browser APIs.

Browser JavaScript execution flow, as well as in Node.js, is based on an *event loop*. The *event loop* concept is very simple. There's an endless loop, where the **JavaScript engine waits for tasks, executes them, and then sleeps, waiting for more tasks**. The general algorithm of the engine:

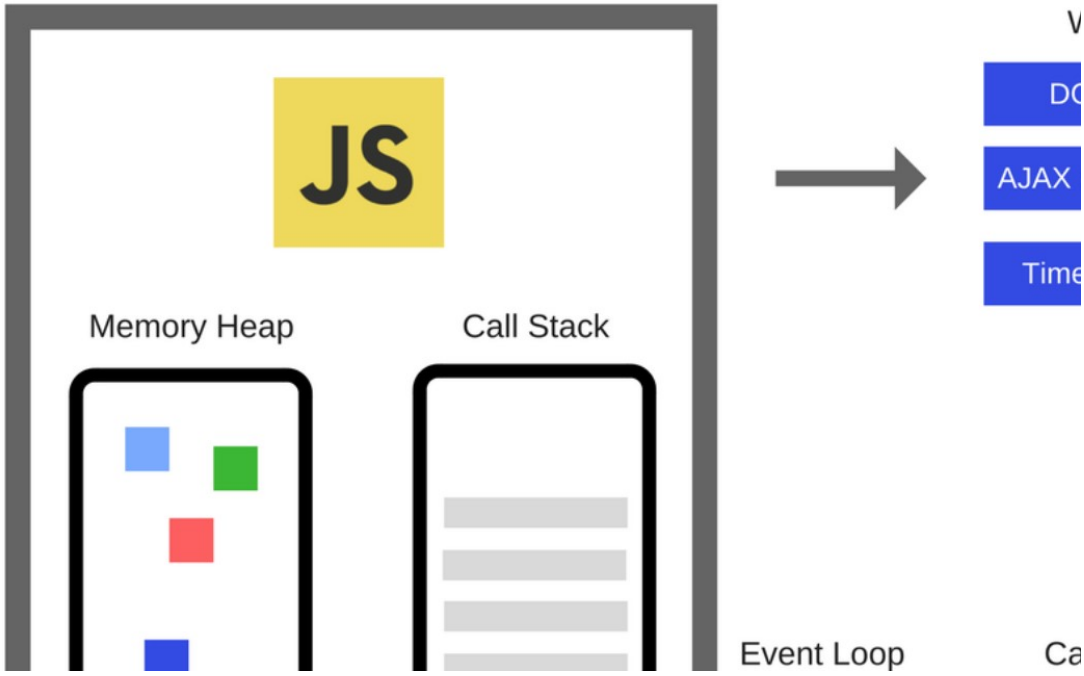
1. While there are tasks - execute them, starting with the oldest task.
2. Sleep until a task appears, then go to 1.

That's a formalization for what we see when browsing a page. The JavaScript engine does nothing most of the time, it only runs if a script/handler/event activates. Examples of tasks:

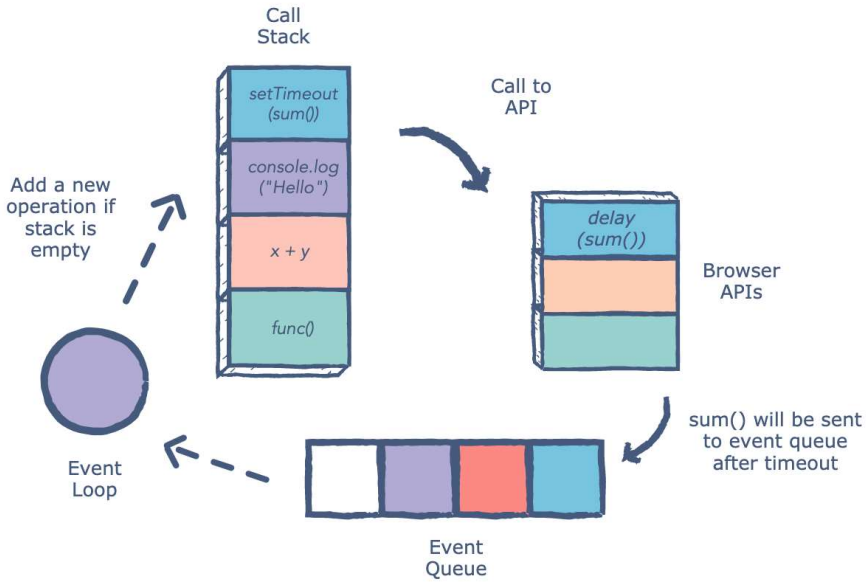
- When an external script `<script src="...">` loads, the task is to execute it.
- When a user moves their mouse, the task is to dispatch `mousemove` event and execute handlers.
- When the time is due for a scheduled `setTimeout`, the task is to run its callback.
- ...and so on.

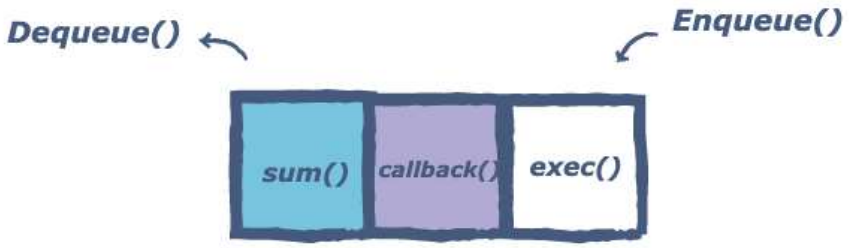
Tasks are set — the engine handles them — then waits for more tasks (while sleeping and consuming close to zero CPU). Let's take a look at what happens on the back-end.

Basic Architecture



    - **Memory Heap** — Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory.
    - **Call Stack** — This represents the single thread provided for JavaScript code execution. Function calls form a stack of frames. It is responsible for keeping track of all the operations in line to be executed. Whenever a function is finished, it is popped from the stack. It is a **LIFO queue** (Last In, First Out).
- **Browser or Web APIs** — They are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. They are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code.
  For example, the Geolocation API provides some simple JavaScript constructs for retrieving location data so you can say, plot your location on a Google Map. In the background, the browser is actually using some complex lower-level code (e.g. C++) to communicate with the device's GPS hardware (or whatever is available to determine position data), retrieve position data, and return it to the browser environment to use in your code. But again, this complexity is abstracted away from you by the API.
- **Event or Callback Queue** — It is responsible for sending new functions to the track for processing. It follows the queue data structure to maintain the correct sequence in which all operations should be sent for execution.



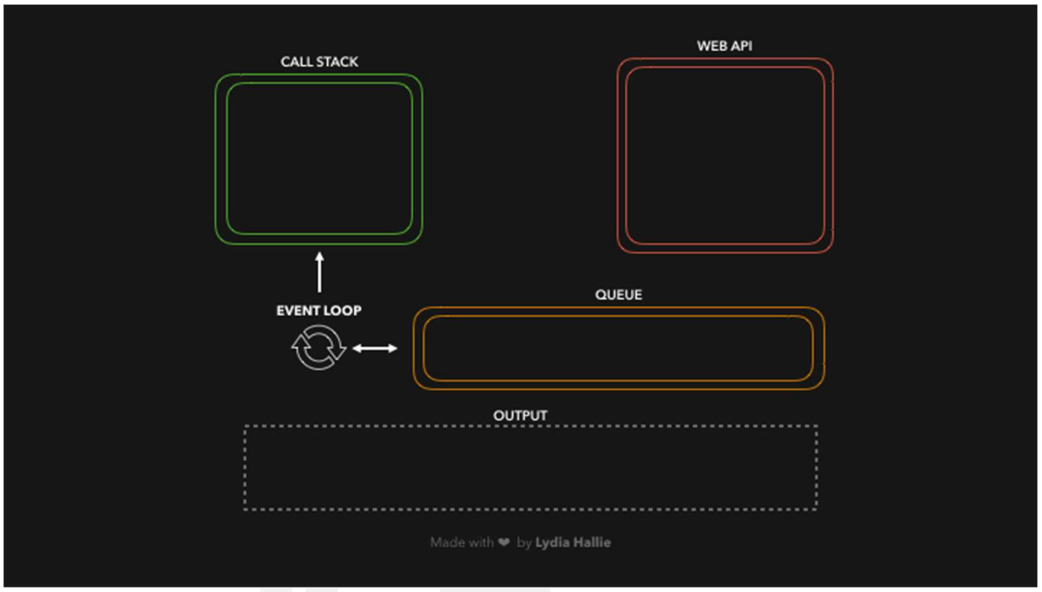


Whenever an async function is called, it is sent to a *browser API*. These are APIs built into the browser. Based on the command received from the call stack, the API starts its own single-threaded operation.
An example of this is the `setTimeout` method. When a `setTimeout` operation is processed in the stack, it is sent to the corresponding API which waits till the specified time to send this operation back in for processing.

Where does it send the operation? The *event queue*. Hence, we have a cyclic system for running async operations in JavaScript. The language itself is single-threaded, but the browser APIs act as separate threads.
The event loop facilitates this process. It has one simple job — **to monitor the call stack and the callback queue**. If the call stack is empty, the event loop will take the first event from the queue and will push it to the call stack, which effectively runs. If it is not, then the current function call is processed.
Let's quickly take a look at an example and see what's happening when we're running the following code in a browser:

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
```

```
baz();Output:First
Third
Second
```



1. We invoke bar. bar returns a setTimeout function.
2. The callback we passed to setTimeout gets added to the Web API, the setTimeout function and bar get popped off the callstack.
3. The timer runs, in the meantime foo gets invoked and logs First. foo returns (undefined),baz gets invoked, and the callback gets added to the queue.
4. baz logs Third. The event loop sees the callstack is empty after baz returned, after which the callback gets added to the call stack.
5. The callback logs Second.

The event loop proceeds to execute all the callbacks waiting in the task queue. Inside the task queue, the tasks are broadly classified into two categories, namely **micro-tasks** and **macro-tasks**.

Micro-tasks within an event loop:

- A micro-task is said to be a function that is executed after the function or program which created it exits and only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the user agent to drive the script's execution environment. A Micro-task is also capable of en-queuing other micro-tasks.
- Micro-tasks are often scheduled for things that are required to be completed immediately after the execution of the current script. **On completion of one macro-task, the event loop moves on to the micro-task queue**. The event loop does not move to the next task outside of the micro-task queue until all the tasks inside the micro-task queue are completed. This implies that the **micro-task queue has a higher priority**.
- Once all the tasks inside the micro-task queue are finished, only then does the event loop shift back to the macro-task queue. The primary reason for prioritizing the micro-task queue is to **improve the user experience**. The micro-task queue is processed after callbacks given that any other JavaScript is not under mid-execution. Micro-tasks include mutation **observer callbacks** as well as **promise callbacks**.
- In such a case wherein new micro-tasks are being added to the queue, these additional micro-tasks are added at the end of the micro-queue and these are also processed. This is because the event loop will keep on calling micro-tasks until there are no more micro-tasks left in the queue, even if new tasks keep getting added. Another important reason for using micro-tasks is to ensure consistent ordering of tasks as well as simultaneously reducing the risk of delays caused by users.

Syntax: Adding micro-tasks:

```
queueMicrotask(() => {
   // Code to be run inside the micro-task
});
```

The micro-task function itself takes no parameters and does not return a value.
Examples: process.nextTick, Promises, queueMicrotask, MutationObserver
Macro-tasks within an event loop:

- Macro-task represents some **discrete and independent work**. These are always the execution of the JavaScript code and micro-task queue is empty. Macro-task queue is often considered the same as the task queue or the event queue. However, the macro-task queue works the same as the task queue. The only small difference between the two is that the **task queue is used for synchronous statements** whereas the **macro-task queue is used for asynchronous statements.**
- In JavaScript, no code is allowed to execute until an event has occurred. It is worth mentioning that the **execution of a JavaScript code execution is itself a macro-task**. The event is queued as a macro-task. When a (macro) task, present in the macro-task queue is being executed, new events may be registered and in turn, created and added to the queue.
- Upon initialization, the JavaScript engine first pulls off the first task in the macro-task queue and executes the callback handler. The JavaScript engine then sends these asynchronous functions to the API module, and the

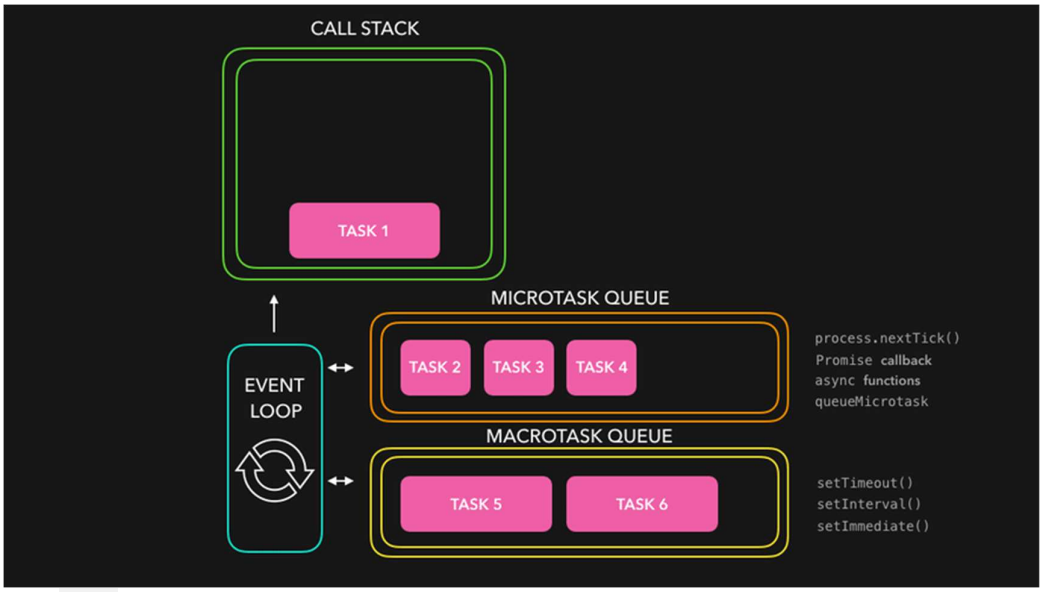
module pushes them to the macro-task queue at the right time. Once inside the macro-task queue, each macro-task is required to wait for the next round of the event loop. In this way, the code is executed.

- All micro-tasks logged are processed in one fell swoop in a single macro-task execution cycle. In comparison, the **macro-task queue has a lower priority**. Macro-tasks include parsing HTML, generating DOM, executing main thread JavaScript code, and other events such as page loading, input, network events, timer events, etc.

Examples: setTimeout, setInterval, setImmediate, requestAnimationFrame, I/O, UI Rendering

Let's take a look at a quick example, simply using:

- Task1: a function that's added to the call stack immediately, for example by invoking it instantly in our code.
- Task2, Task3, Task4: microtasks, for example, a promise then callback, or a task added with queueMicrotask.
- Task5, Task6: a (macro)task, for example, a setTimeout or setImmediate callback



First, Task1 returned a value and got popped off the call stack. Then, the engine checked for tasks queued in the microtask queue. Once all the tasks were put on the call stack and eventually popped off, the engine checked for tasks on the macro task queue, which got popped onto the call stack, and popped off when they returned a value. Let's use it with some real code:

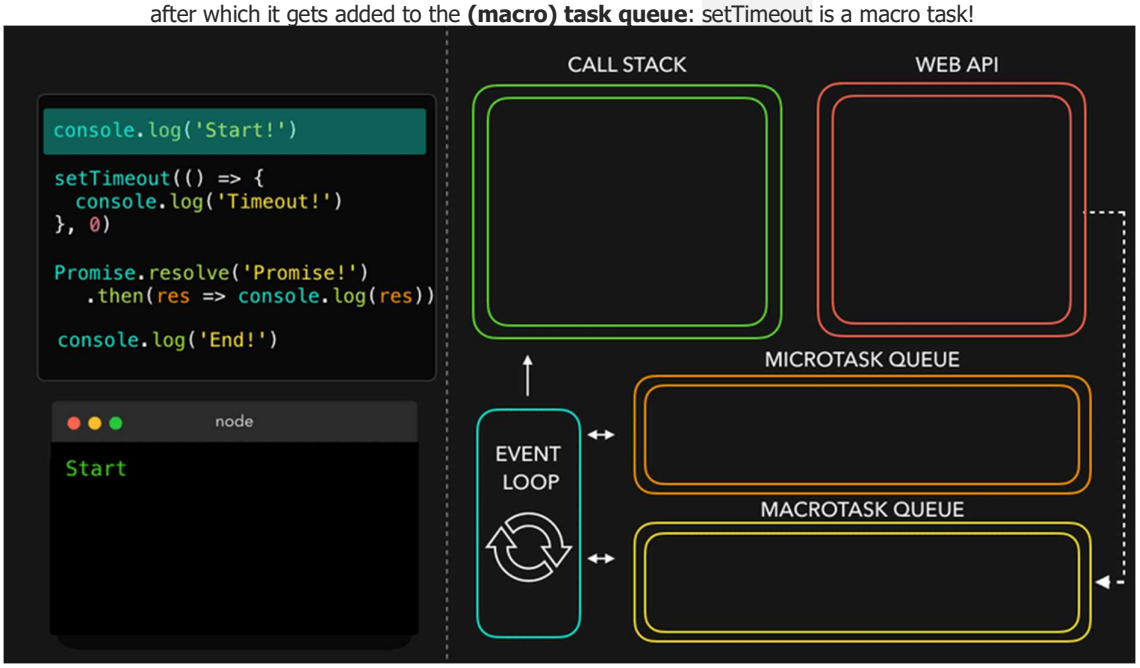
1) Using setTimeout and Promise

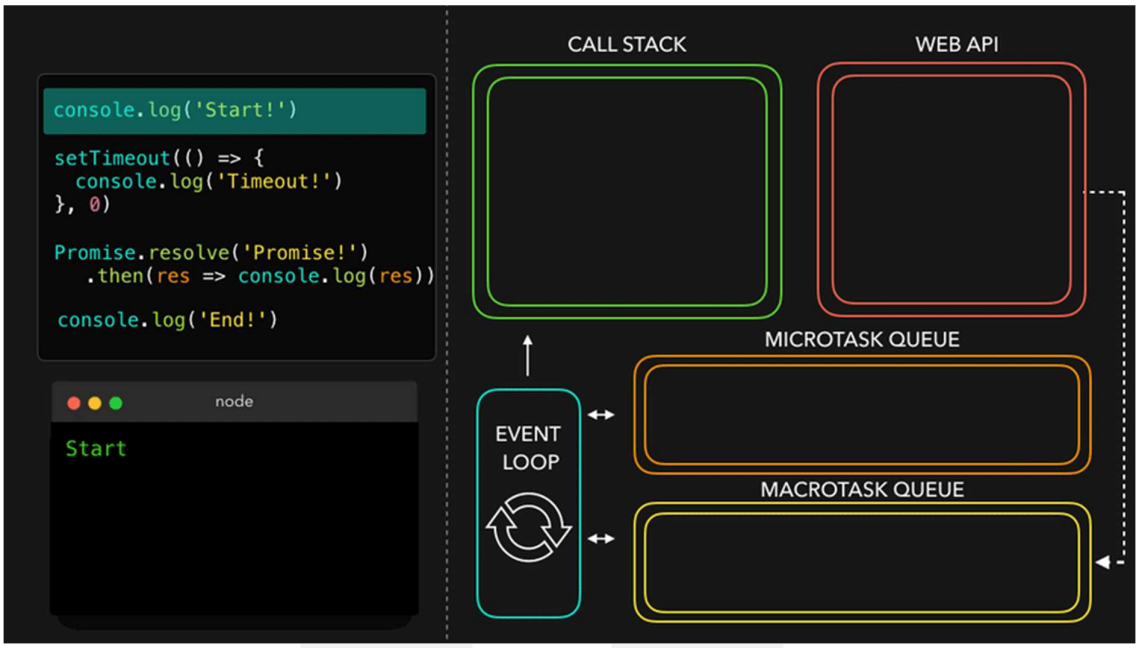
```
console.log('Start!')

setTimeout(() => {
  console.log('Timeout!')
}, 0)

Promise.resolve('Promise!')
    .then(res => console.log(res))

console.log('End!')
```

In this code, we have the macro task setTimeout and the microtask promise then() callback. Let's run this code step-by-step, and see what gets logged:

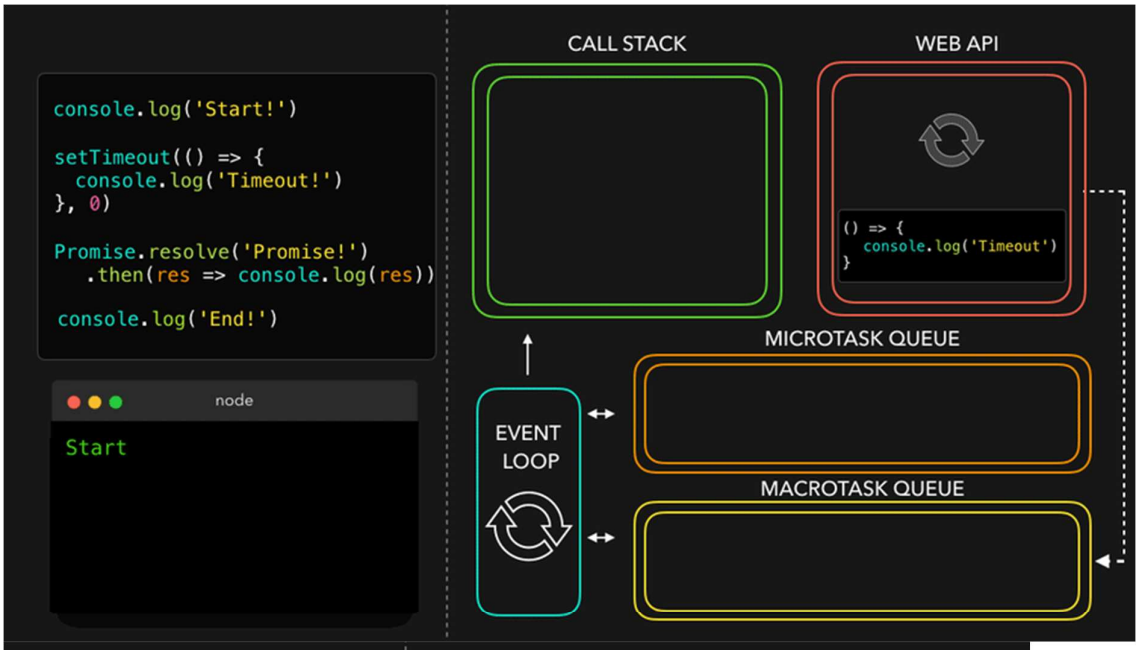
1. On the first line, the engine encounters the console.log() method. It gets added to the call stack, after which it logs the value Start! to the console. The method gets popped off the call stack, and the engine continues.

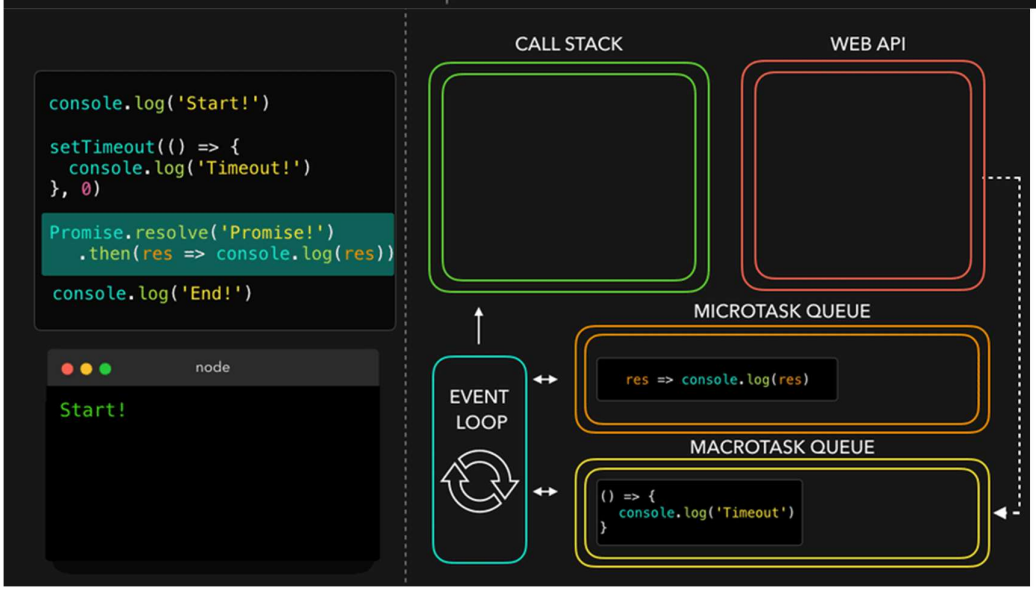
2. The engine encounters the setTimeout method, which gets popped onto the call stack. The setTimeout method is native to the browser: its callback function (() => console.log('Timeout!')) will get added to the Web API until the timer is done. Although we provided the value 0 for the timer, the call back still gets pushed to the Web API first, after which it gets added to the **(macro) task queue**: setTimeout is a macro task!



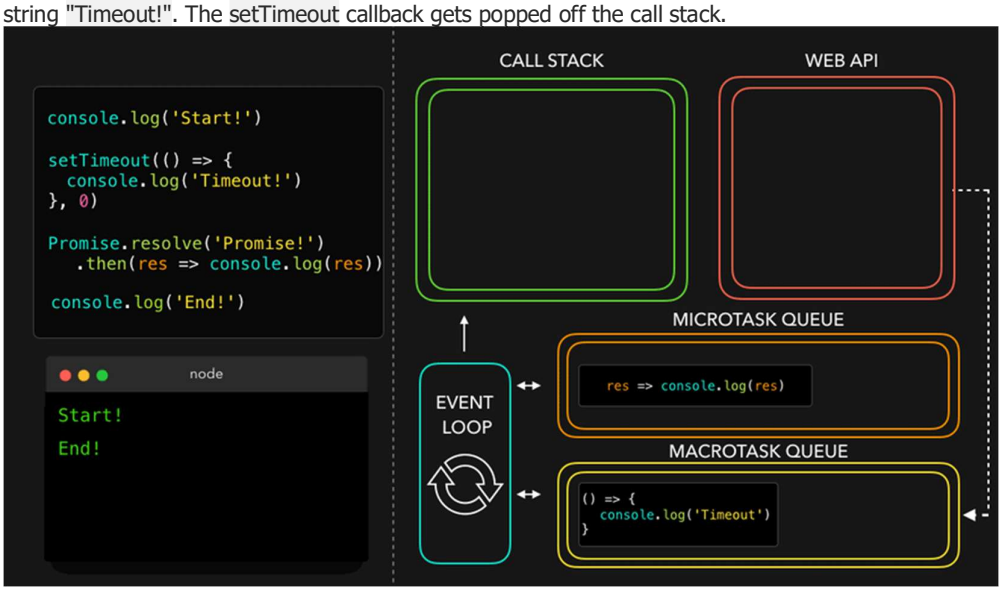


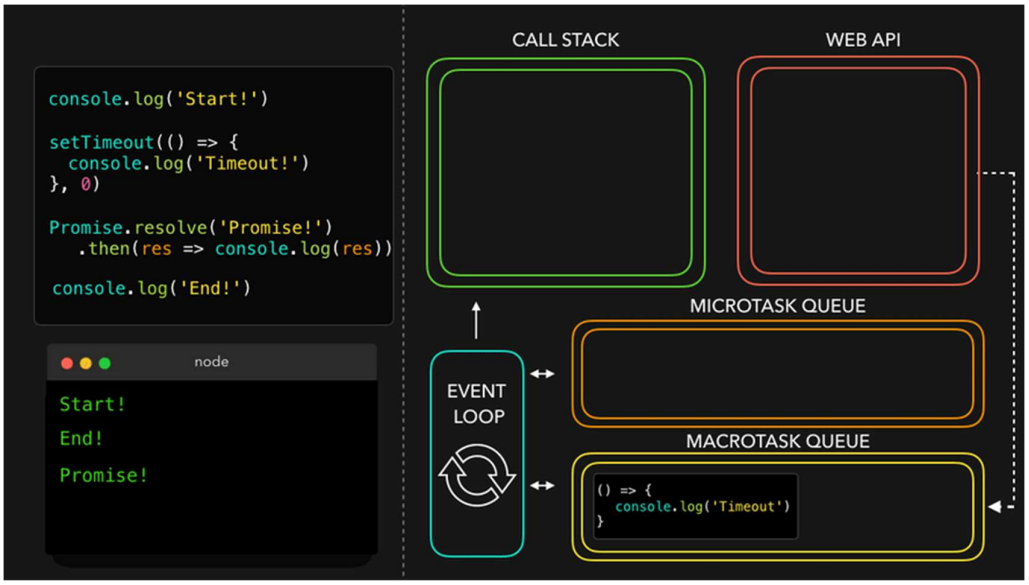
3. The engine encounters the Promise.resolve() method. The Promise.resolve() method gets added to the call stack, after which is resolves with the value Promise!. Its then callback function gets added to the **microtask queue**.
4. The engine encounters the console.log() method. It gets added to the call stack immediately, after which it logs the value End! to the console, gets popped off the call stack, and the engine continues.



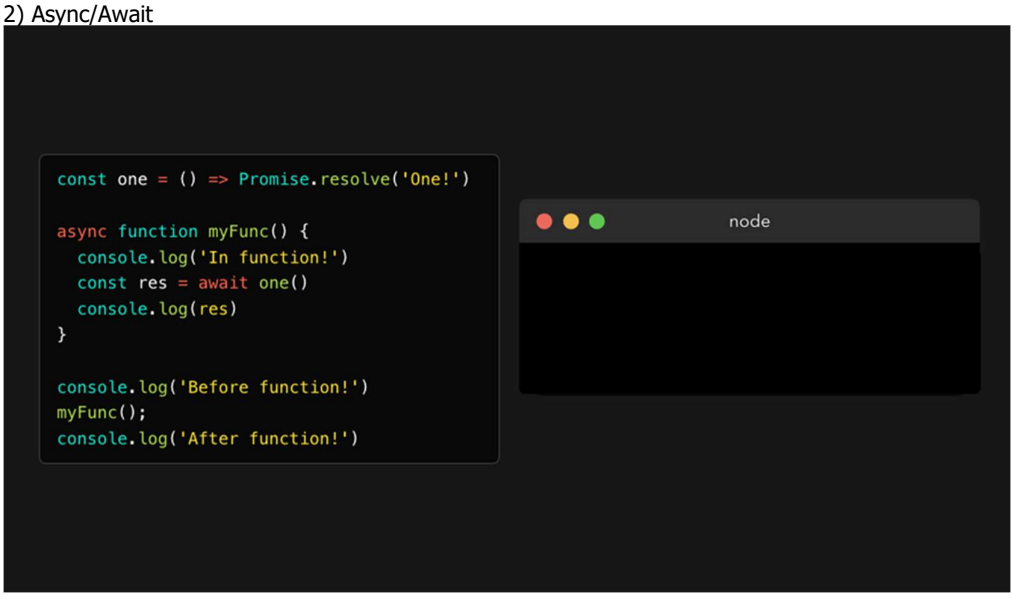


5. The engine sees the call stack is empty now. Since the call stack is empty, it's going to check whether there are queued tasks in the **microtask queue**! And yes there are, the promise then callback is waiting for its turn! It gets popped onto the call stack, after which it logs the resolved value of the promise: the string Promise!in this case.
6. The engine sees the call stack as empty, so it's going to check the microtask queue once again to see if tasks are queued. Nope, the microtask queue is all empty. It's time to check the **(macro)task queue**: the setTimeout callback is still waiting there! The setTimeout callback gets popped onto the call stack. The callback function returns the console.log method, which logs the string "Timeout!". The setTimeout callback gets popped off the call stack.





2) Async/Await



1. First, the engine encounters a console.log. It gets popped onto the call stack, after which Before function! gets logged.
2. Then, we invoke the async function myFunc(), after which the function body myFunc runs. On the very first line within the function body, we call another console.log, this time with the string In function!. The console.log gets added to the call stack, logs the value, and gets popped off.
3. The function body keeps on being executed, which gets us to the second line. Finally, we see a await keyword! 🎉 The first thing that happens is that the value that gets awaited gets executed: the function one in this case. It gets popped onto the call stack, and eventually returns a resolved promise. Once the promise has resolved and one returned a value, the engine encounters the await keyword.
4. When encountering an await keyword, the async function gets *suspended*. ✋ The execution of the function body **gets paused**, and the rest of the async function gets to run in a *microtask* instead of a regular task!
5. Now that the async function myFunc is suspended as it encountered the await keyword, the engine jumps out of the async function and continues executing the code in the execution context in which the async function got called: the **global execution context** in this case! 🏃‍♀️
6. Finally, there are no more tasks to run in the global execution context! The event loop checks to see if there are any microtasks queued up: and there are! The async myFunc function is queued up after resolving the valued of one. myFunc gets popped back onto the call stack, and continues running where it previously left off.
7. The variable res finally gets its value, namely the value of the resolved promise that one returned! We invoke console.log with the value of res: the string One! in this case. One! gets logged to the console and gets popped off the call stack! 😊

Finally, all done! Did you notice how async functions are different compared to a promise then?
The await keyword *suspends* the async function, whereas the Promise body would've kept on being executed if we would've used then!

How Closures Work in JavaScript: A Guide

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

In other words, a closure gives you **access to an outer function's scope from an inner function**. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

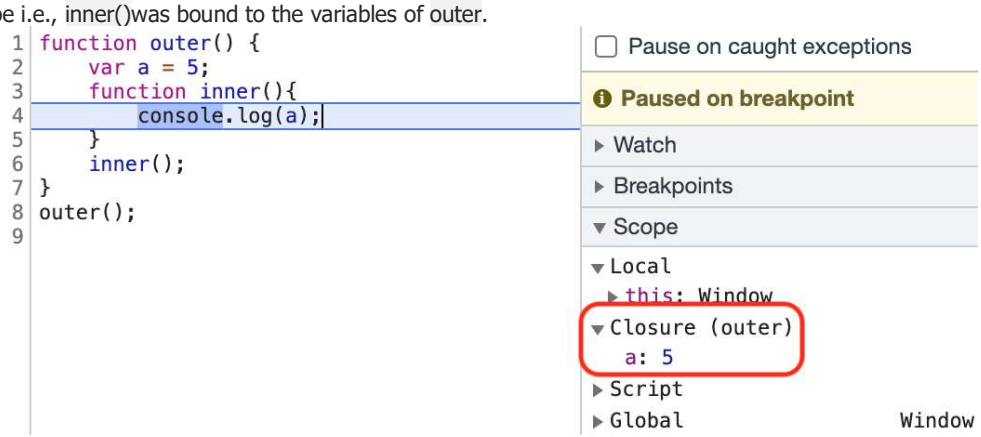
Consider the following example code:

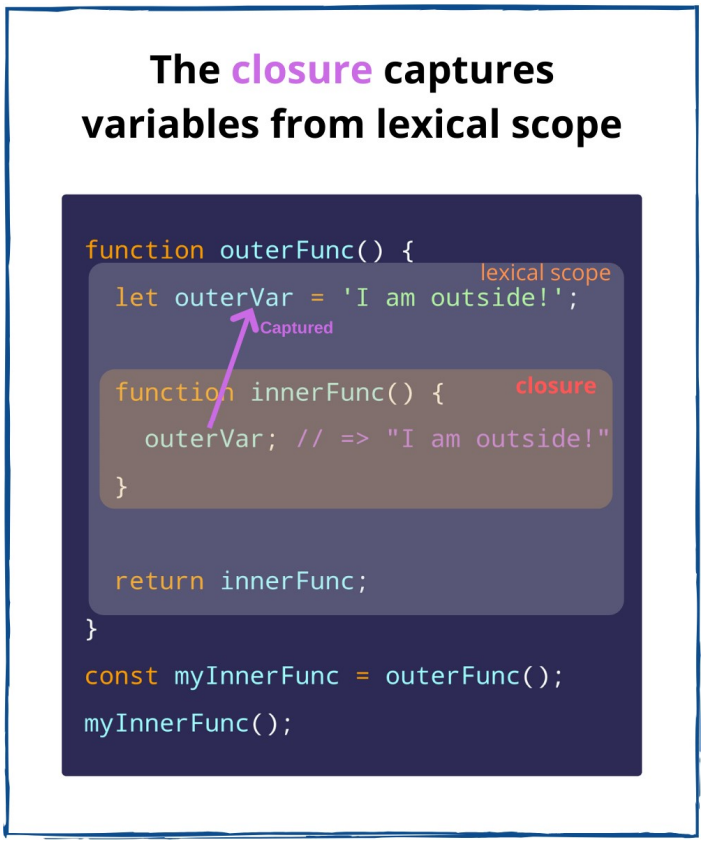
```
function outer() {
   var a = 5;
   function inner(){
      console.log(a);
   }
   inner();
}outer(); //5
```

Here, outer() creates a local variable called aand a function called inner().The inner() function is an inner function that is defined inside outer() and is available only within the body of the outer() function. Note that the inner() function has no local variables of its own. However, since inner functions have access to the variables of outer functions, inner() can access the variable a declared in the parent function, outer().

Notice that the console statement within the inner() function successfully displays the value of the a variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word *lexical* refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

Basically here, inner()is bundled together with its lexical environment of outer(). First, inner()will check its local scope; if it does not find a, it will go to the lexical parent. So inside inner(), it forms a closure with the variable which was part of outer's lexical scope i.e., inner()was bound to the variables of outer.





- When functions are returned from another function, they still maintain their lexical scope i.e., they remember where they were actually present.

```
function outer() {
   var a = 5;
   function inner(){
      console.log(a);
   }
   return inner;
}var x = outer();
x(); //5outer()() //5
```

Here though, outerno longer exists. Still, inner()remembers its lexical scope i.e., when we return inner(), not just the function code is returned, but a closure was returned(function along with lexical scope).

- One important characteristic of closure is that it keeps the state of the outer variable between the multiple calls. The inner function does not contain a separate copy of the variable, it just keeps the reference of the outer variable.

```
function outer() {
   var a = 5;
   function inner(){
      console.log(a);
   }
   a = 100;
   return inner;
}
var x= outer();
x(); //100
```

Here also inner()will come along with its lexical scope i.e. awill not refer to the value, it will refer to its reference.

- If we change the local variable declaration to let and if we pass an extra parameter bin outer(), in both the cases inner function still forms a closure. It is because bis also part of outer environment of inner function.

```
function outer(b) {
   function inner(){
      console.log(a,b);
   }
   let a = 50;
   return inner;
}
var x= outer('Hello');
x(); //50 "Hello"
```

```
//Example
1:
            function ParentFunction() {
               var parentVariable = 60;
               function ChildFunction() {
                  return parentVariable += 1;
               }
               return ChildFunction;
            }

            var executeChild = ParentFunction();
            console.log(executeChild()); //61
            console.log(executeChild()); //62
            console.log(executeChild()); //63
            console.log(executeChild()); //64

            //Example 2:
            var add = (function () {
              var counter = 0;
              return function () {counter += 1; return counter}
            })();

            add(); //1
            add(); //2
            add(); //3
```

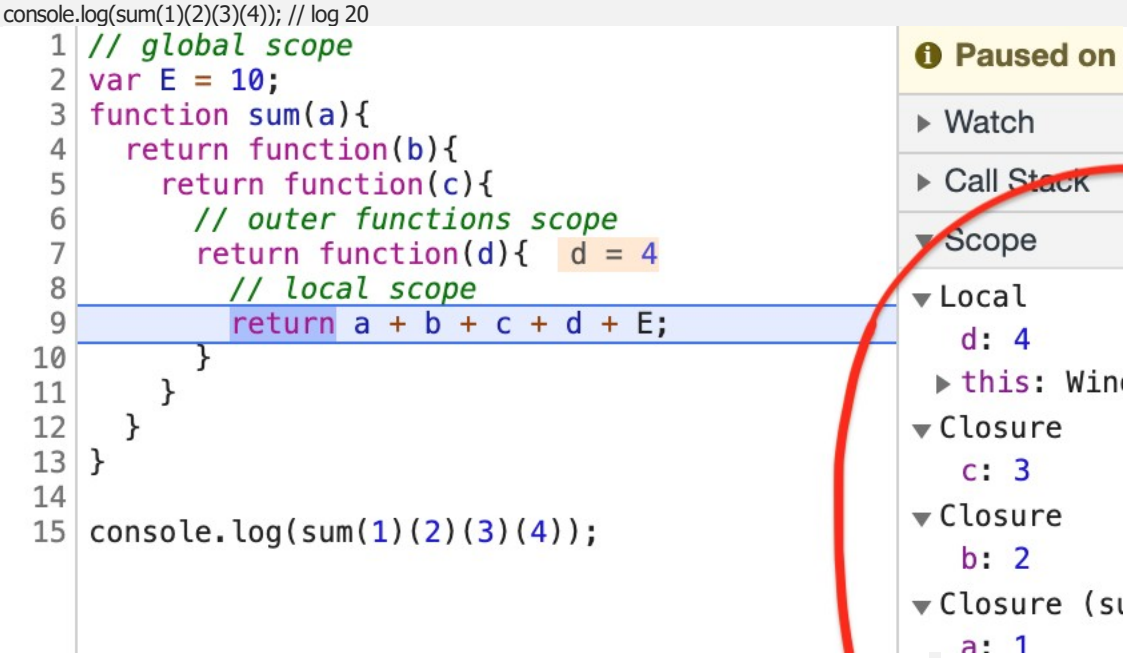
Closure Scope Chain

Every closure has three scopes:

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

To demonstrate, consider the following example code:

```
// global scope
var e = 10;
function sum(a){
  return function(b){
    return function(c){
      // outer functions scope
      return function(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}

console.log(sum(1)(2)(3)(4)); // log 20
```



- Now if we have a global variable with conflicting name i.e same variable name a is present globally and in outer function. Then in this case inner aor the other function scope a value will be returned. And the global ais completely new variable in the global scope. So both the a variables are completely different. Here a is present in the outer environment, if it was not present then it will global a .

```
var a = 10;
function sum(){
  var a = 1;
  return function(b){
    return function(c){
       return a + b + c;
    }
  }
}console.log(sum()(2)(3));
//6 (1+2+3)var a = 10;
function sum(){
  //var a = 1;
  return function(b){
    return function(c){
       return a + b + c;
    }
  }
}
console.log(sum()(2)(3));
//15 (10+2+3)
```

Advantages of Closures

1) Using Private Variables and Methods

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class. JavaScript does not provide a native way of doing this, but it is possible to emulate private variables and methods using closures.

2) Data Hiding and Encapsulation

Closures make data hiding possible by creating private variables i.e other functions or pieces of code will not have access to that particular data. We can say we can encapsulate the data so that other part of program cannot access it. Lets understand this with help of a example:

```
function counter(){
   let count = 0;

   return function incrementCounter(){
     count++;
     console.log(count);
   }
}console.log(count); // Uncaught ReferenceError: count is not definedvar counter1 = counter();
counter1(); //1
counter1(); //2var counter2 = counter();
counter2(); //1
counter2(); //2
counter2(); //3
```

Here, count variable is private or hidden and will not be accessible outside the counter function. counter1 and counter2 are altogether different variables with different scopes.
Now if we want to add decrement counter also then good and scalable way to do is to create function constructors.

```
function Counter(){
   let count = 0;

   this.incrementCounter = function(){
     count++;
     console.log(count);
   }
   this.decrementCounter  = function(){
     count--;
     console.log(count);
   }
}var counter1 = new Counter(); counter1.incrementCounter(); //1
counter1.incrementCounter(); //2counter1.decrementCounter(); //1
```

3) Function Currying
Closure makes currying possible in JavaScript. Currying is an advanced technique of working with functions. Currying is a transformation of functions that translates a function from callable as f(a, b, c) into callable as f(a)(b)(c). It is when you break down a function that takes multiple arguments into a series of functions that each take only one argument.

```
function add (a, b) {
  return a + b;
}add(3, 4); // returns 7
```

This is a function that takes two arguments, a and b, and returns their sum. We will now curry this function:

```
function add (a) {
  return function (b) {
    return a + b;
  }
}
```

This is a function that takes one argument, a, and returns a function that takes another argument, b, and that function returns its sum.

```
add(3)(4); //7
var add3 = add(3);
add3(4);  //7
```

The first statement returns 7, like the add(3, 4) statement. The second statement defines a new function called add3 that will add 3 to its argument. This is what some people may call a closure. The third statement uses the add3 operation to add 3 to 4, again producing 7 as a result.
For more detail refer — **JavaScript Currying: Comprehensive Guide**
4) Function Factories
One powerful use of closures is to use the outer function as a factory for creating functions that are somehow related.

```
function Job(title) {
   return function(prefix) {
      return prefix + ' ' + title;
   };
}var sales = Job('Salesman');
var manager = Job('Manager');alert(sales('Top'));  // Top Salesman
alert(manager('Assistant to the Regional')); // Assistant to the Regional Manager
alert(manager('Regional')); // Regional Manager
```

Using closures as function factories is a great way to keep your JavaScript DRY. Five powerful lines of code allow us to create any number of functions with similar, yet unique purposes.
5) Run function only once
Functions remember how many times the function has been run by forming a closure.

```
var something = (function() {
   var executed = false;
   return function() {
      if (!executed) {
         executed = true;
         console.log("do something");
      }
```

```
  };
})();something(); // "do something" happens
something(); // nothing happens
```

6) setTimeout
It is a method that calls a function or evaluates an expression after a specified number of milliseconds.

```
function x(){
  var i = 1;
  setTimeout(function(){
     console.log(i)
  },3000)
  console.log("After setTimeout")
}x();
//After setTimeout
// 1 (after 3 sec)
```

Here, function in setTimeout forms a closure. So this function remembers the reference to "i". So wherever this function goes it takes the value of "i" with it. setTimeout takes a callback function and stores it in someplace and attaches a timer to that. Once the timer expires, it will take the function and put it to the current call stack, and runs it.
7) Memoization
Memoization is the programmatic practice of making long recursive/iterative functions run much faster. Closures are used here as well. Here's an example of a basic memoize function:
8) Closures are also used to maintain state in the async world, in iterators, and in module design patterns.
Disadvantages of Closures

- Closures prevent variables inside functions from being released by memory i.e. as long as the closure is active, the memory can't be garbage collected. These variables will occupy memory and consume a lot of memory, which may lead to **memory leakage.** The solution to this problem is to delete all unnecessary local variables in time when these variables are not used i.e., set closure to null.
- Creating a function inside a function leads to duplicity in memory and causes the **slowing down of the application**. The solution to this problem is to use closures only when you need privacy. Otherwise, use module patterns to create new objects with shared methods.

What is better for HTTP Requests: Fetch or Axios Comparison?

Fetch — The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.
Fetch() is part of a JavaScript window-object method within the Fetch API. It is built-in, so users don't have to install it. Fetch() allows us to get data from the API asynchronously without installing any additional libraries.
The fetch() method takes one mandatory argument—the path to the resource you want to fetch—and returns a Promise that resolves with an object of the built-in Response class as soon as the server responds with headers. Let's take a look at the syntax of the .fetch() method.

```
fetch(url)
        .then((res) =>
          // handle response
        )
        .catch((error) => {
          // handle error
        })
```

The second argument in .fetch() method are options, and it's optional. If we won't pass the options the request is always GET, and it downloads the content from the given URL. Inside the options parameter, we can pass methods or headers, so if we would like to use the POST method or any other, we have to use this optional array.

```
{
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache',
     // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
      'Content-Type': 'application/json'
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *client
    body: JSON.stringify(data)
     // body data type must match "Content-Type" header
  }
```

Once a Response is retrieved, the returned object contains the following properties:

- response.body: A simple getter exposing a ReadableStream of the body contents
- response.bodyUsed: Stores a Boolean that declares whether the body has been used in response yet
- response.headers: The headers object associated with the response
- response.ok: A Boolean indicating whether the response was successful or not

    - response.redirected: Indicates whether or not the response is the result of a redirect
    - response.status: The status code of the response
    - response.statusText: The status message corresponding to the status code
  - response.type: The type of the response
  - response.url: The URL of the response

There are a few different methods that we can use, depends on the format of the body that we need: **response.json(), response.text(), response.formData(), response.blob(), response.arrayBuffer().**
Let's take a look at the code example with an optional parameter.

```
fetch(url, {
              method: 'POST',
              headers: {
                'Content-Type': 'application/json'
              },
              body: JSON.stringify(data)
            });
              .then((response) => response.json())
              .catch((error) => console.log(error))
```

In the code example above, you can see the simple POST request with method, header, and body params.
Axios — Axios is a Javascript library used to make HTTP requests from node.js or XMLHttpRequests from the browser, and it supports the Promise API that is native to JS ES6.
Some core features of Axios, according to the documentation, are:

- It can be used to intercept HTTP requests and responses.
- It automatically transforms request and response data.
- It enables client-side protection against CSRF.
- It has built-in support for download progress.
- It has the ability to cancel requests.

- To be able to use axios library, we have to install it and import it to our project. axios can be installed using CDN, npm, or bower. Now let's take a look at the syntax of a simple GET method.

```
axios.get(url)
              .then(response => console.log(response));
              .catch((error) => console.log(error));
```

In the code above, you can see how axios is used to create a simple GET request using .get() method. If you'd like to use the POST method in the function, then it's enough to use .post() method instead and pass the request data as a parameter.
Axios also provides more functions to make other network requests as well, matching the HTTP verbs that you wish to execute, such as:

- axios.request(config)
- axios.get(url[, config])
- axios.delete(url[, config])
- axios.head(url[, config])
- axios.options(url[, config])
- axios.post(url[, data[, config]])
- axios.put(url[, data[, config]])
- axios.patch(url[, data[, config]])

When we are creating a config object we can define a bunch of properties, the most common are: **baseUrl, params, headers, auth, responseType**
As a response, axios returns a promise that will resolve with the response object or an error object. The response from a request contains the following information:

- response.data: The response provided by the server
- response.status: The HTTP status code from the server response, like 200 or 404
- response.statusText: HTTP status message from the server response, for example, ok
- response.headers: The headers that the server responded with
- response.config: The config that was provided to axios for the request
- response.request: The request that generated this response

Let's take a look at the code example with the POST method with data.

```
axios.post({
              '/url',
              { name: 'Ayush', age: 30},
              { options }
            })
```

In the code above, you can see the post method, where we put the config object as a param, with URL, data, and additional options.
We can also define the config object as a variable and pass it to the axios like in the example below.

```
const config = {
  url: 'http://api.com',
```

```
  method: 'POST',
  header: {
    'Content-Type': 'application/json'
  },
  data: {
    name: 'Ayush',
    age: 30
  }
}
axios(config);
```

Here, you can see that all the parameters, including URL, data, or method, are in the config object, so it may be easier to define everything in one place.

JSON data

Fetch — There is a two-step process when handling JSON data with fetch(). First, we have to make the actual request, and then we call the .json() method on the response i.e we need to use some kind of method on the response data, and when we are sending the body with the request, we need to stringify the data.

Axios — In axios it's done automatically, so we just pass data in the request or get data from the response. It's automatically stringified, so no other operations are required.

Let's see how we can get data from fetch() and from axios.

```
//
fetch
        fetch('url')
          .then((response) => response.json())
          .then((data) => console.log(data))
          .catch((error) => console.log(error))

        // axios
        axios.get('url')
          .then((response) => console.log(response))
          .catch((error) => console.log(error))
```

In the example above, you can see that with axios we don't have an additional line of code, where we have to convert data to JSON format, and we have this line in .fetch() example. In the case of a bigger project where you create a lot of calls, it's more comfortable to use axios to avoid repeating the code.

Error handling

Axios — At this point, we also need to give points for axios as handling errors is pretty easy. If there will be a bad response like 404, the promise will be rejected and will return an error, so we need to catch an error, and we can check what kind of error it was, that's it. Let's see the code example.

```
axios.get('url')
                .then((response) => console.log(response))
                .catch((error) => {
                  if (error.response) {
                    // When response status code is out of 2xx range
                    console.log(error.response.data)
                    console.log(error.response.status)
                    console.log(error.response.headers)
                  } else if (error.request) {
                    // When no response was received after request was made
                    console.log(error.request)
                  } else {
                    // Error
                    console.log(error.message)
                  }
                })
```

In the code above, we've returned data when the response was good, but if the request failed in any way, we were able to check the type of error in .catch() part and return the proper message.

Fetch — With the .fetch() method, it's a little bit more complicated. Every time we get a response from the .fetch() method, we need to check if the status is a success because even if it's not, we will get the response. In case of .fetch() , a promise won't be resolved if and only if the request won't be completed. Fetch() doesn't throw network errors. Therefore, you must always check the response.ok property when you work with fetch(). Let's see the code example.

```
fetch('url')
              .then((response) => {
                if (!response.ok) {
                  throw Error(response.statusText);
                }
                return response.json()
              })
              .then((data) => console.log(data))
              .catch((error) => console.log(error))
```

In this code, we've checked the status of the code in the promise object, and if the response had status ok, then we could process and use .json() method, but if not, we had to return an error inside .then().

For easy and proper error handling, axios will be definitely a better solution for your project, but still, if you are building a small project with one or two requests, it's fine to use .fetch(), but you need to remember to handle errors correctly.

HTTP interception

HTTP interception can be important when we need to check or change our HTTP requests from the application to the server, or in the other way, for example, for authentication, logging.

Axios — In the case of axios HTTP interception is one of the key features of this library, that's why we don't have to create additional code to use it. Let's take a look at the code example to see how easy we can do it.

```
// request interceptor
axios.interceptors.request.use((config) => {
  console.log('Request was sent');
  return config;
})
// response interceptor
axios.interceptors.response.use((response) => {
  // do an operation on response
  return response;
})
axios.get('url')
  .then((response) => console.log(response))
  .catch((error) => console.log(error))
```

In the code, you can see the request interception and response interception. In the first case, we created a console.log informing about sending requests, and in the response interception, we can do any action on response and then return it.

Fetch — .fetch() doesn't provide the HTTP interception by default, there's a possibility to overwrite the .fetch() method and define what needs to happen during sending the request, but of course, it will take more code and can be more complicated than using axios functionality.

Response timeout

Axios — In Axios, you can use the optional timeout property in the config object to set the number of milliseconds before the request is aborted. For example:

```
axios({
  method: 'post',
  url: '/login',
  timeout: 4000,    // 4 seconds timeout
  data: {
    firstName: 'Ayush',
    lastName: 'Verma'
  }
})
.then(response => {/* handle the response */})
.catch(error => console.error('timeout exceeded'))
```

Fetch — Fetch() provides similar functionality through the AbortController interface. It's not as simple as the Axios version, though:

```
const controller = new AbortController();
const options = {
  method: 'POST',
  signal: controller.signal,
  body: JSON.stringify({
    firstName: 'Ayush',
    lastName: 'Verma'
  })
};
const promise = fetch('/login', options);
const timeoutId = setTimeout(() => controller.abort(), 4000);
promise
  .then(response => {/* handle the response */})
  .catch(error => console.error('timeout exceeded'));
```

Here, we create an AbortController object using the AbortController.AbortController() constructor, which allows us to later abort the request. signal is a read-only property of AbortController providing a means to communicate with a request or abort it. If the server doesn't respond in less than four seconds, controller.abort() is called, and the operation is terminated.

The simplicity of setting timeout in Axios is one of the reasons some developers prefer it to fetch().

Simultaneous requests

Axios — To make multiple simultaneous requests, Axios provides the axios.all() method. Simply pass an array of requests to this method, then use axios.spread() to assign the properties of the response array to separate variables:

```
axios.all([
            axios.get('https://api.github.com/users/iliakan'),
            axios.get('https://api.github.com/users/taylorotwell')
          ])
          .then(axios.spread((obj1, obj2) => {
            // Both requests are now complete
            console.log(obj1.data.login + ' has ' + obj1.data.public_repos + ' public repos on GitHub');
            console.log(obj2.data.login + ' has ' + obj2.data.public_repos + ' public repos on GitHub');
          }));
```

Fetch — We can achieve the same result by using the built-in `Promise.all()` method. Pass all fetch requests as an array to `Promise.all()`. Next, handle the response by using an `async` function, like this:

```
Promise.all([
              fetch('https://api.github.com/users/iliakan'),
              fetch('https://api.github.com/users/taylorotwell')
            ])
            .then(async([res1, res2]) => {
              const a = await res1.json();
              const b = await res2.json();
              console.log(a.login + ' has ' + a.public_repos + ' public repos on GitHub');
              console.log(b.login + ' has ' + b.public_repos + ' public repos on GitHub');
            })
            .catch(error => {
              console.log(error);
            });
```

Download progress

When we have to download a large amount of data, a way to follow the progress would be useful, especially when users have slow internet. Earlier, to implement progress indicators developers used `XMLHttpRequest.onprogress` callback. In `.fetch()` and `axios`, there are different ways to do it.

Fetch — To track the progress of the download in `.fetch()` we can use one of the `response.body` properties, a `ReadableStream` object. It provides body data chunk by chunk, and it allows us to count how much data is consumed in time.

Axios — In `axios`, implementing a progress indicator is possible as well, and it's even easier because there exists a ready module, which can be installed and implemented; it's called **Axios Progress Bar**.

If you have a lot of large data to download and you want to track the progress in progress indicator, you can manage that easier and faster with `axios` but `.fetch()` gives the possibility as well, just it needs more code to be developed for the same result.

Upload progress

Fetch — In `fetch()`, you can't monitor the progress of your uploads.

Axios — In Axios, you can monitor the progress of your uploads. This could be a deal-breaker if you're developing an application for video or photo uploading.

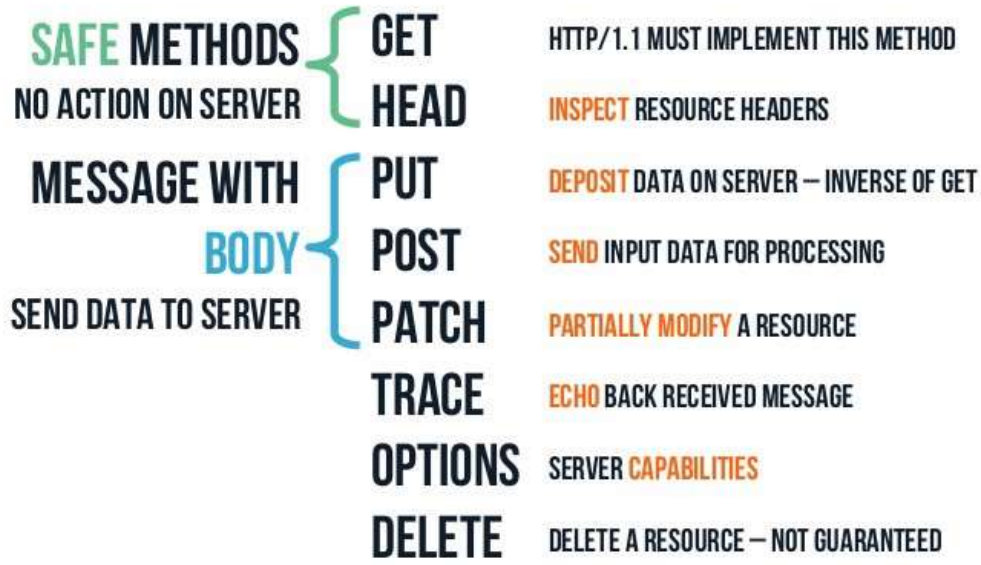
Browser support (Backward Compatibility)

Axios — Axios has **wide browser support**.

Compatibility table

Fetch — Fetch only supports Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+. If you need to support older browsers, a polyfill is available.

HTTP Request Methods: A Complete Guide



GET

- GET requests are the most common and widely used methods in APIs and websites. Simply put, the GET method is used to **retrieve data from a server at the specified resource**. For example, say you have an API with a /users endpoint. Making a GET request to that endpoint should return a list of all available users.
- Since a GET request is only requesting data and not modifying any resources, it's considered a **safe** and **idempotent method**.

,

Request has body	No
Successful response has body	Yes
Safe	Yes
Idempotent	Yes
Cacheable	Yes
Allowed in HTML forms	Yes

//
fetch

```
fetch('url')
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.log(error))

// axios
axios.get('url')
  .then((response) => console.log(response))
  .catch((error) => console.log(error))
```

POST

- In web services, POST requests are used to **send data to the API server** to create or update a resource. The data sent to the server is stored in the request body of the HTTP request.
- When a new resource is POSTed to the server, the API service will automatically associate the new resource by assigning it an ID (new resource URI). In short, this method is used to create a new data entry.
- The simplest example is a contact form on a website. When you fill out the inputs in a form and hit *Send*, that data is put in the **response body** of the request and sent to the server. This may be JSON, XML, or query parameters (there's plenty of other formats, but these are the most common).

- It's worth noting that a `POST` request is **non-idempotent**. It mutates data on the backend server (by creating or updating a resource), as opposed to a `GET` request which does not change any data.

Request has body	Yes
Successful response has body	Yes
Safe	No
Idempotent	No
Cacheable	Only if freshness information is included
Allowed in HTML forms	Yes

```
const
someData
= {
            name: 'Ayush',
            age: 30,
            completed: false
           }
           const postMethod = {
            method: 'POST', // Method itself
            headers: {
             'Content-type': 'application/json; charset=UTF-8' // Indicates the content
            },
            body: JSON.stringify(someData) // We send data in JSON format
           }

           // make the HTTP post request using fetch api
           fetch(url, postMethod)
             .then(response => response.json())
             .then(data => console.log(data)) // Manipulate the data retrieved back, if we want to do something with it
             .catch(err => console.log(err)) // Do something with the error

           //axios
           const config = {
             url: 'http://api.com/users',
             method: 'POST',
             header: {
               'Content-Type': 'application/json'
             },
             data: {
               name: 'Ayush',
               age: 30
             }
           }
           axios(config);
```

	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
Encoding type	application/x-www-form-urlencoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
History	Parameters remain in browser history	Parameters are not saved in browser history
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
Security	GET is less secure compared to POST because data sent is part of the URL Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

PUT

- Similar to POST, PUT requests are used to send data to the API to **update or create a resource**. The difference is that **PUT requests are idempotent**. That is, calling the same PUT request multiple times **will always produce the same result**. In contrast, calling a POST request repeatedly make have side effects of creating the same resource multiple times.
- Generally, when a PUT request *creates* a resource the server will respond with a 201 (Created), and if the request *modifies* existing resource the server will return a 200 (OK) or 204 (No Content).

Request has body	Yes
Successful response has body	No
Safe	No
Idempotent	Yes
Cacheable	No
Allowed in HTML forms	No

```
const
someData
= {
            name: 'Ayush',
            age: 30,
            completed: false
           }
           const putMethod = {
            method: 'PUT', // Method itself
            headers: {
             'Content-type': 'application/json; charset=UTF-8' // Indicates the content
            },
            body: JSON.stringify(someData) // We send data in JSON format
           }

           // make the HTTP put request using fetch api
           fetch(url, putMethod)
             .then(response => response.json())
             .then(data => console.log(data)) // Manipulate the data retrieved back, if we want to do something with it
             .catch(err => console.log(err)) // Do something with the error
```

PUT	POST
It is idempotent, meaning that putting a resource twice will have no effect	It is not idempotent, and thus calling a POST request repeatedly is discouraged
Identity is selected by the client	Identity is returned by the server
Operates as specific	Operates as abstract

PATCH

- A PATCH request is one of the lesser-known HTTP methods, but it is similar to POST and PUT. The difference with PATCH is that you **only apply partial modifications to the resource**.
- The difference between PATCH and PUT, is that a **PATCH request is non-idempotent** (like a POST request).
- To expand on partial modification, say you're API has a /users/{{userid}} endpoint, and a user has a *username*. With a PATCH request, **you may only need to send the updated username** in the request body - as opposed to POST and PUT which require the full user entity.

Request has body	Yes
Successful response has body	Yes
Safe	No
Idempotent	No
Cacheable	No
Allowed in HTML forms	No

```
const
someData
= {
            completed: true // update that task is completed
           }
           const patchMethod = {
            method: 'PATCH', // Method itself
            headers: {
             'Content-type': 'application/json; charset=UTF-8' // Indicates the content
            },
            body: JSON.stringify(someData) // We send data in JSON format
           }

           // make the HTTP patch request using fetch api
           fetch(url, patchMethod)
             .then(response => response.json())
             .then(data => console.log(data)) // Manipulate the data retrieved back, if we want to do something with it
             .catch(err => console.log(err)) // Do something with the error
```

As you can see here, the request is very similar to the PUT request, but the body of the request contains only the property of the resource that needs to be changed

DELETE

- The DELETE method is exactly as it sounds: **delete the resource at the specified URL**. This method is one of the more common in RESTful APIs so it's good to know how it works.
- If a new user is created with a POST request to /users, and it can be retrieved with a GET request to /users/{{userid}}, then making a DELETE request to /users/{{userid}} will completely remove that user.

Request has body	May
Successful response has body	May
Safe	No
Idempotent	Yes
Cacheable	No
Allowed in HTML forms	No

```
const
deleteMethod
= {
                method: 'DELETE', // Method itself
                headers: {
                 'Content-type': 'application/json; charset=UTF-8' // Indicates the content
                },
                // No need to have body, because we don't send nothing to the server.
               }
               const url = '/users/{{userid}}';

               // Make the HTTP Delete call using fetch api
               fetch(url, deleteMethod)
                .then(response => response.json())
                .then(data => console.log(data)) // Manipulate the data retrieved back, if we want to do something with it
                .catch(err => console.log(err)) // Do something with the error
```

HEAD

- The HEAD method is almost identical to GET, **except without the response body**. In other words, if GET /users returns a list of users, then HEAD /users will make the same request but won't get back the list of users.
- HEAD requests are **useful for checking what a GET request will return** before actually making a GET request. For example, if a URL might produce a large download, a HEAD request could read its Content-Length header to check the filesize without actually downloading the file.

Request has body	No
Successful response has body	No
Safe	Yes
Idempotent	Yes
Cacheable	Yes
Allowed in HTML forms	No

The following example demonstrates sending an HTTP HEAD request to the query about the users:

```
HEAD /users HTTP 1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: medium.com
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
```

The response contains the same as the GET method.

```
HTTP/1.1 200 OK
Date: Mon, 23 Sept 2021 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2021 19:15:56 GMT
ETag: "34aa387-d-1568eb00"
Vary: Authorization,Accept
```

```
Accept-Ranges: bytes
Content-Length: 88
Content-Type: application/json
Connection: Closed
```

OPTIONS

- The HTTP `OPTIONS` method requests permitted communication options for a given URL or server. A client can specify a URL with this method, or an asterisk (*) to refer to the entire server. The OPTIONS request should **return data describing what *other* methods and operations the server supports** at the given URL.
- OPTIONS requests are more loosely defined and used than the others, making them a good candidate to **test for fatal API errors**. If an API isn't expecting an OPTIONS request, it's good to put a test case in place that verifies failing behavior.
- In CORS, a preflight request is sent with the `OPTIONS` method so that the server can respond if it is acceptable to send the request.

Request has body
Successful response has body
Safe
Idempotent
Cacheable

The following example requests a list of methods supported by a web server running on medium.com:

```
OPTIONS * HTTP/1.1
Host: medium.com
Origin: https://medium.com/
```

And the server response:

```
HTTP/1.1 200 OK
Allow: GET,POST,PUT,PATCH,DELETE,HEAD,OPTIONS
Access-Control-Allow-Origin: https://medium.com/
Access-Control-Allow-Methods: GET,POST,PUT,PATCH,DELETE,HEAD,OPTIONS
Access-Control-Allow-Headers: Content-Type
```

TRACE

- The **HTTP `TRACE` method** is designed for diagnostic purposes. It performs a message loop-back test along the path to the target resource, providing a useful debugging mechanism.
- The final recipient of the request should reflect the message received, excluding some fields described below, back to the client as the message body of a 200 (OK) response with a Content-Type of `message/http`. The final recipient is either the origin server or the first server to receive a Max-Forwards value of 0 in the request.

Request has body
Successful response has body
Safe
Idempotent
Cacheable

The following example shows the usage of the TRACE method:

```
TRACE / HTTP/1.1
Host: www.medium.com
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
```

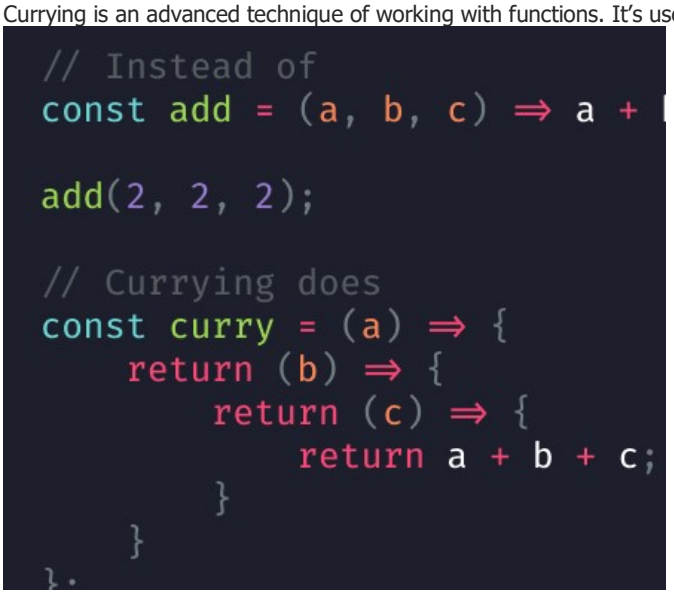
The server response is sent back for the above client request.

```
HTTP/1.1 200 OK
Date: Mon, 23 Sept 2021 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Connection: close
```

```
Content-Type: message/http
Content-Length: 39TRACE / HTTP/1.1
Host: www.medium.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
```

JavaScript Currying

Currying is an advanced technique of working with functions. It's used not only in JavaScript but in other languages as well.



Currying is a process in functional programming in which we can transform a function with multiple arguments into a sequence of nesting functions. It returns a new function that expects the next argument inline.

- In other words, when a function, instead of taking all arguments at one time, takes the first one and returns a new function that takes the second one and returns a new function which takes the third one, and so forth, until all arguments have been fulfilled.
- Currying is a transformation of functions that translates a function from callable as `f(a, b, c)` into callable as `f(a)(b)(c)`. Currying doesn't call a function. It just transforms it.
- The number of arguments a function takes is also called **arity**.

```
function sum(a, b) {
  // do something
}
function _sum(a, b, c) {
  // do something
}
```

function `sum` takes two arguments (2-arity function) and `_sum` takes three arguments (3-arity function).
Curried functions are constructed by chaining closures by defining and immediately returning their inner functions simultaneously.
Example:

```
function sum(a, b, c) {
  return a + b + c;
}sum(1,2,3); // 6
```

As we see, function with the full arguments. Let's create a curried version of the function and see how we would call the same function (and get the same result) in a series of calls:

```
function sum(a) {
  return (b) => {
    return (c) => {
      return a + b + c
    }
  }
}

console.log(sum(1)(2)(3)) // 6
```

We could separate this sum(1)(2)(3) to understand it better:

```
const sum1 = sum(1);
const sum2 = sum1(2);
const result = sum2(3);
console.log(result); // 6
```

Let's get to know how it works:
We passed 1 to the `sum` function: `let sum1 = sum(1);`
The `sum` returns the function:

```
return (b) => {
  return (c) => {
     return a + b + c
  }
}
```

Now, sum1 holds the above function definition which takes an argument b. We called the sum1 function, passing in 2: let sum2 = sum1(2);

The sum1 will return the third function:

```
return (c) => {
  return a + b + c
}
```

The returned function is now stored in sum2 variable. sum2 will be:

```
sum2 = (c) => {
    return a + b + c
}
```

When sum2 is called with 3 as the parameter, const result = sum2(3); It does the calculation with the previously passed in parameters: a = 1, b = 2 and returns 6.

```
console.log(result); // 6
```

Being a nested function, sum2 has access to the variable scope of the outer functions, sum and sum1. This is how sum2 could perform the sum operation with variables defined in the already exit-ed functions. Though the functions have long since returned and garbage collected from memory, yet its variables are somehow still kept "alive".

You see that the three numbers were applied one at a time to the function, and at each time, a new function is returned until all the numbers are exhausted. The last function only accepts c variable but will perform the operation with other variables whose enclosing function scope has long since returned. It works nonetheless because of Closure.

Currying & Partial application

Some might start to think that the number of nested functions a curried function has depends on the number of arguments it receives. Yes, that makes it a curry. Let's take same sum example:

```
function sum(a) {
   return (b, c) => {
      return a + b + c;
   }
}
```

It can be called like this:

```
let x = sum(10);
x(3,12);
x(20,12);
x(20,13);// ORsum(10)(3,12);
sum(10)(20,12);
sum(10)(20,13);
```

The above function expects 3 arguments and has 2 nested functions, unlike our previous version that expects 3 arguments and has 3nesting functions. **This version isn't a curry.** We just did a partial application of the sum function.

Currying and Partial Application are related (because of closure), but they are of different concepts. **Partial application** transforms a function into another function with smaller arity.

```
function sum1(x, y, z) {
   return sum2(x,y,z)
}// tofunction sum1(x) {
   return (y,z) => {
      return sum2(x,y,z)
   }
}
```

For Currying, it would be like this:

```
function sum1(x) {
   return (y) = > {
      return (z) = > {
         return sum2(x,y,z)
      }
   }
}
```

Currying creates nesting functions according to the number of the arguments of the function. Each function receives an argument. If there is no argument there is no currying.

More advanced implementations of currying, such as _.curry from lodash library, return a wrapper that allows a function to be called both normally and partially:

```
function sum(a, b) {
  return a + b;
}let curriedSum = _.curry(sum); // using _.curry from lodash libraryalert( curriedSum(1, 2) ); // 3, still callable normally
alert( curriedSum(1)(2) ); // 3, called partially
```

Note: We can achieve the same behaviour using bind.The problem here is we have to alter this binding.

```
function sum(a, b) {
  return a + b;
}var addBy4 = sum.bind(this,2);
console.log(addBy4(4));
// => 6
```

Advanced Curry Implementation

Let's develop a function that takes any function and returns a curried version of the function. Here's the "advanced" curry implementation for multi-argument functions that we could use above.

The new curry may look complicated, but it's actually easy to understand. The result of curry(func) call is the wrapper curried that looks like this:

```
// func is the function to transform
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

When we run it, there are two if execution branches:

1. If passed args count is the same or more than the original function has in its definition (func.length) , then just pass the call to it using func.apply.
2. Otherwise, get a partial: we don't call func just yet. Instead, another wrapper is returned, that will re-apply curried providing previous arguments together with the new ones.

Then, if we call it, again, we'll get either a new partial (if not enough arguments) or, finally, the result.

Infinite Curry

Now we will build an infinite currying function i.e currying a function without knowing its arity. You can make the above curry technique by infinitely returning a function which takes one argument and accumulates the result till you say it's enough. Something like,

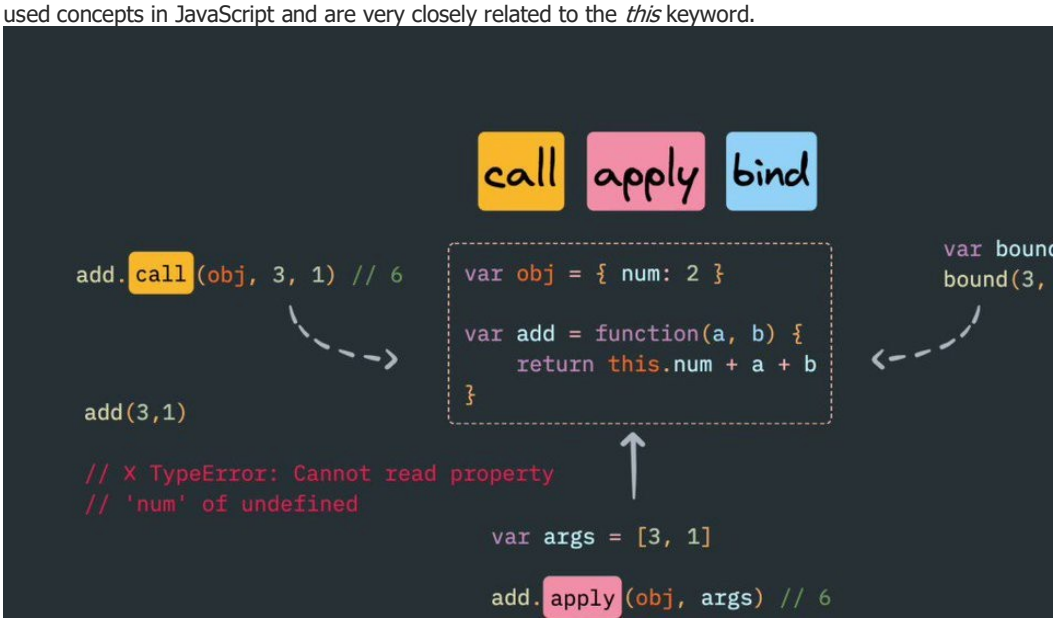
```
sum(a)(b)(c)...(n)()
//Output sum of n numbers -> sum of a + b + c...+ n;Example: sum(1)(2)(3)(4)(); //10
```

Advantages of Currying

- The main benefit of currying is when you need to use the same call with some of the same parameters a lot i.e it helps to avoid passing the same variable again and again. In these situations, currying becomes a good technique to use.
- Currying will make your code easier to refactor. Currying also creates a more declarative code base, e.g. it's easier to read the code and understand what it's doing.
- It helps to create a higher-order function.

Call, Apply and Bind Methods in JavaScript

We'll talk about the call, apply, and bind methods of the function prototype chain. They are some of the most important and often-used concepts in JavaScript and are very closely related to the *this* keyword.



Call method

The **call()** method calls a function with a given this value and arguments provided individually.

```
func.call([thisArg[, arg1, arg2, ...argN]])
```

Parameters:

thisArg — This is optional. It's the value to use as this when calling func.

arg1, arg2, ...argN— Optional arguments for the function.

Return value: The result of calling the function with the specified **this** value and arguments.

Description

- Using a call method we can do **function/method borrowing**, we can borrow functions from other objects and use it with the data of some other objects.

- With call(), you can write a method once and then inherit it in another object, without having to rewrite the method for the new object.
- In the call method, the first argument will be the reference or what we want 'this' to be pointing to.
- And the later arguments can be the arguments to that function. We can pass any number of arguments comma-separated.

Practical Applications

1) Using call to chain constructors for an object

Use call to chain constructors for an object (similar to Java).

In the following example, the constructor for the Person object is defined with two parameters: name and age. Two other functions, Engineer and Doctor, invoke Person, passing this, name, and age. Person initializes the properties name and age, both specialized functions define the category.

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}function Engineer(name, age) {
  Person.call(this, name, age);
  this.category = 'Engineer';
}function Doctor(name, age) {
  Person.call(this, name, age);
  this.category = 'Doctor';
}const engineer = new Engineer('Ayush', 28);
console.log(engineer);
//Engineer {name: "Ayush", age: 28, category: "Engineer"}const doctor = new Doctor('Anu', 30);
console.log(doctor);
//Doctor {name: "Anu", age: 30, category: "Doctor"}
```

2) Using call to invoke a function and specifying the context for 'this'

In the example below, when we call print, the value of this will be bound to object person.

```
const person = {
    name: 'Ayush',
    age: '28'
};
function print() {
  const reply = [this.name, 'is', this.age, 'years old.'].join(' ');
  console.log(reply);
}print.call(person);
//Ayush is 28 years old.
```

3) Using call to invoke a function and without specifying the first argument

In the example below, we invoke the display function without passing the first argument. If the first argument is not passed, the value of this is bound to the global object.

```
var name = 'Ayush';function display() {
  console.log('Your name: ', this.name);
}display.call();
// Your name:  AyushCaution: In strict mode, the value of this will be undefined.'use strict'var name = 'Ayush';function display() {
  console.log('Your name: ', this.name);}display.call();
//Uncaught TypeError: Cannot read property 'name' of undefined
```

Apply method

The **apply()** method calls a function with a given this value, and arguments provided as an array (or an array-like object).

```
func.apply(thisArg, [ argsArray])
```

Parameters:

thisArg The value of this provided for the call to func.

Note that this may not be the actual value seen by the method: if the method is a function in non-strict mode code, null and undefined will be replaced with the global object, and primitive values will be boxed. This argument is required.

argsArray Optional. An array-like object, specifying the arguments with which func should be called, or null or undefined if no arguments should be provided to the function.

Return value: The result of calling the function with the specified **this** value and arguments.

Description

- With apply, you can write a method once, and then inherit it in another object, without having to rewrite the method for the new object.
- apply is very similar to call(), except for the type of arguments it supports.
- With apply, you can also use an array literal, for example, func.apply(this, ['eat', 'bananas']), or an Array object, for example, func.apply(this, new Array('eat', 'bananas')).
- You can also use arguments for the argsArray parameter. arguments is a local variable of a function. It can be used for all unspecified arguments of the called object. Thus, you do not have to know the arguments of the called object when you use the apply method.

Practical Applications

1) Using apply to append an array to another

Use push to append an element to an array. And, because push accepts a variable number of arguments, you can also push multiple elements at once.

But, if you pass an array to push, it will actually add that array as a single element, instead of adding the elements individually. So you end up with an array inside an array.

What if that is not what you want? concat does have the desired behavior in this case, but it does not append to the *existing* array—it instead creates and returns a new array.
But you wanted to append to the existing array… So what now? Write a loop? Surely not?
apply to the rescue!

```
const arr = [1, 2, 3];
const numbers = [4, 5, 6];arr.push.apply(arr, numbers);
console.log(arr);
//[1, 2, 3, 4, 5, 6]
```

2) Using apply and built-in functions
Clever usage of apply allows you to use built-in functions for some tasks that would probably have otherwise been written by looping over the array values.
As an example, here are Math.max/Math.min, used to find out the maximum/minimum value in an array.

```
// Min/Max number in an array
const numbers = [9, 8, 1, 2, 3, 5, 6, 7];// Using Math.min/Math.max apply
let max = Math.max.apply(null, numbers);
console.log(max); //9// This about equal to Math.max(numbers[0], ...)
// or Math.max(5, 6, ...)let min = Math.min.apply(null, numbers);
console.log(min); //1
```

Bind method
The **bind()** method returns a new function, when invoked, has its this sets to a specific value.

```
func.bind(thisArg[, arg1[, arg2[, ...]]])
```

Parameters:
thisArgThe value to be passed as the this parameter to the target function func when the bound function is called. If no arguments are provided to bind , or if the thisArg is null or undefined, the this of the executing scope is treated as the thisArg for the new function.
arg1, arg2, ...argN Optional. Arguments to prepend to arguments provided to the bound function when invoking func.
Return value: A copy of the given function with the specified this value, and initial arguments (if provided).
Description

- The bind() function creates a new **bound function**, which is an *exotic function object* (a term from ECMAScript 2015) that wraps the original function object. Calling the bound function generally results in the execution of its wrapped function.
- The bind method looks exactly the same as the call method but the only difference is instead of directly calling this method here, the bind method binds this method with the object and returns a copy of that method. And will **return a function.**
- So there is a catch over here, it doesn't directly call that method rather it will return a method that can be called later. This is basically used to just bind and keep the copy of that method and use it later.

Practical Applications
1) Creating a bound function
The simplest use of bind() is to make a function that, no matter how it is called, is called with a particular this value.
A common mistake for new JavaScript programmers is to extract a method from an object, then to later call that function and expect it to use the original object as its this (e.g., by using the method in callback-based code).
Without special care, however, the original object is usually lost. Creating a bound function from the function, using the original object, neatly solves this problem:

```
this.x = 9;    // 'this' refers to global 'window' object here in a browser
const module = {
  x: 81,
  getX: function() { return this.x; }
};

module.getX();
//  returns 81

const retrieveX = module.getX;
retrieveX();
//  returns 9; the function gets invoked at the global scope

//  Create a new function with 'this' bound to module
//  New programmers might confuse the
//  global variable 'x' with module's property 'x'const boundGetX = retrieveX.bind(module);
boundGetX();
//  returns 81
```

2) Partially applied functions
The next simplest use of bind() is to make a function with pre-specified initial arguments.
These arguments (if any) follow the provided this value and are then inserted at the start of the arguments passed to the target function, followed by whatever arguments are passed bound function at the time it is called.

```
function addArguments(arg1, arg2) {
  return arg1 + arg2
}const result1 = addArguments(1, 2);
console.log(result1); //3// Create a function with a preset first argument.
const addThirtySeven = addArguments.bind(null, 37);
const result2 = addThirtySeven(5);console.log(result2);
//  37 + 5 = 42const result3 = addThirtySeven(5, 10);
```

```
console.log(result3);
//  37 + 5 = 42
//  (the second argument is ignored)
```

3) Using JavaScript **bind()** for function binding (with **setTimeout()**)

When you pass a method an object is to another function as a callback, the this is lost. For example:

```
let person = {
    firstName: 'John Doe',
    getName: function() {
        console.log(this.firstName);
    }
};setTimeout(person.getName, 1000); //undefinedlet f = person.getName;
setTimeout(f, 1000); //undefined
```

The this inside the setTimeout() function is set to the global object in non-strict mode and undefined in the strict mode.

Therefore, when the callback person.getName is invoked, the name does not exist in the global object, it is set to undefined.

To fix the issue, you can wrap the call to the person.getName method in an anonymous function, like this:

```
setTimeout(function () {
    person.getName();
}, 1000);
// "John Doe"
```

This works because it gets the person from the outer scope and then calls the method getName().

Or you can use the bind() method:

```
let f = person.getName.bind(person);
setTimeout(f, 1000);
// "John Doe"
```

4) Using **bind()** to borrow methods from a different object

The ability to borrow a method of an object without making a copy of that method and maintain it in two separate places is very powerful in JavaScript.

```
let runner = {
    name: 'Runner',
    run: function(speed) {
        console.log(this.name + ' runs at ' + speed + ' mph.');
    }
};let flyer = {
    name: 'Flyer',
    fly: function(speed) {
        console.log(this.name + ' flies at ' + speed + ' mph.');
    }
};let run = runner.run.bind(flyer, 20);
run();
// Flyer runs at 20 mph.
```

Polyfill for bind method

- First, you need to make it available for all the method so we're going to take help from the prototype of Function
- The customBind is a user-defined function you can name anything you want and, now it will be available for any function you define in your code like other functions toString(), toLocaleString(), etc.
- customBind function should return a function which when called should call a function on which customBind is applied.
- All the scope of the object should be passed as **context**.
- Fetch all arguments passed in customBind and returned function via **args1 and args2** and created a bigger array having both array elements and passed that as arguments to our binded function.

What is Cross-Origin Resource Sharing (CORS)?

Cross-Origin Resource Sharing (CORS) is a protocol that enables scripts running on a browser client to interact with resources from a different origin. This is useful because thanks to the same-origin policy followed by XMLHttpRequest and fetch, JavaScript can only make calls to URLs that live on the same origin as the location where the script is running.
For example, if a JavaScript app wishes to make an AJAX call to an API running on a different port, it would be blocked from doing so thanks to the same-origin policy. CORS error in the console will look like this:

```
Access to XMLHttpRequest at 'http://localhost:5000/global_config'   step1:1
from origin 'http://localhost:8080' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
```

CORS — Why Is It Needed?

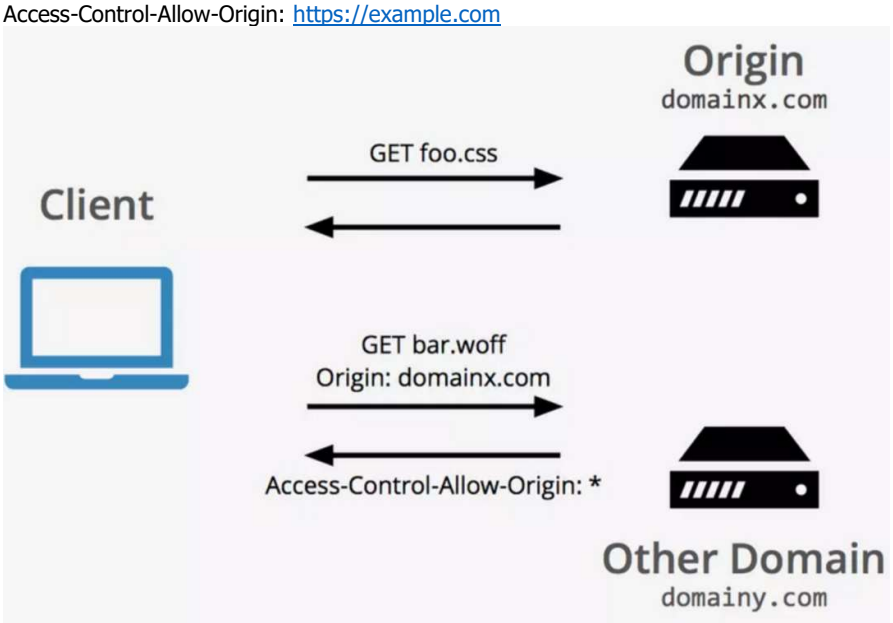
Most of the time, a script running in the user's browser would only ever need to access resources on the same origin (think about API calls to the same backend that served the JavaScript code in the first place). So the fact that JavaScript can't normally access resources on other origins is a good thing for security.
The same-origin policy fights one of the most common cyber-attacks out there: **Cross-Site Request Forgery CSRF**). In this maneuver, a malicious website attempts to take advantage of the browser's cookie storage system.
In this context, "other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

- a **different scheme** (HTTP or HTTPS)
- a **different domain**
- a **different subdomain**
- a **different port**

For example:
Long back, sharing resources from https://ayushv.medium.com with a different domain (google.com/api/getdata), different subdomain
(api.ayushv.medium.com), different port (ayushv.medium.com:5050), different protocols (http://ayushv.medium.com) were not allowed. After CORS became a standard, browsers do allow all these.
However, there are legitimate scenarios where cross-origin access is desirable or even necessary. For example, if you're running a React SPA that makes calls to an API backend running on a different domain. Web fonts also rely on CORS to work. CORS introduces a standard mechanism that can be used by all browsers for implementing cross-domain requests. The spec defines a set of headers that allow the browser and server to communicate about which requests are (and are not) allowed. CORS continues the spirit of the open web by bringing API access to all.

Identifying a CORS Response

When a server has been configured correctly to allow cross-origin resource sharing, some special headers will be included. Their presence can be used to determine that a request supports CORS. Web browsers can use these headers to determine whether or not an XMLHttpRequest call should continue or fail.
There are a few headers that can be set, but the primary one that determines who can access a resource is Access-Control-Allow-Origin. This header specifies which origins can access the resource. For example, to allow access from any origin, you can set this header as follows:
Access-Control-Allow-Origin: *
Or it can be narrowed down to a specific origin:
Access-Control-Allow-Origin: https://example.com



Understanding CORS Request Types

There are two types of CORS requests: **"simple" requests**, and **"preflight" requests**, and it's the *browser* that determines which is used. As the developer, you don't normally need to care about this when you are constructing requests to be sent to a server. However, you may see the different types of requests appear in your network log and, since it may have a performance impact on your application, it may benefit you to know *why* and *when* these requests are sent.
Let's have a look at what that means in more detail in the next couple of sections.

Simple requests (GET, POST, and HEAD)

The browser deems the request to be a "simple" request when the request itself meets a certain set of requirements:

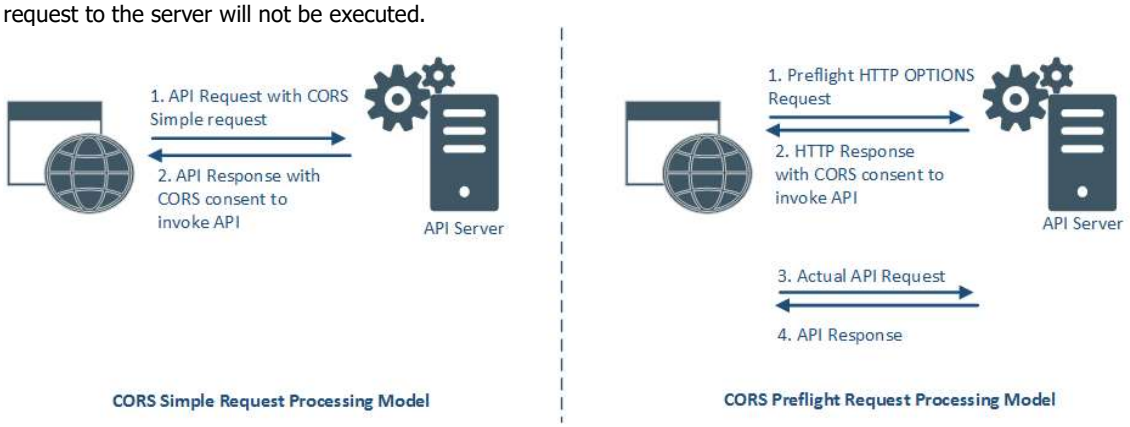
- One of these methods is used: GET, POST, or HEAD
- A CORS safe-listed header is used (Accept, Accept-Language and Content-Language)
- When using the Content-Type header, only the following values are allowed: application/x-www-form-urlencoded, multipart/form-data, or text/plain
- No event listeners are registered on any XMLHttpRequestUpload object
- No ReadableStream object is used in the request

The request is allowed to continue as normal if it meets these criteria and the Access-Control-Allow-Origin header is checked when the response is returned.

Preflight requests (OPTIONS)

If a request does not meet the criteria for a simple request, the browser will instead make an automatic preflight request using the OPTIONS method. This preflight request is made by some browsers as a **safety measure** to ensure that the request being done is trusted by the server. Meaning the server understands that the method, origin, and headers being sent on the request are safe to act upon.

This call is used to determine the exact CORS capabilities of the server, which is in turn used to determine whether or not the intended CORS protocol is understood. If the result of the OPTIONS call dictates that the request cannot be made, the actual request to the server will not be executed.



The preflight request sets the mode as OPTIONS and sets a couple of headers to describe the actual request that is to follow:

- Access-Control-Request-Method: The intended method of the request (e.g., GET or POST)
- Access-Control-Request-Headers: An indication of the custom headers that will be sent with the request
- Origin: The usual origin header that contains the script's current origin

An example of such a request might look like this:

```
# Request
curl -i -X OPTIONS localhost:3001/api/ping \
-H 'Access-Control-Request-Method: GET' \
-H 'Access-Control-Request-Headers: Content-Type, Accept' \
-H 'Origin: http://localhost:3000'
```

This request basically says "I would like to make a GET request with the Content-Type and Accept headers from http://localhost:3000 - is that possible?".

The server will include some Access-Control-* headers within the response to indicate whether the request that follows will be allowed or not. These include:

- Access-Control-Allow-Origin: The origin that is allowed to make the request, or * if a request can be made from any origin
- Access-Control-Allow-Methods: A comma-separated list of HTTP methods that are allowed
- Access-Control-Allow-Headers: A comma-separated list of the custom headers that are allowed to be sent. If you use custom headers (eg. x-authentication-token you need to return it in this ACA header response to OPTIONS call otherwise, the request will be blocked.
- Access-Control-Expose-Headers: Similarly, this response should contain a list of headers that will be present in the actual response to the call and should be made available to the client. All other headers will be restricted.
- Access-Control-Allow-Credentials: This header is only required to be present in the response if your server supports authentication via cookies. The only valid value for this case is true.
- Access-Control-Max-Age: The maximum duration that the response to the preflight request can be cached before another call is made

The response would then be examined by the browser to decide whether to continue with the request or to abandon it.

So a response to the earlier example might look like this:

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
Vary: Access-Control-Request-Headers
Access-Control-Allow-Headers: Content-Type, Accept
Content-Length: 0
Date: Fri, 05 Apr 2019 11:41:08 GMT
Connection: keep-alive
```

The Access-Control-Allow-Origin header, in this case, allows the request to be made from any origin, while the Access-Control-Allow-Methods header describes only the accepted HTTP methods. If a given HTTP method is not accepted, it will not appear in this list.

In this example, Access-Control-Allow-Headers echos back the headers that were asked for in the OPTIONS request. This indicates that all the requested headers are allowed to be sent. If for example, the server doesn't allow the Accept header, then that header would be omitted from the response and the browser would reject the call.

Handling CORS Error

The best way is to follow CORS standards and define a set of headers that allow the browser and server to communicate about which requests are (and are not) allowed. This is the best-case scenario — you should be able to implement the proper CORS response on the server which you're calling. If the API is using express for node you can use the simple cors package. If you want to make your site properly secure, consider using a whitelist for the Access-Control-Allow-Origin header.

For local development, there are some hacks**:**

- Plugins for chrome by which browser bypasses all CORS filters.

- One more way is to start chrome with a flag of **disable-web-security**. That particular web session will be without CORS. This is actually bypassing the web security that chrome has.

chrome --disable-web-security --user-data-dir

The JavaScript Event Propagation: Explained

What is Event Propagation?

Let's start with event propagation. This is the blanket term for both **event bubbling** and **event capturing**. Consider the typical markup to build a list of linked images, for a thumbnails gallery for example:

```
<ul>
   <li><a href="..."><img src="..." alt=""></a>
   <li><a href="..."><img src="..." alt=""></a>
   ...
   <li><a href="..."><img src="..." alt=""></a>
</ul>
```

A click on an image does not only generate a click event for the corresponding IMG element, but also for the parent A, for the grandfather LI and so on, going all the way up through all the element's ancestors, before terminating at the window object.

In DOM terminology, the image is the *event target*, the innermost element over which the click originated. The event target, plus its ancestors, from its parent up through to the window object, form a branch in the DOM tree. For example, in the image gallery, this branch will be composed of the nodes: IMG, A, LI, UL, BODY, HTML, document, window.

Note that window is not actually a DOM node but it implements the EventTarget interface, so, for simplicity, we are handling it like it was the parent node of the document object.

This branch is important because it is the path along which the events propagate (or flow). This propagation is the process of calling all the listeners for the given event type, attached to the nodes on the branch. Each listener will be called with an event object that gathers information relevant to the event (more on this later).

Remember that several listeners can be registered on a node for the same event type. When the propagation reaches one such node, listeners are invoked in the order of their registration.

It should also be noted that the branch determination is static, that is, it is established at the initial dispatch of the event. Tree modifications occurring during event processing will be ignored.

The propagation is bidirectional, from the window to the event target and back. This propagation can be divided into three phases:

1. From the window to the event target parent: this is the *capture phase*
2. The event target itself: this is the *target phase*
3. From the event target parent back to the window: the *bubble phase*

What differentiates these phases is the type of listeners that are called.

The Event Capture Phase

In this event capturing phase, the event is **first captured by the outermost element** and **propagated to the inner elements.** Also, **only the *capturer* listeners are called**, namely, those listeners that were registered using a value of true for the third parameter of addEventListener:

```
el.addEventListener('click', listener, true)
```

If this parameter is omitted, its **default value is false** and the listener is not a capturer.

So, during this phase, only the capturers found on the path from the window to the event target parent are called.

The Event Target Phase

In this phase, all the listeners registered on the event target will be invoked, regardless of the value of their capture flag.

The Event Bubbling Phase

In this event bubbling phase, the event is **first captured and handled by the innermost element** and then **propagated to outer elements**. Also, **only the non-capturers will be called**. That is, only the listeners registered with a value of false for the third parameter of addEventListener():

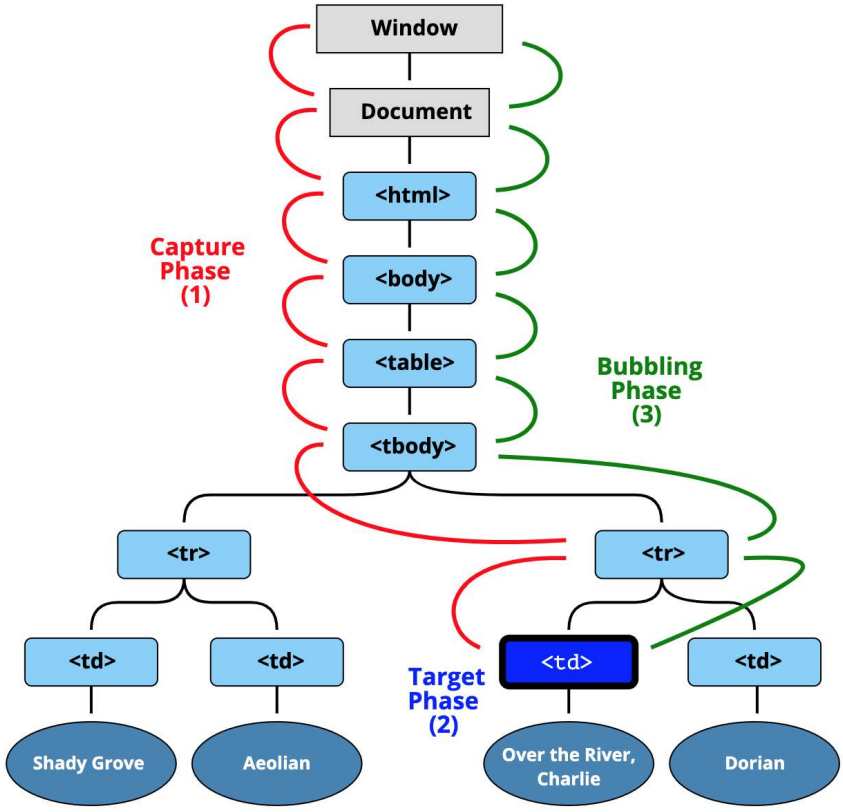
```
el.addEventListener('click', listener, false) // listener doesn't capture
el.addEventListener('click', listener) // listener doesn't capture
```

Note that while all events flow down to the event target with the capture phase, focus, blur, load and some others, don't bubble up. That is, their travel stops after the *target* phase.

Therefore, at the end of the propagation, each listener on the branch has been called exactly once.

Event bubbling does not take place for every kind of event. During propagation, it is possible for a listener to know if an event bubbles by reading the .bubbles Boolean property of the event object.

The three-event flow phase is illustrated in the following diagram



Event Delegation

- Event delegation is the technique of handling events on our web page in a better way. Event delegation is **based upon event bubbling**. So just because **event bubbling exists, event delegation also exists.**
- On our web page we have a number of events, and as an application grows events also keep on increasing. At some point in time, we have a lot of event handlers just hanging around on our web page, which is a critical performance bottleneck. So that is why we use event delegation.
- Suppose in e-commerce sites we have a lot of categories, like laptops, shoes, fashion, etc. So whenever we click on a laptop it takes to laptops and so on. Generally, we attach event listeners to each category. And that is not a good way to do it, we can have infinite categories.
- A better way to handle this is, instead of attaching event handlers to each and every child element or HTML element individually, we should rather attach event handlers to the parent of these elements.
- **Single event handler to parent.** Then on click of the child element, events will bubble out to their parents. The parent is listening to all the events happening in the child elements.

Accessing Propagation Information

I already mentioned the .bubbles property of the event object. There are a number of other properties provided by this object that are available to the listeners to access information relative to the propagation.

- **e.target** references the event target.
- **e.currentTarget** is the node on which the running listener was registered on. This is the same value of the listener invocation context, i.e, the value referenced by the this keyword.
- We can even find out the current phase with **e.eventPhase**. It is an integer that refers to one of the three Event constructor constants CAPTURING_PHASE, BUBBLING_PHASE and AT_TARGET.

Putting it into Practice

Let's see the above concepts into practice. In the following pen, there are five nested square boxes, named b0…b4. Initially, only the outer box b0 is visible; the inner ones will show when the mouse pointer hovers over them. When we click on a box, a log of the propagation flow is shown on the table to the right.

It is even possible to click outside the boxes: in this case, the event target will be the BODY or the HTML element, depending on the click screen location.

Stopping Propagation

The event propagation can be stopped in any listener by invoking the **stopPropagation** method of the event object. This means that all the listeners registered on the nodes on the propagation path that follow the current target will not be called. Instead, all the other remaining listeners attached on the current target will still receive the event.

In short, the stopPropagation method prevents propagation of any handlers at top-level DOM hierarchy to execute. It stops the click event from bubbling to the parent elements. It will allow other handlers on the same element to be executed, prevent handlers on parent elements from running. It does not stop the browser's default behavior.

For example- If you create a table containing <table>, <tr> and <td>. If you set three event handlers for <td> and in the second event handler you call stopPropagation. Then the other two event handlers will also run with this one.

Here we have prepended this new listener as a capturer to the list of callbacks registered on window:

```
window.addEventListener('click', e => { e.stopPropagation(); }, true);
window.addEventListener('click', listener('c1'), true);
window.addEventListener('click', listener('c2'), true);
```

```
window.addEventListener('click', listener('b1'));
window.addEventListener('click', listener('b2'));
```

This way, whatever box is clicked, the propagation halts early, reaching only the capturer listeners on window.

Stopping Immediate Propagation

As indicated by its name, **stopImmediatePropagation** throws the brakes on straight away, preventing even the siblings of the current listener from receiving the event.

In short, the stopImmediatePropagation method prevents both propagation of any other handlers and those at top level DOM hierarchy. It stops the other events which were assigned after this event.

```
stopImmediatePropagation = stopPropagation + (other event listeners removed)
```

For example- If you create a table containing <table>, <tr> and <td>. If you set three event handlers for <td> and in the second event handler you call stopImmediatePropagation. Then the first two event handlers will only run with this one. Thrid one will not run. It will depend on you where you used this, that handler will be the last one to be executed.

We can see this with a minimal change to the last example:

```
window.addEventListener('click', e => { e.stopImmediatePropagation(); }, true);
window.addEventListener('click', listener('c1'), true);
window.addEventListener('click', listener('c2'), true);
window.addEventListener('click', listener('b1'));
window.addEventListener('click', listener('b2'));
```

Now, nothing is output in the log table, neither the c1 and c2 window capturers rows, because the propagation stops after the execution of the new listener.

Event Cancellation

Some events are associated with a **default action that the browser executes at the end of the propagation**. For instance, the click on a link element or the click on a form submit button causes the browser to navigate to a new page, or submit the form respectively.

It is possible to avoid the execution of such default actions with the event cancellation, by calling yet another method of the event object, **e.preventDefault**, in a listener. But this method does not stop the event from bubbling up the DOM.

However, there is one more way to **return false**. It prevents the browser's default behavior, prevents the event from bubbling up the DOM, and immediately returns from any callback.

```
return false = e.preventDefault + stopPropagation + (stops callback execution)
```

JavaScript Promises

What is a Promise?

- A promise in JavaScript is similar to a promise in real life. When we make a promise in real life, it is a guarantee that we are going to do something in the future. Because promises can only be made for the future. A promise has two possible outcomes: it will either be kept when the time comes, or it won't. This is also the same for promises in JavaScript. When we define a promise in JavaScript, it will be resolved when the time comes, or it will get rejected.
- Promises are one of the ways we can deal with **asynchronous operations in JavaScript**. The promise is commonly defined as a **proxy for a value that will eventually become available**. The Promise is a way of defining a function in such a way that we can **synchronously control its flow.**
- "**Producing code**" is code that **can take some time**. "**Consuming code**" is a code that **must wait for the result**. A Promise is a JavaScript object that **links producing code and consuming code.** JavaScript Promise object contains both the producing code and calls to the consuming code.

Creating a promise: The Promise constructor

To create a promise in JavaScript, you use the Promise constructor:

```
const myPromise = new Promise((resolve, reject) => {
   // "Producing Code" (May take some time)
   let condition;

   if(condition is met) {
     resolve('Promise is resolved successfully.');//when successful
   } else {
     reject('Promise is rejected'); // when error
   }
});
```

The Promise constructor accepts a function as an argument. This function is called the executor.

The executor accepts two functions with the names, by convention, resolve() and reject().

When you call the new Promise(executor), the executor is called automatically.

Inside the executor, you manually call the resolve() function if the executor is completed successfully and invoke the reject() function in case of an error occurs.

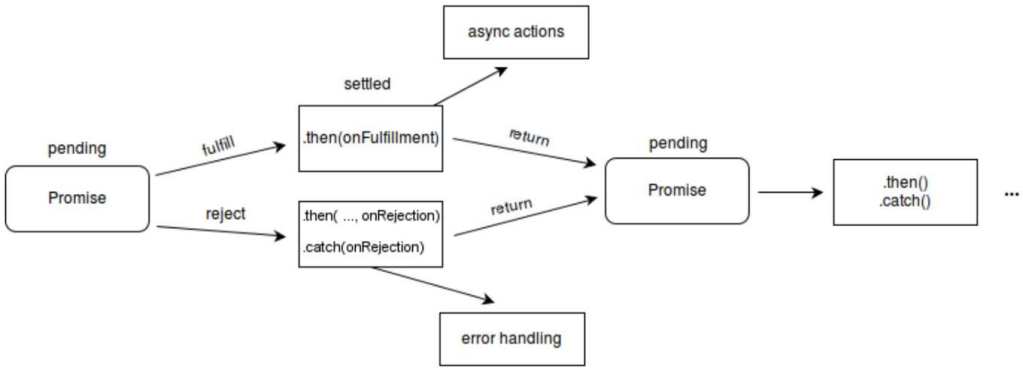
The Promise object supports two properties: **state** and **result**. A promise is in one of these states:

- ***pending***: initial state, neither fulfilled nor rejected.
- ***fulfilled***: meaning that the operation was completed successfully.
- ***rejected***: meaning that the operation failed.

```
▼Promise {<pending>} i
  ▶__proto__: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

```
    [[PromiseResult]]: "Promise is rejected successfully."
    [[PromiseState]]: "fulfilled"
  ▶__proto__: Promise
▲Promise {<fulfilled>: "Promise is rejected successfully."} i
▼Promise {<rejected>: "Promise is rejected"} i
  ▶__proto__: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: "Promise is rejected"
```

Once the promise reaches either fulfilled state or rejected state, it stays in that state and can't switch.
In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. It also cannot go back from the fulfilled state or rejected state to the pending state.
Once a new Promise object is created, it is in the pending state until it is resolved. To schedule a callback when the promise is either resolved or rejected, you call the methods of the Promise object: then(), catch(), and finally().



Consuming a Promise: then, catch, finally
1) The then() method
The then() method is used to schedule a callback to be executed when the promise is successfully resolved.
The then() method takes two callback functions:

```
myPromise.then(onFulfilled, onRejected);
```

The onFulfilled callback is called if the promise is fulfilled. The onRejected callback is called when the promise is rejected.
2) The catch() method
If you want to schedule a callback to be executed when the promise is rejected, you can use the catch() method of the Promise object.
Internally, the catch() method invokes the then(undefined, onRejected) method.

```
// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then((message) => {
   console.log(message);
}).catch((message) => {
   console.log(message);
});
```

3) The finally() method
Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected.
To remove duplicate and execute the createApp() whether the promise is fulfilled or rejected, you use the finally() method, like this:

```
myPromise
   .then(success => console.log(success))
   .catch(reason => console.log(reason))
   .finally(()=> createApp());
```

Promise Chaining
The instance method of the Promise object such as then(), catch()or finally() returns a separate promise object. Therefore, you can call the promise's instance method on the return Promise. The successively calling methods in this way are referred to as the promise chaining.
Consider the following example. First, create a new promise that resolves to the value 10 after 3 seconds:

```
let p = new Promise((resolve, reject) => {
   setTimeout(() => {
      resolve(10);
```

```
    }, 3000);
});
```

Note that we use the setTimeout() method to simulate an asynchronous operation. Then, invoke the then() method on the promise:

```
p.then((result) => {
    console.log(result);
    return result * 2;
});
```

The callback passed to the then() method executes once the promise is resolved. In the callback, we showed the result of the promise and returned a new value: result*2.

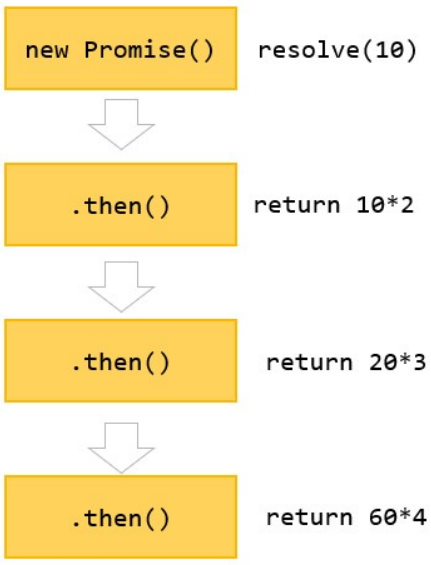
Because the then() method returns a new Promise whose value is resolved to the return value, you can call the then() method on the return Promise, like this:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});p.then((result) => {
    console.log(result);
    return result * 2;
}).then((result) => {
    console.log(result);
    return result * 3;
});Output:
10
20
```

In this example, the return value in the first then() method is passed to the second then() method. You can keep calling the then() method successively as follows:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});p.then((result) => {
    console.log(result); // 10
    return result * 2;
}).then((result) => {
    console.log(result); // 20
    return result * 3;
}).then((result) => {
    console.log(result); // 60
    return result * 4;
});Output:
10
20
60
```

The following picture illustrates the promise chaining:



Multiple handlers for a promise

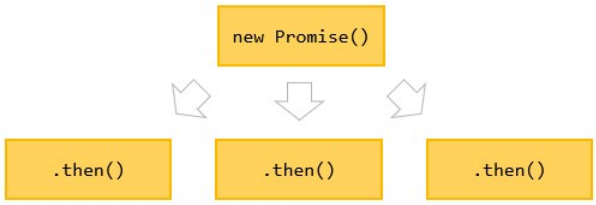
When you call the then() method multiple times on a promise, it is not promise chaining. For example:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
```

```
});p.then((result) => {
    console.log(result); // 10
    return result * 2;
})p.then((result) => {
    console.log(result); // 10
    return result * 3;
})p.then((result) => {
    console.log(result); // 10
    return result * 4;
});Output:
10
10
10
```

In this example, you have multiple handlers for one promise. These handlers have no relationships. They execute independently and also don't pass the result from one to another like the promise chaining above.

The following picture illustrates a promise that has multiple handlers:



In practice, you will rarely use multiple handlers for one promise.

Returning a Promise

When you return a value in the then() method, the then() method returns a new Promise that immediately resolves to the return value.

Also, you can return a new promise in the then() method, like this:

```
let p = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(10);
    }, 3 * 100);
});p.then((result) => {
    console.log(result);
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(result * 2);
        }, 3 * 1000);
    });
}).then((result) => {
    console.log(result);
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(result * 3);
        }, 3 * 1000);
    });
}).then(result => console.log(result));Output:
10
20
60
```

This example shows 10, 20, and 60 after every 3 seconds. This code pattern allows you to execute some tasks in sequence.

The following refactors the above example:

```
function generateNumber(num) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve(num);
        }, 3 * 1000);
    });
}generateNumber(10)
    .then(result => {
        console.log(result);
        return generateNumber(result * 2);
    })
    .then((result) => {
        console.log(result);
        return generateNumber(result * 3);
    })
    .then(result => console.log(result));
```

Promise chaining syntax

Sometimes, you have multiple asynchronous tasks that you want to execute in sequence. In addition, you need to pass the result of the previous step to the next one. In this case, you can use the following syntax:

```
step1()
   .then(result => step2(result))
   .then(result => step3(result))
   ...
```

If you need to pass the result from the previous task to the next one without passing the result, you use this syntax:

```
step1()
   .then(step2)
   .then(step3)
   ...
```

Suppose that you want to perform the following asynchronous operations in sequence:

1. Get the user from the database.
2. Get the services of the selected user.
3. Calculate the service cost from the user's services.

The following functions illustrate the three asynchronous operations:

```
function getUser(userId) {
   return new Promise((resolve, reject) => {
      console.log('Get the user from the database.');
      setTimeout(() => {
         resolve({
            userId: userId,
            username: 'admin'
         });
      }, 1000);
   })
}function getServices(user) {
   return new Promise((resolve, reject) => {
      console.log(`Get the services of ${user.username} from the API.`);
      setTimeout(() => {
         resolve(['Email', 'VPN', 'CDN']);
      }, 3 * 1000);
   });
}function getServiceCost(services) {
   return new Promise((resolve, reject) => {
      console.log(`Calculate the service cost of ${services}.`);
      setTimeout(() => {
         resolve(services.length * 100);
      }, 2 * 1000);
   });
}
```

The following uses the promises to serialize the sequences:

```
getUser(100)
   .then(getServices)
   .then(getServiceCost)
   .then(console.log);Output:
Get the user from the database.
Get the services of admin from the API.
Calculate the service cost of Email,VPN,CDN.
300
```

Benefits of Promises

1. Improves Code Readability
2. Better handling of asynchronous operations
3. Better flow of control definition in asynchronous logic
4. Better Error Handling

Applications

1. Promises are used for asynchronous handling of events.
2. Promises are used to handle asynchronous HTTP requests.

Polyfill for Promises

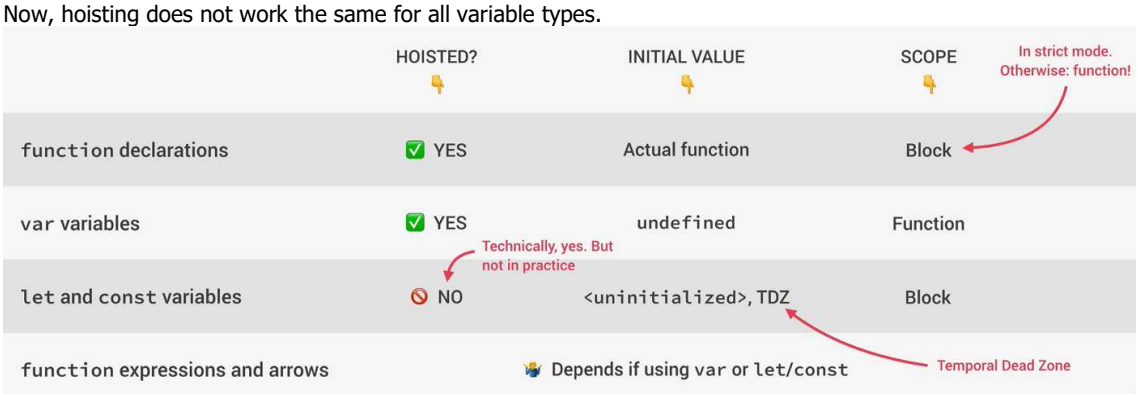
Let us implement our polyfill (say PromisePolyFill). From the above we know the following :

- The promise constructor function must accept a callback as an argument. We will call it as executor.
- It must return an object with at least two properties, then and catch
- then and catch are functions that again accept a callback and also they can be chained. Hence both must return a reference to this
- We need to store the reference to callback function passed to then and catch somewhere so that they should be executed at a later point of time, depending on the status of the executor. If executor resolved we must invoke the then callback. If executor rejects, we must invoke catch callback.
- We must invoke this executor function which will accept two arguments, resolve and reject.

Now we require three more additional variables :
fulfilled : Boolean indicating if the executor has been resolved or not
rejected : Boolean indicating if the executor has been rejected or not
called: Boolean indicating if the then or catchcallback has been called or not.

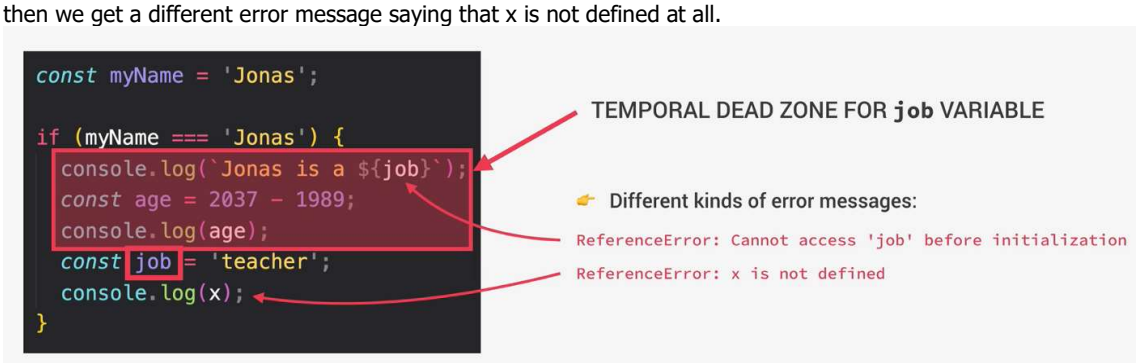
JavaScript Hoisting and Temporal Dead Zone(TDZ)- var, let, const and function declarations

Variable Environment in JavaScript
An Execution Context (EC) always contains three parts — a **variable environment**, **the scope chain** in the current context, and **this** keyword. In this post, we'll be discussing the variable environment in JavaScript.
In JavaScript, we have a mechanism called **hoisting**. And hoisting basically makes some types of variables accessible/usable in the code before they are actually declared in the code.
Variables are magically **lifted** or moved to the **top of their scope** for example, to the top of a function.
How hoisting really works?
Behind the scenes, the code is basically scanned for variable declarations before it is executed. So this happens during the so-called creation phase of the execution context.
Then for each variable that is found in the code, a **new property** is created in a **variable environment object**.
How does hoisting work for variable types?
Now, hoisting does not work the same for all variable types.



- **Function declarations** are actually hoisted and the initial value in the variable environment is set to the **actual function**. So in practice, what this means is that we can use function declarations before they are actually declared in the code, again, because they are stored in the variable environment object, even before the code starts executing. Function declarations are **block-scoped** for strict mode. So if you're using a sloppy mode, which you shouldn't, then functions have functioned scope.
- Variables declared with **var** are also **hoisted**, but hoisting works in a different way here. So unlike functions, when we try to access a var variable before it's declared in a code, we don't get the declared value but we get **undefined**.
- **Let** and **Const** variables are **not hoisted** i.e. technically they are actually hoisted but their value is basically set to **uninitialized**. So there is no value to work with at all. And so in practice, it is as if hoisting was not happening at all. Instead, we say that these variables are placed in a so-called **Temporal Dead Zone** or **TDZ** which makes it so that we can't access the variables between the beginning of the scope and to place where the variables are declared. So as a consequence, if we attempt to use a let or const variable before it's declared, we get an error.
  Let and const are **block-scoped**. So they exist only in the block in which they were created.
- In Function expressions and arrow functions, it depends if they were created using var or const or let. These functions are simply variables. And so they behave the exact same way as variables in regard to hoisting.

Temporal Dead Zone: Let and Const
Basically the region of the scope in which the variable is defined, but can't be used in any way. So it is as if the variable didn't even exist. Now, if we still tried to access the variable while in the TDZ like we actually do in the first line of this if block, then we get a reference error telling us that we can't access the job before initialization.
However, if we tried to access a variable that was actually never even created, like in the last line here where we want to log x, then we get a different error message saying that x is not defined at all.



Basically each and every let and const variable get their own Temporal Dead Zone that starts at the beginning of the scope until the line where it is defined.
Why Temporal Dead Zone?
Makes it way **easier** to **avoid** and **catch errors**. Because using a variable that is set to undefined before it is actually declared can cause serious bugs that might be hard to find. So accessing variables before the declaration is bad practice and should be avoided. And the best way to avoid it is by simply getting an error when we attempt to do so.
Make **const variables actually work** the way they are supposed to. So as we know, we can't reassign const variables. So it will not be possible to set them to undefined first and then assign their real value later. Const should never be reassigned. And so it's only assigned when execution actually reaches the declaration. And that makes it impossible to use the variable before.
Why Hoisting?
Using functions before the actual declarations.

Essential for some programming techniques, such as **mutual recursion**. Some people also think that it makes code a lot **more readable**.
The fact that it also works for var declarations is because that was the only way hoisting could be implemented at the time. So the hoisting of var variables is basically just a byproduct of hoisting functions.

Hoisting and TDZ in practice

```
Variables:
console.log(me);   //undefined
console.log(job);  //Uncaught ReferenceError: Cannot access 'job'    before initialization
console.log(year); //Uncaught ReferenceError: Cannot access 'year'    before initialization
var me = 'Ayush';
let job = 'developer';
const year = 2021;
Functions:
console.log(addDec(2,3)) //5
console.log(addExpr(2,3)) //Uncaught ReferenceError: Cannot access 'addExpr' before initialization
console.log(addArrow(2,3)) //Uncaught ReferenceError: Cannot access 'addArrow' before initialization
function addDec(a,b){
   return a + b;
}
const addExpr = function(a,b){
   return a + b;
}
const addArrow = (a,b) => a + b;
console.log(addExpr1(2,3)) //Uncaught TypeError: addExpr1 is not a function (set to undefined - undefined(2,3))
var addExpr1 = function(a,b){
   return a + b;
}
Example:
if(!numProducts) deleteShoppingCart();
var numProducts = 10;
function deleteShoppingCart(){
   console.log('All products deleted')
}
Output - All products deleted (numProducts is undefined - falsy value)
```

Web Storage APIs — Local Storage and Session Storage

Web Storage APIs are used by developers to **store data into the browser** to **improve the user experience** and **performance** of the web apps. The data here refers to the **key-value pair of strings**. Now storing this data can be done by two mechanisms: either by using the **sessionStorage** API and the **localStorage** API.

Session storage

- The data is stored in the browser's memory for that **specific session**. The session here means until we close the browser window or tab.
- The storage limit of session storage is very high when compared to cookies. The cookies which can generally store around 4000 bytes of data and session storage can store at least **5 MB of data** or even more than that depending on the device and browser.

Note:

1. Data survives a page refresh.
2.While restoring the tab (Ctrl+Shift+T), sessionStorage is maintained in the Chrome, Firefox but not in the Safari browser . It is browser dependent feature while restoring the tab.

Local storage

- It is almost the same as the session storage but the only difference is that it **does not have an expiration time.** So even if we close the browser window/tabs and restart the system and come back again anytime the data persists till we manually delete it. That makes it unique and very useful.
- localStorage can also store about 5 MB of data.

Note:

Data stored on normal browsing sessions will not be available when you open a browser in private browsing or in Incognito mode. It is also browser dependent feature.

Now let's discuss common features to both local and session storage:
1) Both are used for storing the data on the **client-side**. the data is never transferred to the server while making a network request.
2) Both are stored on the **window object** of the browser
— **window.localStorage** or **localStorage** and **window.sessionStorage** or **sessionStorage.**
3) Both are Object type — **typeof localStorage** and **typeof sessionStorage** is **"object"**
4) Both storage objects provide the same methods and properties:

- setItem(key, value) – store key/value pair.
- getItem(key) – get the value by key.
- removeItem(key) – remove the key with its value.
- clear() – delete everything.
- key(index) – get the key on a given position.
- length – the number of stored items.

As we can see, it's like a `Map` collection (`setItem/getItem/removeItem`), but also allows access by index with `key(index)`.
5) Due to security reasons both follow the **same-origin policy**. Same-Origin refers to the **same protocol (HTTP or HTTPS), same host or domain(abc.com), and the same port(8080 or wherever the app is hosted).** Session storage is different even for the document with the same-origin policy open in different tabs, so the same web page open in two different tabs ***cannot*** share the same session storage.

Suppose we store localStorage data in https://ayushv.medium.com/. Can we access the data on:

- https://ayushv.medium.com/data.php — Yes, we are of the same origin.
- http://ayushv.medium.com/— No, we are on different protocols.
- https://blog.ayushv.medium.com/— No, we are on a different domain. (subdomain added)
- https://ayushv.medium.com/:8080- No, we are on a different port.

6) Both local and session storage are also scoped by ***browser vendors***. So storage data saved by IE cannot be read by Chrome or FF.
We can see all the local and session storage data for each site under the **Application** *tab in chrome developer tools.*
Examples and Practical usage:

```
1. localStorage.setItem(key, value)localStorage.setItem("name", "Ayush")
localStorage.setItem("name", "Verma")
//it will override2. localStorage.getItem(key)localStorage.getItem("name") 3.
localStorage.removeItem(key)localStorage.removeItem("name")4. localStorage.clear()
5. localStorage.key(0)
6. localStorage.length
```

Generally, Web Storage APIs are used to store objects. **JSON.stringify() and JSON.parse() are used to store and get objects.**
Web Storage APIs are used a lot by many big companies to store some less relevant user-specific data into their browsers. Some companies even use it to optimize the performance of the web page speed as accessing local storage is faster than making a request to the server and getting the data. Companies like Flipkart and Paytm use localStorage for keeping a lot of data.

- Flipkart — Some information such as browsed products, navigation menu, autosuggest history all this user-specific data are being stored in the local storage.
- Paytm — They store a lot of data such as the recent searches for flights, recent cities we select, and even some session data into localStorage.

Use Storage objects for Browser Caching — We can cache some application data for later usage/run the app offline. For example, imagine you need to load currency data for all the countries. It can be saved in LocalStorage and not make HTTP requests every time you needed the list.
However, If you want it to load per session, you can use SessionStorage.
Use sessionStorage for modal — Suppose we have a modal on the page that displays some long-form to enter details. Many times it happens that, the user enters some data in the form, and by mistake clicks outside the modal and the modal gets closed. Because of which he loses all the entered data.
To avoid the loss of data, we can use sessionStorage so the stored data in modal will only be applicable to the current tab.
Tip — create own generic functions something like setLocalStorageData() and getLocalStorageData()
How Performance gets affected — Large complex applications can be slow due to the synchronous behaviour of storage objects. If you need more storage and performance, you can use IndexedDB or **cache API**.
Vulnerability to Cross-Site Scripting (XSS) Attacks
XSS attacks can be used to get data from storage objects and add malicious scripts to the data stored. Both are vulnerable to XSS Attacks. Therefore avoid storing sensitive data in browser storage.
It is not recommended to save sensitive data as Username/Password, Credit card info, JWT tokens, API keys, Personal info, and Session ids.
How to Mitigate Security Attacks?

- Do not use the same origin for multiple web applications. Instead, use subdomains since otherwise, the storage will be shared with all.
- Validate, encode and escape data read from browser storage.
- Encrypt data before saving.

Recursion in JavaScript

What is Recursion?

A **process** (a function in our case) that **calls itself**.

Why do we need to know Recursion?

It's EVERYWHERE!

- Methods using recursion internally — JSON.parse/JSON.stringify, document.getElementById
- DOM traversal algorithms and Object traversal
- A cleaner alternative to iteration.

Call Stack *— First let's talk about functions. In almost all program languages there is a built data structure that manages what happens when functions are invoked. Its named as Call Stack in JavaScript.*
It's a ***stack*** *data structure. Any time a function is invoked it is placed (****pushed****) on the top of the call stack. When JavaScript sees the* ***return*** *keyword or when the function ends, the complier will remove(****pop****).*
We are used to functions being pushed on the call stack and popped off when they are done. When we write recursive functions, we keep pushing new functions (in fact the same function)onto the call stack!

How recursive functions work?

Two essential parts of any recursive functions — **Base case** and **different input.** Invoke the **same** function with a different input until you reach your base case — the condition where the recursion ends.

Examples:

1. **Countdown** — print numbers to the console from whatever number we pass till 1.

Without recursion:
```
function countDown(num){
  for(var i = num; i > 0; i--) {
    console.log(i);
  }
  console.log("All done!");
}countDown(5);
```
With recursion:
```
function countDown(num){
  if (num <= 0) {
    console.log("All done!");
    return;
  }
  console.log(num);
  num--;
  countDown(num);
}countDown(3);//print 3
//countDown(2)
//print 2
//countDown(1)
//print 1
//countDown(0) - base case
//print "All done"
```

2. **SumRange** — print sum to the console from whatever number we pass till 1

```
function sumRange(num){
  if (num === 1) return 1;
  return num + sumRange(num-1);
}
sumRange(3);//return 3 + sumRange(2)
//          return  2 + sumRange(1)
//                    return 1 - base case
// 3 + 2 + 1 = 6
```

3. **Factorial-**print multiplication to the console from whatever number we pass till 1.

Without recursion:
```
function factorial(num){
  let total = 1;
  for(var i = num; i > 0; i--) {
      total *= i;
  }
  return total;
}factorial(5); //120
```
With recursion:
```
function factorial(num){
  if(num === 1) return 1;
  return num * factorial(num-1);
}factorial(3); //6
```

Common Recursion Pitfalls

- No base case
- Forgetting to return or returning the wrong thing!
- Maximum call stack size exceeded — stack overflow!

Helper Method Recursion:

A design pattern that's commonly used with recursion.

```
function outer(input){
  var outerScopedVariable = [];
  function helper(helperInput){
    //modify the outerScopedVariable
     helper(helperInput--)
  }
```

```
    helper(input)
    return outerScopedVariable;
}//Two functions - outer(main) and helper (recursive)
//Commonly done when we need to compile an array or list of data.
```

Example —

Collect all the odd values in an array.

```
function collectOddValues(arr){
    let result = [];

    function helper(helperInput){
        if(helperInput.length === 0){
            return;
        }
        if(helperInput[0] % 2 !== 0){
            result.push(helperInput[0]);
        }
        helper(helperInput.slice(1))
    }
    helper(arr) return result;
}collectOddValues([1,2,2,4,4,5,6,7,8]) //(3) [1, 5, 7]
```

Pure Recursion:

```
function collectOddValues(arr){
    let newArr = [];

    if(arr.length === 0){
        return newArr;
    }
    if(arr[0] % 2 !== 0){
        newArr.push(arr[0]);
    }
    newArr = newArr.concat(collectOddValues(arr.slice(1)));    return newArr;
}
collectOddValues([1,2,3,4,5]) //(3) [1, 3, 5]//[1].concat(collectOddValues([2,3,4,5]));
//          [].concat(collectOddValues([3,4,5]));
//                  [3].concat(collectOddValues([4,5]));
//                    [].concat(collectOddValues([5]));
//                      [5].concat(collectOddValues([]));
//                          []
//[1,3,5]
```

Pure Recursion Tips

- For arrays, use methods like **slice**, the **spread** operator, and **concat** that makes copies of arrays so we do not mutate them.
- Remember strings are immutable, so we will need to use methods like **slice**, **substr**, or **substring** to make copies of strings.
- To make copies of object use **Object.assign**, or the **spread** operator.

Recursion examples:

1. **power** — Write a function called power which accepts a base and an exponent. The function should return the power of the base to the exponent. This function should mimic the functionality of **Math.pow()** — do not worry about negative bases and exponents.

```
function power(base, exponent){
    if(exponent === 0) return 1;
    return base * power(base,exponent-1)
}
power(2,0) // 1
power(2,2) // 4
power(2,4) // 16
```

2. **productOfArray** — Write a function called productOfArray which takes in an array of numbers and returns the product of them all.

```
function productOfArray(arr){
    if(arr.length === 0) return 1;
    return arr[0] * productOfArray(arr.slice(1))
}

productOfArray([1,2,3]) // 6
productOfArray([1,2,3,10]) // 60
```

3. **Fibonacci** — Write a recursive function called fib which accepts a number and returns the nth number in the Fibonacci sequence. Recall that the Fibonacci sequence is the sequence of whole numbers 1, 1, 2, 3, 5, 8, … which starts with 1 and 1, and where every number thereafter is equal to the sum of the previous two numbers.

```
function fib(n){
    if (n <= 2) return 1;
    return fib(n-1) + fib(n-2);
}fib(4) //3
```

```
fib(6) //8
fib(10) //55//n = 4
//fib(3) + fib(2)
//[fib(2)+ fib(1)] + 1
//1 + 1 + 1
//3//n = 6
//fib(5) + fib(4)
//[fib(4)+ fib(3)] + [fib(3)+ fib(2)]
//[fib(3)+ fib(2) + fib(2)+ fib(1)] + [fib(2)+ fib(1) + 1]//[fib(2) + fib(1)+ fib(2) + fib(2)+ fib(1)] + [fib(2)+ fib(1) + 1]
//[1 + 1 + 1 + 1 + 1] + [1 + 1 + 1]
//8
```

4. **reverse** — Write a recursive function called reverse which accepts a string and returns a new string in reverse.

```
function reverse(str){
    if(str.length === 1) return str[0];
    return str[str.length - 1] + reverse(str.slice(0, str.length-1))
}// reverse('awesome') // 'emosewa'
// reverse('rithmschool') // 'loohcsmhtir'
```

5. **flatten** — Write a recursive function called flatten which accepts an array of arrays and returns a new array with all values flattened.

Helper method:

```
function flatten(arr){
  let resultArr = [];  function inner(arr){
    for(let i = 0; i< arr.length; i++){
        if(Array.isArray(arr[i])){
            inner(arr[i]);
        }
        else{
            resultArr.push(arr[i])
        }
    }
  }
  inner(arr);
  return resultArr;
}flatten([1, 2, [3, 4, [5, [6, 7, [[[8]]]]]]]) //[1, 2, 3, 4, 5, 6, 7, 8]
```

Pure recursion:

```
function flatten(arr){
  let resultArr = [];
  for(let i = 0; i < arr.length; i++){
      if(Array.isArray(arr[i])){
          resultArr = resultArr.concat(flatten(arr[i]))
      }
      else{
          resultArr.push(arr[i])
      }
  }
  return resultArr;
}// flatten([1, 2, 3, [4, 5] ]) // [1, 2, 3, 4, 5]
// flatten([1, [2, [3, 4], [[5]]]]) // [1, 2, 3, 4, 5]
// flatten([[1],[2],[3]]) // [1,2,3]
// flatten([[[[1], [[[2]]], [[[[[[[3]]]]]]]]]) // [1,2,3]
```

Object-Oriented Programming (OOP) patterns in JavaScript

What Is Object-Oriented Programming?

Object-Oriented Programming(OOP) is a programming paradigm based on the **concept of objects**. And paradigm simply means the style of the code,

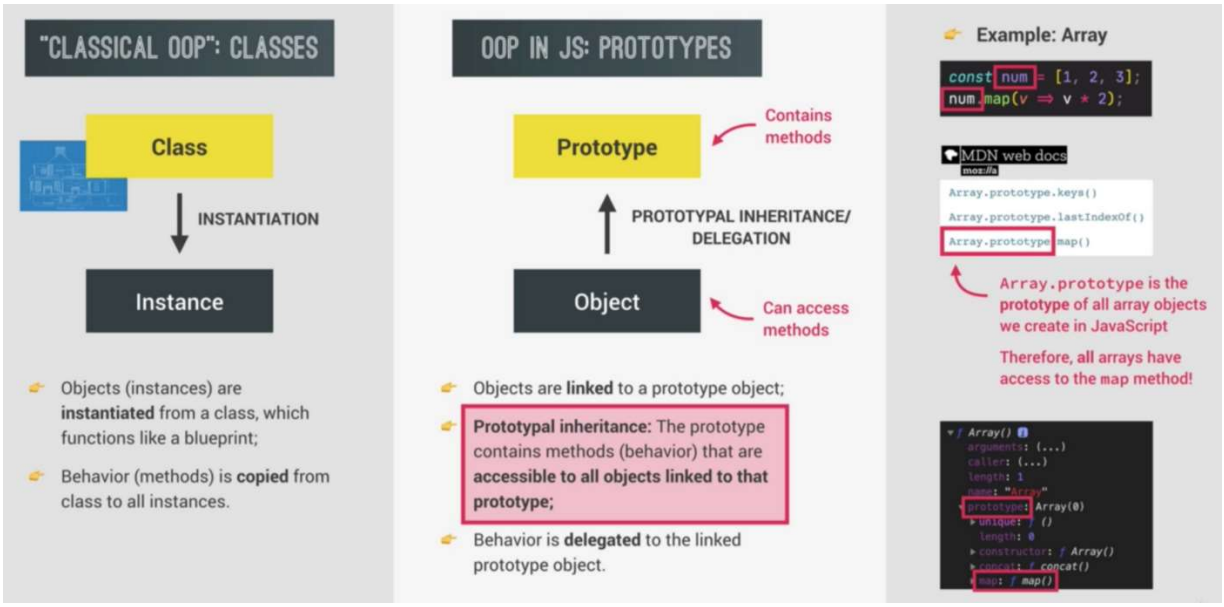
how we write and organize code.

We use objects to **model** (describe) aspects of the real world, like a user or a to-do list item, or even more abstract features like an HTML component or some kind of data structure.

Objects may contain data (properties) and also code (methods). By using objects, we **pack all the data and the corresponding behavior into one big block**.

In OOP objects are **self-contained** pieces/blocks of code like small applications on their own. Objects are **building blocks** of applications and **interact** with one another. Interactions happen through a **public interface** (API). This interface is basically a bunch of methods that a code outside of the objects can access and use to communicate with the object.

OOP was developed with the goal of **organizing** code, to make it **more flexible** and **easier to maintain**.



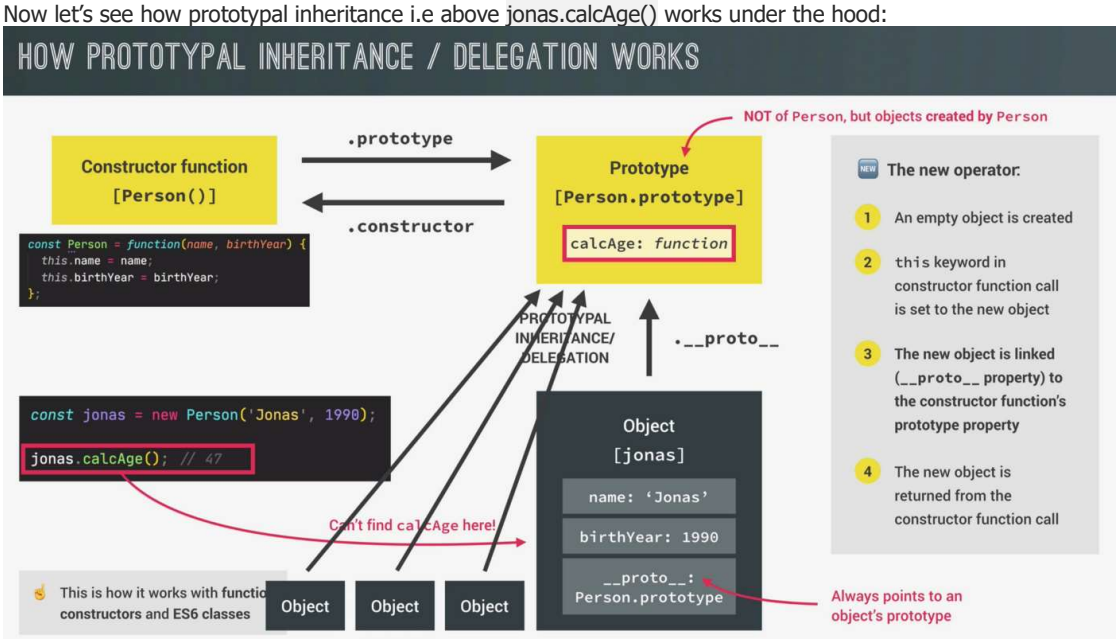
Three ways of implementing Prototypal Inheritance

1. Constructor function
   - Technique to create objects from a function.
   - This is how built-in objects like Arrays, Maps, or Sets are actually implemented.
2. ES6 Classes
   - Modern alternative to constructor function syntax.
   - "Syntactic sugar": behind the scenes, ES6 classes work **exactly** like constructor functions.
   - ES6 classes do NOT behave like classes in "classical OOP".
3. Object.create()
   - The easiest and most straightforward way of linking an object to a prototype object.

Constructor Functions and the 'new' Operator

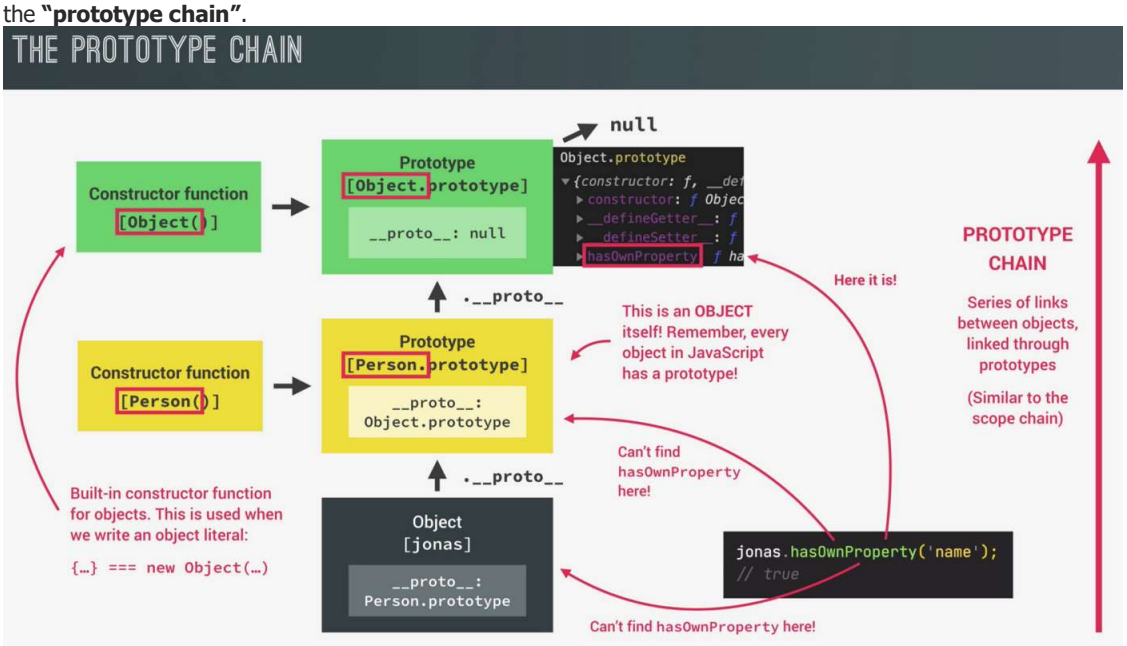
- In JavaScript, there's really no difference between a "regular" function and a constructor function. They're actually all the same. But as a convention, functions that are meant to be used as constructors are generally first letter **capitalized**.
- In JavaScript, all functions are also objects, which means that they can have properties. And as it so happens, they all have a property called `**prototype**`, which is also an object.
- Any time you create a function, it will *automatically* have a property called **prototype**, which will be initialized to an empty object.
- prototype is a property of a Function object. It is the prototype of objects constructed by that function. It is used to build __proto__ when you create an object with new.
- __proto__ is an internal property of an object, pointing to its prototype. It is the actual object that is used in the lookup chain to resolve methods.

instanceof —operator tests to see if the prototype property of a constructor appears anywhere in the prototype chain of an object. The return value is a boolean value.
isPrototypeOf — method checks if an object exists in another object's prototype chain.
hasOwnProperty — method returns a boolean indicating whether the object has the specified property as its own property.
Now let's see how prototypal inheritance i.e above jonas.calcAge() works under the hood:



Prototype Chain

JavaScript uses an inheritance model called "**differential inheritance**". What that means is that methods aren't copied from parent to child. Instead, children have an "invisible link" back to their parent object. More commonly, it's referred to as the **"prototype chain"**.




```
console.log(jonas.__proto__);
//{ species: "Homo sapiens", calcAge: f, constructor: f}console.log(jonas.__proto__.__proto__);
//{ constructor: f, _defineGetter_: f, _defineSetter_: f, hasOwnProperty: f, ....}console.log(jonas.__proto__.__proto__.___proto__);
//nullconsole.dir(Person.prototype.constructor);
//constructor property points back at personconst arr = [1,2,3,4,5];
console.log(arr.__proto__); //contains all the built-in array methods
console.log(arr.__proto__ === Array.prototype); //trueconsole.log(arr.__proto__.__proto__); //all object properties
console.log(arr.__proto__.__proto__.__proto__); //nullconst h1 = document.querySelector("h1");
console.dir(h1); //object
console.log(x=> x*2); //object
```

We can add new methods to this prototype and all the arrays will then inherit it.

```
Array.prototype.unique = function() {
   return [...new Set(this)];
};let arr = [1,2,3,2,1,4,5];
console.log(arr.unique()) // [1,2,3,4,5]
```

ES6 Classes

Classes in JavaScript do not work like traditional classes in other languages like Java or C++ instead, classes in JavaScript are just syntactic sugar of Constructor Functions. They still implement prototypal inheritance behind the scenes, but the syntax makes more sense to people coming from other programming languages. And that was basically the goal of adding classes to JavaScript. Some important points to note for classes:

- Classes are **not hoisted** even if they are class declarations.
- Just like functions classes are also **first-class citizens,** which means that we can pass them into functions and also return them from functions. That is because classes are really just a special kind of function behind the scenes.
- The body of a class is **always executed in strict mode**. Classes are executed in strict mode even if we didn't activate it for our entire script, all the code that is in the class will be executed in strict mode.

Constructor functions are not like old or deprecated syntax. It's 100% fine to keep using them. This is more a question of personal preference.

Setters and Getters

Every object in JavaScript can have setter and getter properties. And we call these special properties **assessor properties**, while the more normal properties are called **data properties**. Getters and setters are basically functions that get and set a value.

For objects:

```
const account = {
 owner: 'Ayush',
 movements: [100,50, 300, 40],

 // read something as a property
 get latest(){
   return this.movements.slice(-1).pop();
 },

 set latest(mov){
   this.movements.push(mov);
 }
}console.log(account.latest); //40
account.set  = 60;
console.log(account.movements); //[100, 50, 300, 40]
```

Classes do also have getters and setters, and they do indeed work in the exact same way. Getter is indeed just like any other regular method that we set on the prototype. Setters and getters can actually be very useful for data validation.

For classes:class PersonCl{
constructor(firstName, birthYear){
this.firstName = firstName;
this.birthYear = birthYear;
}

// Method will be added to .prototype property
get age (){
return 2020 - this.birthYear;
}
}const ayush = new PersonCl('Ayush','1992');
console.log(ayush.age); //28

Static Methods

Now a good example to actually understand the static method is the built-in Array.from() method. The Array.from() method converts any array-like structure to a real array.

Array.from(document.querySelectorAll('h1'));
//[h1]

This from() method here is really a method that is attached to the Array constructor. So we could not use the from method on an Array because this from method is not attached to the prototype property of the constructor. Therefore all the arrays do not inherit this method.

[1,2,3].from();
//Uncaught TypeError: [1,2,3].from is not a function

Another static method is Number.parseFloat() and it's static on the Number constructor. So it's not available on numbers, but only on this very constructor.

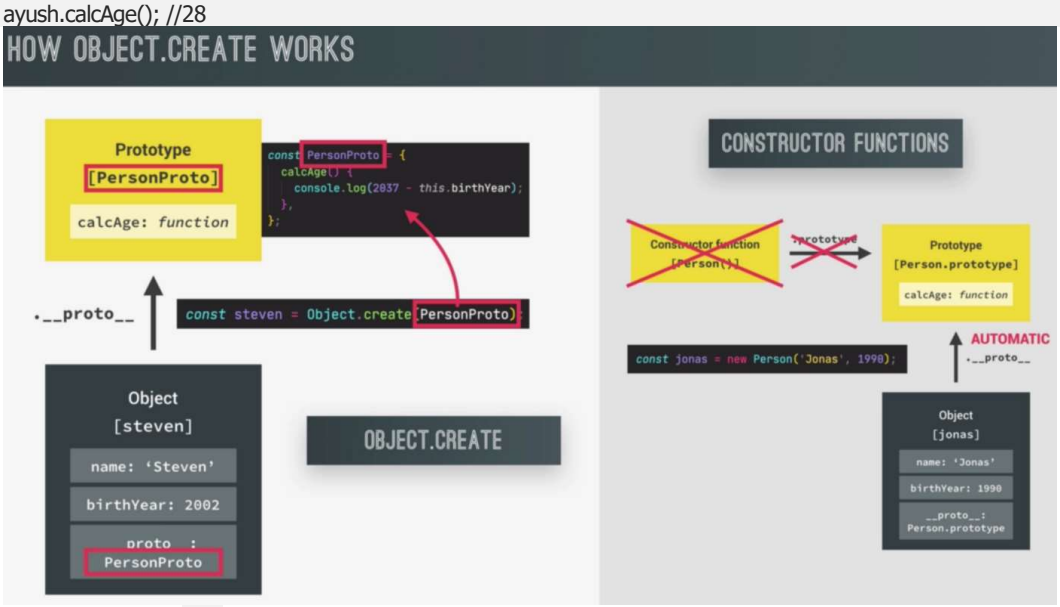
Number.parseFloat(12);
//12

These are some good examples that we understand what a static method is. And we usually use these kinds of helpers, that should be related to a certain constructor. Now let's do that for both or constructor function and also for the class.

Object.create()

The third way is to use a function called Object.create(), which works in a pretty different way than constructor functions and classes work. Now, with Object.create(), there is still the idea of prototypal inheritance. However, there are no prototype properties involved. Nor any constructor functions or the new operator. So instead, we can use Object.create() to essentially manually set the prototype of an object to any other object that we want.

const PersonProto = {
calcAge (){
console.log(2020 - this.birthYear);
}
}const ayush = Object.create(PersonProto);
console.log(ayush); //{}
//empty object and in the prototype we have calcAge.ayush.name = 'Ayush';
ayush.birthYear = 1992;
ayush.calcAge(); //28



When we use the newoperator in constructor functions or classes, it automatically sets the prototype of the instances to the constructors, prototype property. This happens automatically.

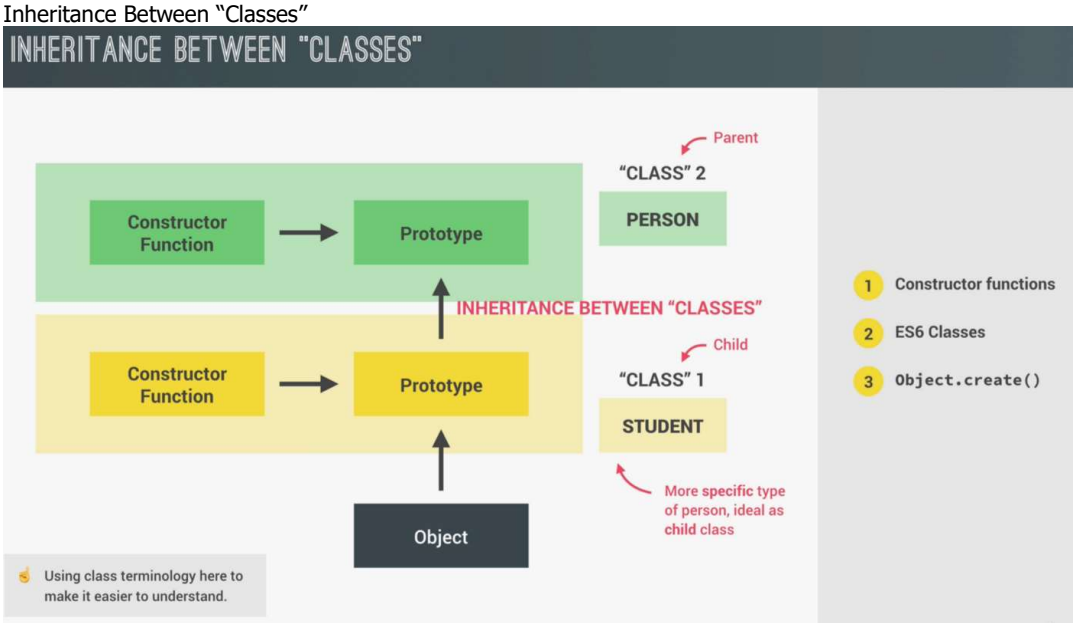
On the other hand, with Object.create(), we can set the prototype of objects manually to any object that we want. And in this case, we manually set the prototype of the steven object to the PersonProto object.

Now the two objects are effectively linked through the proto property, just like before. Now looking at properties, or methods in a prototype chain works just like it worked in function constructors, or classes. And so the prototype chain, in fact, looks exactly the

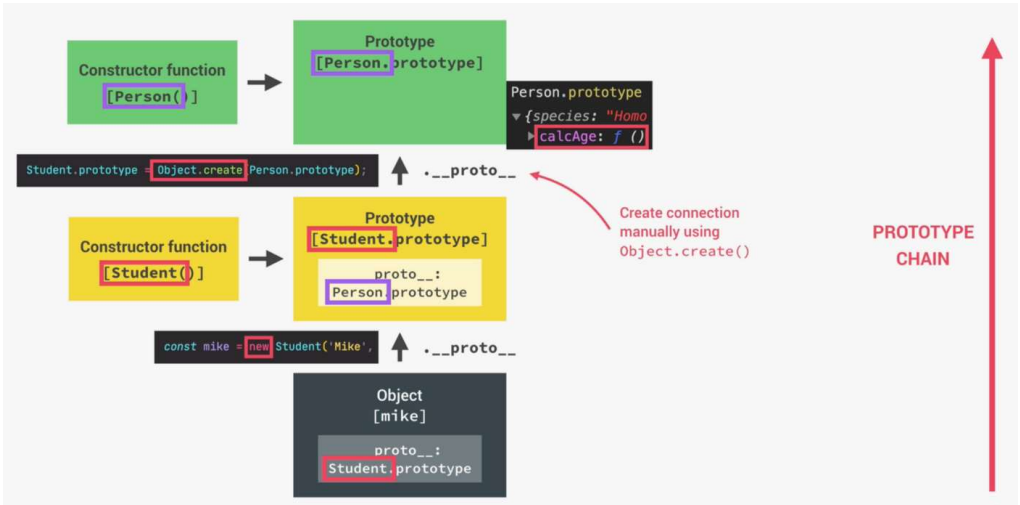
same here. The big difference is that we didn't need any constructor function, and also no prototype property at all, to achieve the exact same thing. This is actually the least used way of implementing prototypal inheritance.

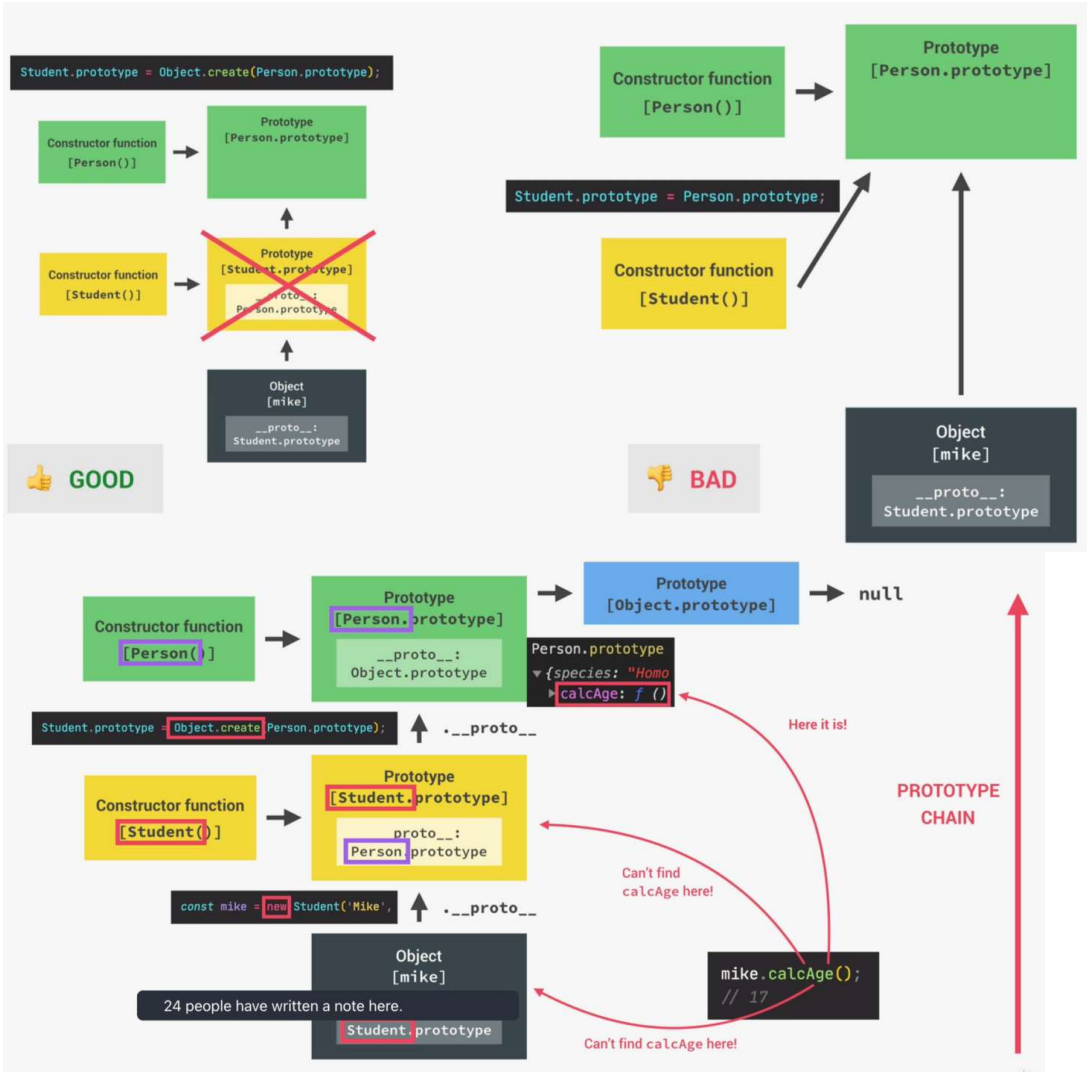
```
const PersonProto = {
 calcAge (){
  console.log(2020 - this.birthYear);
 },

 init(firstName, birthYear){
    this.firstName = firstName;
    this.birthYear = birthYear;
 }
}const ayush = Object.create(PersonProto);console.log(ayush.__proto__ === PersonProto); //trueconst anu =
Object.create(PersonProto);
anu.init('Anu', 1990);
anu.calcAge(); //30
```

Inheritance Between "Classes"



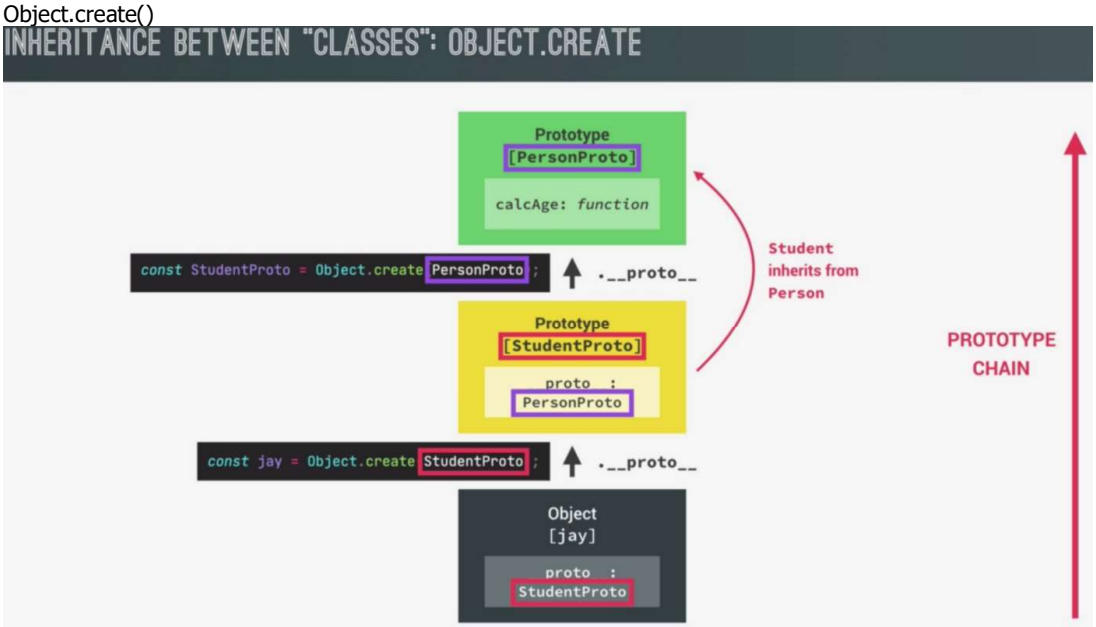
Constructor Functions





ES6 Classes

Object.create()

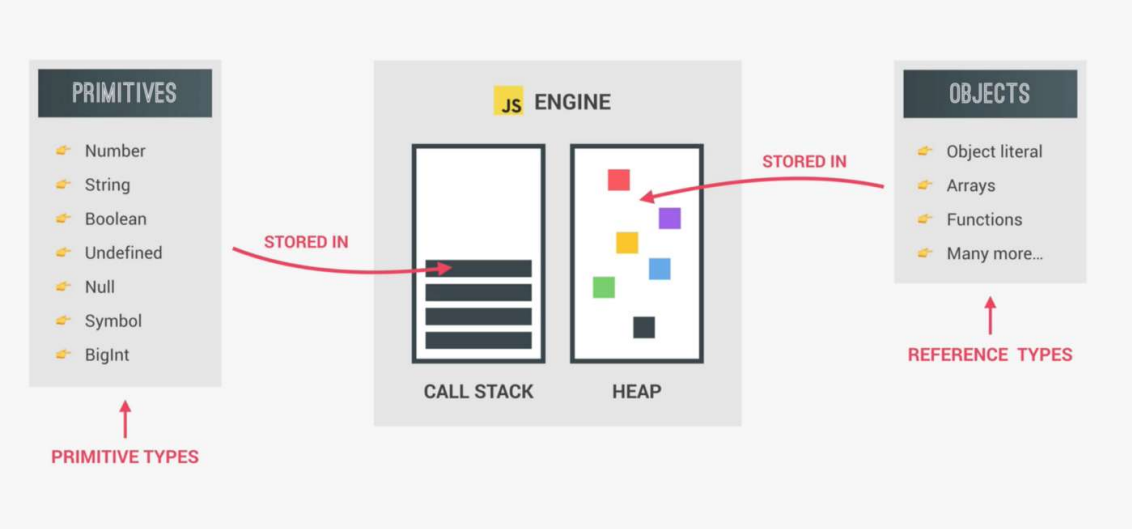


In this version, we don't even worry about constructors anymore, and also not about prototype properties, and not about the new operator. It's really just objects linked to other objects. And it's all really simple and beautiful.

In this technique with Object.create(), we are, in fact, not faking classes. All we are doing is simply linking objects together, where some objects then serve as the prototype of other objects. Although ES6 classes and constructor functions are actually way more used in the real world.

Shallow Copy and Deep Copy in JavaScript

Data types in JavaScript can be divided into two main categories— ***primitive data types*** and ***reference data types.***



Primitive data types: The value assigned to the variable of primitive data type is **tightly coupled**. That means, whenever you create a copy of a variable of primitive data type, the value is copied to a new memory location to which the new variable is pointing to. When you make a copy, it will be a **real copy.**

Reference data types: Whereas, for reference data type it stores the address of the memory location where the object is stored. There are two types of copying reference data types namely **shallow copy** and **deep copy**.

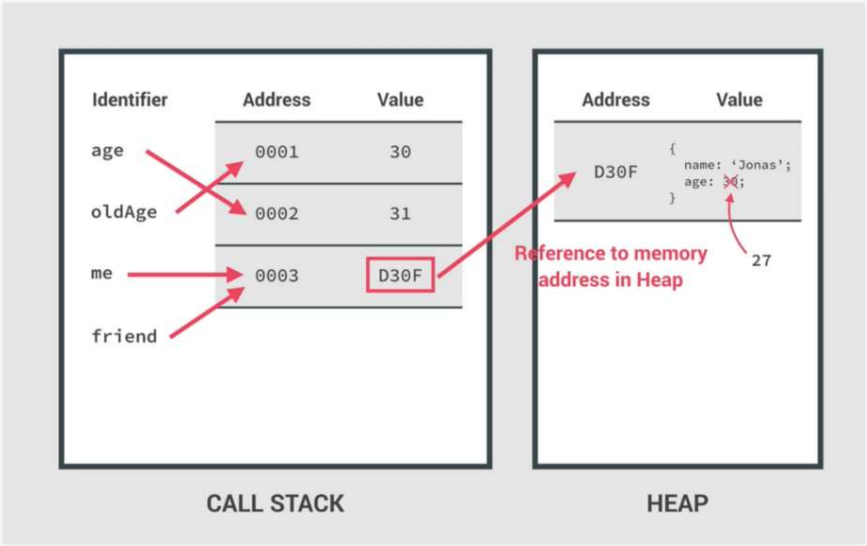
Primitive values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

Reference values example:

```
const me = {
  name: 'Jonas'
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

No problem, because we're NOT changing the *value* at address 0003!



Shallow copy

A shallow copy simply **points to the reference address** of the original collection structure (object or array) which holds the value in the new variable i.e., only the collection structure is copied, not the element.

When the field value is a reference type it just copies the reference address, no new object will be created. The referenced objects are thus *shared.*

let originalObject= {name: "apple", type: "fruit"};
*let clonedObject= originalObject;clonedObject.name = 'orange';**Output**:*
clonedObject= {name: "orange", type: "fruit"}
originalObject= {name: "orange", type: "fruit"}

Shallow copy is simple and typically cheap, as they can be usually implemented by simply copying the reference address. Similarly, this can be observed for arrays.

let originalArr = [1,2,3];
let clonedArr = originalArr;
*clonedArr.push(4);**Output**:*
clonedArr = [1,2,3,4]
originalArr= [1,2,3,4]

However, there are workarounds to copying objects without reference.

1.**Spread operator** (...) is a convenient way to make a shallow copy of an array or object —when there is no nesting, it works great.

It is useful for creating new instances of arrays that do not behave unexpectedly due to old references. The spread operator is thus useful for adding to an array in React state.

Example 1:
let originalObject = {name: "apple"};
let clonedObject = {...originalObject};
*clonedObject.name = 'orange';**Output**:*
clonedObject= {name: "orange"}
*originalObject= {name: "apple"}**Example 2**:*
let originalObject = {name: "apple", price: {chennai: 120}};
let clonedObject = {...originalObject};*clonedObject.name ="orange"; // will not reflect in originalObject*
*clonedObject.price.chennai = "100"; //will reflect in originalObject also**Output**:*
clonedObject = {name: "orange", price: {chennai: 100}}
originalObject = {name: "apple", price: {chennai: 100}}

2.**Object.assign()** method copies enumerable properties from a source object to a target object. Further, this can be used only if the object/array contains primitive type values.

Example 1:
let originalObject = {name: "apple"};
let clonedObject = Object.assign({}, originalObject);
*clonedObject.name = 'orange';**Output**:*
clonedObject= {name: "orange"}
*originalObject= {name: "apple"}**Example 2**:*
let originalObject = {name: "apple", price: {chennai: 120}};
let clonedObject = Object.assign({}, originalObject);clonedObject.name ="orange"; // will not reflect in originalObject
*clonedObject.price.chennai = "100"; //will reflect in originalObject also**Output**:*
clonedObject = {name: "orange", price: {chennai: 100}}
originalObject = {name: "apple", price: {chennai: 100}}

3.**Slice()** — For arrays specifically, using the built-in .slice() method works the same as the spread operator — creating a shallow copy of one level:

Example 1:
let originalArr = [1,2,3,4,5]
let clonedArr = originalArr.slice();
clonedArr.push(6);***Output**:*
originalArr = [1, 2, 3, 4, 5]
clonedArr = [1, 2, 3, 4, 5, 6]

4.**Array.from()** — Another method to copy a JavaScript array is using Array.from()which will also make a shallow copy.

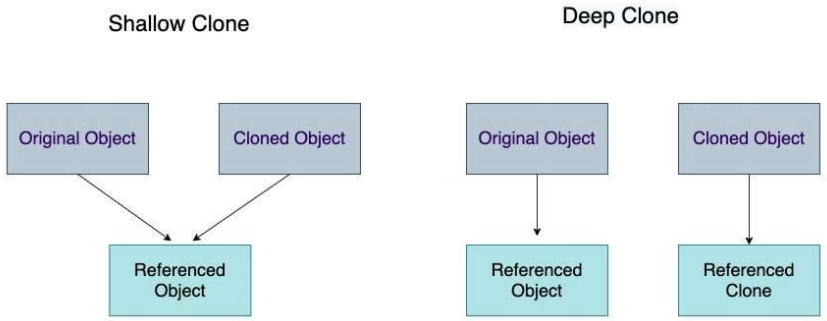
Example 1:
let originalArr = [1,2,3,4,5]
let clonedArr = Array.from(originalArr);
clonedArr.push(6);***Output**:*
originalArr = [1, 2, 3, 4, 5]
clonedArr = [1, 2, 3, 4, 5, 6]

These methods will **copy up to the first level**. If the object contains any nested object or array then the internally copied reference type value will refer to its memory address i.e shallow copies will work unexpectedly, because nested objects are not actually cloned. To overcome this we need to iterate and copy till the last level, but this approach is expensive and not recommended. For deeply-nested objects, a deep copy will be needed.

Deep copy

It simply creates a **duplicate** of all the properties of the *source object* into *the target object.* In other words, both the primitive type and reference type properties will be allocated to **new memory locations**.

In this way, if the *source object* becomes non-existent, the *target object* still exists in the memory.



The correct term to use would be **cloning**, where you know that they both are totally the same, but yet different (i.e. stored as two different entities in the memory space).

It is used to copy JavaScript objects to a new variable NOT by reference.

The strict equality operator (===) shows that the nested references are the same for shallow copy and different for deep copy.

```
const nestedArr = [["1"] ,["2"] ,["3"]];
const nestedCopyWithSpread = [...nestedArr];console.log(nestedArr[0] === nestedCopyWithSpread[0]);
// true -- Shallow copy (same reference)const nestedCopyWithJSON = JSON.parse(JSON.stringify(nestedArr));console.log(nestedArr[0]
=== nestedCopyWithJSON[0]);
// false -- Deep copy (different references)
```

There are many methods of making a deep copy (or deep clone):

1.**JSON.parse/stringify**— If you do not use Date, functions, undefined, Infinity, [NaN], RegExps, Maps, Sets, Blobs, FileLists, ImageDatas, sparse Arrays, Typed Arrays or other complex types within your object, a very simple one-liner to deep clone an object is:

var clonedObject = JSON.parse(JSON.stringify(originalObject));Example:*let originalObject = {name: "apple", price: {chennai: 120}};*
let clonedObject = JSON.parse(JSON.stringify(originalObject));*clonedObject.name ="orange";*
*clonedObject.price.chennai = "100";***Output***:*
clonedObject = {name: "orange", price: {chennai: 100}}
originalObject = {name: "apple", price: {chennai: 120}}

2.**Lodash —** It is a JavaScript library that provides multiple utility functions and one of the most commonly used functions of the Lodash library is the **cloneDeep()** method. This method helps in the deep cloning of an object and also clones the non-serializable properties which were a limitation in the JSON.stringify() approach.

```
import _ from "lodash" // Import the entire lodash library//import { clone, cloneDeep } from "lodash" // Alternatively: Import just the clone methods from lodashconst nestedArr = [["1"] ,["2"] ,["3"]];const shallowCopyWithLodashClone = _.clone(nestedArr)
console.log(nestedArr[0] === shallowCopyWithLodashClone[0]);
// true -- Shallow copy (same reference)const deepCopyWithLodashCloneDeep = _.cloneDeep(nestedArray)
console.log(nestedArr[0] === deepCopyWithLodashCloneDeep[0]);
// false -- Deep copy (different reference)
```

3.**Ramda —** The functional programming library Ramda includes the R.clone() method, which makes a deep copy of an object or array.

```
import R from "ramda" // Import the entire ramda library// import { clone } from "ramda" // Alternatively: Import just the clone methods from ramdaconst nestedArr = [["1"] ,["2"] ,["3"]];const deepCopyWithRamdaClone = R.clone(nestedArray)
console.log(nestedArr[0] === deepCopyWithRamdaClone[0]);
// false -- Deep copy (different reference)
```

4.**Custom Function —** It is pretty easy to write a recursive JavaScript function that will make a deep copy of nested objects or arrays.

5.**rdfc —** For the best performance, the library rfdc (Really Fast Deep Clone) will deep copy about 400% faster than lodash's _.cloneDeep.Using rfdc is pretty straight-forward, much like the other libraries:

```
const clone = require('rfdc')() // Returns the deep copy function
clone({a: 37, b: {c: 3700}}) // {a: 37, b: {c: 3700}}
```

- rdfc clones all JSON types: Object, Array, Number, String, null
- With additional support for: Date (copied), undefined (copied), Function (referenced), AsyncFunction (referenced), GeneratorFunction (referenced), arguments (copied to a normal object)
- All other types have output values that match the output of JSON.parse(JSON.stringify(o)).
- The rfdc library supports all types and also supports circular references with an optional flag that decreases performance by about 25%.
- Circular references will break the other deep copy algorithms discussed. Such a library would be useful if you are dealing with a **large**, complex object such as one loaded from JSON files from 3MB-15MB in size.

Performance of Copy Algorithms

Of the various copy algorithms, the shallow copies are the fastest, followed by deep copies using a custom function or rfdc.Ranked from best to worst*:-*

1. Reassignment "=" (string arrays, number arrays — only)
2. Slice (string arrays, number arrays — only)
3. Custom function: for-loop or recursive copy /rfdc
4. JSON.parse (string arrays, number arrays, object arrays — only)
5. Lodash's _.cloneDeep