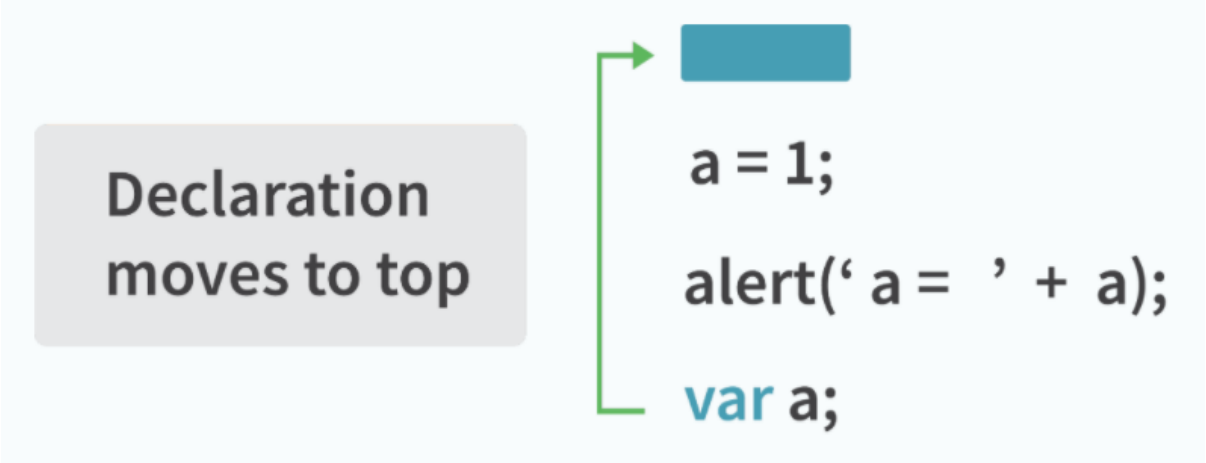
https://www.interviewbit.com/javascript-interview-questions/

Magyarázza el a hoisting-et javascriptben.

A hoisting a javascript alapértelmezett viselkedése, ahol az összes változó- és függvénydeklaráció a tetejére kerül. Ez azt jelenti, hogy függetlenül attól, hogy a változók és függvények hol vannak deklarálva, a hatókör tetején helyezkednek el. A hatókör lehet helyi és globális.



Example 1:

```
hoistedVariable = 3;
console.log(hoistedVariable); // outputs 3 even when the variable is declared after it is initialized
var hoistedVariable;
```

Example 2:

```
hoistedFunction();   // Outputs " Hello world! " even when the function is declared after calling

function hoistedFunction(){
  console.log(" Hello world! ");
}
```

Example 3:

```
// Hoisting takes place in the local scope as well
function doSomething(){
  x = 33;
  console.log(x);
  var x;
}
```

doSomething(); // Outputs 33 since the local variable “x” is hoisted inside the local scope

****Note - Variable initializations are not hoisted, only variable declarations are hoisted:**

```
var x;
console.log(x); // Outputs "undefined" since the initialization of "x" is not hoisted
x = 23;
```

**Note - To avoid hoisting, you can run javascript in strict mode by using “use strict” on top of the code:

```
"use strict";
x = 23; // Gives an error since 'x' is not declared
var x;
```

Mi az NaN tulajdonság a JavaScriptben?

A NaN tulajdonság a „Not-a-Number”

A NaN típusa szám (number).

Az isNaN () függvény az adott értéket számtípussá konvertálja, majd egyenlővé teszi a NaN-t.

```
isNaN("Hello")   // Returns true
isNaN(345)    // Returns false
isNaN('1')   // Returns false, since '1' is converted to Number type which results in 0 ( a number)
isNaN(true) // Returns false, since true converted to Number type results in 1 ( a number)
isNaN(false) // Returns false
isNaN(undefined) // Returns true
```

Magyarázza el a higher order függvényeket a javascriptben.

Azokat a függvényeket, amelyek más függvényeken operálnak akár argumentumként átadva a másik függvényen, akár visszaadva őket, magasabb rendű (higher order) függvényeknek nevezzük.

```
unction higherOrder(fn) {
  fn();
}

higherOrder(function() { console.log("Hello world") });
```

```
function higherOrder2() {
  return function() {
    return "Do something";
  }
}

var x = higherOrder2();
x()    // Returns "Do something"
```

Magyarázza a call (), apply () és a bind () metódusokat.

call()

Ez a függvény meghív egy metódust (függvényt) a tulajdonos objektum megadásával.

1. példa:

```
function sayHello(){
  return "Hello " + this.name;
}

var obj = {name: "Sandy"};

sayHello.call(obj);

// Returns "Hello Sandy"
```

A call () metódus lehetővé teszi egy objektum számára egy másik objektum metódusának (függvényének) használatát.

2. példa:

```
var person = {
  age: 23,
  getAge: function(){
   return this.age;
  }
}

var person2 = {age: 54};
person.getAge.call(person2);

// Returns 54
```

call() accepts arguments:

```
function saySomething(message){
  return this.name + " is " + message;
}

var person4 = {name: "John"};

saySomething.call(person4, "awesome");
// Returns "John is awesome"
```

apply()

Az Apply metódus hasonló a call () metódushoz. Az egyetlen különbség az, hogy

A call () metódus külön veszi az argumentumokat, míg az Apply () metódus tömbként veszi fel az argumentumokat.

```
function saySomething(message){
  return this.name + " is " + message;
}

var person4 = {name: "John"};

saySomething.apply(person4, ["awesome"]);
```

bind ()

Ez a módszer egy új függvényt ad vissza, ahol az „this” kulcsszó értéke a paraméterként megadott tulajdonos objektumhoz lesz kötve.

Példa érvekkel:

```
var bikeDetails = {
    displayDetails: function(registrationNumber,brandName){
    return this.name+ " , "+ "bike details: "+ registrationNumber + " , " + brandName;
  }
}

var person1 = {name: "Vivek"};

var detailsOfPerson1 = bikeDetails.displayDetails.bind(person1, "TS0122", "Bullet");

// Binds the displayDetails function to the person1 object


detailsOfPerson1();
// Returns Vivek, bike details: TS0452, Thunderbird
```

Mi a currying a JavaScriptben?

A currying egy olyan fejlett technika, amellyel az n argumentumok függvényét egy vagy kevesebb argumentum n függvényévé alakíthatjuk át.

Példa egy curried függvényre:

```
function add (a) {
  return function(b){
   return a + b;
  }
}

add(3)(4)
```

Például, ha van egy f (a, b) függvényünk, akkor a curry után a függvény átalakul f (a) (b) -re.

A curry technikával nem változtatjuk meg a függvény funkcionalitását, hanem megváltoztatjuk a hívás módját.

Lássuk a curry-t működés közben:

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

function currying(fn){
  return function(a){
```

```
    return function(b){
      return fn(a,b);
    }
  }
}

var curriedMultiply = currying(multiply);

multiply(4, 3); // Returns 12

curriedMultiply(4)(3); // Also returns 12
```

Magyarázza el a hatókört (scope) és a hatókörláncot (scope chain) javascriptben.

A JS hatóköre meghatározza a változók és függvények hozzáférhetőségét a kód különböző részein.

Általánosságban elmondható, hogy a hatókör tudatja velünk a kód egy adott részén, hogy melyek azok a változók és függvények, amelyekhez hozzáférhetünk, vagy amelyekhez nem férhetünk hozzá.

Háromféle hatókör létezik a JS-ben:

- Global Scope
- Local or Function Scope
- Block Scope

1.) Globális hatókör (global scope)

A globális névtérben deklarált változók vagy függvények globális hatókörűek, ami azt jelenti, hogy az összes globális hatókörű változó és függvény a kód belsejéből bárhonnan elérhető

```
var globalVariable = "Hello world";

function sendMessage(){
  return globalVariable; // can access globalVariable since it's written in global space
}

function sendMessage2(){
  return sendMessage(); // Can access sendMessage function since it's written in global space
}

sendMessage2();   // Returns “Hello world”
```

2.) Függvény hatóköre (function scope)

A függvényen belül deklarált bármely változónak vagy függvénynek lokális / függvény scope-ja van, ami azt jelenti, hogy a függvényen belül deklarált összes változó és függvény a függvényen belül érhető el, és nem azon kívül.

```
function awesomeFunction(){
  var a = 2;

  var multiplyBy2 = function(){
    console.log(a*2); // Can access variable "a" since a and multiplyBy2 both are written inside the same
function
```

```
  }
}
console.log(a); // Throws reference error since a is written in local scope and cannot be accessed outside

multiplyBy2(); // Throws reference error since multiplyBy2 is written in local scope
```

3.) Blokk hatókör (Block scope)

A blokk hatóköre a let és const használatával deklarált változókhoz kapcsolódik. A var-al deklarált változók nem rendelkeznek blokk hatókörrel.

A blokk hatóköre elmondja, hogy a (z) {} blokkon belül deklarált bármely változó csak abban a blokkban érhető el, és azon kívül nem érhető el.

```
{
  let x = 45;
}

console.log(x); // Gives reference error since x cannot be accessed outside of the block

for(let i=0; i<2; i++){
  // do something
}

console.log(i); // Gives reference error since i cannot be accessed outside of the for loop block
```

Scope Chain

A JavaScript motor a Scope-ot is használja a változók megtalálásához.

Értsük meg, hogy egy példa segítségével:

```
var y = 24;

function favFunction(){
  var x = 667;
  var anotherFavFunction = function(){
    console.log(x); // Does not find x inside anotherFavFunction, so looks for variable inside favFunction, outputs 667
  }

  var yetAnotherFavFunction = function(){
    console.log(y); // Does not find y inside yetAnotherFavFunction, so looks for variable inside favFunction and does not find it, so looks for variable in global scope, finds it and outputs 24
  }

  anotherFavFunction();
  yetAnotherFavFunction();
}


favFunction();
```

Amint a fenti kódban láthatja, ha a javascript motor nem találja a változót a helyi hatókörben (local scope), akkor megpróbálja ellenőrizni a változót a külső hatókörben (outer scope). Ha a változó nem létezik a külső hatókörben, akkor megpróbálja megtalálni a változót a globális hatókörben.

Ha a változó nem található a globális térben sem, akkor referenciahibát dob.

Magyarázza el a closure-okat JavaScript-ben.?

A closure-ok egy függvény azon képessége, hogy emlékezzen a külső hatókörében deklarált változókra és függvényekre.

```
var Person = function(pName){
  var name = pName;

  this.getName = function(){
    return name;
  }
}

var person = new Person("Neelesh");
console.log(person.getName());
```

Példa:

```
function randomFunc(){
  var obj1 = {name:"Vivian", age:45};

  return function(){
    console.log(obj1.name + " is "+ "awesome"); // Has access to obj1 even when the randomFunc function is executed

  }
}

var initialiseClosure = randomFunc(); // Returns a function

initialiseClosure();
```

Értsük meg a fenti kódot,

A randomFunc () függvény végrehajtásra kerül, és visszaad egy függvényt, amikor egy változóhoz rendeljük:

```
var initialiseClosure = randomFunc();
```

Ezután a visszaküldött függvény végrehajtásra kerül, amikor meghívjuk az initiseClosure-t:

```
initialiseClosure();
```

A fenti kódsor a „Vivian awesome” kimenetet adja, és ez a closure miatt lehetséges.

Amikor a randomFunc () függvény fut, látja, hogy a visszatérő függvény a benne lévő obj1 változót használja:

```
console.log(obj1.name + " is "+ "awesome");
```

Ezért a randomFunc () ahelyett, hogy a végrehajtás után megsemmisítené az obj1 értékét, további értékek céljából elmenti az értéket a memóriába. Ez az oka annak, hogy a visszatérő függvény képes a külső hatókörben deklarált változó használatára a függvény már végrehajtása után is.

A függvénynek ezt a képességét, hogy egy változót a végrehajtása után is további hivatkozás céljából használjon, closure-nak nevezzük.

Mik az objektum prototípusai?

Az összes javascript objektum egy prototípustól örököl tulajdonságokat.

Például,

- A dátumobjektumok öröklik a tulajdonságokat a datum (Date) prototípusától
- A matematikai objektumok a Math prototípustól örökölik a tulajdonságokat
- A tömb objektumok öröklik a tulajdonságokat a tomb (Array) prototípusától.
- A lánc tetején található az Object.prototype. Minden prototípus tulajdonságokat és módszereket örököl az Object.prototype-ból.

A prototípus egy objektum tervrajza. A prototípus lehetővé teszi számunkra, hogy tulajdonságokat és módszereket alkalmazzunk egy objektumon, még akkor is, ha a tulajdonságok és a módszerek nem léteznek az aktuális objektumon.

Példa:

```
var arr = [];
arr.push(2);

console.log(arr); // Outputs [2]
```

A fenti kódban, amint az látható, nem definiáltunk egyetlen tulajdonságot vagy metódust sem, amelyet pushnak hívunk az „arr” tömbön, de a javascript engine nem dob hibát.

Ennek oka a prototípusok használata. Ahogy korábban megbeszéltük, az Array objektumok örökölnek tulajdonságokat az Array prototípusból.

A javascript motor úgy látja, hogy a metódus nem létezik az aktuális tömbobjektumon, ezért megkeresi a metódust az Array prototípus belsejében, és megtalálja a módszert.

Amikor a tulajdonság vagy metódus nem található az aktuális objektumon, a javascript motor mindig megpróbálja a prototípusát keresni, és ha még mindig nem létezik, akkor a prototípus prototípusába néz, és így tovább.

Mi az memoization?

A memoization a gyorsítótár egyik formája, ahol a függvény visszatérési értéke a paraméterek alapján kerül gyorsítótárba. Ha az adott függvény paramétere nem változik, akkor a függvény gyorsítótárazott változata jelenik meg.

Értsük meg a memorization-t úgy, hogy egy egyszerű függvényt átalakítunk memoized függvénnyé:

Megjegyzés - A memóriát drága függvényhívásokhoz használják, de a következő példában egy egyszerű függvényt fontolgatunk, hogy jobban megértsük a memória fogalmát.

Vegye figyelembe a következő függvényt:

```
function addTo256(num){
  return num + 256;
}

addTo256(20); // Returns 276
addTo256(40); // Returns 296
addTo256(20); // Returns 276
```

A fenti kódba írtunk egy függvényt, amely hozzáadja a paramétert a 256-hoz, és visszaadja.

Amikor ugyanazzal a paraméterrel (a fenti esetben „20”) ismételten meghívjuk az addTo256 függvényt, akkor újra kiszámoljuk az eredményt ugyanarra a paraméterre.

Az eredmény ugyanazon paraméterrel történő újraszámítása a fenti esetben nem nagy baj, de képzeljük el, ha a függvény nagy teljesítményű munkát végez, akkor az eredmény ugyanazon paraméterrel történő újra és újra kiszámítása időpazarláshoz vezet.

Itt jön létre a memoizáció, a memoization segítségével tárolhatjuk (cache) a kiszámított eredményeket a paraméterek alapján. Ha ugyanazt a paramétert ismét használjuk a függvény meghívása közben, az eredmény kiszámítása helyett közvetlenül a tárolt (gyorsítótárazott) értéket adjuk vissza.

Konvertáljuk a fenti addTo256 függvényt memorized-ra:

```
function memoizedAddTo256(){
  var cache = {};

  return function(num){
    if(num in cache){
      console.log("cached value");
      return cache[num]

    }
    else{
```

```
      cache[num] = num + 256;
      return cache[num];
    }
  }
}

var memoizedFunc = memoizedAddTo256();

memoizedFunc(20); // Normal return
memoizedFunc(20); // Cached return
```

A fenti kódban, ha a memoizedFunc függvényt ugyanazzal a paraméterrel futtatjuk, az eredmény újbóli kiszámítása helyett a gyorsítótárazott eredményt adja vissza.

Megjegyzés - Bár a memoizálás időt takarít meg, nagyobb memóriafelhasználást eredményez, mivel az összes számított eredményt tároljuk.

Mik az osztályok a javascriptben?

Az ES6 változatban bevezetett osztályok nem más, mint a konstruktor függvényre syntactic sugar.

Új módszert kínálnak a konstruktor függvények javascriptben történő deklarálására.

Az alábbiakban bemutatjuk az osztályok deklarálásának és használatának példáit:

```
// Before ES6 version, using constructor functions
function Student(name,rollNumber,grade,section){
  this.name = name;
  this.rollNumber = rollNumber;
  this.grade = grade;
  this.section = section;
}

// Way to add methods to a constructor function
Student.prototype.getDetails = function(){
  return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade: ${this.grade}, Section:${this.section}';
}


let student1 = new Student("Vivek", 354, "6th", "A");
student1.getDetails();
// Returns Name: Vivek, Roll no:354, Grade: 6th, Section:A

// ES6 version classes
class Student{
  constructor(name,rollNumber,grade,section){
    this.name = name;
    this.rollNumber = rollNumber;
    this.grade = grade;
    this.section = section;
  }

  // Methods can be directly added inside the class
  getDetails(){
    return 'Name: ${this.name}, Roll no: ${this.rollNumber}, Grade:${this.grade}, Section:${this.section}';
  }
}
```

```
let student2 = new Student("Garry", 673, "7th", "C");
student2.getDetails();
// Returns Name: Garry, Roll no:673, Grade: 7th, Section:C
```

Az osztályokkal kapcsolatos legfontosabb megjegyzések:

A függvényektől eltérően az osztályok nem hoisted. Egy osztály nem használható a deklarálás előtt.

Egy osztály örökölheti a tulajdonságokat és a metódusokat más osztályoktól az extend kulcsszó használatával.

Az osztályon belüli összes szintaxisnak meg kell felelnie a javascript strict mode (‘use strict ‘).

Mi az az a Temporal Dead Zone?

Az Temporal Dead Zone olyan viselkedés, amely let és const kulcsszavak használatával deklarált változókkal fordul elő.

Ez egy olyan viselkedés, amikor megpróbálunk hozzáférni egy változóhoz, mielőtt inicializálnánk.

Példák:

```
x = 23; // Gives reference error

let x;


function anotherRandomFunc(){
  message = "Hello"; // Throws a reference error

  let message;
}
anotherRandomFunc();
```

A fenti kódban mind globális, mind funkcionális scope-ban olyan változókhoz próbálunk hozzáférni, amelyeket még nem deklaráltunk. Ezt hívják Temporal Dead Zone-nak.