https://github.com/ganqqwerty/123-Essential-JavaScript-Interview-Questions

Mi a hátránya a metódusok közvetlen deklarálásának a JavaScript objektumokban?

A metódusok közvetlenül a JavaScript objektumokban történő deklarálásának egyik hátránya, hogy nagyon memória-hatékonytalanok. Amikor ezt megteszi, a metódus új példánya jön létre az objektum minden példányához. Íme egy példa:

```
var Employee = function (name, company, salary) {
  this.name = name || "";
  this.company = company || "";
  this.salary = salary || 5000;

  // We can create a method like this:
  this.formatSalary = function () {
      return "$ " + this.salary;
  };
};

// Alternatively we can add the method to Employee's prototype:
Employee.prototype.formatSalary2 = function() {
    return "$ " + this.salary;
}

//creating objects
var emp1 = new Employee('Yuri Garagin', 'Company 1', 1000000);
var emp2 = new Employee('Dinesh Gupta', 'Company 2', 1039999);
var emp3 = new Employee('Erich Fromm', 'Company 3', 1299483);
```

Ebben az esetben az emp1, emp2, emp3 minden változónak megvan a maga másolata a formatSalary metódusból. A formatSalary2 azonban csak egyszer kerül hozzáadásra az Employee.prototype fájlhoz.

Mi a „closure” a javascriptben? Mondanál példát?

A closure egy másik függvényben definiált függvény (az úgynevezett szülőfüggvény), és mint ilyen hozzáférhet a szülőfüggvény hatókörében deklarált és definiált változókhoz.

A closure három skálán fér hozzá a változókhoz:

- A saját hatókörében deklarált változó
- A szülőfüggvény hatókörében deklarált változó
- A globális névtérben deklarált változó

```
var globalVar = "abc"; //Global variable

// Parent self-invoking function
(function outerFunction (outerArg) { // start of outerFunction's scope

  var outerFuncVar = 'x'; // Variable declared in outerFunction's function scope

  // Closure self-invoking function
  (function innerFunction (innerArg) { // start of innerFunction's scope

    var innerFuncVar = "y"; // variable declared in innerFunction's function scope
    console.log(
```

```
      "outerArg = " + outerArg + "\n" +
      "outerFuncVar = " + outerFuncVar + "\n" +
      "innerArg = " + innerArg + "\n" +
      "innerFuncVar = " + innerFuncVar + "\n" +
      "globalVar = " + globalVar);

  // end of innerFunction's scope

  })(5); // Pass 5 as parameter to our Closure

// end of outerFunction's scope

})(7); // Pass 7 as parameter to the Parent function
```

A internalFunction egy closure, amelyet az externalFunction belsejében definiálnak, és következésképpen hozzáfér az összes olyan változóhoz, amelyet deklaráltak és definiáltak az externalFunction hatókörében, valamint a program globális hatókörében található minden változóhoz.

A fenti kód kimenete a következő lenne:

```
outerArg = 7
outerFuncVar = x
innerArg = 5
innerFuncVar = y
globalVar = abc
```

Hogyan ellenőrizhető, hogy egy objektum tömb vagy sem?

A Object.prototype toString metódusának legjobb módja annak megállapításához, hogy egy objektum egy adott osztály példánya-e vagy sem.

```
if(Object.prototype.toString.call(arrayList) === '[object Array]') {
  console.log('Array!');
}
```

Mi lesz a következő kód kimenete?

```
var bar = true;
console.log(bar + 0);
console.log(bar + "xyz");
console.log(bar + true);
console.log(bar + false);
```

the code above will output `1, "truexyz", 2, 1` as output. Here's a general guideline for the plus operator:

- Number + Number -> Addition
- Boolean + Number -> Addition
- Boolean + Boolean -> Addition
- Number + String -> Concatenation
- String + Boolean -> Concatenation
- String + String -> Concatenation

Mi lesz a következő kód kimenete?

```
var z = 1, y = z = typeof y;
console.log(y);
```

A fenti kód a "undefined" karakterláncot fogja kiírni. Az asszociativitási szabály szerint az azonos prioritású operátorokat az operátor asszociativitási tulajdonságai alapján dolgozzák fel. Itt jobbról-balra megy a kiértékelés. A teljes sorrend így fog kinézni:

```
var z;
z = 1;
var y;
z = typeof y;
y = z;
```

Mi a különbség a függvénynek az alább felsorolt formátumokban történő deklarálása között?

```
var foo = function() {
  // Some code
}
function bar () {
  // Some code
}
```

A fő különbség az, hogy a foo függvény futás közben van definiálva, és függvény kifejezésnek (function expression) nevezzük, míg a bar()-t parse időben definiáljuk, és függvény utasításnak (function statement) hívjuk. Hogy jobban megértsük, vessünk egy pillantást az alábbi kódra:

```
// Run-Time function declaration
  foo(); // Call foo function here, It will give an error
  var foo = function() {
    console.log("Hi I am inside Foo");
  };

// Parse-Time function declaration
bar(); // Call bar function here, It will not give an Error
function bar() {
  console.log("Hi I am inside Foo");
}
```

Melyik esetben a függvénydefiníció nem lesz hoisted a JavaScriptben?

Vegyük a következő függvény kifejezést:

```
var foo = function foo() {
     return 12;
 }
```

A JavaScript-ben a deklarált változókat és függvények hoisted-ok. Alapvetően a JavaScript-fordító előre tekint, hogy megtalálja az összes változó deklarációt, és a függvény tetejére emeli, ahol deklarálva van. Például:

```
foo(); // Here foo is still undefined
var foo = function foo() {
  return 12;
};
```

A jelenet mögött található kód körülbelül így néz ki:

```
var foo = undefined;
foo(); // Here foo is undefined
foo = function foo() {
  // Some code stuff
}

var foo = undefined;
foo = function foo() {
  // Some code stuff
}
foo(); // Now foo is defined here
```

Mi lesz a következő kód kimenete?

```
var salary = "1000$";

(function () {
  console.log("Original salary was " + salary);

  var salary = "5000$";

  console.log("My New Salary " + salary);
})();
```

Válasz:

A fenti kód kimenete: undefined, 5000 $ a hoisting miatt

```
var salary = "1000$";

(function () {
  var salary = undefined;
  console.log("Original salary was " + salary);

  salary = "5000$";

  console.log("My New Salary " + salary);
})();
```

Mi a különbség a typeof és a instanceof között?

A typeof egy olyan operátor, amely egy karakterláncot ad vissza az átadott adatok típusával.

A typeof operátor ellenőrzi, hogy az érték a hét alaptípus valamelyikéhez tartozik-e: szám, karakterlánc, logikai érték, objektum, függvény, undefined vagy Symbol.

A typeof (null) az objektumot adja vissza.

Az instanceof sokkal intelligensebb: prototípusok szintjén működik. Különösen azt vizsgálja, hogy a job oldali operandus megjelenik-e valahol a bal prototípus láncában. instanceof nem működik primitív típusokkal. Az instanceof operátor ellenőrzi az aktuális objektumot, és igaz értéket ad vissza, ha az objektum a megadott típusú, például:

```
var dog = new Animal();
dog instanceof Animal; // Output : true


var name = new String("xyz");
name instanceof String; // Output : true
```

Hogyan ellenőrizhető, hogy létezik-e kulcs egy JavaScript-objektumban, vagy sem.?

```
var person = {
          name: 'Nishant',
          age: 24
}
```

Első módszer:

```
console.log('name' in person); // checking own property print true
console.log('salary' in person); // checking undefined property print false
```

Második módszer:

```
console.log(person.hasOwnProperty('toString')); // print false
console.log(person.hasOwnProperty('name')); // print true
console.log(person.hasOwnProperty('salary')); // print false
```

Hogyan működik az Object.create módszer a JavaScript-ben?

Az ECMAScript 5 Object.create () metódus a legegyszerűbb módja annak, hogy az egyik objektum örököljön a másiktól, konstruktor függvény meghívása nélkül.

```
var employee = {
  name: 'Nishant',
  displayName: function () {
    console.log(this.name);
  }
};

var emp1 = Object.create(employee);
console.log(emp1.displayName());  // output "Nishant"
```

A fenti példában létrehozunk egy új objektumot emp1, amely örököl az employee-tól. Más szavakkal, az emp1 prototípusát employee-ra állítják be. Ezt követően az emp1 képes elérni ugyanazokat a tulajdonságokat és metódust mint az employee, amíg új tulajdonságokat vagy azonos nevű metódust meg nem határoznak.

Például: A displayName () metódus definiálása az emp1-en nem fogja automatikusan felülírni az employee displayName metódust.

```
emp1.displayName = function() {
          console.log('xyz-Anonymous');
};

employee.displayName(); //Nishant
emp1.displayName();//xyz-Anonymous
```

Ezen az Object.create () metódus egy második argumentum megadását is lehetővé teszi, amely egy olyan objektum, amely további tulajdonságokat és metódusokat tartalmaz az új objektumhoz való hozzáadáshoz.

```
var emp1 = Object.create(employee, {
          name: {
                    value: "John"
          }
});

emp1.displayName(); // "John"
employee.displayName(); // "Nishant"
```

Hogyan lehet a konstruktor függvényeket használni az örökléshez a JavaScriptben?

Tegyük fel, hogy van Person osztályunk, amelynek neve, kora, fizetési tulajdonságai és inkrementSalary () metódusa van.

```
function Person(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
  this.incrementSalary = function (byValue) {
    this.salary = this.salary + byValue;
  };
}
```

Most szeretnénk létrehozni egy Employee osztályt, amely tartalmazza a Person osztály összes tulajdonságát, és további tulajdonságokat akartunk hozzáadni az Employee osztályhoz.

```
function Employee(company){
          this.company = company;
}

//Prototypal Inheritance
Employee.prototype = new Person("Nishant", 24,5000);
```

A fenti példában az Employee típus örökli a Person. Ezt úgy teszi meg, hogy a Person to Employee új példányát hozzárendeli a prototípushoz. Ezt követően az Employee minden példánya örököli tulajdonságait és metódusokat a Person-től.

```
//Prototypal Inheritance
Employee.prototype = new Person("Nishant", 24,5000);

var emp1 = new Employee("Google");

console.log(emp1 instanceof Person); // true
console.log(emp1 instanceof Employee); // true
```

Értsük meg a konstruktor öröklődését:

```
//Defined Person class
function Person(name){
          this.name = name || "Nishant";
}

var obj = {};

// obj inherit Person class properties and method
```

```
Person.call(obj); // constructor inheritance

console.log(obj); // Object {name: "Nishant"}
```

Itt láthattuk, hogy a Person.call (obj) hívás meghatározta a név tulajdonságait Person-től obj-ig.

```
console.log(name in obj); // true
```

A típus alapú öröklődést a fejlesztő által definiált konstruktor függvénnyel lehet a legjobban használni,. Emellett rugalmasságot is lehetővé tesz a hasonló típusú objektumok létrehozásában.

Hogyan akadályozhatjuk meg az objektum módosítását a JavaScript-ben?

Az ECMAScript 5 számos módszert vezet be az objektum módosításának megakadályozására, amelyek lezárják az objektumot annak biztosítására, hogy senki ne változtassa meg az Object funkcionalitását.

A módosítás megakadályozásának három szintje van:

1: A kiterjesztések megakadályozása (Prevent extensions):

Új tulajdonságok vagy metódusok nem adhatók hozzá az objektumhoz, de a meglévő tulajdonságok és metódus megváltoztatható.

Például:

```
var employee = {
          name: "Nishant"
};

// lock the object
Object.preventExtensions(employee);

// Now try to change the employee object property name
employee.name = "John"; // work fine

//Now try to add some new property to the object
employee.age = 24; // fails silently unless it's inside the strict mode
```

2: Seal:

Ez megegyezik a kiterjesztés megakadályozásával (prevent extension), emellett megakadályozza a meglévő tulajdonságok és metódusok törlését is.

Egy objektum lezárásához (seal) az Object.seal () metódust használjuk. az Object.isSealed () használatával ellenőrizheti, hogy egy objektum lezárva (sealed) van-e vagy sem;

```
var employee = {
          name: "Nishant"
```

```
};

// Seal the object
Object.seal(employee);

console.log(Object.isExtensible(employee)); // false
console.log(Object.isSealed(employee)); // true

delete employee.name // fails silently unless it's in strict mode

// Trying to add new property will give an error
employee.age = 30; // fails silently unless in strict mode
```

egy objektum lezárásakor (seales) a meglévő tulajdonságai és metódusai nem távolíthatók el. A lezárt (sealed) objektumok szintén nem származtathatóak le.

3: Freeze:

Hasonló, mint a seal, de ezen kívül az összes tulajdonság és metódus csak olvasható.

Objektum lefagyasztásához (freeze) használja az Object.freeze () metódust. Azt is meg tudjuk határozni, hogy egy objektum lefagyott-e az Object.isFrozen () használatával;

```
var employee = {
          name: "Nishant"
};

//Freeze the object
Object.freeze(employee);

// Seal the object
Object.seal(employee);

console.log(Object.isExtensible(employee)); // false
console.log(Object.isSealed(employee));     // true
console.log(Object.isFrozen(employee));     // true


employee.name = "xyz"; // fails silently unless in strict mode
employee.age = 30;     // fails silently unless in strict mode
delete employee.name   // fails silently unless it's in strict mode
```

A frozen objektumok nem terjeszthetőek ki és seales is tekintjük.

Ajánlott:

Ha úgy dönt, hogy megakadályozza a módosítást, az objektum legyen sealed és freeze, majd használja a strict mode-ot, hogy el tudja érni a hibát.

```
"use strict";

var employee = {
          name: "Nishant"
};

//Freeze the object
Object.freeze(employee);

// Seal the object
Object.seal(employee);

console.log(Object.isExtensible(employee)); // false
console.log(Object.isSealed(employee));     // true
console.log(Object.isFrozen(employee));     // true
```

```
employee.name = "xyz"; // fails silently unless in strict mode
employee.age = 30;     // fails silently unless in strict mode
delete employee.name;  // fails silently unless it's in strict mode
```

Mi a névtelen függvény tipikus felhasználási esete a JavaScriptben?

A névtelen függvények alapvetően a következő esetekben használatosak.

1.) Nincs szükség névre, ha a függvényt csak egy helyen használják, akkor nem szükséges nevet adni a függvényhez.

Vegyük a setTimeout függvény példáját:

```
setTimeout(function(){
          alert("Hello");
},1000);
```

Itt nincs szükség a megnevezett függvény használatára, ha biztosak vagyunk abban, hogy a hello alert-et csak egyszer használja az alkalmazás.

2.) Az anonim függvényeket inline-nak nyilvánítják, és az inline függvényeknek előnyei vannak abban az esetben, ha hozzáférhetnek a szülő hatókörében lévő változóhoz.

Vegyünk egy példát az eseménykezelőre. Értesítés egy adott típusú eseményről (például kattintás) egy adott objektumhoz.

Tegyük fel, hogy van HTML elemünk (gombunk), amelyre kattintási eseményt szeretnénk hozzáadni, és amikor a felhasználó rákattint a gombra, szeretnénk végrehajtani valamilyen logikát.

```
<button id="myBtn"></button>
```

Event Listener:

```
var btn = document.getElementById('myBtn');
btn.addEventListener('click', function () {
  alert('button clicked');
});
```

A fenti példa az anonim függvény callback függvényként történő felhasználását mutatja be az eseménykezelőben.

3.) A névtelen függvény paraméterének átadása a hívó függvénynek.

```
// Function which will execute callback function
function processCallback(callback){
          if(typeof callback === 'function'){
                    callback();
          }
}

// Call function and pass anonymous function as callback
processCallback(function(){
          alert("Hi I am anonymous callback function");
```

```
});
```

A névtelen függvény használatával kapcsolatos döntéshozatal legjobb módja a következő kérdés feltevése:

Használni fogom azt a függvényt, amelyet meghatározni fogok, bárhol máshol?

Ha nem, hozzon létre egy névtelen függvényt.

A névtelen függvény használatának előnye:

- Csökkenthet egy kis kódot, különösen rekurzív és callback függvények esetén.
- globális névtér-szennyezések elkerülése.

Hogyan állítsunk be egy alapértelmezett paraméterértéket?

Method 1: Setting default parameter value

```
function sentEmail(configuration, provider) {
  // Set default value if user has not passed value for provider
  provider = typeof provider !== 'undefined' ? provider : 'Gmail'
  // Your code logic
;
}
// In this call we are not passing provider parameter value
sentEmail({
  from: 'xyz@gmail.com',
  subject: 'Test Email'
});
// Here we are passing Yahoo Mail as a provider value
sentEmail({
  from: 'xyz@gmail.com',
  subject: 'Test Email'
}, 'Yahoo Mail');
```

Method 2: Setting default parameter value

```
function sentEmail(configuration, provider) {
  // Set default value if user has not passed value for provider
  provider = provider || 'Gmail'
  // Your code logic
;
}
// In this call we are not passing provider parameter value
sentEmail({
  from: 'xyz@gmail.com',
  subject: 'Test Email'
});
// Here we are passing Yahoo Mail as a provider value
sentEmail({
  from: 'xyz@gmail.com',
  subject: 'Test Email'
}, 'Yahoo Mail');
```

Method 3: Setting default parameter value in ES6

```
function sendEmail(configuration, provider = "Gmail") {
  // Set default value if user has not passed value for provider

  // Value of provider can be accessed directly
  console.log(`Provider: ${provider}`);
}

// In this call we are not passing provider parameter value
sentEmail({
```

```
  from: 'xyz@gmail.com',
  subject: 'Test Email'
});
// Here we are passing Yahoo Mail as a provider value
sentEmail({
  from: 'xyz@gmail.com',
  subject: 'Test Email'
}, 'Yahoo Mail');
```

Mi a non-enumerable tulajdonság a JavaScript-ben, és hogyan hozhat létre egyet?

Az objektumnak lehetnek olyan tulajdonságai, amelyek nem jelennek meg, amikor az objektumon keresztül iterál a for...in ciklusban, vagy az Object.keys () segítségével megkapja a tulajdonságok tömbjét. Ezeket a tulajdonságokat nem enumerálható (non-enumerable) tulajdonságoknak ismerjük.

Mondjuk, hogy van egy következő objektumunk:

```
var person = {
          name: 'John'
};
person.salary = '10000$';
person['country'] = 'USA';

console.log(Object.keys(person)); // ['name', 'salary', 'country']
```

Mint tudjuk, az objektum tulajdonságainak neve, fizetése, országa megszámlálható, ezért megjelenik, amikor az Object.keys (személy) nevet kaptuk.

non-enumerable tulajdonság létrehozásához az Object.defineProperty () függvényt kell használnunk. Ez egy speciális módszer a nem megszámlálhatatlan tulajdonság létrehozására a JavaScript-ben.

```
var person = {
          name: 'John'
};
person.salary = '10000$';
person['country'] = 'USA';

// Create non-enumerable property
Object.defineProperty(person, 'phoneNo',{
          value : '8888888888',
          enumerable: false
})

Object.keys(person); // ['name', 'salary', 'country']
```

A fenti példában a phoneNo tulajdonság nem jelent meg, mert a enumerable:false-ra állítottuk.

Mi a függvénykötés (function binding)?

A függvénykötés (function binding) eleve a JavaScript kategóriába tartozik, és ez nagyon népszerű technika, amelyet az eseménykezelővel és a callback függvénnyel együtt használnak a kódfuttatási kontextus megőrzéséhez, miközben a függvényt paraméterként adja át.

Vizsgáljuk meg a következő példát:

```
var clickHandler = {
```

```
          message: 'click event handler',
          handleClick: function(event) {
                    console.log(this.message);
          }
};

var btn = document.getElementById('myBtn');
// Add click event to btn
btn.addEventListener('click', clickHandler.handleClick);
```

Ebben a példában itt jön létre a clickHandler objektum, amely tartalmazza a message property-t és a handleClick metódust.

A DOM gombhoz hozzárendeltük a handleClick metódust, amelyet a kattintásra válaszul hajtunk végre. Ha rákattint a gombra, akkor a handleClick metódust hívja meg. Itt a console.log naplózza a kattintási eseménykezelő üzenetet, de valójában undefined-et logol.

A undefined megjelenítésének problémája az oka, hogy a clickHandler.handleClick metódus végrehajtási kontextusa nem mentésre kerül, ezért ez a button btn objektumra mutat. Megoldhatjuk ezt a problémát a bind módszerrel.

```
var clickHandler = {
          message: 'click event handler',
          handleClick: function(event) {
                    console.log(this.message);
          }
};

var btn = document.getElementById('myBtn');
// Add click event to btn and bind the clickHandler object
btn.addEventListener('click', clickHandler.handleClick.bind(clickHandler));
```

Értékek átadása referencia és érték alapján

A JS-fejlesztők számára alapvető fontosságú megérteni, hogy mely értékeket adják át referenciaként, és melyeket adják át az értékek. Ne felejtsük el, hogy az objektumokat, beleértve a tömböket is, referenciaként adjuk át, míg a karakterláncokat, logikai értékeket és számokat érték szerint adjuk át.

Mi a kimenete az alábbi kódnak?

```
var strA = "hi there";
var strB = strA;
strB="bye there!";
console.log (strA)
```

Kimenet: ‘hi there’, mert string-ek érték szerint adódnak át.

Mi a kimenete az alábbi kódnak?

```
var objA = {prop1: 42};
```

```
var objB = objA;
objB.prop1 = 90;
console.log(objA)
```

{prop1: 90}, mert objektumokkal foglakozunk.

Mi a kimenete az alábbi kódnak?

```
var objA = {prop1: 42};
var objB = objA;
objB = {};
console.log(objA)
```

A kimenet {prop1: 42} lesz. Amikor az objB-t hozzárendeljük az objB-hez, az objB változó ugyanarra az objektumra mutat, mint az objB változó. Amikor azonban az objB-t újból hozzárendeljük egy üres objektumhoz, egyszerűen megváltoztatjuk az objB változó hivatkozási helyét. Ez nem befolyásolja, hogy az objA változó hol hivatkozik.

Mi a kimenete az alábbi kódnak?

```
var arrA = [0,1,2,3,4,5];
var arrB = arrA;
arrB[0]=42;
console.log(arrA)
```

A kimenet [42,1,2,3,4,5] lesz. A tömbök a JavaScript objektumai, és referenciával adják át őket. Ezért az arrA és az arrB is ugyanarra a tömbre mutat [0,1,2,3,4,5]. Ezért az arrB első elemének megváltoztatása módosítja az arrA-t is: ugyanaz a tömb a memóriában.

Mi a kimenete az alábbi kódnak?

```
var arrA = [0,1,2,3,4,5];
var arrB = arrA.slice();
arrB[0]=42;
console.log(arrA)
```

A kimenet [0,1,2,3,4,5] lesz. A slice () függvény az új tömböt ad vissza. Ezért hivatkozik az arrA és az arrB két teljesen különböző tömbre.

Mi a kimente az alábbi kódnak?

```
var arrA = [{prop1: "value of array A!!"},  {someProp: "also value of array A!"}, 3,4,5];
var arrB = arrA;
arrB[0].prop1=42;
console.log(arrA);
```

A kimenet [{prop1: 42}, {someProp: " also value of array A!"}, 3,4,5]. A tömbök objektumok a JS-ben, így az arrA és az arrB változók is ugyanazon tömbre mutatnak. Az arrB [0] megváltoztatása megegyezik az arrA [0] módosításával

Mi a kimente az alábbi kódnak?

```
var arrA = [{prop1: "value of array A!!"}, {someProp: "also value of array A!"},3,4,5];
var arrB = arrA.slice();
arrB[0].prop1=42;
arrB[3] = 20;
console.log(arrA);
```

A kimenet [{prop1: 42}, {someProp: "szintén az A tömb értéke!"}, 3,4,5]. A slice() függvény az új tömböt visszaadó tömb összes elemét lemásolja. Azonban nem végez mély másolást. Ehelyett sekély másolást végez. Elképzelheti, hogy a slice() így valósul meg:

```
function slice(arr) {
   var result = [];
   for (i = 0; i< arr.length; i++) {
       result.push(arr[i]);
   }
   return result;
}
```

Nézze meg az result.push (arr [i]) sort. Ha az arr [i] történetesen szám vagy karaktersorozat, akkor érték szerint adja át. Ha az arr [i] objektum, akkor hivatkozást adunk át. Tömbünk esetén az arr [0] egy {prop1: " value of array A!!"} objektum. Csak az erre az objektumra való hivatkozás kerül másolásra. Ez gyakorlatilag azt jelenti, hogy az arrA és az arrB tömbök megosztják az első két elemet. Ezért változtatja meg az arrB [0] tulajdonságát az arrB-ben az arrA [0] is.