https://github.com/sudheerj/javascript-interview-questions

Hogyan lehet objektumokat létrehozni a JavaScript-ben

1.) Object constructor

Üres objektum létrehozásának legegyszerűbb módja az Object konstruktor. Jelenleg ez a megközelítés nem ajánlott.

```
var object = new Object();
```

2.) Object's create method:

Object's create method új objektumot hoz létre azáltal, hogy a prototípus objektumot paraméterként adja át.

```
var object = Object.create(null);
```

3.) Object literal syntax:

Object literal syntax megegyezik az object create methoddal, mikor ott null-t adunk meg paraméterként.

```
var object = {};
```

4.) Function constructor:

Hozzon létre bármilyen függvényt, és alkalmazza az new operátort objektumpéldányok létrehozására:

```
function Person(name){
   var object = {};
   object.name=name;
   object.age=21;
   return object;
}
var object = new Person("Sudheer");
```

5.) Function constructor with prototype:

Ez hasonló a függvény konstruktorhoz, de prototípust használ tulajdonságaikhoz és metódusaihoz

```
function Person(){}
Person.prototype.name = "Sudheer";
var object = new Person();
```

Ez egyenértékű egy object create method-dal létrehozott példánnyal egy function prototype-al majd argumentumként meghívja ezt a függvényt egy példánnyal és paraméterekkel.

```
function func {};

new func(x, y, z);
```

vagy:

```
// Create a new instance using function prototype.
var newInstance = Object.create(func.prototype)

// Call the function
var result = func.call(newInstance, x, y, z),

// If the result is a non-null object then use it otherwise just use the new instance.
console.log(result && typeof result === 'object' ? result : newInstance);
```

6.) ES6 osztály szintaxisa:

Az ES6 bevezeti az osztály funkciót az objektumok létrehozására:

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

var object = new Person("Sudheer");
```

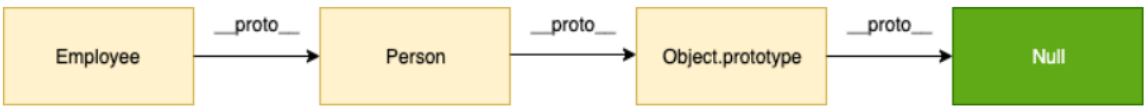
7.) Singleton minta:

A Singleton olyan objektum, amelyet csak egyszer lehet példányosítani. A konstruktorának ismételt hívásai ugyanazt a példányt adják vissza, és így biztos lehet abban, hogy véletlenül nem hoznak létre több példányt.

```
var object = new function(){
   this.name = "Sudheer";
}
```

Mi a prototípus lánc (prototype chain)

A prototípus láncolásával új típusú objektumokat lehet építeni a meglévők alapján. Hasonló az osztályalapú nyelv örökléséhez. Az objektum példányának prototípusa az Object.getPrototypeOf (objektum) vagy proto tulajdonságon keresztül érhető el, míg a konstruktorok függvény prototípusa az object.prototype oldalon érhető el.



Mi a különbség a Call, Apply and Bind között?

A Call, Apply és a Bind közötti különbség az alábbi példákkal magyarázható,

Call: A call () metódus meghív egy függvényt, amelynek egy adott értéke és argumentumai vannak egyesével megadva

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}
```

```
invite.call(employee1, 'Hello', 'How are you?'); // Hello John Rodson, How are you?
invite.call(employee2, 'Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

Apply: Meghívja a függvényt, és lehetővé teszi az argumentumok tömbként történő továbbítását

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

invite.apply(employee1, ['Hello', 'How are you?']); // Hello John Rodson, How are you?
invite.apply(employee2, ['Hello', 'How are you?']); // Hello Jimmy Baily, How are you?
```

bind: egy új függvényt ad vissza, lehetővé téve egy tömb és tetszőleges számú argumentum átadását

```
var employee1 = {firstName: 'John', lastName: 'Rodson'};
var employee2 = {firstName: 'Jimmy', lastName: 'Baily'};

function invite(greeting1, greeting2) {
    console.log(greeting1 + ' ' + this.firstName + ' ' + this.lastName+ ', '+ greeting2);
}

var inviteEmployee1 = invite.bind(employee1);
var inviteEmployee2 = invite.bind(employee2);
inviteEmployee1('Hello', 'How are you?'); // Hello John Rodson, How are you?
inviteEmployee2('Hello', 'How are you?'); // Hello Jimmy Baily, How are you?
```

A call és a bind meglehetősen felcserélhető. Mindkettő azonnal végrehajtja az aktuális függvényt. El kell döntenie, hogy könnyebb-e tömböt vagy vesszővel elválasztott argumentumlistát használni.

Mi a JSON és mik a közös műveletei?

A JSON egy JavaScript alapú szintaxist követő szöveges adatformátum, amelyet Douglas Crockford népszerűsített. Ez akkor hasznos, ha adatokat szeretne hálózaton keresztül továbbítani, és ez alapvetően csak egy .json kiterjesztésű szöveges fájl:

String konvertálása natív objektummá

```
JSON.parse(text)
```

Stringifikáció: ** natív objektum konvertálása karakterláncá, így továbbítható a hálózaton keresztül

```
JSON.stringify(object)
```

array slice method?

A slice () metódus a tömbben kiválasztott elemeket új tömbobjektumként adja vissza. Kiválasztja az adott kezdő argumentumtól kezdődő elemeket, és az adott opcionális vég argumentumnál fejeződik be az utolsó elem nélkül. Ha elhagyja a második argumentumot, akkor az a végéig választ. Néhány példa erre a módszerre:

```
let arrayIntegers = [1, 2, 3, 4, 5];
let arrayIntegers1 = arrayIntegers.slice(0,2); // returns [1,2]
let arrayIntegers2 = arrayIntegers.slice(2,3); // returns [3]
let arrayIntegers3 = arrayIntegers.slice(4); //returns [5]
```

Megjegyzés: A Slice a tömböt új tömbként adja vissza.

array splice method?

A splice () metódust az elemek hozzáadják / eltávolítják egy tömbből, majd visszaadják az eltávolított elemet. Az első argumentum a tömb pozícióját határozza meg a beillesztéshez vagy a törléshez, míg a második opció a törölni kívánt elemek számát jelzi. Minden további argumentum hozzáadódik a tömbhöz. Néhány példa erre a módszerre:

```
let arrayIntegersOriginal1 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal2 = [1, 2, 3, 4, 5];
let arrayIntegersOriginal3 = [1, 2, 3, 4, 5];

let arrayIntegers1 = arrayIntegersOriginal1.splice(0,2); // returns [1, 2]; original array:
[3, 4, 5]
let arrayIntegers2 = arrayIntegersOriginal2.splice(3); // returns [4, 5]; original array: [1,
2, 3]
let arrayIntegers3 = arrayIntegersOriginal3.splice(3, 1, "a", "b", "c"); //returns [4];
original array: [1, 2, 3, "a", "b", "c", 5]
```

A Splice módszer módosítja az eredeti tömböt, és visszaadja a törölt tömböt.

slice and splice közti különbségek?

Slice	Splice
Nem módosítja az eredeti tömböt (immutable)	Módosítja az eredeti tömböt (mutable)
Visszaadja az eredeti tömb részhalmazát	A törölt elemeket tömbként adja vissza
Az elemek tömbből történő kivételére szolgál	Elemek beillesztésére vagy törlésére szolgál a tömbből

Object and Map közti különbségek?

Az object-ek abban hasonlítanak a Maps-hez, hogy mindkettő lehetővé teszi, hogy kulcsok-érték párokat készítsen, lehívja ezeket az értékeket, törölje a kulcsokat, és észlelje, hogy van-e valami tárolva egy kulcson. Ebből az okból kifolyólag az Objektumokat történelmileg Maps-ként használták. Vannak azonban fontos különbségek, amelyek bizonyos esetekben előnyösebbé teszik a Map használatát.

Az objektum kulcsa a karakterlánc és a szimbólum, míg a Map-nél bármely érték lehet, beleértve a függvényeket, az objektumokat és a primitíveket is.

A Map kulcsai sorrendben vannak, míg az Objektumhoz hozzáadott kulcsok nem. Így egy iterálás után a Map objektum a kulcsokat a beszúrás sorrendjében adja vissza.

A size tulajdonsággal egyszerűen megszerezheti a Map méretét, míg az Objektumban lévő tulajdonságok számát manuálisan kell meghatározni.

A Map egy iterálható és így közvetlenül iterálható, míg az objektumon történő iteráláshoz valamilyen módon meg kell szerezni a kulcsait, és iterálni kell rajtuk.

Az objektumnak van prototípusa, ezért vannak olyan alapértelmezett kulcsok a Map-ben, amelyek ütközhetnek a kulcsokkal, ha nem vigyázol. Az ES5-től ez megkerülhető a map = Object.create (null) használatával, de ez ritkán történik meg.

A Map hatékonyabb olyan esetekben, amikor a kulcspárokat gyakran hozzáadják és eltávolítják.

Mi a különbség a == és a === operátorok között?

A JavaScript strict (===,! ==) és type-converting (==,! =) egyenlőség-összehasonlítást biztosít. A strict operátorok a változó típusát veszik figyelembe, míg a nem strict operátorok a típusok javítását / átalakítását a változók értékei alapján végzik. A strict operátorok az alábbi feltételeket követik a különféle típusoknál:

- Két karakterlánc szigorúan egyenlő, ha azonos karaktersorozat, azonos hosszúságú és karakterek vannak a megfelelő pozíciókban.
- Két szám szigorúan egyenlő, ha numerikusan egyenlő. vagyis azonos számértékkel rendelkezik. Két különleges eset van ebben,
    - A NaN nem egyenlő semmivel, beleértve a NaN-t sem.
    - A pozitív és a negatív nulla egyenlő egymással.
- Két logikai operandus szigorúan egyenlő, ha mindkettő igaz vagy mindkettő hamis.
- Két objektum szigorúan egyenlő, ha ugyanarra az objektumra vonatkoznak.
- A Null és a Undefined típusok nem egyenlőek a ===, de a == értékkel igen. azaz null === undefined -> false, de null == undefined -> true

```
0 == false   // true
0 === false  // false
1 == "1"     // true
1 === "1"    // false
null == undefined // true
null === undefined // false
'0' == false // true
'0' === false // false
[]==[] or []===[] //false, refer different objects in memory
{}=={} or {}==={} //false, refer different objects in memory
```

Lambda vagy arrow function?

Az arrow function rövidebb szintaxisa egy függvénykifejezésnek, és nem rendelkezik sajátokkal az alábbiakból: this, arguments, super, or new.target. Ezek a függvények a legalkalmasabbak a nemmetódus függvényekhez, és nem használhatók konstruktorként.

first class function?

A Javascriptben a függvények first class object-ek. Az first class object azt jelentik, amikor az adott nyelv függvényeit úgy kezeljük, mint bármely más változót.

Például egy ilyen nyelvben egy függvény argumentumként átadható más függvényeknek, egy másik függvény visszaadhatja és egy változó értékeként rendelheti hozzá. Például az alábbi példában a handler:

```
const handler = () => console.log ('This is a click handler function');
document.addEventListener ('click', handler);
```

first order function?

olyan függvény, amely nem fogad el argumentumként egy másik függvényt, és a függvényt nem adja vissza visszatérési értékként.

```
const firstOrder = () => console.log ('I am a first order function!');
```

higher order function?

függvény, amely elfogad egy másik függvényt argumentumként, vagy visszatér egy függvényt visszatérési értékként.

```
const firstOrderFunc = () => console.log ('Hello I am a First order function');
const higherOrder = ReturnFirstOrderFunc => ReturnFirstOrderFunc ();
higherOrder (firstOrderFunc);
```

unary function?

függvény, amely pontosan egy argumentumot fogad el.

```
const unaryFunction = a => console.log (a + 10); // Add 10 to the given argument and display
the value
```

currying function?

egy függvényt több argumentummal ehelyett függvények sorozatává alakítjuk, amelyek mindegyikének csak egyetlen argumentuma van. A currying alkalmazásával egy n-áris függvényt unáris függvénnyé alakítja. Vegyünk egy példát az n-ary függvényre, és arra, hogyan alakul át curry függvénnyé:

```
const multiArgFunction = (a, b, c) => a + b + c;
const curryUnaryFunction = a => b => c => a + b + c;
curryUnaryFunction (1); // returns a function: b => c =>  1 + b + c
curryUnaryFunction (1) (2); // returns a function: c => 3 + c
curryUnaryFunction (1) (2) (3); // returns the number 6
```

A curried függvények remekül javítják a kód újrafelhasználhatóságát és a funkcionális összetételt.

pure function?

függvény, ahol a visszatérési értéket csak az argumentumai határozzák meg. azaz ha egy függvényt ugyanazokkal a paraméterekkel 'n'-szer hívunk meg az alkalmazásban, akkor mindig ugyanazt az értéket adja vissza. Vegyünk egy példát a pure és a nem pure függvény közötti különbséget,

```
//Impure
let numberArray = [];
const impureAddNumber = number => numberArray.push (number);
//Pure
const pureAddNumber = number => argNumberArray =>
  argNumberArray.concat ([number]);

//Display the results
console.log (impureAddNumber (6)); // returns 1
console.log (numberArray); // returns [6]
console.log (pureAddNumber (7) (numberArray)); // returns [6, 7]
console.log (numberArray); // returns [6]
```

A fenti kódrészletek szerint a Push függvény nem pure a tömb megváltoztatásával és a paraméterértéktől független push számindex visszaadásával. Míg a Concat viszont felveszi a tömböt és összefűzi a másik tömbvel, egy teljesen új tömböt hoz létre. Ezenkívül a visszatérési érték az előző tömb összefűzése. Ne feledje, hogy a pure függvények fontosak, mivel egyszerűsítik az egység tesztet függőség-injektálás nélkül.

let keyword?

A let utasítás egy blokk hatókörű helyi változót deklarál. Ezért a let kulcsszóval definiált változók hatóköre a blokkra, utasításra vagy kifejezésre korlátozódik, amelyre használják. Míg a var kulcsszóval deklarált változók a változó globális vagy lokális meghatározására szolgálnak, a blokk hatókörétől függetlenül. Vegyünk egy példát a használat bemutatására:

```
let counter = 30;
if (counter === 30) {
  let counter = 31;
  console.log(counter); // 31
}
console.log(counter); // 30 (because if block variable won't exist here)
```

Let és var közti különbségek?

var	let
It is been available from the beginning of JavaScript	Introduced as part of ES6
It has function scope	It has block scope
Variables will be hoisted	Hoisted but not initialized

```
function userDetails(username) {
   if(username) {
     console.log(salary); // undefined(due to hoisting)
     console.log(age); // error: age is not defined
     let age = 30;
     var salary = 10000;
   }
   console.log(salary); //10000 (accessible to due function scope)
   console.log(age); //error: age is not defined(due to block scope)
}
```

Hogyan deklarálhatjuk újra a változókat a switch-ben hiba nélkül?

Ha megpróbálja újra deklarálni a változókat egy switch blokkban, az hibákat okoz, mert csak egy blokk van. Például az alábbi kódblokk az alábbiak szerint szintaxis hibát dob,

```
let counter = 1;
switch(x) {
  case 0:
    let name;
    break;

  case 1:
    let name; // SyntaxError for redeclaration.
    break;
}
```

A hiba elkerülése érdekében létrehozhat egy beágyazott blokkot egy case-ben.

```
let counter = 1;
    switch(x) {
      case 0: {
        let name;
        break;
      }
      case 1: {
        let name; // No SyntaxError for redeclaration.
        break;
      }
    }
```

Temporal Dead Zone?

A Temporal Dead Zone egy olyan viselkedés a JavaScript-ben, amely akkor fordul elő, amikor egy változót a let és const kulcsszavakkal deklarálunk, a var-val azonban nem. Az ECMAScript 6-ban egy let vagy const változó elérése a deklarációja előtt (a hatókörén belül) ReferenceError-t okoz. Az időtartamot, amikor ez megtörténik, temporal dead zone-nak nevezzük. Nézzük meg ezt a viselkedést egy példával,

```
function somemethod() {
  console.log(counter1); // undefined
  console.log(counter2); // ReferenceError
  var counter1 = 1;
  let counter2 = 2;
}
```

IIFE(Immediately Invoked Function Expression)?

A JavaScript funkció, amely a létrehozása után azonnal lefut:

```
(function ()
    {
      // logic here
    }
 )
();
```

Az IIFE használatának elsődleges oka az adatvédelem megszerzése, mert a IIFE-ben deklarált változókhoz a külvilág nem férhet hozzá. azaz ha megpróbál hozzáférni a változókhoz az IIFE segítségével, akkor az az alábbiak szerint hibát dob,

```
(function ()
        {
          var message = "IIFE";
          console.log(message);
        }
 )
();
console.log(message); //Error: message is not defined
```

Mi a modulok használatának előnye?

Karbantarthatóság

Újrahasználhatóság

Namespacing

memoization?

memoization olyan programozási technika, amely megkísérli növelni a funkció teljesítményét azáltal, hogy a korábban kiszámolt eredményeket gyorsítótárba helyezi. Minden alkalommal, amikor egy memóriába felvett függvény meghívásra kerül, annak paramétereit használják a gyorsítótár indexelésére. Ha az adatok vannak, akkor a teljes függvény végrehajtása nélkül visszaadhatók. Ellenkező esetben a függvény végrehajtásra kerül, majd az eredmény hozzáadódik a gyorsítótárhoz.

```
const memoizAddition = () => {
  let cache = {};
 return (value) => {
  if (value in cache) {
   console.log('Fetching from cache');
   return cache[value]; // Here, cache.value cannot be used as property name starts with the number
which is not a valid JavaScript  identifier. Hence, can only be accessed using the square bracket
notation.
  }
  else {
   console.log('Calculating result');
   let result = value + 20;
   cache[value] = result;
   return result;
  }
 }
}
// returned function from memoizAddition
const addition = memoizAddition();
console.log(addition(20)); //output: 40 calculated
console.log(addition(20)); //output: 40 cached
```

Hoisting?

Hoisting olyan JavaScript mechanizmus, ahol a változókat és a függvénydeklarációkat a hatókörük tetejére helyezik a kód végrehajtása előtt. Ne feledje, hogy a JavaScript csak a deklarációkat hoist-olja, az inicializálást nem. Vegyünk egy egyszerű példát:

```
console.log(message); //output : undefined
var message = 'The variable Has been hoisted';
```

A fenti kód az alábbiak szerint néz ki a fordítónak:

```
var message;
console.log(message);
message = 'The variable Has been hoisted';
```

Mik az osztályok az ES6-ban?

Az ES6-ban a Javascript osztályok elsősorban syntactic sugar-t jelentenek a JavaScript meglévő prototípus-alapú öröklődéséhez (prototype-based inheritance) képest. Például a prototípuson alapuló öröklődés, amelyet az alábbi függvénykifejezésben írunk:

```
function Bike(model,color) {
    this.model = model;
    this.color = color;
}

Bike.prototype.getDetails = function() {
    return this.model + ' bike has' + this.color + ' color';
};
```

Míg az ES6 osztályok:

```
class Bike{
  constructor(color, model) {
    this.color= color;
    this.model= model;
  }

  getDetails() {
    return this.model + ' bike has' + this.color + ' color';
  }
}
```

Mik a closure-ok?

A closure egy függvény és a lexikai környezet kombinációja, amelyen belül a függvényt deklarálták. vagyis egy olyan belső függvény, amely hozzáférést biztosít a külső vagy bezáró függvény változóihoz. A bclosure-nak három hatóköre van:

1.) Saját hatókör, ahol változók vannak meghatározva a {} zárójelben.

2.) Külső függvény változói

3.) Globális változók

Vegyünk egy példát,

```
function Welcome(name){
  var greetingInfo = function(message){
   console.log(message+' '+name);
  }
return greetingInfo;
}
var myFunction = Welcome('John');
myFunction('Welcome '); //Output: Welcome John
myFunction('Hello Mr.'); //output: Hello Mr.John
```

A fenti kódnak megfelelően a belső függvény (greetingInfo) a külső függvény (Welcome) változókhoz is hozzáférhet, még miután a külső függvény visszatért is.

Mik azok a modulok?

A modulok független, újrafelhasználható kód kis egységeire utalnak, és számos JavaScript tervezési minta alapjaként is szolgálnak. A legtöbb JavaScript modul objektum literált, függvényt vagy konstruktort exportál.

Mi a hatókör (scope) a javascriptben?

A hatókör (scope) a változók, függvények és objektumok hozzáférhetősége a kód bizonyos részeiben futás közben. Más szavakkal, a hatókör (scope) meghatározza a változók és más erőforrások láthatóságát a kód területein.

Mi az a Service worker?

A Service worker alapvetően egy szkript (JavaScript fájl), amely a háttérben fut, elkülönül a weboldaltól, és olyan szolgáltatásokat nyújt, amelyekhez nincs szükség weboldalra vagy felhasználói beavatkozásra. A Service worker néhány fő jellemzője: Rich offline experiences (offline első webalkalmazás-fejlesztés), időszakos háttérszinkronizálások, push notifications, hálózati kérések lehallgatása és kezelése, valamint a válaszok gyorsítótárának programozott kezelése.

Hogyan manipulálhatja a DOM-ot egy Service worker segítségével?

A Service worker nem férhet hozzá közvetlenül a DOM-hoz. De képes kommunikálni az általa ellenőrzött oldalakkal a postMessage felületen keresztül küldött üzenetekre válaszolva, és ezek az oldalak manipulálhatják a DOM-ot.

Mi az IndexedDB?

Az IndexedDB egy alacsony szintű API nagyobb mennyiségű strukturált adat, beleértve a fájlokat / blob-okat is, kliensoldali tárolásra. Ez az API indexeket használ az adatok nagy teljesítményű kereséseinek engedélyezéséhez.

Mi a web storage?

A web storage olyan API, amely mechanizmust biztosít a böngészők számára a kulcs / érték párok helyi tárolására a felhasználó böngészőjében, sokkal intuitívabban, mint a sütik használata. A web storage két mechanizmust biztosít az adatok kliens oldalon történő tárolására.

1.) Local storage: A jelenlegi származási adatokat lejárati dátum nélkül tárolja.

2.) Session storage: Egy munkamenet adatait tárolja, és az adatok elvesznek, amikor a böngésző lapja bezárul.

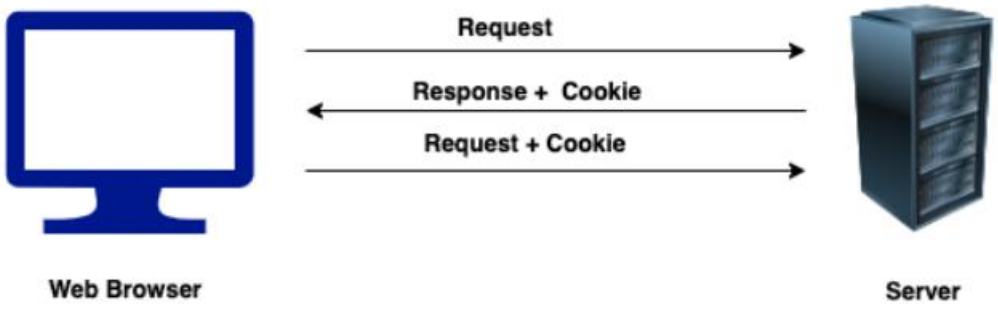
Mi az a post message?

A post message olyan módszer, amely lehetővé teszi a cross-origin kommunikációt a Window objektumok között (azaz egy oldal és egy általa létrehozott felugró ablak, vagy egy oldal és egy benne beágyazott iframe között). Általában a különböző oldalakon található szkriptek csak akkor férhetnek hozzá egymáshoz, ha az oldalak same-origin policy-t követnek (vagyis az oldalak ugyanazt a protokollt, portszámot és host-ot használják).

Mi a Cookie?

A cookie egy olyan adat, amelyet a számítógépe tárol, és amelyet a böngészője elérhet. A sütik kulcs / érték párokként kerülnek mentésre. Például létrehozhat egy felhasználónév nevű sütit az alábbiak szerint:

```
document.cookie = "username=John";
```



Miért van szükség Cookie-ra?

A cookie-kat arra használják, hogy emlékezzenek a felhasználói profilra vonatkozó információkra (például felhasználónév). Alapvetően két lépést tartalmaz,

1.) Amikor egy felhasználó meglátogat egy weboldalt, a felhasználói profil egy cookie-ban tárolható.

2.) Legközelebb, amikor a felhasználó meglátogatja az oldalt, a cookie megjegyzi a felhasználói profilt.

Milyen beállítások vannak egy cookie-nál?

Az alábbiakban kevés beállítás áll rendelkezésre egy sütire,

1.) Alapértelmezés szerint a böngésző bezárásakor a cookie törlődik, de ezt a viselkedést megváltoztathatja a lejárati dátum beállításával (UTC-idő szerint).

```
document.cookie = "username=John; expires=Sat, 8 Jun 2019 12:00:00 UTC";
```

2.) Alapértelmezés szerint a cookie egy aktuális oldalhoz tartozik. De megadhatja a böngészőnek, hogy a cookie milyen útvonalhoz tartozik egy elérési út paraméter használatával.

```
document.cookie = "username=John; path=/services";
```

Hogyan lehet törölni a cookie-kat?

Törölhet egy cookie-t, ha a lejárati dátumot lejárt dátumként állítja be. Ebben az esetben nem kell megadnia a cookie értékét. Például az alábbiak szerint törölhet egy felhasználónév sütit az aktuális oldalon.

```
document.cookie = "username=; expires=Fri, 07 Jun 2019 00:00:00 UTC; path=/;";
```

Megjegyzés: Meg kell határoznia a cookie elérési útját, hogy biztosítsa a megfelelő cookie törlését. Egyes böngészők csak akkor engedélyezik a cookie-k törlését, ha megad egy elérési utat.

Mi a különbség a cookie, local storage and session storage között?

Feature	Cookie	Local storage	Session storage
Hozzáférés kliens vagy szerver oldalon	Szerver és kliens oldal egyaránt	csak kliens oldalon	csak kliens oldalon
Élettartam	A Lejárat opcióval konfigurálva	törlésig	amíg a tab be nem zárul

Feature	Cookie	Local storage	Session storage
SSL támogatás	Támogatott	Nem támogatott	Nem támogatott
Maximális adatméret	4KB	5 MB	5MB

Hogyan érheti el a web storage-t?

A Window objektum végrehajtja a WindowLocalStorage és az WindowSessionStorage objektumokat, amelyek rendelkeznek localStorage (window.localStorage) és sessionStorage (window.sessionStorage) tulajdonságokkal. Ezek a tulajdonságok létrehozzák a Storage objektum egy példányát, amelyen keresztül adatelemeket lehet beállítani, lekérni és eltávolítani egy adott tartományhoz és tárolási típushoz (session or local). Például olvashat és írhat a local storage objektumokra az alábbiak szerint

```
localStorage.setItem('logo', document.getElementById('logo').value);
localStorage.getItem('logo');
```

Milyen metódusok érhetők el a session storage során?

adatainak olvasására, írására és törlésére:

```
// Save data to sessionStorage
sessionStorage.setItem('key', 'value');

// Get saved data from sessionStorage
let data = sessionStorage.getItem('key');

// Remove saved data from sessionStorage
sessionStorage.removeItem('key');

// Remove all saved data from sessionStorage
sessionStorage.clear();
```

Mi a StorageEvent és annak eseménykezelője?

A StorageEvent egy esemény, amely akkor aktiválódik, amikor egy tárterületet egy másik dokumentummal összefüggésben megváltoztattak. Míg az onstorage tulajdonság egy EventHandler a tárolási események feldolgozásához. A szintaxis az alábbiak szerint alakul

```
window.onstorage = functionRef;
```

Vegyük az onstorage eseménykezelő példáját, amely naplózza a storage key-t és annak értékeit

```
window.onstorage = function(e) {
  console.log('The ' + e.key +
    ' key has been changed from ' + e.oldValue +
    ' to ' + e.newValue + '.');
};
```

Miért van szükség web storage-re?

A web storage biztonságosabb, és nagy mennyiségű adat tárolható helyben, anélkül, hogy befolyásolná a webhely teljesítményét. Ezenkívül az információkat soha nem továbbítják a szerverre. Ezért ez egy ajánlottabb megközelítés, mint a sütik.

Hogyan ellenőrizheti a web storage-t támogatja-e a böngésző?

```
if (typeof(Storage) !== "undefined") {
  // Code for localStorage/sessionStorage.
} else {
  // Sorry! No Web Storage support..
}
```

Hogyan ellenőrizheti a web workers-t támogatja-e a böngésző?

```
if (typeof(Worker) !== "undefined") {
  // code for Web worker support.
} else {
  // Sorry! No Web Worker support..
}
```

Mondjon példát egy web worker-re?

1.) Hozzon létre egy Web Worker fájlt: Írnia kell egy szkriptet a számérték növeléséhez. Nevezzük meg counter.js néven

```
let i = 0;

function timedCount() {
  i = i + 1;
  postMessage(i);
  setTimeout("timedCount()",500);
}

timedCount();
```

Itt a postMessage () metódus segítségével üzenetet küldhetünk vissza a HTML oldalra

2.) Web worker-objektum létrehozása: Web worker -objektumot létrehozhat a böngésző támogatásának ellenőrzésével. Nevezzük ezt a fájlt web_worker_example.js néven

```
if (typeof(w) == "undefined") {
  w = new Worker("counter.js");
}
```

és fogadhatunk üzeneteket a web worker-től:

```
w.onmessage = function(event){
  document.getElementById("message").innerHTML = event.data;
};
```

3.) Web worker megszüntetése: A Web worker-ök mindaddig hallgatják az üzeneteket (a külső szkript befejezése után is), amíg le nem áll. A terminate () metódussal befejezheti az üzenetek meghallgatását.

```
w.terminate();
```

4.) A Web Worker újrafelhasználása: Ha a változót undefined-ra állítja, akkor újra felhasználhatja a kódot

```
w = undefined;
```

Mik a web worker korlátozásai a DOM-on?

A web worker-nek nincs hozzáférésük az alábbi javascript objektumokhoz, mivel azokat külső fájlokban definiálják:

- Window object
- Document object
- Parent object

Mi az a Promise?

A Promise olyan objektum, amely valamikor a jövőben egyetlen értéket hozhat létre resolved értékkel, vagy olyan okkal, amely nem resolved (például hálózati hiba). A 3 lehetséges állapot egyikében lesz: teljesítve, elutasítva vagy függőben (fulfilled, rejected, or pending).

Létrehozásának szintaxisa:

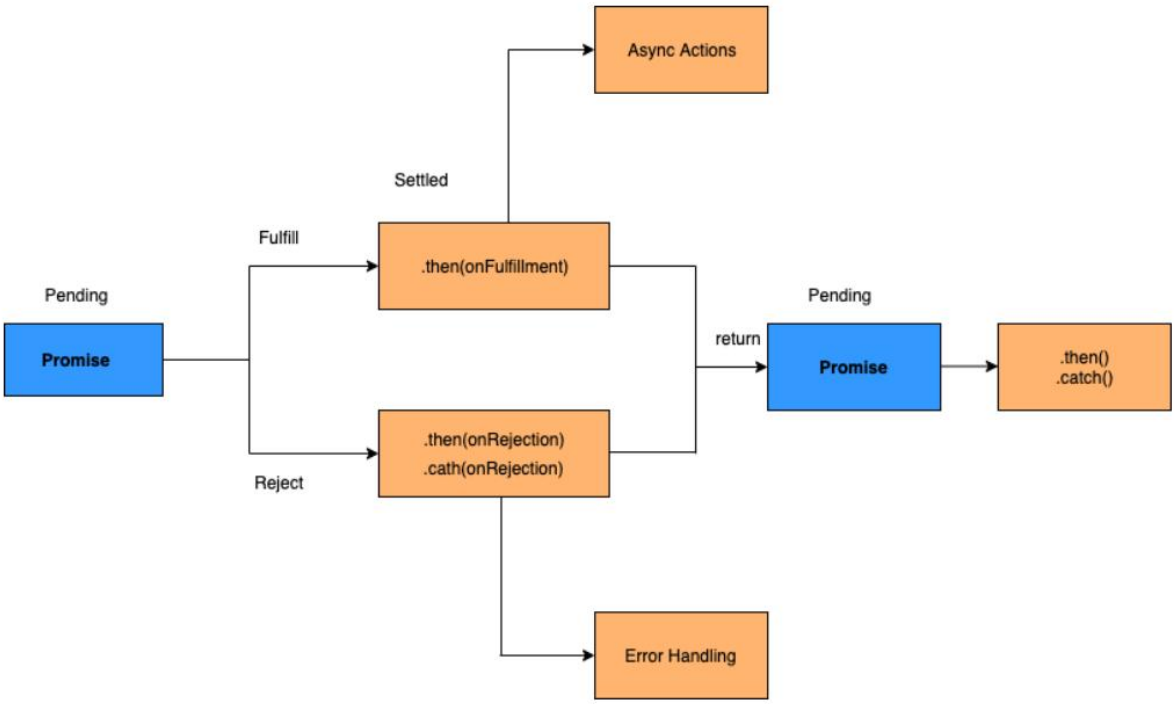
```
const promise = new Promise(function(resolve, reject) {
      // promise description
    })
```

Használata:

```
const promise = new Promise(resolve => {
  setTimeout(() => {
    resolve("I'm a Promise!");
  }, 5000);
}, reject => {

});

promise.then(value => console.log(value));
```



Miért van szükség Promise-ra?

A Promise-ok aszinkron műveletek kezelésére szolgálnak. Alternatív megközelítést kínálnak a callback-re azáltal, hogy csökkentik a callback hell-t és lehetővé teszik, hogy a kód tisztább legyen.

Mi a Promise három állapota?

Függőben (Pending): Ez a promise kezdeti állapota a művelet megkezdése előtt

Teljesítve (Fulfilled): Ez az állapot azt jelzi, hogy a megadott művelet befejeződött.

Elutasítva (Rejected): Ez az állapot azt jelzi, hogy a művelet nem fejeződött be. Ebben az esetben egy hibaérték dobódik.

Mi a callback function?

A callback function egy másik függvénybe argumentumként átadott függvény. Ezt a függvényt a külső függvényen belül hívjuk meg egy művelet befejezéséhez. Vegyünk egy egyszerű példát :

```
function callbackFunction(name) {
  console.log('Hello ' + name);
}

function outerFunction(callback) {
  let name = prompt('Please enter your name.');
  callback(name);
}
```

```
outerFunction(callbackFunction);
```

Miért van szükségünk callback-ekre?

A callback-ekre azért van szükség, mert a javascript eseményvezérelt nyelv. Ez azt jelenti, hogy a javascript a válasz várakozása helyett folytatja a végrehajtást, miközben más eseményeket hallgat. Vegyünk egy példát az első (first) függvényről, amely meghívja az API hívást (amelyet a setTimeout szimulál), és a következő függvénnyel, amely naplózza az üzenetet.

```
function firstFunction(){
  // Simulate a code delay
  setTimeout( function(){
    console.log('First function called');
  }, 1000 );
}
function secondFunction(){
  console.log('Second function called');
}
firstFunction();
secondFunction();

Output
// Second function called
// First function called
```

Amint az a kimenetből kiderült, a javascript nem várta meg az első függvény válaszát, és a fennmaradó kódblokk végrehajtásra került. Tehát a callback-eket arra használják, hogy bizonyos kódok ne hajtsanak végre addig, amíg a másik kód nem fejezi be a végrehajtást.

Mi a Callback Hell?

A Callback Hell egy anti-pattern, több beágyazott callback-el, ami megnehezíti a kód olvasását és hibakeresését az aszinkron logika kezelésekor. A callback hell így néz ki:

```
async1(function(){
    async2(function(){
        async3(function(){
            async4(function(){
                ....
            });
        });
    });
});
```

Mik azok a szerver által küldött események (server-sent events)?

A szerver által küldött események (server-sent events- SSE) egy olyan kiszolgálói push technológia, amely lehetővé teszi a böngésző számára, hogy automatikus frissítéseket kapjon a szerverről HTTP-kapcsolaton keresztül. Ezek egyirányú kommunikációs csatornák - az események csak szerverről kliensre folynak. Ezt használták a Facebook / Twitter frissítésekben, részvényárfolyam-frissítésekben, hírcsatornákban stb.

Hogyan fogadja a szerver által küldött események (server sent events) értesítéseit?

Az EventSource objektumot a szerver által küldött eseményértesítések fogadására használják. Például fogadhat üzeneteket a szerverről az alábbiak szerint:

```
if(typeof(EventSource) !== "undefined") {
  var source = new EventSource("sse_generator.js");
  source.onmessage = function(event) {
    document.getElementById("output").innerHTML += event.data + "<br>";
  };
}
```

Hogyan ellenőrizheti a szerver által küldött események (server sent events) böngészőjének támogatását?

A szerver által küldött események böngészőjének támogatását az alábbiak használata előtt végezheti el,

```
if(typeof(EventSource) !== "undefined") {
  // Server-sent events supported. Let's have some code here!
} else {
  // No server-sent events supported
}
```

Melyek a szerver által küldött események (server sent events) számára elérhető események?

Az alábbiakban felsoroljuk a szerver által küldött eseményekhez elérhető események listáját

Event	Description
onopen	A kiszolgálóval való kapcsolat megnyitásakor használják
onmessage	Ez az esemény akkor használható, amikor üzenet érkezik
onerror	Akkor történik, amikor hiba lép fel

Melyek a Promise fő szabályai?

A promise-oknak meghatározott szabályokat kell követnie,

- Az Promise olyan objektum, amely szabványnak megfelelő .then () metódust szolgáltat
- A függőben lévő Promise teljesített (fulfilled) vagy elutasított (rejected) állapotba kerülhet
- A teljesített vagy elutasított (rejected) Promise nem léphet át más állapotba.
- Az Promise teljesülése (fulfilled) után az érték nem változhat.

Mi a callback in callback (callback-ben callback)?

Az egyik callback hívásba beágyazhatja egy másik callback hívást, hogy egymás után hajtsa végre a műveleteket.

```
loadScript('/script1.js', function(script) {
   console.log('first script is loaded');

  loadScript('/script2.js', function(script) {

    console.log('second script is loaded');

    loadScript('/script3.js', function(script) {

        console.log('third script is loaded');
      // after all scripts are loaded
    });

  })

});
```

Mi az promise chaining (Promise láncolás)?

Az aszinkron feladatok egymás után történő ismert folyamata.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  console.log(result); // 1
  return result * 2;

}).then(function(result) {

  console.log(result); // 2
  return result * 3;

}).then(function(result) {

  console.log(result); // 6
  return result * 4;

});
```

A fenti kezelőkben az eredmény átkerül a .then () kezelők láncára az alábbi munkafolyamat mellett,

- A kezdeti Promise 1 másodperc alatt resolved lesz,
- Ezt követően .then handler-t hívjukaz eredmény (1) naplózásával, majd Promise-t adunk vissza az eredmény * 2 értékével.
- Ezt követően az érték átkerült a következő .then kezelőhöz az eredmény (2) naplózásával és egy Promise visszaadásával * 3 eredménnyel.
- Végül az utolsó .then handler átadott érték az eredmény naplózásával (6) és Promise visszaadásával * 4 eredménnyel.

promise.all?

promise.all olyan Promise, amely Promise-ok sokaságát veszi alapul (iterálható), és akkor less resolved, amikor az összes Promise resolved, vagy bármelyik rejected.

```
Promise.all([Promise1, Promise2, Promise3]) .then(result) => {   console.log(result) }) .catch(error
=> console.log(`Error in promises ${error}`))
```

Megjegyzés: Ne feledje, hogy az Promise sorrendje (az eredmény kiadása) megmarad a bemeneti sorrend szerint.

Promise.race () ?

Az Promise.race () metódus visszaadja a Promise példányát, amelyet először resolved vagy rejected lett. Vegyünk egy példát a race () módszerre, ahol a promise2 lesz először resolved.

```
var promise1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 500, 'one');
});
var promise2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then(function(value) {
  console.log(value); // "two" // Both promises will resolve, but promise2 is faster
});
```

Mi a stict mode a javascriptben?

A stict mode az ECMAScript 5 új szolgáltatása, amely lehetővé teszi egy program vagy egy funkció „szigorú” működési kontextusba helyezését. Így megakadályozza bizonyos cselekvések végrehajtását, és további kivételeket dob. A " use strict " szó szerinti kifejezést; utasítja a böngészőt, hogy a javascript kódot használja strict módban.

Miért van szükség strict módra?

A strict mód hasznos, ha a "biztonságos" JavaScriptet, hogy a "rossz szintaxist" valódi hibákba írja. Például kiküszöböli a globális változó véletlenszerű létrehozását hibával, és hibát dob egy nem módosítható tulajdonsághoz, egy getter-only tulajdonsághoz, egy nem létező tulajdonsághoz, egy nem létező változóhoz vagy egy nem létező objektumhoz.

Hogyan deklarálja a strict mode?

A strict mode a " use strict " hozzáadásával deklaráljuk; egy szkript vagy egy függvény elejére. Ha egy szkript elején deklaráljuk, globális hatókörrel rendelkezik.

```
"use strict";
x = 3.14; // This will cause an error because x is not declared
```

és ha egy függvényen belül, annak lokális hatóköre van:

```
x = 3.14;       // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

Mi a célja a kettős felkiáltójelnek?

A kettős felkiáltójels vagy tagadás (!!) biztosítja, hogy a kapott típus logikai boolean érték legyen. Ha hamis volt (pl. 0, null, undefined, stb.), Akkor hamis lesz, különben igaz. Például kipróbálhatja az IE verzió-ra ezzel a kifejezéssel, az alábbiak szerint:

```
let isIE8 = false;
isIE8 = !! navigator.userAgent.match(/MSIE 8.0/);
console.log(isIE8); // returns true or false
```

Ha nem ezt a kifejezést használja, akkor az az eredeti értéket adja vissza.

```
console.log(navigator.userAgent.match(/MSIE 8.0/));  // returns either an Array or null
```

Megjegyzés: A kifejezés !! nem operátor, de kétszerese a ! operátornak.

Mi a delete operátor célja?

A delete kulcsszó property törlésére szolgál.

```
var user= {name: "John", age:20};
delete user.age;

console.log(user); // {name: "John"}
```

Mi a typeof operator?

Visszaadja egy változó vagy egy kifejezés típusát.

```
typeof "John Abraham"     // Returns "string"
typeof (1 + 2)        // Returns "number"
```

Mi az undefined?

A undefined azt jelzi, hogy egy változóhoz nem rendeltek értéket, vagy egyáltalán nem deklarálták. A undefined érték típusa szintén undefined.

```
var user;    // Value is undefined, type is undefined
console.log(typeof(user)) //undefined
```

Bármely változó ki lehet üríteni az érték undefined beállításával.

```
user = undefined
```

Mi a null érték?

A null érték az objektumértékek szándékos hiányát jelenti. Ez a JavaScript egyik primitív értéke. A null érték típusa objektum. Üresítheti a változót, ha nullára állítja az értéket.

```
var user = null;
console.log(typeof(user)) //object
```

Mi a különbség a null és a undefined között?

Null	Undefined
Ez egy hozzárendelési érték, amely azt jelzi, hogy a változó nem mutat objektumra.	Ez nem egy hozzárendelési érték, ahol egy változót deklaráltunk, de még nem rendeltünk hozzá értéket.
A null típusa az object	A undefined típusa undefined
A null érték egy primitív érték, amely a null, üres vagy nem létező hivatkozást jelenti.	Az undefined érték egy primitív érték, amelyet akkor használnak, amikor egy változóhoz nincs hozzárendelve érték.
Jelzi a változó értékének hiányát	Maga a változó hiányát jelzi
Átalakítva nullára (0) primitív műveletek végrehajtása közben	Átalakítva NaN-ra primitív műveletek végrehajtása közben

Mi az eval?

Az eval () függvény kiértékeli a JavaScript karaktersorozatot. A karakterlánc lehet JavaScript-kifejezés, változó, utasítás vagy utasításssorozat.

```
console.log(eval('1 + 2')); //  3
```

Mi a különbség az window és document között?

Window	Document

Window	Document
Ez minden weboldal gyökérszintű eleme	Az window objektum közvetlen gyermeke. Ez más néven Document Object Model (DOM)
Alapértelmezés szerint az window objektum implicit módon elérhető az oldalon	Hozzáférhet a window.document vagy a document segítségével.
Olyan módszerekkel rendelkezik, mint a alert(), confirm() és a tulajdonságok, például a document, location	Olyan módszereket biztosít, mint a getElementById, getElementByTagName, createElement stb

Hogyan érheti el az előzményeket (history) a javascriptben?

Az window.history objektum tartalmazza a böngésző előzményeit (history). Az előző és a következő URL-eket az előzményekbe a back () és a forward () metódusokkal töltheti be.

```
function goBack() {
  window.history.back()
}
function goForward() {
  window.history.forward()
}
```

Megjegyzés: Az előzményeket (history) window előtag nélkül is elérheti.

Melyek a javascript adattípusok?

- Number
- String
- Boolean
- Object
- Undefined

Mi az isNaN?

Az isNaN () függvény segítségével meghatározható, hogy egy érték nem szám-e (Not-a-Number) vagy sem. vagyis ez a függvény igazzal tér vissza, ha az érték megegyezik a NaN értékkel. Ellenkező esetben hamis értéket ad vissza.

```
isNaN('Hello') //true
isNaN('100') //false
```

Mi a különbség az undeclared és undefined változók között?

undeclared	undefined
Ezek a változók nem léteznek egy programban, és nincsenek deklarálva	Ezek a változók deklaráltak a programban, de nem rendeltek hozzá értéket
Ha megpróbálja beolvasni egy undeclared változó értékét, futásidejű hiba lép fel	Ha megpróbálja olvasni egy undefined változó értékét, akkor egy undefined értéket ad vissza.

Mik a globális változók?

A globális változók azok, amelyek a kód teljes hosszában, hatókör nélkül elérhetők. A var kulcsszó egy helyi változó deklarálására szolgál, de ha kihagyja, akkor globális változóvá válik

```
msg = "Hello" // var is missing, it becomes global variable
```

Milyen problémák vannak a globális változókkal?

A globális változók problémája a lokális és globális hatókörű változónevek konfliktusa. A globális változókra támaszkodó kód hibakeresése és tesztelése is nehéz.

Mi a NaN tulajdonság?

A NaN tulajdonság globális tulajdonság, amely a "Not-a-Number" értéket képviseli, vagyis azt jelzi, hogy az érték nem szám. Nagyon ritka a NaN használata egy programban, de néhány esetben visszatérítési értékként használható

```
Math.sqrt(-1)
parseInt("Hello")
```

Mi a célja az isFinite függvénynek?

Az isFinite () függvény segítségével határozható meg, hogy egy szám véges-e. Hamis értéket ad vissza, ha az érték + végtelen,-végtelen vagy NaN (nem-szám), ellenkező esetben igaz.

```
isFinite(Infinity);  // false
isFinite(NaN);       // false
isFinite(-Infinity); // false

isFinite(100);         // true
```

Mi az event flow?

Az event flow az esemény fogadásának sorrendje a weboldalon. Ha olyan elemre kattint, amely több más elemben be van ágyazva, mielőtt a kattintása ténylegesen elérné célját vagy célelemét, akkor annak először ki kell váltania a kattintási eseményt az egyes szülőelemeihez, a tetejétől kezdve a globális window objektummal. Az eseményáramlásnak (event flow) két módja van:

i. Top to Bottom(Event Capturing)
ii. Bottom to Top (Event Bubbling)

Mi az Event Bubbling?

Az Event Bubbling az esemény terjedésének egy olyan típusa, amikor az esemény először a legbelső célelemre váltódik ki, majd egymás után kiváltja a célelem őseit (szüleit) ugyanazon beágyazott hierarchiában, amíg el nem éri a legkülső DOM elemet:

Mi az Event Capturing?

Az Event Capturing az esemény terjedésének egy olyan típusa, ahol az eseményt először a legkülső elem rögzíti, majd egymás után kiváltja a célelem leszármazottait (gyermekeit) ugyanazon egybeágyazott hierarchiában, amíg el nem éri a legbelső DOM elemet.

Hogyan küldhet (submit) form-ot JavaScript használatával?

Az form-ot küldhet (submit)a JavaScript használatával a document.form [0] .submit () használatával.

```
function submit() {
    document.form[0].submit();
}
```

Hogyan találja meg az operációs rendszer részleteit?

Az window.navigator objektum információkat tartalmaz a látogató böngészőjének operációs rendszeréről. Néhány operációs rendszer tulajdonság elérhető platform tulajdonság alatt,

```
console.log(navigator.platform);
```

Mi a különbség document load és a DOMContentLoaded események között?

A DOMContentLoaded esemény akkor indul el, amikor a kezdeti HTML dokumentum teljesen be van töltve és elemezve, anélkül, hogy megvárná az assets(stylesheets, images, and subframes) betöltésének befejezését. Míg a load esemény akkor indul el, amikor az egész oldal betöltődött, beleértve az összes dependent resources(stylesheets, images).

Mi a különbség native, host és user objects között?

A natív objektumok olyan objektumok, amelyek az ECMAScript specifikáció által meghatározott JavaScript nyelv részét képezik. Például String, Math, RegExp, Object, Function stb. Alapvető objektumok, amelyek az ECMAScript specifikációban vannak meghatározva. A host objektumok a böngésző vagy a futási környezet (Node) által biztosított objektumok. Például a window, az

XmlHttpRequest, a DOM csomópontok stb. A user objektumok a javascript kódban definiált objektumok. Például a profil információkhoz létrehozott felhasználói objektumok.

Milyen előnyei és hátrányai vannak a Promise-oknak a callback-el szemben?

Előnyök:

- Kerüli a callback hell-t, amely olvashatatlan
- Könnyű szekvenciális aszinkron kódot írni a .then () segítségével
- Könnyű párhuzamos aszinkron kódot írni a Promise.all () segítségével
- Megoldja a callback-ek néhány gyakori problémáját (túl későn, túl korán, sokszor hívja a callback-et és nem kapunk a hibákat / kivételeket)

Hátrányok:

- Kicsit nehezebb kódolni
- Ha az ES6 nem támogatott, töltsön be egy polifill-t

Mi a különbség az attribútum és a property között?

Az attribútumok a HTML markup-on, míg a tulajdonságok a DOM-on vannak meghatározva. Például az alábbi HTML elemnek 2 attribútumtípusa és értéke van,

```
<input type="text" value="Name:">
```

Az attribútum értékét az alábbiak szerint nyerhetjük ki:

```
const input = document.querySelector('input');
console.log(input.getAttribute('value')); // Good morning
console.log(input.value); // Good morning
```

És miután megváltoztatja a szövegmező értékét:

```
console.log(input.getAttribute('value')); // Good morning
console.log(input.value); // Good evening
```

Mi a same-origin policy?

Az same-origin policy olyan policy, amely megakadályozza, hogy a JavaScript a domain határain túl is kéréseket küldjön. Az eredetet az URI séma, a hosztnév és a portszám kombinációjaként definiáljuk. Ha engedélyezi ezt a policy-t, akkor megakadályozza, hogy arosszindulatú szkript hozzáférjen a bizalmas adatokhoz egy másik weboldalon a Document Object Model (DOM) használatával.

Mi az Void (0) célja?

A Void (0) az oldal frissítésének megakadályozására szolgál. Ez hasznos lehet a unwanted side-effect kiküszöbölésére, mert visszaadja a undefined primitív értéket. Általában olyan HTML-

dokumentumokhoz használják, az href = "JavaScript: Void (0);" egy <a> elemen belül. vagyis amikor egy linkre kattint, a böngésző új oldalt tölt be, vagy frissíti ugyanazt az oldalt. De ezt a viselkedést megakadályozzuk ezzel a kifejezéssel. Például az alábbi link üzenetet ad:

```
<a href="JavaScript:void(0);" onclick="alert('Well done!')">Click Me!</a>
```

compiled vagy interpreted nyelv a JavaScript?

A JavaScript interpretedt nyelv, nem compiled nyelv. A böngészőben egy interpreter felolvassa a JavaScript kódot, minden sort értelmez és futtat. Manapság a modern böngészők a Just-In-Time (JIT) fordítás néven ismert technológiát használják, amely a JavaScript-et futtatható bájtkódba fordítja, éppen futás közben.

Mik az események?

Az események olyan dolgok, amelyek a HTML elemekkel történnek. Ha a JavaScript-et HTML-oldalakon használják, a JavaScript reagálhat ezekre az eseményekre. Néhány példa a HTML eseményekre,

- A weboldal betöltődött
- A beviteli mező megváltozott
- A gombra kattintottak

Írjuk le a kattintási esemény viselkedését a gombelemnél:

```
<!doctype html>
<html>
 <head>
   <script>
     function greeting() {
       alert('Hello! Good morning');
     }
   </script>
 </head>
 <body>
   <button type="button" onclick="greeting()">Click me</button>
 </body>
</html>
```

Mi a preventDefault metódus használata?

A preventDefault () metódus törli az eseményt, ha törölhető, vagyis az eseményhez tartozó alapértelmezett művelet vagy viselkedés nem fog bekövetkezni. Például a beküldés gombra kattintva tiltsa le az űrlap elküldését, és a hiperhivatkozásra kattintva akadályozza meg az oldal URL megnyitását.

```
document.getElementById("link").addEventListener("click", function(event){
 event.preventDefault();
});
```

Megjegyzés: Ne feledje, hogy nem minden esemény törölhető.

Mi a stopPropagation metódus használata?

A stopPropagation metódust arra használjuk, hogy megakadályozzuk az esemény bubbling-ját (továbbterjedését) az eseményláncban. Például az alábbiakban beágyazott div-ek stopPropagation módszerrel megakadályozzák az alapértelmezett eseményterjedést, amikor a beágyazott div-re (Div1) kattintanak

```
<p>Click DIV1 Element</p>
<div onclick="secondFunc()">DIV 2
  <div onclick="firstFunc(event)">DIV 1</div>
</div>

<script>
function firstFunc(event) {
  alert("DIV 1");
  event.stopPropagation();
}

function secondFunc() {
  alert("DIV 2");
}
</script>
```

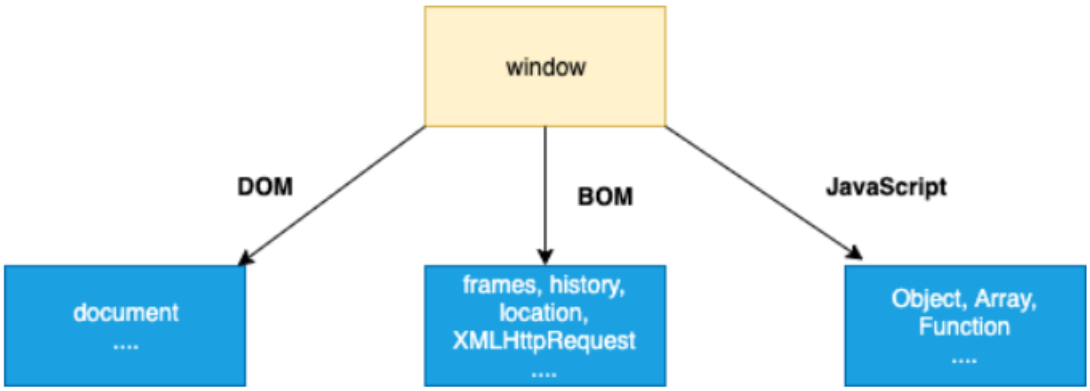
Milyen lépésekkel jár a return false használata?

Az eseménykezelőkben a return false utasítás végrehajtja az alábbi lépéseket,

- Először leállítja a böngésző alapértelmezett műveletét vagy viselkedését.
- Megakadályozza, hogy az esemény továbbterjessze a DOM-ot
- Leállítja a callback végrehajtását, és híváskor azonnal visszatér.

Mi a BOM?

A böngészőobjektum modell (BOM) lehetővé teszi a JavaScript számára, hogy "beszéljen" a böngészővel. A bjects navigator, history, screen, location and document-ből áll, amelyek a window gyermekei. A böngésző objektum modellje nem szabványosított, és különböző böngészők alapján változhat.



Mi a setTimeout használata?

A setTimeout () metódust egy függvény meghívására vagy egy kifejezés kiértékelésére használjuk meghatározott számú milliszekundum után. Például 2 másodperc múlva naplózzunk egy üzenetet a setTimeout módszerrel,

```
setTimeout(function(){ console.log("Good morning"); }, 2000);
```

Mi a setInterval használata?

A setInterval () metódust függvény meghívására vagy egy kifejezés kiértékelésére használjuk meghatározott időközönként (milliszekundumban). Például 2 másodperc múlva naplózunk egy üzenetet a setInterval módszerrel,

```
setInterval(function(){ console.log("Good morning"); }, 2000);
```

a JavaScriptet egyszálú nyelv?

A JavaScript egyszálú nyelv. Mivel a nyelvi specifikáció nem teszi lehetővé a programozó számára, hogy kódot írjon, így az fordító annak egyes részeit párhuzamosan futtathatja több szálban vagy folyamatban. Míg az olyan nyelvek, mint a java, go, a C ++ többszálú és több- processzű programokat készíthet.

Mi az event delegation?

Az event delegation olyan események meghallgatásának technikája, ahol szülői elemet delegál hallgatóként az összes eseményen belül, amely benne történik.

Például, ha egy adott űrlapon belül a mező változását akarta észlelni, használhatja az esemény delegálási technikáját,

```
var form = document.querySelector('#registration-form');

// Listen for changes to fields inside the form
```

```
form.addEventListener('input', function (event) {

// Log the field that was changed
console.log(event.target);

}, false);
```

Mi az ECMAScript?

Az ECMAScript a JavaScript alapját képező szkriptnyelv. Az ECMA Nemzetközi Szabványügyi Szervezet által az ECMA-262 és az ECMA-402 specifikációkban szabványosított ECMAScript. Az ECMAScript első kiadása 1997-ben jelent meg.

Mi a JSON?

A JSON (JavaScript Object Notation) egy lightweight format, amelyet adatcserére használnak. A JavaScript nyelvének egy részhalmazán alapul, mert az objektumok beépítetten vannak a JavaScript-ben.

Melyek a JSON szintaxis szabályai?

Az alábbiakban felsoroljuk a JSON szintaxis szabályainak listáját

- Az adatok név / érték párokban vannak megadva
- Az adatok vesszővel vannak elválasztva
- A {} zárójelben objektumok vannak
- A szögletes zárójelben tömbök találhatók

Mi a célja a JSON stringify-nak?

Amikor adatokat küld egy webszerverre, az adatoknak karakterlánc formátumban kell lenniük. Ezt úgy érheti el, hogy a JSON objektumot stringekké konvertálja a stringify () módszerrel.

```
var userJSON = {'name': 'John', age: 31}
var userString = JSON.stringify(user);
console.log(userString); //"{"name":"John","age":31}"
```

Hogyan elemzi a JSON karakterláncot?

Az adatok webszerverről történő fogadásakor az adatok mindig karakterlánc formátumban vannak. De ezt a karakterlánc-értéket konvertálhatja javascript objektummá a parse () módszerrel.

```
var userString = '{"name":"John","age":31}';
var userJSON = JSON.parse(userString);
console.log(userJSON);// {name: "John", age: 31}
```

Miért van szükség JSON-ra?

A böngésző és a szerver közötti adatcserénél az adatok csak szövegesek lehetnek. Mivel a JSON csak szöveges, könnyen elküldhető egy szerverről és egy szerverről, és bármilyen programozási nyelv adatformátumként használható.

Mik azok a PWA-k?

A progresszív webalkalmazások (PWA-k) egy olyan alkalmazás-típus, amelyet az interneten keresztül szállítanak, és amely általános webes technológiák, például HTML, CSS és JavaScript felhasználásával épül fel. Ezeket a PWA-kat szerverekre telepítik, URL-eken keresztül elérhetik, és a keresőmotorok indexelik.

Mi a clearTimeout módszer célja?

A clearTimeout () függvényt használják a javascriptben az idő törlésére, amelyet a setTimeout () függvény állított be előtte. vagyis a setTimeout () függvény visszatérési értéke egy változóban tárolódik, és az időzítő törléséhez átkerül a clearTimeout () függvénybe.

Például az alábbi setTimeout metódust használják az üzenet megjelenítésére 3 másodperc múlva. Ez az időtúllépés a clearTimeout () módszerrel törölhető.

```
<script>
var msg;
function greeting() {
   alert('Good morning');
}
function start() {
  msg =setTimeout(greeting, 3000);

}

function stop() {
    clearTimeout(msg);
}
</script>
```

Mi a clearInterval módszer célja?

A clearInterval () függvényt a javascriptben használják a setInterval () függvény által beállított intervallum törlésére. azaz a setInterval () függvény által visszaadott visszatérési érték egy változóban tárolódik, és az intervallum törléséhez átkerül a clearInterval () függvénybe.

Például az alábbi setInterval metódust használják az üzenet 3 másodpercenként történő megjelenítésére. Ez az intervallum a clearInterval () módszerrel törölhető

```
<script>
var msg;
function greeting() {
   alert('Good morning');
}
function start() {
  msg = setInterval(greeting, 3000);

}
```

```
function stop() {
    clearInterval(msg);
}
</script>
```

Hogyan ellenőrizheti, hogy egy string tartalmaz-e részstringet?

Háromféle módon ellenőrizhetjük, hogy egy string tartalmaz-e részstringet vagy sem,

1.) Using includes: ES6-ban haszálhatuk a String.prototype.includes módszerz, ami tesztel egy karakterláncot, hogy tartalmaz egy részsztringet

```
var mainString = "hello", subString = "hell";
mainString.includes(subString)
```

2.) Az indexOf használata: ES5 vagy régebbi környezetben használhatja a String.prototype.indexOf fájlt, amely a részszöveg indexét adja vissza. Ha az index értéke nem egyenlő -1, ez azt jelenti, hogy a részszöveg szerepel a fő karakterláncban.

```
var mainString = "hello", subString = "hell";
mainString.indexOf(subString) !== -1
```

3.) RegEx használata: A fejlett megoldás a reguláris kifejezés tesztmódszerét (RegExp.test) használja

```
var mainString = "hello", regex = "/hell/";
regex.test(mainString)
```

Hogyan kérheti le az aktuális URL-t javascript-tel?

Használhatja a window.location.href kifejezést az aktuális URL elérési útjának megszerzéséhez, és ugyanazt a kifejezést használhatja az URL frissítéséhez is.

```
console.log('location.href', window.location.href); // Returns full URL
```

url properties of location object?

- href - A teljes URL
- protokoll - Az URL protokollja
- host - The hostname and port of the URL
- hostname - The hostname of the URL
- port - The port number in the URL
- pathname - The path name of the URL
- search - The query portion of the URL
- hash - The anchor portion of the URL

Hogyan lehet lekérdezni a query string values-t a javascriptben?

Az URLSearchParams segítségével

```
const urlParams = new URLSearchParams(window.location.search);
const clientCode = urlParams.get('clientCode');
```

Hogyan ellenőrizheti, hogy létezik-e kulcs az objektumban?

Három megközelítéssel ellenőrizheti, hogy létezik-e kulcs az objektumban, vagy sem,

1.) Using in operator: Használhatja az in operátorban, hogy létezik-e kulcs az objektumban, vagy sem

```
"key" in obj
```

és ha ellenőrizni szeretné, hogy nem létezik-e kulcs, ne felejtse el zárójeleket használni,

```
!("key" in obj)
```

2.) A hasOwnProperty használata: A hasOwnProperty használatával különösen tesztelheti az objektumpéldány tulajdonságait (és nem öröklött tulajdonságokat)

```
obj.hasOwnProperty("key") // true
```

3.) undefined comparison: Ha egy nem létező tulajdonsághoz fér hozzá egy objektumtól, az eredmény undefined.

```
const user = {
  name: 'John'
};

console.log(user.name !== undefined);     // true
console.log(user.nickName !== undefined); // false
```

Hogyan lehet végigiterálni a javascript objektumon?

Használhatja a for-in ciklust a javascript objektumon való iteráláshoz. Azt is ellenőrizheti, hogy a kapott kulcs egy objektum tényleges tulajdonsága, és nem a hasOwnProperty módszerrel származik-e a prototípusból.

```
var object = {
    "k1": "value1",
    "k2": "value2",
    "k3": "value3"
};

for (var key in object) {
    if (object.hasOwnProperty(key)) {
        console.log(key + " -> " + object[key]); // k1 -> value1 ...
    }
}
```

Hogyan nézi meg, hogy üres-e az objektum?

Különböző megoldások léteznek az ECMAScript verziók alapján.

1.) Using Object entries(ECMA 7+): Az objektum bejegyzések hosszát a konstruktor típusával együtt használhatja.

```
Object.entries(obj).length === 0 && obj.constructor === Object // Since date object length is 0, you need to check constructor check as well
```

2.) Using Object keys(ECMA 5+): Használhatja az objektumkulcsok hosszát a konstruktor típusával együtt.

```
Object.keys(obj).length === 0 && obj.constructor === Object // Since date object length is 0, you need to check constructor check as well
```

3.) For-in használata a hasOwnProperty (Pre-ECMA 5) használatával:

```
function isEmpty(obj) {
  for(var prop in obj) {
    if(obj.hasOwnProperty(prop)) {
      return false;
    }
  }

  return JSON.stringify(obj) === JSON.stringify({});
}
```

Mi az argumentum objektum?

Az argumentum objektum egy tömbszerű objektum, amely elérhető a függvényeken belül, és tartalmazza az adott függvénynek átadott argumentumok értékeit.

```
function sum() {
    var total = 0;
    for (var i = 0, len = arguments.length; i < len; ++i) {
        total += arguments[i];
    }
    return total;
}

sum(1, 2, 3) // returns 6
```

Hogyan hasonlíthat össze két dátumobjektumot?

A date.getTime () metódust kell használnia a dátumértékek összehasonlításához az összehasonlító operátorok helyett (==,! =, === és! == operátorok)

```
var d1 = new Date();
var d2 = new Date(d1);
console.log(d1.getTime() === d2.getTime()); //True
console.log(d1 === d2); // False
```

Hogyan adhat hozzá egy kulcsértéket a javascriptben?

Kétféle megoldás lehetséges új tulajdonságok hozzáadásához egy objektumhoz.

```
var object = {
    key1: value1,
    key2: value2
```

```
};
```

1.) dot notation: Ez a megoldás akkor hasznos, ha ismeri a tulajdonság nevét

```
object.key3 = "value3";
```

2.) Szögletes zárójeles jelölés használata: Ez a megoldás akkor hasznos, ha a tulajdonság nevét dinamikusan meghatározzák.

```
obj["key3"] = "value3";
```

Hogyan rendelhet alapértelmezett értékeket a változókhoz?

Használhatja a logikai vagy az operátort hozzárendelési kifejezésben az alapértelmezett érték megadásához. A szintaxis az alábbiak szerint néz ki,

```
var a = b || c;
```

A fenti kifejezés szerint az „a” változó csak akkor kapja meg a „c” értékét, ha a „b” hamis (ha null, hamis, undefined, 0, üres karakterlánc vagy NaN), különben az „a” megkapja a „b” értéket.

Hogyan definiálja a többsoros karakterláncokat?

Meghatározhat többsoros karakterláncokat a '' karakterrel, majd a sorvégzővel.

```
var str = "This is a \
very lengthy \
sentence!";
```

De ha van szóköz a '' karakter után, akkor a kód pontosan ugyanúgy fog kinézni, de egy SyntaxError-t kapunk.

Mi az app shell model?

Az application shell (or app shell) architecture az egyik módja annak, hogy olyan Progresszív Webalkalmazást készítsen, amely megbízhatóan és azonnal betöltődik a felhasználói képernyőkön, hasonlóan a natív alkalmazásokban látottakhoz. Hasznos, ha egy kezdeti HTML-t hálózat nélkül gyorsan a képernyőre juttat.

Meghatározhatjuk a függvények tulajdonságait?

Igen, Megadhatjuk a függvények tulajdonságait, mert a függvények is objektumok.

```
fn = function(x) {
```

```
   //Function code goes here
}

fn.name = "John";

fn.profile = function(y) {
  //Profile code goes here
}
```

Hogyan lehet megtalálni a függvény által várt paraméterek számát?

A function.length szintaxissal megkeresheti a függvény által várt paraméterek számát. Vegyünk egy példát az összeg függvényre a számok összegének kiszámításához,

```
function sum(num1, num2, num3, num4){
    return num1 + num2 + num3 + num4;
}
sum.length // 4 is the number of parameters expected.
```

Mi az a polyfill?

A polyfill egy olyan JS-kóddarab, amelyet modern funkciók biztosítására használnak a régebbi böngészőkben, amelyek natív módon nem támogatják.

Mik azok a js címkék?

A label utasítás lehetővé teszi számunkra a ciklusok és blokkok elnevezését a JavaScript-ben. Ezután felhasználhatjuk ezeket a címkéket arra, hogy később a kódra hivatkozhassunk. Például az alábbi címkével ellátott kód elkerüli a számok kiírását, ha azonosak,

```
ar i, j;

loop1:
for (i = 0; i < 3; i++) {
   loop2:
   for (j = 0; j < 3; j++) {
      if (i === j) {
         continue loop1;
      }
      console.log('i = ' + i + ', j = ' + j);
   }
}

// Output is:
//   "i = 1, j = 0"
//   "i = 2, j = 0"
//   "i = 2, j = 1"
```

Milyen előnyökkel jár a deklarációk legfelső szinten tartása?

Javasoljuk, hogy minden deklarációt minden szkript vagy függvény tetején tartson. Ennek az az előnye,

- Tisztább kódot ad
- Egyetlen helyet biztosít a helyi változók kereséséhez
- Könnyen elkerülhető a nem kívánt globális változók
- Csökkenti a nem kívánt újbóli deklarációk lehetőségét

Milyen előnyei vannak a változók inicializálásának?

Javasoljuk a változók inicializálását az alábbi előnyök miatt,

- Tisztább kódot ad
- Egyetlen helyet biztosít a változók inicializálásához
- Elkerüli az undefined értékeket a kódban

Melyek az új objektum létrehozásának ajánlásai?

Javasoljuk, hogy kerülje az új objektumok létrehozását az new Object () használatával. Ehelyett inicializálhatja az értékeket a típusa alapján az objektumok létrehozásához.

- Assign {} instead of new Object()
- Assign "" instead of new String()
- Assign 0 instead of new Number()
- Assign false instead of new Boolean()
- Assign [] instead of new Array()
- Assign /()/ instead of new RegExp()
- Assign function (){} instead of new Function()

```
var v1 = {};
var v2 = "";
var v3 = 0;
var v4 = false;
var v5 = [];
var v6 = /()/;
var v7 = function(){};
```

Mi a tree shaking?

A tree shaking a holt kód megszüntetésének egyik formája. Ez azt jelenti, hogy a fel nem használt modulok nem lesznek benne a csomagban a készítés során, és ez az ES2015 modul szintaxisának statikus struktúrájára (azaz importálásra és exportra) támaszkodik. Kezdetben ezt népszerűsítette az ES2015 modul kötegelt összesítése.

Mi szükség van tree shaking-re?

A Tree Shaking jelentősen csökkentheti a kód méretét bármely alkalmazásban. vagyis minél kevesebb kódot küldünk a, annál jobban teljesít az alkalmazás. Például, ha csak egy „Hello World” alkalmazást akarunk létrehozni SPA-keretrendszer segítségével, akkor ez csak néhány 100 kb. A Tree shaking a Rollup és a Webpack csomagban valósul meg.

Javasoljuk-e az eval használatát?

Nem, lehetővé teszi tetszőleges kód futtatását, amely biztonsági problémát okoz. Mint tudjuk, hogy az eval () függvényt a szöveg futtatására használják kódként. Az esetek többségében nem szükséges használni.

Mi az a reguláris kifejezés?

A reguláris kifejezés olyan karaktersorozat, amely keresési mintát képez. Ezt a keresési mintát használhatja adatok szöveges keresésére. Ezekkel minden típusú szöveges keresés és szövegcsere művelet elvégezhető. Lássuk most a szintaxis formátumát:

```
/pattern/modifiers;
```

kis- és nagybetűkkel nem rendelkező felhasználónévvel a következő lenne:

```
/John/i
```

Milyen sztring módszerek érhetők el a reguláris kifejezésben?

A reguláris kifejezéseknek két karakterlánc-metódusuk van: search() és replace(). A search () metódus kifejezés segítségével keres egyezést, és visszaadja az egyezés helyzetét.

```
var msg = "Hello John";
var n = msg.search(/John/i); // 6
```

A replace() metódus egy módosított karakterlánc visszaküldésére szolgál, ahol a minta kicserélődik.

```
var msg = "Hello John";
var n = msg.replace(/John/i, "Buttler"); // Hello Buttler
```

Mik a módosítók a reguláris kifejezésben?

A módosítók case insensitive és globális keresések végrehajtására használhatók. Soroljunk fel néhány módosítót:

Modifier	**Description**
i	Perform case-insensitive matching
g	Perform a global match rather than stops at first match
m	Perform multiline matching

```
var text = "Learn JS one by one";
var pattern = /one/g;
```

```
var result = text.match(pattern); // one,one
```

Mik a szabályos kifejezési minták?

A reguláris kifejezések minták egy csoportját adják meg a karakterek egyeztetése érdekében. Alapvetően 3 típusba sorolják őket,

- Zárójelek: Ezeket a karakterek tartományának megkeresésére használják. Például az alábbiakban bemutatunk néhány felhasználási esetet,
    - [abc]: A zárójelek közötti karakterek bármelyikének megkeresésére szolgál (a, b, c)
    - [0-9]: A zárójelek közötti számjegyek bármelyikének megkeresésére szolgál
    - (a | b): A | -vel elválasztott alternatívák bármelyikének megtalálásához használatos
- Metakarakterek: Ezek különleges jelentéssel rendelkező karakterek. Például az alábbiakban bemutatunk néhány felhasználási esetet,
    - \ d: Számjegy keresésére szolgál
    - \ s: white space karakter keresésére szolgál
    - \ b: Megtalálható egyezés a szó elején vagy végén
- Kvantorok: Ezek hasznosak a mennyiségek meghatározásához. Például néhány felhasználási eset,
    - n +: Bármely karaktersorozat találatainak keresésére, amely legalább egy n-t tartalmaz
    - n *: Minden nulla vagy annál több előfordulást tartalmazó karakterlánc találatainak keresésére szolgál
    - n ?: Minden olyan karaktersorozat találatainak keresésére, amely nulla vagy n előfordulást tartalmaz

Mi az a RegExp objektum?

A RegExp objektum egy reguláris kifejezés objektum, előre meghatározott tulajdonságokkal és módszerekkel. Nézzük meg a RegExp objektum egyszerű használatát,

```
var regexp = new RegExp('\\w+');
console.log(regexp);
// expected output: /\w+/
```

Hogyan kereshet egy karakterláncban egy mintát?

A test () metódust használhatja a reguláris kifejezés metódusának keresésére egy karakterláncban, és az eredménytől függően igaz vagy hamis értéket adhat vissza.

```
var pattern = /you/;
console.log(pattern.test("How are you?")); //true
```

Mi az exec módszer célja?

Egy megadott karakterláncban keres egyezést, és az true / false érték visszaadása helyett eredménytömböt, vagy null-t ad vissza.

```
var pattern = /you/;
console.log(pattern.exec("How are you?")); //["you", index: 8, input: "How are you?", groups:
undefined]
```

Hogyan készítsünk szinkron HTTP kérést?

A böngészők biztosítanak egy XMLHttpRequest objektumot, amely felhasználható szinkron HTTP kérések készítésére JavaScript-ből

```
function httpGet(theUrl)
{
    var xmlHttpReq = new XMLHttpRequest();
    xmlHttpReq.open( "GET", theUrl, false ); // false for synchronous request
    xmlHttpReq.send( null );
    return xmlHttpReq.responseText;
}
```

Hogyan készítsen aszinkron HTTP kérést?

A böngészők biztosítanak egy XMLHttpRequest objektumot, amellyel aszinkron HTTP kéréseket lehet készíteni a JavaScript-ből a 3. paraméter igazként való átadásával.

```
function httpGetAsync(theUrl, callback)
{
    var xmlHttpReq = new XMLHttpRequest();
    xmlHttpReq.onreadystatechange = function() {
        if (xmlHttpReq.readyState == 4 && xmlHttpReq.status == 200)
            callback(xmlHttpReq.responseText);
    }
    xmlHttp.open("GET", theUrl, true); // true for asynchronous
    xmlHttp.send(null);
}
```

Hogyan konvertálhatja a dátumot egy másik időzónára a javascriptben?

A toLocaleString () metódus segítségével konvertálhat egy dátumot egy időzónából egy másikba. Például konvertáljuk az aktuális dátumot angol időzónára az alábbiak szerint,

```
console.log(event.toLocaleString('en-GB', { timeZone: 'UTC' })); //29/06/2019, 09:56:00
```

Milyen módon lehet a javascriptet futtatni az oldal betöltése után?

i. **window.onload:**

```
window.onload = function ...
```

i. **document.onload:**

```
document.onload = function ...
```

i. **body onload:**

```
<body onload="script();">
```

Mi a freeze method?

A freeze () metódust egy objektum lefagyasztására használják. Az objektum lefagyasztása nem teszi lehetővé új tulajdonságok hozzáadását egy objektumhoz, megakadályozza az eltávolítást, és megakadályozza a meglévő tulajdonságok, konfigurálhatóságának vagy írhatóságának megváltoztatását.

```
const obj = {
  prop: 100
};

Object.freeze(obj);
obj.prop = 200; // Throws an error in strict mode

console.log(obj.prop); //100
```

Mi a freeze módszer célja?

- Objektumok és tömbök fagyasztására szolgál.
- Arra használják, hogy egy tárgy változhatatlanná váljon.

Miért kell freeze módszert használnom?

Az Objektumorientált paradigmában egy meglévő API bizonyos elemeket tartalmaz, amelyeket nem kívánnak kiterjeszteni, módosítani vagy újrafelhasználni a jelenlegi környezetükön kívül. Ezért ez a final kulcsszóként működik, amelyet különféle nyelveken használnak.

Milyen különféle operátorokat támogat a javascript?

Az operátor képes egy bizonyos érték vagy operandus manipulálására (matematikai és logikai számítások). Különböző operátorokat támogat a JavaScript, az alábbiak szerint:

i. **Arithmetic Operators:** Includes + (Addition),– (Subtraction), * (Multiplication), / (Division), % (Modulus), + + (Increment) and – – (Decrement)
ii. **Comparison Operators:** Includes = =(Equal),!= (Not Equal), ===(Equal with type), > (Greater than),> = (Greater than or Equal to),< (Less than),<= (Less than or Equal to)
iii. **Logical Operators:** Includes &&(Logical AND),||(Logical OR),!(Logical NOT)
iv. **Assignment Operators:** Includes = (Assignment Operator), += (Add and Assignment Operator), – = (Subtract and Assignment Operator), *= (Multiply and Assignment), /= (Divide and Assignment), %= (Modules and Assignment)
v. **Ternary Operators:** It includes conditional(: ?) Operator
vi. **typeof Operator:** It uses to find type of variable. The syntax looks like `typeof variable`

Mi a Rest parameter?

A Rest paraméter a függvény paraméterek kezelésének továbbfejlesztett módja, amely lehetővé teszi számunkra, hogy határozatlan számú argumentumot jelenítsünk meg tömbként. A szintaxis az alábbiak szerint alakul:

```
function f(a, b, ...theArgs) {
  // ...
}
```

Vegyünk például egy összegző példát a paraméterek dinamikus számának kiszámításához,

```
function total(…args){
let sum = 0;
for(let i of args){
sum+=i;
}
return sum;
}
console.log(fun(1,2)); //3
console.log(fun(1,2,3)); //6
console.log(fun(1,2,3,4)); //13
console.log(fun(1,2,3,4,5)); //15
```

Mi történik, ha a rest parameter nem utolsó argumentumként használja?

A rest paraméter legyen az utolsó argumentum, mivel feladata az összes többi argumentum tömbbe gyűjtése. Például, ha definiál egy függvényt, mint az alábbiak, annak semmi értelme, és hibát fog dobni.

```
function someFunc(a,…b,c){
//You code goes here
return;
}
```

Melyek a javascriptben elérhető bitenkénti operátorok?

Az alábbiakban felsoroljuk a JavaScriptben használt bitenkénti logikai operátorokat

i. Bitwise AND (&)
ii. Bitwise OR (|)
iii. Bitwise XOR (^)
iv. Bitwise NOT (~)
v. Left Shift (<<)
vi. Sign Propagating Right Shift (>>)
vii. Zero fill Right Shift (>>>)

Mi a spread operator?

A Spread operátor lehetővé teszi az iterable (tömbök / objektumok / karakterláncok) kibővítését egyetlen argumentumra / elemre. Vegyünk egy példát, hogy lássuk ezt a viselkedést

```
function calculateSum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];

console.log(calculateSum(...numbers)); // 6
```

Hogyan állapíthatja meg, hogy az objektum frozen-e vagy sem?

Az Object.isFrozen () metódust annak megállapítására használják, hogy egy objektum frozen-e vagy sem. Az objektum frozen, ha az alábbi feltételek mindegyike teljesül,

- Ha nem bővíthető.
- Ha az összes tulajdonsága nem konfigurálható.
- Ha az összes adattulajdonsága nem írható. A felhasználás a következő lesz,

```
const object = {
   property: 'Welcome JS world'
};
Object.freeze(object);
console.log(Object.isFrozen(object));
```

Hogyan határozhatja meg azt, hogy két érték azonos vagy sem object-et használva?

Az Object.is () módszer meghatározza, hogy két érték azonos-e. Például a különböző típusú értékek használata:

```
Object.is('hello', 'hello');     // true
Object.is(window, window);   // true
Object.is([], []) // false
```

Két érték megegyezik, ha az alábbiak egyike érvényes:

i. both undefined
ii. both null
iii. both true or both false
iv. both strings of the same length with the same characters in the same order
v. both the same object (means both object have same reference)
vi. both numbers and both +0 both -0 both NaN both non-zero and both not NaN and both have the same value.

Hogyan másolja a tulajdonságokat az egyik objektumból a másikba?

Használhatja az Object.assign () metódust, amely az értékek és tulajdonságok egy vagy több forrásobjektumból a célobjektumba másolására szolgál. Visszaadja a célobjektumot, amelynek tulajdonságai és értékei át vannak másolva a célobjektumból. A szintaxis az alábbiak szerint alakul,

```
Object.assign(target, ...sources)
```

Vegyünk példát egy forrás és cél objektumról:

```
const target = { a: 1, b: 2 };
const source = { b: 3, c: 4 };

const returnedTarget = Object.assign(target, source);

console.log(target); // { a: 1, b: 3, c: 4 }

console.log(returnedTarget); // { a: 1, b: 3, c: 4 }
```

Amint az a fenti kódban megfigyelhető, van egy közös tulajdonság.

Melyek a assign metódus alkalmazásai?

Az alábbiakban bemutatjuk az Object.assign () metódus néhány fő alkalmazását,

- Egy objektum klónozására szolgál.
- Ugyanazokkal a tulajdonságokkal rendelkező objektumok egyesítésére szolgál.

Mi az a proxy objektum?

A Proxy objektum az alapvető műveletek egyedi viselkedésének meghatározására szolgál, mint például tulajdonságkeresés, hozzárendelés, felsorolás, függvényhívás stb. A szintaxis a következő,

```
var p = new Proxy(target, handler);
```

Vegyünk egy példát a proxy objektumra,

```
var handler = {
    get: function(obj, prop) {
        return prop in obj ?
            obj[prop] :
            100;
    }
};

var p = new Proxy({}, handler);
p.a = 10;
p.b = null;

console.log(p.a, p.b); // 10, null
console.log('c' in p, p.c); // false, 100
```

A fenti kódban a get handlert használja, amely meghatározza a proxy viselkedését, amikor egy műveletet végrehajtanak rajta

Mi a seal célja?

Az Object.seal () metódust egy objektum lezárására használják, megakadályozva új tulajdonságok hozzáadását. De a jelenlegi tulajdonságok értékei mindaddig módosítható. Lássuk az alábbi példát, hogy többet megtudjunk a seal () módszerről

```
const object = {
    property: 'Welcome JS world'
 };
 Object.seal(object);
 object.property = 'Welcome to object world';
 console.log(Object.isSealed(object)); // true
 delete object.property; // You cannot delete when sealed
 console.log(object.property); //Welcome to object world
```

Melyek a seal alkalmazásai?

Az alábbiakban bemutatjuk az Object.seal () metódus fő alkalmazásait,

- Objektumok és tömbök lezárására szolgál.

Milyen különbségek vannak a freeze és a seal módszerek között?

Ha egy objektum freeze az Object.freeze () módszerrel, akkor tulajdonságai megváltoztathatatlanná válnak, és azokon nem lehet változtatni, míg ha egy objektumot az Object.seal () metódussal seal, akkor a meglévő tulajdonságokban változtatni lehet az objektum.

Hogyan állapíthatja meg, hogy egy objektum sealed vagy sem?

Az Object.isSealed () metódust használjuk. Az objektum sealed, ha az alábbi feltételek mindegyike teljesül

- Ha nem bővíthető.
- Ha nem törölhető (de nem feltétlenül nem írható).

```
const object = {
property: 'Hello, Good morning'
};

Object.seal(object); // Using seal() method to seal the object

console.log(Object.isSealed(object));      // checking whether the object is sealed or not
```

Hogyan lehet enumerable kulcs- és értékpárot kapni?

Az Object.entries () metódus segítségével az adott objektum saját megszámlálható karakterlánc-kulcsú tulajdonság [kulcs, érték] párjainak tömbjét adja vissza, ugyanabban a sorrendben, mint amit a for ... in ciklus biztosít. Nézzük meg az object.entries () metódus működését egy példában,

```
const object = {
  a: 'Good morning',
  b: 100
};

for (let [key, value] of Object.entries(object)) {
  console.log(`${key}: ${value}`); // a: 'Good morning'
                                   // b: 100
}
```

Mi a fő különbség az Object.values és az Object.entries módszer között?

Az Object.values () módszer viselkedése hasonló az Object.entries () metódushoz, de a [kulcs, érték] párok helyett értéktömböt ad vissza.

```
const object = {
   a: 'Good morning',
   b: 100
 };

 for (let value of Object.values(object)) {
   console.log(`${value}`); // 'Good morning' 100
 }
```

Hogyan lehet megszerezni az objektumok kulcslistáját?

Használhatja az Object.keys () metódust, amelyet egy adott objektum saját tulajdonságneveinek tömbjének visszaadására használunk, ugyanabban a sorrendben, mint egy normál ciklus esetén. Például megkaphatja egy felhasználói objektum kulcsait:

```
const user = {
  name: 'John',
  gender: 'male',
  age: 40
};

console.log(Object.keys(user)); //['name', 'gender', 'age']
```

Hogyan hozhat létre objektumot prototípussal?

Az Object.create () metódussal új objektumot lehet létrehozni a megadott prototípus objektummal és tulajdonságokkal, vagyis egy meglévő objektumot használ az újonnan létrehozott objektum prototípusaként. Új objektumot ad vissza a megadott prototípus objektummal és tulajdonságokkal.

```
const user = {
   name: 'John',
   printInfo: function () {
     console.log(`My name is ${this.name}.`);
   }
 };

 const admin = Object.create(user);

 admin.name = "Nick"; // Remember that "name" is a property set on "admin" but not on "user" object

 admin.printInfo(); // My name is Nick
```

Mi az a WeakSet?

A WeakSet gyengén (gyenge referenciákkal) tárolt objektumok gyűjteményének tárolására szolgál. A szintaxis a következő:

```
new WeakSet([iterable]);
```

Nézzük meg az alábbi példát a viselkedés magyarázatára,

```
var ws = new WeakSet();
var user = {};
ws.add(user);
ws.has(user);    // true
ws.delete(user); // removes user from the set
ws.has(user);    // false, user has been removed
```

Mi a különbség a WeakSet és a Set között?

A fő különbség az, hogy a Set objektumokra való hivatkozások erősek, míg a WeakSet objektumokra való hivatkozások gyengék. azaz egy objektum a WeakSet-ben a garbage collector törölheti, ha nincs rá már referencia. Egyéb különbségek:

- A Set-ek bármilyen értéket tárolhatnak, míg a WeakSets csak az objektumok gyűjteményeit tárolhatja
- A WeakSet nem rendelkezik méret tulajdonsággal, ellentétben a Set-el
- A WeakSet nem rendelkezik olyan módszerekkel, mint például clear, keys, values, entries, forEach.
- A WeakSet nem iterálható.

Sorolja fel a WeakSet-en elérhető módszerek gyűjteményét

- add (value): Új objektumot adunk a megadott értékkel a weaksethez
- delete(value): Törli az értéket a WeakSet gyűjteményből.
- has (value): true értéket ad vissza, ha az érték szerepel a WeakSet gyűjteményben, különben hamis értéket ad vissza.
- length (): Visszaadja a silpSetObject hosszát.

```
var weakSetObject = new WeakSet();
var firstObject = {};
var secondObject = {};
// add(value)
weakSetObject.add(firstObject);
weakSetObject.add(secondObject);
console.log(weakSetObject.has(firstObject)); //true
console.log(weakSetObject.length()); //2
weakSetObject.delete(secondObject);
```

Mi az a WeakMap?

A WeakMap objektum kulcs / érték párok gyűjteménye, amelyben a kulcsokra gyengén hivatkoznak. Ebben az esetben a kulcsoknak objektumoknak kell lenniük, és az értékek tetszőleges értékek lehetnek. A szintaxis az alábbiak szerint néz ki,

```
new WeakMap([iterable])
```

Nézzük meg az alábbi példát a viselkedés magyarázatára:

```
var ws = new WeakMap();
 var user = {};
 ws.set(user);
 ws.has(user);    // true
 ws.delete(user); // removes user from the map
 ws.has(user);    // false, user has been removed
```

Mi a különbség a WeakMap és a Map között?

A fő különbség az, hogy a Map kulcsobjektumaira történő hivatkozások erősek, míg a WeakMap kulcsobjektumaira való hivatkozások gyengék. vagyis a WeakMap egyik kulcsobjektumát a garbage collector törölheti, ha nincs rá már referencia. Egyéb különbségek:

- A Maps bármilyen kulcsfajtát tárolhat, míg a WeakMaps csak a legfontosabb objektumok gyűjteményeit tárolhatja
- A WeakMap a Map-től eltérően nem rendelkezik méret tulajdonsággal
- A WeakMap nem rendelkezik olyan módszerekkel, mint a clear, keys, values, entries, forEach.
- A WeakMap nem iterálható.

Sorolja fel a WeakMap-on elérhető módszerek gyűjteményét

Az alábbiakban felsoroljuk a WeakMap-en elérhető módszerek listáját

- set (kulcs, érték): Beállítja a kulcs értékét a WeakMap objektumban. Visszaadja a WeakMap objektumot.
- delete (kulcs): eltávolítja a kulcshoz társított értékeket.
- has (kulcs): Logikai értéket ad vissza, amely azt állítja, hogy értéket társítottak-e a kulcshoz a WeakMap objektumban, vagy sem.
- get (kulcs): Visszaadja a kulcshoz társított értéket, vagy ha nincs megadva akkor undefined lesz.

```
var weakMapObject = new WeakMap();
var firstObject = {};
var secondObject = {};
// set(key, value)
weakMapObject.set(firstObject, 'John');
weakMapObject.set(secondObject, 100);
console.log(weakMapObject.has(firstObject)); //true
console.log(weakMapObject.get(firstObject)); // John
weakMapObject.delete(secondObject);
```

Mi az uneval célja?

Az uneval () egy beépített függvény, amelyet az objektum forráskódjának karakterlánc-reprezentációjának létrehozására használnak. Ez egy legfelső szintű függvény, és nincs társítva egyetlen objektumhoz sem. Lássuk az alábbi példát, hogy többet megtudjunk a funkcionalitásáról,

```
var a = 1;
uneval(a); // returns a String containing 1
uneval(function user() {}); // returns "(function user(){})"
```

Mi a különbség az uneval és az eval között?

Az uneval függvény egy adott objektum forrását adja vissza; mivel az eval függvény ennek ellenkezőjét teszi, ha a forráskódot egy másik memóriaterületen értékeli. Nézzünk meg egy példát a különbség tisztázására,

```
var msg = uneval(function greeting() { return 'Hello, Good morning'; });
var greeting = eval(msg);
greeting(); // returns "Hello, Good morning"
```

Mi az a névtelen függvény?

Az anonim függvény név nélküli függvény! Az anonim függvényeket általában egy változó nevéhez rendelik, vagy callback függvényként használják. A szintaxis az alábbiak szerint alakul,

```
function (optionalParameters) {
  //do something
}

const myFunction = function(){ //Anonymous function assigned to a variable
  //do something
};

[1, 2, 3].map(function(element){ //Anonymous function used as a callback function
  //do something
});
```

Lássuk a fenti névtelen függvényt egy példában,

```
var x = function (a, b) {return a * b};
var z = x(5, 10);
console.log(z); // 50
```

Mi a fontossági sorrend a lokális és globális változók között?

A lokális változó elsőbbséget élvez az azonos nevű globális változóval szemben. Lássuk ezt a viselkedést egy példában.

```
var msg = "Good morning";
function greeting() {
   msg = "Good Evening";
   console.log(msg);
}
greeting();
```

Mik azok a javascript- accessors?

Az ECMAScript 5 bevezette a javascript objektum- accessors-t vagy a kiszámított tulajdonságokat (computed properties) getterek és setterek segítségével. A Getters a get kulcsszót használja, míg a Setters a set kulcsszót.

```
var user = {
  firstName: "John",
  lastName : "Abraham",
```

```
  language : "en",
  get lang() {
    return this.language;
  }
  set lang(lang) {
  this.language = lang;
  }
};
console.log(user.lang); // getter access lang as en
user.lang = 'fr';
console.log(user.lang); // setter used to set lang as fr
```

Hogyan definiálja a tulajdonságot az Object konstruktoron?

Az Object.defineProperty () statikus metódus segítségével új tulajdonságot lehet meghatározni közvetlenül egy objektumon, vagy módosítani lehet egy objektum meglévő tulajdonságát, és visszaadja az objektumot. Nézzünk meg egy példát a tulajdonság meghatározásának megismerésére:

```
const newObject = {};

Object.defineProperty(newObject, 'newProperty', {
  value: 100,
  writable: false
});

console.log(newObject.newProperty); // 100

newObject.newProperty = 200; // It throws an error in strict mode due to writable setting
```

Mi a különbség a get és a defineProperty között?

Mindkettőnek hasonló eredményei vannak, amíg nem használunk osztályokat. Ha a get parancsot használja, akkor a tulajdonság az objektum prototípusán lesz megadva, míg az Object.defineProperty () használatával a tulajdonság abban a példányban kerül meghatározásra, amelyre alkalmazzák.

Melyek a Getters és a Setters előnyei?

- Egyszerűbb szintaxist nyújtanak
- kiszámított tulajdonságok és beállítások meghatározására használják a JS-ben.
- Hasznos a tulajdonságok és a módszerek közötti ekvivalencia-kapcsolat biztosításához
- Jobb adatminőséget tudnak biztosítani

Hozzáadhatok gettert és setter-t a defineProperty módszerrel?

Igen, használhatja az Object.defineProperty () metódust a Getters és a Setters hozzáadásához.

```
var obj = {counter : 0};

// Define getters
Object.defineProperty(obj, "increment", {
  get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
  get : function () {this.counter--;}
});
```

```
// Define setters
Object.defineProperty(obj, "add", {
  set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
  set : function (value) {this.counter -= value;}
});

obj.add = 10;
obj.subtract = 5;
console.log(obj.increment); //6
console.log(obj.decrement); //5
```

Mi az error object?

A error object egy beépített hibaobjektum, amely hibainformációt nyújt hiba bekövetkezésekor. Két tulajdonsága van: név és üzenet. Például az alábbi függvény naplózza a hiba részleteit,

```
try {
  greeting("Welcome");
}
catch(err) {
  console.log(err.name + "<br>" + err.message);
}
```

Melyek a hibanevek ?

A hibaobjektumból 6 különböző típusú hibanév van.

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	An error has occurred with a number "out of range"
ReferenceError	An error due to an illegal reference
SyntaxError	An error due to a syntax error
TypeError	An error due to a type error
URIError	An error due to encodeURI()

Melyek a hibakezelés különféle utasításai?

- try: Ez az utasítás egy kódblokk hibáinak tesztelésére szolgál
- catch: Ez az utasítás a hiba kezelésére szolgál
- throw: Ez az utasítás egyedi hibák létrehozására szolgál.
- finally: Ez az utasítás a kód végrehajtására szolgál a try és a catch után, függetlenül az eredménytől.

Mi a nodejs?

A Node.js egy szerveroldali platform, amely a Chrome JavaScript runtime-ra épül, a gyors és skálázható hálózati alkalmazások egyszerű felépítéséhez. Ez egy eseményalapú, nem blokkoló, aszinkron I / O, amely a Google V8 JavaScript motorját és libuv könyvtárát használja.

Mi az iterátor?

Az iterátor olyan objektum, amely meghatároz egy szekvenciát és egy visszatérési értéket. Az Iterator protokollt egy next () metódussal valósítja meg, amely két tulajdonságú objektumot ad vissza: value (a sorozat következő értéke) és done (ami igaz, ha a sorozat utolsó értéke elfogyott).

Hogyan működik a szinkron iteráció?

A szinkron iterációt az ES6-ban vezették be, és az alábbi komponensekkel működik,

- Iterable: Ez egy olyan objektum, amelyet át lehet állítani egy olyan módszerrel, amelynek kulcsa a Symbol.iterator.
- Iterator: Ez egy olyan objektum, amelyet a [Symbol.iterator] () meghívásával adunk vissza. Ez az iterátor objektum minden egyes iterált elemet beburkol egy objektumba, és egyesével visszatér next () metóduson keresztül.
- IteratorResult: Ez egy objektum, amelyet a next () metódus ad vissza. Az objektum két tulajdonságot tartalmaz; az value tulajdonság tartalmaz egy iterált elemet, és a done tulajdonság meghatározza, hogy az elem az utolsó elem vagy sem.

```
const iterable = ['one', 'two', 'three'];
const iterator = iterable[Symbol.iterator]();
console.log(iterator.next());  // { value: 'one', done: false }
console.log(iterator.next());  // { value: 'two', done: false }
console.log(iterator.next());  // { value: 'three', done: false }
console.log(iterator.next());  // { value: 'undefined, done: true }
```

Mi az az event loop?

Az Event Loop a callback függvények sora. Amikor egy aszinkron függvény végrehajtódik, a callback függvény a sorba kerül. A JavaScript motor csak akkor kezdi meg az eseményhurok (event loop) feldolgozását, ha az async függvény befejezte a kód végrehajtását. Megjegyzés: Lehetővé teszi a Node.js számára, hogy nem blokkoló I / O műveleteket hajtson végre, annak ellenére, hogy a JavaScript egyszálú.

Mi a call stack?

A Call Stack egy adatstruktúra a javascript fordító számára, hogy nyomon kövesse a program függvényhívásait. Két fő akciója van,

- Amikor meghív egy függvényt annak végrehajtására, akkor stack-be tolja.
- Amikor a végrehajtás befejeződik, a függvény kiugrik a stack-ből.

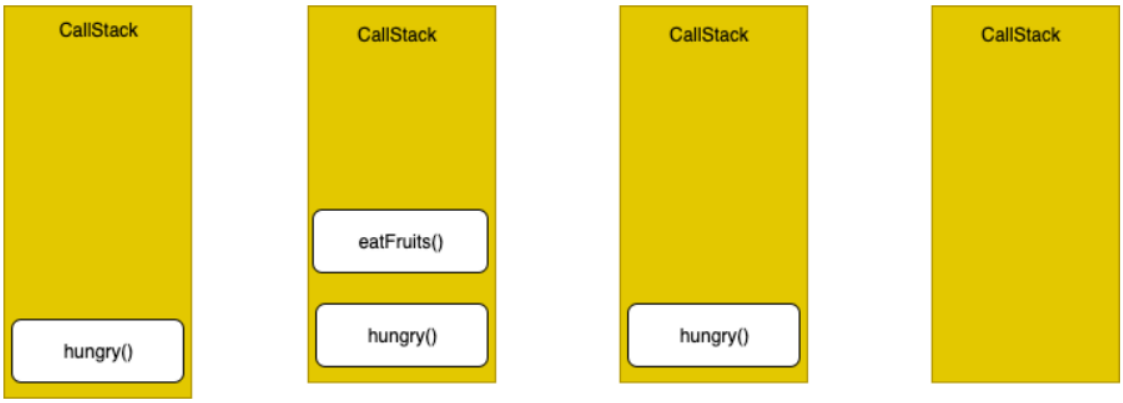
Vegyünk egy példát, és ez az állapot ábrázolása diagram formátumban

```
function hungry() {
   eatFruits();
}
function eatFruits() {
   return "I'm eating fruits";
}

// Invoke the `hungry` function
hungry();
```

A fenti kódot az alábbiakban egy hívásveremben (call stack) dolgozták fel,

- Adja hozzá az hungry () függvényt a hívásverem listájához (call stack list), és hajtsa végre a kódot.
- Adja hozzá az eatFruits () függvényt a hívásverem listájához (call stack list), és hajtsa végre a kódot.
- Törölje az eatFruits () függvényt a hívásverem-listánkból (call stack list).
- Törölje az hungry () függvényt a hívásverem listából, (call stack list) mivel már nincsenek elemek.



Mi az Unary operator?

Az unary (+) operátor segítségével egy változó számokká konvertálható. Ha a változó nem konvertálható, akkor is szám lesz, de NaN értékkel.

```
var x = "100";
var y = + x;
console.log(typeof x, typeof y); // string, number

var a = "Hello";
var b = + a;
```

```
console.log(typeof a, typeof b, b); // string, number, NaN
```

Mi a vessző operator (comma operator)?

A vessző operátor (comma operator) segítségével minden operandusát balról jobbra értékelik, és az utolsó operandus értékét adja vissza. Ez teljesen eltér a vesszők használatától a tömbökön, az objektumokon és a függvény argumentumain és paraméterein. Például a numerikus kifejezések használata az alábbiak szerint alakul:

```
var x = 1;
x = (x++, x);

console.log(x); // 2
```

Mi a vessző operator (comma operator) előnye?

Általában arra használatos, hogy több kifejezést tartalmazzon egy helyen, amelyhez egyetlen kifejezés szükséges. Ennek a vessző operátornak az egyik általános használata, hogy több paramétert ad meg egy for ciklusban. Például az alábbi ciklusnál több kifejezést használ egy helyen vesszővel,

```
for (var a = 0, b =10; a <= 10; a++, b--)
```

Használhatja a vessző operátort egy visszatérési utasításban, ahol a visszatérés előtt feldolgozza.

```
function myFunction() {
   var a = 1;
   return (a += 10, a); // 11
}
```

Mi a különbség a javascript és a typescript között?

feature	typescript	javascript
Language paradigm	Objektumorientált programozási nyelv	Szkript nyelv
Typing support	Támogatja a statikus típust	Dinamikus típussal rendelkezik
Modules	Támogatott	Nem támogatott
Interface	Interfész-koncepcióval rendelkezik	Nem támogatja az interfészeket
Optional parameters	A függvények támogatják az opcionális paramétereket	A függvények nem támogatják az opcionális paramétereket

Milyen előnyei vannak a typescript-nek a javascripttel szemben?

- A TypeScript csak a fejlesztési időben képes fordítási időbeli hibákat találni, és ezáltal kevesebb futásidejű hiba fordul elő. Míg a javascript interpretált nyelv.
- A TypeScript erősen típusos, támogatja a statikus típust, amely lehetővé teszi a típus helyességének ellenőrzését fordításkor. Javascriptben ez nem érhető el.
- A TypeScript fordító a .ts fájlokat az ES3, ES4 és ES5 fájlokba fordíthatja, ellentétben a javascript ES6 jellemzőivel, amelyeket egyes böngészők nem támogatnak.

Mi az objektum inicializáló?

Az objektum inicializáló olyan kifejezés, amely leírja az objektum inicializálását. Ennek a kifejezésnek a szintaxisa vesszővel elválasztott listaként jelenik meg, amely nulla vagy annál több tulajdonságnév-párból és egy objektum kapcsolódó értékéből áll, göndör zárójelek közé zárva ({}). Ez az objektumok létrehozásának egyik módja:

```
var initObject = {a: 'John', b: 50, c: {}};

console.log(initObject.a); // John
```

Hogyan lehet megszerezni egy objektum prototípusát?

Az Object.getPrototypeOf (obj) módszerrel visszaadhatja a megadott objektum prototípusát, azaz a belső prototípus tulajdonságának értéke. Ha nincsenek öröklött tulajdonságok, akkor a null értéket adja vissza.

```
const newPrototype = {};
const newObject = Object.create(newPrototype);

console.log(Object.getPrototypeOf(newObject) === newPrototype); // true
```

Mi történik, ha egy a string típust a getPrototype metódusnak?

Az ES5-ben egy TypeError kivételt dob, ha az obj paraméter nem objektum. Míg ES2015-ben a paraméter egy objektumra lesz kényszerítve.

```
// ES5
Object.getPrototypeOf('James'); // TypeError: "James" is not an object
// ES2015
Object.getPrototypeOf('James'); // String.prototype
```

Hogyan állíthatja az egyik objektum prototípusát egy másik objektumra?

Használhatja az Object.setPrototypeOf () metódust, amely egy adott objektum prototípusát (azaz a belső Prototype tulajdonságát) egy másik objektummá vagy null értékre állítja. Például, ha egy négyzet alakú objektum prototípusát téglalap alakú objektummá kívánja állítani, az a következő,

```
Object.setPrototypeOf(Square.prototype, Rectangle.prototype);
Object.setPrototypeOf({}, null);
```

Hogyan ellenőrizheti, hogy egy objektum kibővíthető-e vagy sem?

Az Object.isExtensible () metódust arra használják, hogy meghatározzák, hogy egy objektum kibővíthető-e vagy sem. azaz, hogy hozzá lehet-e adni új tulajdonságokat, vagy sem.

```
const newObject = {};
console.log(Object.isExtensible(newObject)); //true
```

Megjegyzés: Alapértelmezés szerint az összes objektum kibővíthető, vagyis az új tulajdonságok hozzáadhatók vagy módosíthatók.

Hogyan akadályozhatja meg egy objektum kibővítését?

Az Object.preventExtensions () metódust arra használják, hogy megakadályozzák az új tulajdonságok hozzáadását egy objektumhoz. Más szavakkal, megakadályozza az objektum jövőbeli kiterjesztését. Nézzük meg ennek a tulajdonságnak a használatát,

```
const newObject = {};
Object.preventExtensions(newObject); // NOT extendable

try {
  Object.defineProperty(newObject, 'newProperty', { // Adding new property
    value: 100
  });
} catch (e) {
  console.log(e); // TypeError: Cannot define property newProperty, object is not extensible
}
```

Milyen módon lehet egy objektumot nem-kiterjeszhetővé (non-extensible) tenni ?

- Object.preventExtensions
- Object.seal
- Object.freeze

```
var newObject = {};

Object.preventExtensions(newObject); // Prevent objects are non-extensible
Object.isExtensible(newObject); // false

var sealedObject = Object.seal({}); // Sealed objects are non-extensible
Object.isExtensible(sealedObject); // false

var frozenObject = Object.freeze({}); // Frozen objects are non-extensible
Object.isExtensible(frozenObject); // false
```

Hogyan definiálhat több tulajdonságot egy objektumon?

Az Object.defineProperties () metódussal új objektumokat definiálhatunk vagy meglévő tulajdonságokat módosíthatunk közvetlenül egy objektumon, és visszaadjuk az objektumot. Definiáljunk több tulajdonságot egy üres objektumon,

```
const newObject = {};

Object.defineProperties(newObject, {
  newProperty1: {
```

```
    value: 'John',
    writable: true
  },
  newProperty2: {}
});
```

Mi az Obfuscation (elfedés) a javascriptben?

Az Obfuscation a javascript-kód olyan létrehozása, amelyet az emberek nehezen értenek meg. Valami hasonló a titkosításhoz, de egy gép megérti a kódot és végrehajtja. Nézzük meg az alábbi függvényt obfuscation előtt,

```
function greeting() {
   console.log('Hello, welcome to JS world');
}
```

És az Obfuscation kód után az alábbiak szerint jelenik meg,

```
eval(function(p,a,c,k,e,d){e=function(c){return c};if(!''.replace(/^/,String)){while(c--
){d[c]=k[c]||c}k=[function(e){return d[e]}];e=function(){return'\\w+'};c=1};while(c--
){if(k[c]){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('2 1(){0.3(\'4, 7 6 5
8\')}',9,9,'console|greeting|function|log|Hello|JS|to|welcome|world'.split('|'),0,{}))
```

Miért van szükség Obfuscation-ra?

- A kód mérete csökken. Tehát a szerver és a kliens közötti adatátvitel gyors lesz.
- Elrejti az üzleti logikát a külvilág elől, és megvédi a kódot másoktól
- A letöltési idő lerövidül

Mi a Minification?

A Minification az összes felesleges karakter eltávolításának folyamata (a szóközök eltávolításra kerülnek), és a változókat a funkcionalitás megváltoztatása nélkül átnevezzük. Ez egyfajta elfedés is.

Melyek a minification előnyei?

Általában a forgalom csökkentése és az erőforrások intenzív igénye esetén ajánlatos a tömörítést használni. Csökkenti a fájlméretet az alábbi előnyökkel,

- Csökkenti a weboldal betöltési idejét
- A hálózat terhelése csökken

Mi a különbség az Obfuscation és Encryption között?

Feature	Obfuscation	Encryption

Feature	Obfuscation	Encryption
Definition	Bármely adat formájának megváltoztatása bármilyen más formában	Az információ formájának megváltoztatása olvashatatlan formátumra egy kulcs használatával
A key to decode	Dekódolható kulcs nélkül	Szükség van rá (kulcs)
Target data format	Összetett formára konvertálódik	Olvashatatlan formátumra konvertálva

Melyek az elérhető ellenőrzési DOM tulajdonságok?

- validity: Megadja a bemeneti elem érvényességéhez kapcsolódó logikai tulajdonságok listáját.
- validationMessage: Megjeleníti az üzenetet, ha a nem érvényes a bemeneti elem.
- willValidate: Azt jelzi, hogy egy bemeneti elem érvényes-e vagy sem.

Mi a validációs tulajdonságok listája?

- customError: Igaz értéket ad vissza, ha egyéni validációs üzenet van beállítva.
- patternMismatch: Igaz értéket ad vissza, ha az elem értéke nem egyezik meg a minta attribútumával.
- rangeOverflow: Igaz értéket ad vissza, ha az elem értéke nagyobb, mint a max attribútuma.
- rangeUnderflow: Igaz értéket ad vissza, ha az elem értéke kisebb, mint a min attribútuma.
- stepMismatch: Igaz értéket ad vissza, ha az elem értéke érvénytelen a step attribútum szerint.
- tooLong: Igaz, ha egy elem értéke meghaladja a maxLength attribútumot.
- typeMismatch: Igaz értéket ad vissza, ha az elem értéke érvénytelen a type attribútum szerint.
- valueMissing: Igaz értéket ad vissza, ha a kötelező attribútummal rendelkező elemnek nincs értéke.
- valid: Igaz értéket ad vissza, ha egy elem értéke érvényes.

Mondjon példát a rangeOverflow tulajdonság használatára?

Ha egy elem értéke nagyobb, mint a max attribútuma, akkor a rangeOverflow tulajdonság igaz értéket ad vissza. Például az alábbi űrlap hibát dob, ha az érték meghaladja a 100 értéket:

```
<input id="age" type="number" max="100">
<button onclick="myOverflowFunction()">OK</button>
function myOverflowFunction() {
  if (document.getElementById("age").validity.rangeOverflow) {
    alert("The mentioned age is not allowed");
```

```
  }
}
```

Hogyan sorolja fel az objektum összes tulajdonságát?

Használhatja az Object.getOwnPropertyNames () metódust, amely az adott objektumban található összes tulajdonság tömbjét adja vissza. Használjuk egy példában,

```
const newObject = {
  a: 1,
  b: 2,
  c: 3
};

console.log(Object.getOwnPropertyNames(newObject));  ["a", "b", "c"]
```

Hogyan lehet megszerezni egy objektum tulajdonságleíróit?

Használhatja az Object.getOwnPropertyDescriptors () metódust, amely az adott objektum összes saját tulajdonságleíróját visszaadja.

```
const newObject = {
   a: 1,
   b: 2,
   c: 3
 };
const descriptorsObject = Object.getOwnPropertyDescriptors(newObject);
console.log(descriptorsObject.a.writable); //true
console.log(descriptorsObject.a.configurable); //true
console.log(descriptorsObject.a.enumerable); //true
console.log(descriptorsObject.a.value); // 1
```

Milyen tulajdonságokat ad meg egy tulajdonságleíró?

- value: A tulajdonsághoz társított érték
- writable: Meghatározza, hogy a tulajdonsághoz társított érték megváltoztatható-e vagy sem
- configurable: true értéket ad vissza, ha a tulajdonságleíró típusa megváltoztatható, és ha a tulajdonság törölhető a megfelelő objektumból.
- enumerable: Meghatározza, hogy a tulajdonság megjelenik-e a tulajdonságok felsorolása során a megfelelő objektumon, vagy sem.
- set: beállítja a tulajdonságot
- get: visszaadja a tulajdonságot

Hogyan hasonlítja össze a skalár tömböket?

```
const arrayFirst = [1,2,3,4,5];
const arraySecond = [1,2,3,4,5];
console.log(arrayFirst.length === arraySecond.length && arrayFirst.every((value, index) => value ===
arraySecond[index])); // true
```

Ha a tömböket sorrendtől függetlenül szeretné összehasonlítani, akkor korábban rendeznie kell őket,

```
const arrayFirst = [2,3,1,4,5];
const arraySecond = [1,2,3,4,5];
```

```
console.log(arrayFirst.length === arraySecond.length && arrayFirst.sort().every((value, index) =>
value === arraySecond[index])); //true
```

Hogyan kapjuk meg az értéket a get paraméterekből?

Az new URL () objektum elfogadja az URL karakterláncot, és az objektum searchParams tulajdonsága felhasználható a get paraméterek eléréséhez. Ne feledje, hogy előfordulhat, hogy a régebbi böngészők (beleértve az IE-t is) URL-jének eléréséhez a polyfill vagy az window.location alkalmazást kell használnia.

```
let urlString = "http://www.some-domain.com/about.html?x=1&y=2&z=3"; //window.location.href
let url = new URL(urlString);
let parameterZ = url.searchParams.get("z");
console.log(parameterZ); // 3
```

A javascript támogatja a névteret?

A JavaScript alapértelmezés szerint nem támogatja a névteret. Tehát, ha létrehoz egy elemet (függvényt, metódust, objektumot, változót), akkor globális lesz, és szennyezi a globális névteret. Vegyünk egy példát két függvény meghatározására névtér nélkül,

```
function func1() {
    console.log("This is a first definition");

}
function func1() {
    console.log("This is a second definition");
}
func1(); // This is a second definition
```

Mindig a második függvénydefiníciót hívja meg. Ebben az esetben a névtér megoldja a névütközés problémáját.

Hogyan deklarálja a névteret?

Annak ellenére, hogy a JavaScriptnek nincsenek névterei, az Objects, IIFE használatával névtereket hozhatunk létre.

1.) Objektum literál jelölés használata: Vegyünk be változókat és függvényeket egy objektum literál belsejébe, amely névtérként működik. Ezután az objektum jelöléssel érheti el őket

```
var namespaceOne = {
   function func1() {
       console.log("This is a first definition");
   }
}
var namespaceTwo = {
     function func1() {
         console.log("This is a second definition");
     }
 }
namespaceOne.func1(); // This is a first definition
namespaceTwo.func1(); // This is a second definition
```

2.) Az IIFE (Immediately invoked function expression) használata: Az IIFE külső zárójelpárja létrehozza a benne lévő összes kód lokális hatókörét, és az anonim függvényt függvény kifejezéssé teszi. Ennek köszönhetően létrehozhatja ugyanazt a függvényt két különböző függvény kifejezésben, hogy névtérként működjön.

```
(function() {
 function fun1(){
   console.log("This is a first definition");
   } fun1();
}());

(function() {
    function fun1(){
       console.log("This is a second definition");
   } fun1();
 }());
```

3.) Blokk és let / const deklaráció használata: Az ECMAScript 6-ban egyszerűen használhat egy blokkot és egy let deklarációt a változó hatókörének blokkra szűkítésére.

```
{
  let myFunction= function fun1(){
  console.log("This is a first definition");
  }
  myFunction();
 }
  //myFunction(): ReferenceError: myFunction is not defined.

 {
  let myFunction= function fun1(){
  console.log("This is a second definition");
  }
  myFunction();
 }
  //myFunction(): ReferenceError: myFunction is not defined.
```

Mi a V8 JavaScript motor?

A V8 egy nyílt forráskódú, nagy teljesítményű JavaScript-motor, amelyet a Google Chrome böngésző használ, C ++ nyelven írva. A node.js projektben is használják. Megvalósítja az ECMAScript-et

Miért hívjuk a javascriptet dinamikus nyelvnek?

A JavaScript lazán típusos (loosely typed) vagy dinamikus nyelv, mivel a JavaScript-ben szereplő változók nincsenek közvetlenül társítva egy adott értéktípushoz, és bármely változó minden típusú értékhez hozzárendelhető / átrendelhető.

```
let age = 50;    // age is a number now
age  = 'old'; // age is a string now
age  = true;  // age is a boolean
```

Mi az a void operator?

A void operátor kiértékeli az adott kifejezést, majd undefined-al tér vissza (azaz visszatérő érték nélkül). A szintaxis az alábbiak szerint alakul,

```
void (expression)
void expression
```

Jelenítsünk meg üzenetet átirányítás és újratöltés nélkül:

```
<a href="javascript:void(alert('Welcome to JS world'))">Click here to see a message</a>
```

Sorolja fel az ES6 néhány jellemzőjét?

i. Support for constants or immutable variables
ii. Block-scope support for variables, constants and functions
iii. Arrow functions
iv. Default parameters
v. Rest and Spread Parameters
vi. Template Literals
vii. Multi-line Strings
viii. Destructuring Assignment
ix. Enhanced Object Literals
x. Promises
xi. Classes
xii. Modules

Mi az ES6?

Az ES6 a javascript nyelv hatodik kiadása, és 2015 júniusában jelent meg. Eredetileg ECMAScript 6 (ES6) néven ismert, később ECMAScript 2015 névre keresztelték. Szinte az összes modern böngésző támogatja az ES6-ot, de a régi böngészőknél sok transzpiler , mint a Babel.js stb.

A const a változó értéket megváltoztathatatlanná (immutable) teszi-e?

nem engedélyezi a későbbi hozzárendeléseket

```
const userList = [];
userList.push('John'); // Can mutate even though it can't re-assign
console.log(userList); // ['John']
```

Mik az alapértelmezett (default) paraméterek?

Az ES5-ben logikai VAGY operátoroktól kell függenünk a függvényparaméterek alapértelmezett értékeinek kezeléséhez. Míg az ES6-ban az alapértelmezett (default) függvény paraméterek lehetővé teszik a paraméterek inicializálását az alapértelmezett értékekkel, ha nincs átadva érték vagy undefined. Hasonlítsuk össze a viselkedést egy példával:

```
//ES5
var calculateArea = function(height, width) {
   height =  height || 50;
   width = width || 60;
```

```
   return width * height;
}
console.log(calculateArea()); //300
```

Az alapértelmezett paraméterek egyszerűbbé teszik az inicializálást

```
//ES6
var calculateArea = function(height = 50, width = 60) {
   return width * height;
}

console.log(calculateArea()); //300
```

Mik azok a sablon literálok (template literals)?

A Template literals vagy template strings olyan karakterláncok, amelyek lehetővé teszik a beágyazott kifejezéseket. Ezeket a back-tick (`) karakter zárja be kettős vagy egyes idézőjelek helyett. Az Es6-ban ez a funkció lehetővé teszi az alábbi dinamikus kifejezések használatát:

```
var greeting = `Welcome to JS World, Mr. ${firstName} ${lastName}.`
```

Az ES5-ben az alábbiak szerint megszakító karakterláncra van szükség,

```
var greeting = 'Welcome to JS World, Mr. ' + firstName + ' ' + lastName.`
```

Mik a nesting templates?

A nesting template a sablon literal (template literals) szintaxisában támogatott szolgáltatás, amely lehetővé teszi a inner backticks beillesztését a sablon $ {} helyőrzőjébe. Például az alábbi beágyazási sablont használják az ikonok megjelenítésére a felhasználói engedélyek alapján, míg a külső sablon ellenőrzi a platform típusát:

```
const iconStyles = `icon ${ isMobilePlatform() ? '' :
 `icon-${user.isAuthorized ? 'submit' : 'disabled'}` }`;
```

A fenti használati esetet nesting template nélkül is megírhatja. A nesting template szolgáltatás azonban kompaktabb és olvashatóbb.

```
//Without nesting templates
 const iconStyles = `icon ${ isMobilePlatform() ? '' :
  (user.isAuthorized ? 'icon-submit' : 'icon-disabled'}`;
```

Mik a címkézett sablonok (tagged templates)?

A címkézett sablonok (tagged templates) a sablonok speciális formája, amelyben a címkék lehetővé teszik a sablon literálok elemzését egy függvénnyel. A tag függvény elfogadja az első paramétert stringek tömbjeként, a többi paramétert pedig kifejezésekként. Ez a függvény paraméterek alapján

visszaadhatja a manipulált karakterláncokat is. Lássuk, hogyan használják ezt a címkézett sablon viselkedést egy informatikai szakmai készség egy szervezet példában:

```
var user1 = 'John';
var skill1 = 'JavaScript';
var experience1 = 15;

var user2 = 'Kane';
var skill2 = 'JavaScript';
var experience2 = 5;

function myInfoTag(strings, userExp, experienceExp, skillExp) {
  var str0 = strings[0]; // "Mr/Ms. "
  var str1 = strings[1]; // " is a/an "
  var str2 = strings[2]; // "in"

  var expertiseStr;
  if (experienceExp > 10){
    expertiseStr = 'expert developer';
  } else if(skillExp > 5 && skillExp <= 10) {
    expertiseStr = 'senior developer';
  } else {
    expertiseStr = 'junior developer';
  }

  return ${str0}${userExp}${str1}${expertiseStr}${str2}${skillExp};
}

var output1 = myInfoTag`Mr/Ms. ${ user1 } is a/an ${ experience1 } in ${skill1}`;
var output2 = myInfoTag`Mr/Ms. ${ user2 } is a/an ${ experience2 } in ${skill2}`;

console.log(output1);// Mr/Ms. John is a/an expert developer in JavaScript
console.log(output2);// Mr/Ms. Kane is a/an junior developer in JavaScript
```

raw strings?

Az ES6 a String.raw () módszerrel raw strings szolgáltatást nyújt, amelyet a sablon-karakterláncok nyers karakterlánc (raw strings)-alakjának megszerzésére használnak. Például a felhasználás az alábbiak szerint alakul:

```
var calculationString = String.raw `The sum of numbers is \n${1+2+3+4}!`;
console.log(calculationString); // The sum of numbers is 10
```

Ha nem használ nyers karakterláncokat (raw strings), akkor az újsoros karaktersorozatot a kimenet több soros megjelenítésével dolgozzuk fel:

```
var calculationString = `The sum of numbers is \n${1+2+3+4}!`;
console.log(calculationString);
// The sum of numbers is
// 10
```

A raw tulajdonság elérhető a tag függvény első argumentumánál is:

```
function tag(strings) {
   console.log(strings.raw[0]);
}
```

Mi destructuring assignment?

A destructuring assignment egy olyan JavaScript kifejezés, amely lehetővé teszi az értékek tömbökből vagy objektumokból történő kibontását az különböző változókba. :

```
var [one, two, three] = ['JAN', 'FEB', 'MARCH'];

console.log(one); // "JAN"
console.log(two); // "FEB"
console.log(three); // "MARCH"


var {name, age} = {name: 'John', age: 32};

console.log(name); // John
console.log(age); // 32
```

Melyek az alapértelmezett értékek a destructuring assignment során?

Egy változóhoz alapértelmezett érték rendelhető, ha a tömbből vagy az objektumból kicsomagolt érték nincs meghatározva a destructuring assignment során. Segít elkerülni az alapértelmezett értékek külön-külön történő beállítását az egyes hozzárendeléseknél. Vegyünk egy példát mind tömbökre, mind objektumhasználati esetekre:

Arrays destructuring:

```
var x, y, z;

[x=2, y=4, z=6] = [10];
console.log(x); // 10
console.log(y); // 4
console.log(z); // 6
```

Objects destructuring:

```
var {x=2, y=4, z=6} = {x: 10};

console.log(x); // 10
console.log(y); // 4
console.log(z); // 6
```

Hogyan cserélhet változókat a destructuring assignment segítségével?

Ha nem használja a destructuring assignment-et, két érték cseréjéhez ideiglenes változóra van szükség

```
var x = 10, y = 20;

[x, y] = [y, x];
console.log(x); // 20
console.log(y); // 10
```

Mik a enhanced object literals?

Az objektum literálok megkönnyítik a {} zárójelben lévő tulajdonságokkal rendelkező objektumok gyors létrehozását. Például rövidebb szintaxist biztosít a közös objektumtulajdonság-meghatározáshoz, az alábbiak szerint.

```
//ES6
var x = 10, y = 20
obj = { x, y }
console.log(obj); // {x: 10, y:20}
//ES5
var x = 10, y = 20
obj = { x : x, y : y}
console.log(obj); // {x: 10, y:20}
```

Mi a dinamikus import?

A dinamikus importálás az import () függvény szintaxisa segítségével lehetővé teszi számunkra a modulok igény szerinti betöltését promise vagy az async / await szintaxis használatával. A dinamikus import fő előnye a csomagméretünk csökkentése, kéréseink válasza a méret / job terhelésre és a felhasználói élmény általános javulása. A dinamikus import szintaxisa az alábbiak szerint alakul:

```
import('./Module').then(Module => Module.method());
```

Melyek a dinamikus import felhasználási esetei?

Az alábbiakban bemutatunk néhányat a dinamikus import statikus import helyett történő felhasználásáról,

1.) Importálhat modult igény szerint vagy feltételesen. Például, ha polifillet szeretne betölteni a régi böngészőbe

```
if (isLegacyBrowser()) {
    import(···)
    .then(···);
}
```

2.) Modul meghatározása dinamikusan:

```
import(`messages_${getLocale()}.js`).then(···);
```

Mi a célja a double tilde operator-nak?

A double tilde operator (~~) kettős NOT bitenkénti operátor. Ez az operátor gyorsabban helyettesíti a Math.floor () –t.

Mi az Error objektum célja?

A Error konstruktor létrehoz egy hibaobjektumot, és a hibaobjektumok példányai dobásra kerülnek, amikor futásidejű hibák fordulnak elő. A Error objektum a felhasználó által definiált kivételek alapobjektumaként is használható. A hibaobjektum szintaxisa az alábbiak szerint alakul,

```
new Error([message[, fileName[, lineNumber]]])
```

A felhasználó által definiált kivételeket vagy hibákat a Error objektum try ... catch blokk használatával dobhatja az alábbiak szerint:

```
try {
  if(withdraw > balance)
  throw new Error("Oops! You don't have enough balance");
} catch (e) {
  console.log(e.name + ': ' + e.message);
}
```

Mi az EvalError objektum célja?

Az EvalError objektum hibát jelez a globális eval () függvénnyel kapcsolatban. Annak ellenére, hogy ezt a kivételt a JavaScript már nem dobta meg, az EvalError objektum továbbra is a kompatibilitás érdekében marad. Ennek a kifejezésnek a szintaxisa az alábbiak szerint alakul,

```
new EvalError([message[, fileName[, lineNumber]]])
```

EvalError with in try...catch block:

```
try {
  throw new EvalError('Eval function error', 'someFile.js', 100);
} catch (e) {
  console.log(e.message, e.name, e.fileName);              // "Eval function error", "EvalError",
"someFile.js"
```

Melyek azok a hibák listája, amelyek a non-strict mode-ban nem, de a strict mode-ban hibák?

Amikor alkalmazza a „use strict” beállítást; szintaxis, az alábbi esetek egy része SyntaxError-t dob a parancsfájl végrehajtása előtt:

i. When you use Octal syntax

```
var n = 022;
```

i. Using `with` statement
ii. When you use delete operator on a variable name
iii. Using eval or arguments as variable or function argument name
iv. When you use newly reserved keywords
v. When you declare a function in a block

```
if (someCondition) { function f() {} }
```

Mi a célja some módszernek a tömbökben?

A Some () metódust arra használják, hogy teszteljék, hogy a tömbben legalább egy elem megfelel-e a megadott függvény által végrehajtott tesztnek. A módszer logikai értéket ad vissza.

```
var array = [1, 2, 3, 4, 5, 6 ,7, 8, 9, 10];

var odd = element ==> element % 2 !== 0;
```

```
console.log(array.some(odd)); // true (the odd element exists)
```

Hogyan kombinálhat két vagy több tömböt?

A concat () metódust két vagy több tömb összekapcsolására használjuk egy új, az összes elemet tartalmazó tömb visszaadásával. A szintaxis az alábbiak szerint alakul:

```
array1.concat(array2, array3, ..., arrayX)
```

Vegyünk egy példát:

```
var veggies = ["Tomato", "Carrot", "Cabbage"];
  var fruits = ["Apple", "Orange", "Pears"];
  var veggiesAndFruits = veggies.concat(fruits);
  console.log(veggiesAndFruits); // Tomato, Carrot, Cabbage, Apple, Orange, Pears
```

Mi a különbség a sekély (Shallow) és a mély másolás (Deep copy) között?

1.) Sekély másolás (Shallow copy): A sekély másolás (Shallow copy) az objektum bitenkénti másolata. Új objektum jön létre, amely az eredeti objektumban található értékek pontos másolatával rendelkezik. Ha az objektum bármelyik mezője hivatkozás más objektumokra, akkor csak a referencia címek kerülnek másolásra, azaz csak a memória címek kerülnek másolásra.

Példa:

```
var empDetails = {
  name: "John", age: 25, expertise: "Software Developer"
}
```

duplikátum létrehozásához:

```
var empDetailsShallowCopy = empDetails    //Shallow copying!
```

ha valamilyen tulajdonságértéket megváltoztatunk a másolatban, így:

```
empDetailsShallowCopy.name = "Johnson"
```

A fenti állítás megváltoztatja az empDetails name property értékét is, mivel van egy sekély példányunk. Ez azt jelenti, hogy az eredeti adatokat is elveszítjük.

2.) Mély másolás (Deep copy) : A mély másolat (Deep copy) az összes mezőt lemásolja, és a mezők által mutatott, dinamikusan lefoglalt memóriából másolatokat készít. A mély másolat akkor következik be, amikor egy objektumot másolnak azokkal az objektumokkal, amelyekre hivatkoznak.

Példa:

```
var empDetails = {
  name: "John", age: 25, expertise: "Software Developer"
}
```

Hozzon létre egy mély másolatot az eredeti objektum tulajdonságainak felhasználásával egy új változóba:

```
var empDetailsDeepCopy = {
  name: empDetails.name,
  age: empDetails.age,
  expertise: empDetails.expertise
}
```

Mi a thunk függvény?

A thunk csak egy olyan függvény, amely késlelteti az érték kiértékelését. Ez nem tartalmaz argumentumokat, de megadja az értéket, amikor a thunkot meghívjuk. vagyis nem most hajtják végre, hanem valamikor a jövőben. Vegyünk egy szinkron példát,

```
const add = (x,y) => x + y;

const thunk = () => add(2,3);

thunk() // 5
```

Mik azok az asynchronous thunks?

Az asynchronous thunks hasznosak hálózati kérések során. Lássunk egy példát a hálózati kérésekre:

```
function fetchData(fn){
  fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => fn(json))
}

const asyncThunk = function (){
   return fetchData(function getData(data){
      console.log(data)
  })
}

asyncThunk()
```

A getData függvény nem lesz azonnal meghívva, de csak akkor lesz meghívva, ha az adatok rendelkezésre állnak az API végpontból. A setTimeout függvényt arra is használjuk, hogy aszinkron legyen a kódunk. A legjobb valós idejű példa a redux állapotmenedzsment könyvtár, amely az aszinkron töredékeket használja a műveletek késleltetésére.

Mi történik egy tömb tagadásával?

Egy tomb tagadása, azaz a! karakter kényszeríti a tömböt logikai értékre. Mivel a tömböket igaznak tekintik, ezért a tagadás hamis lesz.

```
console.log(![]); // false
```

Mi történik, ha két tömböt egymáshoz adunk?

Ha két tömböt adunk egymáshoz, akkor mindkettőt karakterláncokká alakítja és összefűzi. Például

```
console.log(['a'] + ['b']);  // "ab"
console.log([] + []); // ""
console.log(![] + []); // "false", because ![] returns false.
```

Mi az destructuring aliases?

Előfordul, hogy szeretne egy destructured változót, amelynek neve más, mint a tulajdonság neve.

```
const obj = { x: 1 };
// Grabs obj.x as as { otherName }
const { x: otherName } = obj;
```

Mi a legkönnyebb egy tömböt objektummá konvertálni?

Tömböt átalakíthat objektummá ugyanazokkal az adatokkal a spread (...) operátorral.

```
var fruits = ["banana", "apple", "orange", "watermelon"];
var fruitsObject = {...fruits};
console.log(fruitsObject); // {0: "banana", 1: "apple", 2: "orange", 3: "watermelon"}
```

Hogyan hozhat létre tömböt néhány adattal?

Létrehozhat tömböt néhány adattal vagy tömböt ugyanazokkal az értékekkel a fill módszer segítségével.

```
var newArray = new Array(5).fill("0");
console.log(newArray); // ["0", "0", "0", "0", "0"]
```

Melyek a console object placeholder-ei?

% o - objektum

% s - string

% d - decimal or integer

```
const user = { "name":"John", "id": 1, "city": "Delhi"};
console.log("Hello %s, your details %o are available in the object form", "John", user); // Hello
John, your details {name: "John", id: 1, city: "Delhi"} are available in object
```

Hogyan jelenítheti meg az adatokat táblázatos formátumban a console object segítségével?

A console.table () arra szolgál, hogy táblázatos formában jelenítse meg az adatokat a konzolban, az összetett tömbök vagy objektumok megjelenítéséhez.

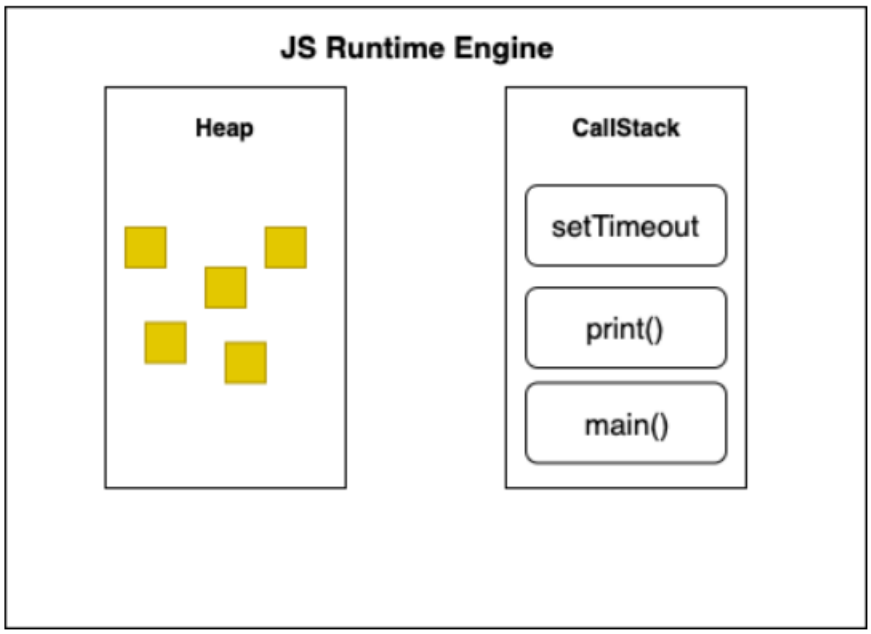
```
const users = [{ "name":"John", "id": 1, "city": "Delhi"}, { "name":"Max", "id": 2, "city": "London"},
{ "name":"Rod", "id": 3, "city": "Paris"} ];
console.table(users);
```

Milyen módon lehet kezelni az aszinkron kódot?

i. Callbacks
ii. Promises
iii. Async/await
iv. Third-party libraries such as async.js,bluebird etc

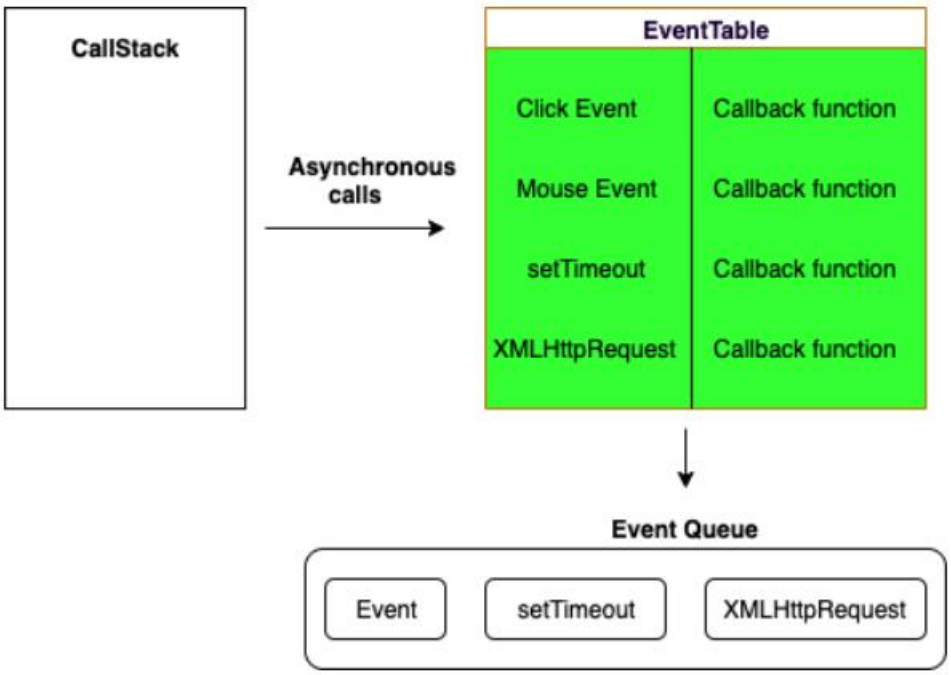
Mi a heap?

A heap (vagy memory heap) az a memóriahely, ahol az objektumok tárolódnak, amikor változókat definiálunk. azaz ezen a helyen történik az összes memória-allokáció és de-allokáció. A heap és a call-stack is két konténer JS futási környezeteben. Valahányszor a futásidejű változók és függvénydeklarációk találkoznak a kódban, azokat tárolja a heap-ben.



Mi az esemény táblázat (event table)?

Az Eseménytábla (event table) egy olyan adatstruktúra, amely tárolja és nyomon követi az összes olyan eseményt, amelyeket aszinkron módon hajtanak végre, például bizonyos időintervallumok után vagy néhány API-kérés után, azaz amikor meghív egy setTimeout függvényt vagy meghívja az aszinkron műveletet, az hozzáadódik az Eseménytáblához. Nem egyedül hajtja végre a függvényeket. Az eseménytábla fő célja az események nyomon követése és az eseménysorba történő elküldése az alábbi ábra szerint.



Mi a microTask queue?

A Microtask Queue az a queue, ahol a promise objektumok által kezdeményezett összes feladat feldolgozásra kerül a callback queue előtt. A Microtask Queue feldolgozása a következő renderelés előtt történik. De ha ezek a Microtask Queue- sokáig működnek, ez vizuális romláshoz vezet (sok idő míg kirenderelődik az adat).

Mi a babel?

A Babel egy JavaScript-transzpiler, amely az ECMAScript 2015+ kódot a visszafelé kompatibilis JavaScript-verzióvá konvertálja a jelenlegi és régebbi böngészőkben vagy környezetekben. Az alábbiakban felsoroljuk a főbb jellemzőket,

- Szintaxis átalakítása
- Polyfill funkciók, amelyek hiányoznak a célkörnyezetből (@ babel / polyfill használatával)
- Forráskód-transzformációk

Melyek az observable elemek általános használati esetei?

A observable elemek közül a legelterjedtebbek közé tartoznak a web sockets with push notifications, user input changes, repeating intervals stb

Mi az RxJS?

Az RxJS (Reactive Extensions for JavaScript) egy könyvtár a reaktív programozás megvalósításához Observable elemek felhasználásával, amely megkönnyíti az aszinkron vagy callback-alapú kód összeállítását. Segédfunkciókat is biztosít az Observable létrehozásához és azokkal való munkához.

Mi a különbség a Function konstruktor és a függvény deklaráció között?

A Function konstruktorral létrehozott függvények nem zárják le létrehozási környezetüket, de mindig globális hatókörben jönnek létre. vagyis a függvény csak a saját lokális változóihoz és a globális hatókörű változókhoz férhet hozzá. Míg a függvény deklarációk hozzáférhetnek a külső függvény változókhoz is.

1.) Function Constructor:

```
var a = 100;
function createFunction() {
    var a = 200;
    return new Function('return a;');
}
console.log(createFunction()()); // 100
```

2.) Function declaration:

```
var a = 100;
function createFunction() {
    var a = 200;
    return function func() {
        return a;
    }
}
console.log(createFunction()()); // 200
```

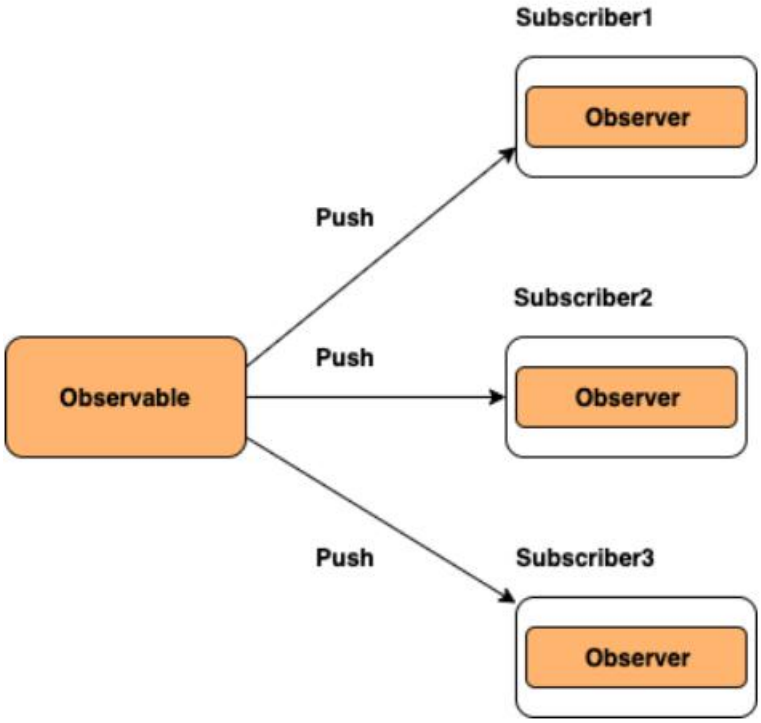
Mi az observable?

Az Observable alapvetően egy olyan függvény, amely az értékeket akár szinkron, akár aszinkron módon képes visszajuttatni egy observer-hez az idő múlásával. A consumer az subscribe () metódus meghívásával kaphatja meg az értéket. Nézzünk meg egy egyszerű példát:

```
import { Observable } from 'rxjs';

const observable = new Observable(observer => {
  setTimeout(() => {
    observer.next('Message from a Observable!');
  }, 3000);
});

observable.subscribe(value => console.log(value));
```



Mi az aszinkron függvény?

Az aszinkron függvény az async kulcsszóval deklarált függvény, amely lehetővé teszi az aszinkron, promise-alapú viselkedés tisztább stílusú megírását az promise láncolásának (chain) elkerülésével. Ezek a függvények nulla vagy több await kifejezést tartalmazhatnak.

Vegyünk egy alábbi aszinkron függvény példát:

```
async function logger() {

  let data = await fetch('http://someapi.com/users'); // pause until fetch returns
  console.log(data)
}
logger();
```

Milyen különbségek vannak az argumentum objektum és a rest paraméter között?

- Az argumentum objektum tömbszerű, de nem tömb. Míg a rest paraméter tömbpéldány.
- Az argumentumobjektum nem támogatja az olyan módszereket, mint a sort, map, forEach, vagy pop. Mivel ezek a módszerek alkalmazhatók a rest paraméterekben.
- A rest paraméter csak azok, amelyek nem kaptak külön nevet, míg az argumentumok objektum tartalmazza a függvénynek átadott összes argumentumot

Milyen különbségek vannak a spread operátor és a rest paraméter között?

A Rest paraméter összegyűjti az összes megmaradt elemet egy tömbbe. Míg a Spread operátor lehetővé teszi az iterables(arrays / objects / strings) kibővítését egyetlen argumentumra / elemre. azaz a Rest paraméter ellentétes a spread operátorral.

Melyek a built-in iterables?

i. Arrays and TypedArrays
ii. Strings: Iterate over each character or Unicode code-points
iii. Maps: iterate over its key-value pairs
iv. Sets: iterates over their elements
v. arguments: An array-like special variable in functions
vi. DOM collection such as NodeList

Milyen különbségek vannak a for...of és for...in utasítások között?

Mind a for...in és a for... of utasítások javascript adatstruktúrán iterálnak. A különbség az, hogyan iterálnak:

- for..in i: egy objektum property kulcsain
- for..of : az értékeken iterál

```
let arr = ['a', 'b', 'c'];

arr.newProp = 'newVlue';

// key are the property keys
for (let key in arr) {
  console.log(key);
}

// value are the property values
for (let value of arr) {
  console.log(value);
}
```

Mivel a for..in ciklusban az objektum kulcsai felett iterál, ezért 0, 1, 2 amit kiír. A for..of ciklus egy adatstruktúra értékein iterál, és a, b, c lesz ezért az értéke.

Hogyan definiálja a példány és a nem példány szintű tulajdonságokat?

A példány tulajdonságait az osztály metódusokon belül kell meghatározni. Például a név és az életkor tulajdonságai az alábbiak szerint definiálták a konstruktorban:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

De a statikus (osztály) és prototípus tulajdonságokat a ClassBody deklaráción kívül kell meghatározni. Rendeljük az alábbiak szerint a Person osztály életkor értékét,

```
Person.staticAge = 30;
Person.prototype.prototypeAge = 40;
```