https://github.com/sudheerj/ECMAScript-features

Mi az ECMAScript?

Az ECMAScript a script nyelv, amely a JavaScript alapja. Az ECMA Nemzetközi Szabványügyi Szervezet által az ECMA-262 és az ECMA-402 specifikációkban szabványosított ECMAScript. Az ECMAScript szolgáltatás minden egyes javaslata a következő lejárati szakaszokon megy keresztül:

1. Stage 0: Strawman;
2. Stage 1: Proposal;
3. Stage 2: Draft;
4. Stage 3: Candidate;
5. Stage 4: Finished.

Edition	Date
ES2015 Or ES6	June 2015
ES2016 Or ES7	June 2016
ES2017 Or ES8	June 2017
ES2018 Or ES9	June 2018
ES2019 Or ES10	June 2019
ES2020 Or ES11	June 2020

ES2015 vagy ES6

Variable Scoping

A változó hatóköre meghatározza a változó láthatóságát vagy hozzáférhetőségét a program vagy régió bizonyos részén belül. Az ES6-ban az const és a let kulcsszavak lehetővé teszik a fejlesztők számára, hogy a változókat a blokk hatókörében deklarálják. A let utasítás egy blokk-hatókörű helyi változót deklarál, amely átrendelhető. azaz a let deklaráció létrehoz egy mutatható változót.

```
let a = 1;

if (a === 1) {
  let a = 2;

  console.log(a); //2
}

console.log(a); //1
```

A const változók hasonlóak a let változókhoz, nem változtathatók meg. vagyis a const deklaráció csak olvasható hivatkozást hoz létre egy értékre.

```
const x = 1;

if (x === 1) {
    const y = 2; // You cannot re-assign the value similar to let variable

    console.log(y); //2
}

console.log(x); //1
```

Arrow function

Arrow function tömörebb szintaxist nyújtanak a függvénykifejezések írásához azáltal, hogy nem kell kiírni a function és return kulcsszavakat.. Lássuk, hogyan néz ki:

```
// Function Expression
var multiplyFunc = function(a, b) {
    return a * b;
}
console.log(multiplyFunc(2, 5)); // 10

// Arrow function
var multiplyArrowFunc = (a, b) => a * b;
console.log(multiplyArrowFunc(2, 5)); // 10
```

Kihagyhatja a zárójeleket (()) is, ha a függvénynek pontosan egy paramétere van (vagy nulla, vagy egynél több paraméter). Ettől eltekintve zárójeleket ({}) is kiteheti, ha a függvénynek több kifejezése van a függvény törzsben.

Soroljuk fel az arrow függvények összes változatát:

```
//1. Single parameter and single statement
var message = name => console.log("Hello, " + name + "!");
message("Sudheer"); // Hello, Sudheer!

//2. Multiple parameters and single statement
var multiply = (x, y) => x * y;
console.log(multiply(2, 5)); // 10


//3. Single parameter and multiple statements
var even = number => {
    if(number % 2) {
        console.log("Even");
    } else {
        console.log("Odd");
    }
}
even(5); // odd

//4. Multiple parameters and multiple statements
var divide = (x, y) => {
    if(y != 0) {
        return x / y;
    }
}
console.log(divide(100, 5)); // 20

//5. No parameter and single statement
var greet = () => console.log('Hello World!');
greet(); // Hello World!
```

Osztályok

Az osztályokat syntactic sugar-ként vezetik be a prototípuson alapuló öröklődés és konstruktor függvények felett. Tehát ez nem hoz új objektum-orientált öröklési modellt a JavaScript-be.

Az osztályok meghatározásának két módja van:

i. **Class declarations:**

```
class Square {
  constructor(length) {
    this.length = length;
  }

  get area() {
    return this.length * this.length;
  }

  set area(value) {
    this.area = value;
  }
}
```

ii. **Class expressions:**

```
const square = class Square {
  constructor(length) {
    this.length = length;
  }

  get area() {
    return this.length * this.length;
  }

  set area(value) {
    this.area = value;
  }
}
```

Az extend kulcsszóval használhatja az öröklést. Ez lehetővé teszi az alosztály számára, hogy megszerezze a szülőosztály összes tulajdonságát.

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

  start() {
    console.log(`${this.name} vehicle started`);
  }
}

class Car extends Vehicle {
  start() {
    console.log(`${this.name} car started`);
  }
}

const car = new Car('BMW');
console.log(car.start()); // BMW car started
```

Megjegyzés: Annak ellenére, hogy az ES6 osztályok hasonlítanak más objektumorientált nyelvek osztályaihoz, például Java, PHP stb., De nem teljesen ugyanúgy működnek.

Enhanced object literals

i. **Property Shorthand:**

Az objektum tulajdonságai gyakran azonos nevű változókból jönnek létre.
Lássuk az ES5 ábrázolást

```
var a = 1, b = 2, c = 3;
  obj = {
    a: a,
    b: b,
    c: c
  };
  console.log(obj);
```

és az alábbiakban rövidebb szintaxissal ábrázolható,

```
var a = 1, b = 2, c = 3;
  obj = {
    a,
    b,
    c
  };
  console.log(obj);
```

ii. **Method Shorthand:** Az ES5-ben az Object metódusok megkövetelik a függvény utasítást, az alábbiak szerint:

```
var calculation = {
  sum:  function(a, b) { return a + b; },
  multiply: function(a, b) { return a * b; }
};

console.log( calculation.add(5, 3) );  // 15
console.log( calculation.multiply(5, 3) ); // 15
```

Ez elkerülhető az ES6-ban,

```
var calculation = {
  sum(a, b) { return a + b; },
  multiply(a, b) { return a * b; }
};

console.log( calculation.add(5, 3) );  // 15
console.log( calculation.multiply(5, 3) ); // 15
```

iii. **Computed Property Names:** Az ES5-ben nem lehetett változót használni egy kulcshoz az objektum létrehozásának szakaszában.

```
var
  key = 'three',
  obj = {
    one: 1,
    two: 2
  };

obj[key] = 3;
```

Az objektumkulcsokat dinamikusan lehet hozzárendelni az ES6-ban szögletes zárójelbe helyezésével ([])

```
const
```

```
  key = 'three',
  computedObj = {
    one: 1,
    two: 2,
    [key]: 3
  };
```

Template literals

Az ES6 előtt a JavaScript fejlesztőknek csúnyán kellett összefűzniük a string-eket a dinamikus string-ekhez.

A sablon literálok (template literals) lehetővé teszik a beágyazott kifejezéseket, amelyeket dollárjel és kapcsos zárójelek ($ {kifejezés}) jelölnek. Ezeket a literálokat kettős vagy egyszeres idézőjelek helyett a backtick () karakter veszi körül.

Az ES6-nak két új típusú literálja van:

1.) Sablon literálok (template literals): karakterláncok, amelyek több soron keresztül is mehetnek, és interpolált kifejezéseket tartalmaznak (azaz $ {kifejezés})

```
const firstName = 'John';
console.log(`Hello ${firstName}!
Good morning!`);
```

2.) Tagged template literals: Függvény hívások, amelyek a sablon literál előtti függvény megemlítésével jönnek létre.

```
const Button = styled.a`
  display: inline-block;
  border-radius: 3px;
`
```

Destructuring

Destructuring egy javascript kifejezés, amely több érték kinyerésére szolgál az objektumokban (objektum tulajdonságai) és tömbökben tárolt adatokból.

1.) Object destructuring:

Ez a property értékek kivonására szolgál egy objektumból.

```
const user = { firstName: 'John', lastName: 'Kary' };
const {firstName, lastName} = user;
console.log(firstName, lastName); // John, Kary
```

2.) Array destructuring

Ez az értékek kinyerésére szolgál egy tömbből.

```
const [one, two, three] = ['one', 'two', 'three'];
console.log(one, two, three); // one, two, three
```

i. Variable declarations
ii. Assignments
iii. Parameter definitions
iv. for-of loop

Alapértelmezett paraméterek

Az alapértelmezett paraméterek lehetővé teszik egy függvény paramétereinek inicializálását alapértelmezett értékekkel, ha nincs átadva érték vagy undefined.

Az ES6 előtt ellenőriznie kell a definiálatlan értékeket, és meg kell adnia a definiálatlan értékek alapértelmezett értékét az if / else vagy a ternary operátor használatával

```
function add(a, b) {
  a = (typeof a !== 'undefined') ? a : 10;
  b = (typeof b !== 'undefined') ? b : 20;
  return a + b;
}
add(20); // 40
add(); // 30
```

Az ES6-ban ezek az ellenőrzések elkerülhetők az alapértelmezett paraméterek használatával

```
function add(a = 10, b = 20) {
  return a + b;
}
add(20); // 40
add(); // 30
```

Rest paraméter

A rest paraméter egy határozatlan számú argumentumot mutat tömbként. A fontos pont itt csak a függvény utolsó paramétere lehet "rest paraméter". Ezt azért vezették be, hogy csökkentse az argumentumok által kiváltott nehézkes kódot.

```
function sum(...args) {
  return args.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum(1, 2, 3)); // 6
console.log(sum(1, 2, 3, 4)); // 10
console.log(sum(1, 2, 3, 4, 5)); // 15
```

Spread Operator

A Spread Operator lehetővé teszi az iterálhatóak (tömbök / objektumok / karakterláncok) kibővítését egyetlen argumentumra / elemre.

A függvény és a konstruktor hívásokban a spread operátor az iterálható értékeket argumentumokká változtatja

```
console.log(Math.max(...[-10, 30, 10, 20])); //30
console.log(Math.max(-10, ...[-50, 10], 30)); //30
```

A tömb literálokban és karakterláncokban a spread operátor az iterálható értékeket tömb elemekké alakítja

```
console.log([1, ...[2,3], 4, ...[5, 6, 7]]); // 1, 2, 3, 4, 5, 6, 7
```

Iterators & For..of

A String, Array, TypedArray, Map és Set mind beépített iterable (iterálhatóak), de az objektumok alapértelmezés szerint nem iterable (iterálhatóak). Az iterátorok egy új módja annak, hogy iterálják a JavaScript bármely gyűjteményét. Ezek olyan objektumok, amelyek meghatároznak egy szekvenciát és egy visszatérési értéket annak leállításakor. Az iterátor az Iterator úgy valósítja meg, hogy rendelkezik egy next () metódussal, amely két tulajdonságú objektumot ad vissza:

value: Az iterációs sorrend következő értéke.

done: akkor adja vissza a true értéket, ha a sorozat utolsó értéke már elfogyott.

Az objektum iterálhatóvá tehető, ha meghatároz egy Symbol.iterator tulajdonságot rajta.

```
const collection = {
   one: 1,
   two: 2,
   three: 3,
   [Symbol.iterator]() {
     const values = Object.keys(this);
     let i = 0;
     return {
       next: () => {
         return {
           value: this[values[i++]],
           done: i > values.length
         }
       }
     };
   }
 };

 const iterator = collection[Symbol.iterator]();

 console.log(iterator.next());    // → {value: 1, done: false}
 console.log(iterator.next());    // → {value: 2, done: false}
 console.log(iterator.next());    // → {value: 3, done: false}
 console.log(iterator.next());    // → {value: undefined, done: true}

 for (const value of collection) {
   console.log(value);
 }
```

A for ...of utasítás létrehoz egy ciklust, amely a felhasználó által meghatározott gyűjteményobjektumon iterál. De ez a beépített objektumokhoz is használható.

Megjegyzés: Az iteráció hirtelen megszakadását break, throw vagy return okozhatja.

Generátorok

A generátor egy olyan függvény, amely megállhat, majd folytathatja onnan, ahol leállt, miközben fenntartja a kontextust . Meghatározható egy fuction kulcsszóval, amelyet csillag követ (azaz a function * ()).

Ez a függvény visszaad egy iterátor objektumot, és ez az iterátor next () metódusa egy olyan objektumot ad vissza, amelynek értéke olyan tulajdonságot tartalmaz, amely tartalmazza a kapott értéket és egy done tulajdonságot, amely jelzi, hogy a generátor megadta-e az utolsó értékét.

```
function* myGenerator(i) {
  yield i + 10;
  yield i + 20;
  return i + 30;
}

const myGenObj = myGenerator(10);

console.log(myGenObj.next().value); // 20
console.log(myGenObj.next().value); // 30
console.log(myGenObj.next().value); // 40
```

Modulok

A modulok független, újrafelhasználható kód kis egységei, amelyeket építőkockaként kell használni a Javascript alkalmazásban.

Az ES6 előtt nem volt natív modul támogatás a JavaScript-ben. 3 fő modul szabványt használtak,

- Asynchronous Module Definition (AMD)
- RequireJS Modules
- CommonJS Modules (modul.exports, és a Node.js-ben használt szintaxist igényel)

Az ES6 biztosította a modulok beépített támogatását. A modul belsejében minden alapértelmezés szerint privát, és strict módban fut. A publikus változókat, függvényeket és osztályokat az export-taltesszük ki, és ezeket importáljuk az import utasítás használatával.

Export Statement:

Kétféle export létezik:

1.) Named Exports: (nulla vagy több export modulonként)

Minden elemet vagy egyetlen exportálási utasítást exportálhat az összes elem egyszerre történő exportálásához

```
// module "my-module.js"

const PI = Math.PI;

function add(...args) {
  return args.reduce((num, tot) => tot + num);
}
```

```
function multiply(...args) {
  return args.reduce((num, tot) => tot * num);
}

// private function
function print(msg) {
  console.log(msg);
}

export { PI, add, multiply };
```

2.) Default Exports: (modulonként egy)

Ha egyetlen értéket akarunk exportálni, használhat alapértelmezett (default) exportálást.

```
// module "my-module.js"

export default function add(...args) {
                 return args.reduce((num, tot) => tot + num);
}
```

Import Statement:

A statikus import utasítás csak olvasható kötések importálására szolgál, amelyeket egy másik modul exportál.

Az importálási forgatókönyvek számos változata létezik, az alábbiak szerint:

```
// 1. Import an entire module's contents
import * as name from "my-module";

//2.Import a single export from a module
import { export1 } from "my-module";

//3.Import multiple exports from a module
import { export1 , export2 } from "my-module";

//4.Import default export from a module
import defaultExport from "my-module";

//5.Import an export with an alias
import { export1 as alias1 } from "my-module";
```

Set

A Set egy beépített objektum bármilyen típusú egyedi értékű gyűjtemények tárolására.

```
let mySet = new Set()

mySet.add(1);
mySet.add(2);
mySet.add(2);
mySet.add('some text here');
mySet.add({one: 1, two: 2 , three: 3});
console.log(mySet); // Set [ 1, 2, 'some text here', {one: 1, two: 2 , three: 3} ]
console.log(mySet.size) // 4
console.log(mySet.has(2)); // true
```

Weakset

A Set bármilyen típusú adat, például primitívek és objektumtípusok tárolására szolgál. Míg a WeakSet egy gyengén tartott tárgyak tárolására szolgáló objektum egy gyűjteményben. (vagyis a WeakSet csak az objektumok gyűjteménye). Itt a gyenge azt jelenti, hogy ha a WeakSet-ben tárolt objektumra más hivatkozás nem létezik, akkor ezeket az objektumokat a szemétgyűjtő (garbage collector) gyűjti.

```
let myUserSet = new WeakSet();

let john = { name: "John" };
let rocky = { name: "Rocky" };
let alex = { name: "Alex" };
let nick = { name: "Nick" };

myUserSet.add(john);
myUserSet.add(rocky);
myUserSet.add(john);
myUserSet.add(nick);

console.log(myUserSet.has(john)); // true
console.log(myUserSet.has(alex)); // false
console.log(myUserSet.delete(nick));
console.log(myUserSet.has(nick)); // false

john = null;
```

Map

A Map olyan elemek gyűjteménye, ahol minden elem kulcs - érték párként van tárolva. Az objektumokat és a primitív értékeket kulcsként vagy értékként is tarthatja, és elemeit iterációs sorrendben iterálja.

Vegyünk egy map-et különféle primitívekkel és objektumokkal, mint kulcs-érték párokat és különféle metódusokat rajta:

```
let typeMap = new Map();

 var keyObj = {'one': 1}

 typeMap.set('10', 'string');   // a string key
 typeMap.set(10, 'number');     // a numeric key
 typeMap.set(true, 'boolean'); // a boolean key
 typeMap.set(keyObj, 'object'); // an object key


 console.log(typeMap.get(10)   ); // number
 console.log(typeMap.get('10') ); // string
 console.log(typeMap.get(keyObj)) // object
 console.log(typeMap.get({'one': 1})) // undefined

 console.log(typeMap.size ); // 3

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


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

WeakMap

A WeakMap objektum kulcs / érték párok gyűjteménye, amelyben a kulcsok gyengén hivatkoznak. Ehhez az objektumhoz a kulcsoknak objektumoknak kell lenniük, és az értékek tetszőleges értékek lehetnek.

Nézzük meg a WeakMap különböző módszereit az alábbi példával:

```
var weakMap = new WeakMap();

var obj1  = {}
var obj2  = {}


weakMap.set(obj1, 1);
weakMap.set(obj2, 2);
weakMap.set({}, {"four": 4});

console.log(weakMap.get(obj2)); // 2
console.log(weakMap.has({})); // return false even though empty object exists as key. Because the keys
have different references

delete obj2;
console.log(weakMap.get(obj2)); // 2
weakMap.delete(obj1)
console.log(weakMap.get(obj1)); //undefined
```

Unicode

Az ES6 előtt a JavaScript karakterláncokat 16 bites karakterkódolás (UTF-16) képviseli. Minden karaktert 16 bites szekvencia képvisel, amelyet kódegységnek nevezünk. Mivel a karakterkészletet az Unicode kibővítette, váratlan eredményeket kaphatunk az UTF-16 kódolású karaktersorozatoknál, amelyek helyettesítő párokat tartalmaznak (vagyis mivel nem elegendő bizonyos karaktereket csak 16 bitben ábrázolni, két 16 bites kódegységre van szükség).

```
let str = '吉';

console.log(str.length);             // 2
console.log(text.charAt(0));        // ""
console.log(text.charAt(1));        // ""
console.log(text.charCodeAt(0));    // 55362(1st code unit)
console.log(text.charCodeAt(1));    // 57271(2nd code unit)

console.log(/^.$/.test(str)); // false, because length is 2
console.log('\u20BB7); // 7!(wrong value)
console.log(str === '\uD842\uDFB7'); // true
```

Az ECMAScript 6 teljes támogatást adott az UTF-16 számára a string-eken és a reguláris kifejezéseken belül. Bevezeti az új Unicode-t karakterláncokban és az új RegExp u módot, valamint új API-kat (codePointAt, fromCodePoint) a karakterláncok feldolgozásához.

```
let str = '吉';

// new string form
console.log('\u{20BB7}'); // "吉"

// new RegExp u mode
console.log(new RegExp('\u{20BB7}', 'u'));
console.log(/^.$/u.test(str)); // true

//API methods
```

```
console.log(str.codePointAt(0)); // 134071
console.log(str.codePointAt(1)); // 57271

console.log(String.fromCodePoint(134071));  // "吉"
```

Szimbólumok

A Symbol egy új sajátos primitív adattípus a JavaScript-en, más primitív típusokkal együtt, mint a string, szám, logikai érték, null és undefined. Az új szimbólum csak a Symbol függvény meghívásával jön létre. vagyis minden alkalommal, amikor a Symbol függvényt meghívja, új és teljesen egyedi értéket kap. Átadhat egy paramétert a Symbol () -nak is, amely csak hibakeresési célokra hasznos.

Annak ellenére, hogy két szimbólum egyenlőség-ellenőrzése mindig hamis, akkor viszont igaz less ha a szimbólumokat a .for metódussal hasonlítjuk össze (azaz: Symbols.for ('key') === Symbols.for ('key'))

```
//1. Object properties
let id = Symbol("id");
let user = {
  name: "John",
  age: 40,
  [id]: 111
};

for (let key in user) {
console.log(key); // name, age without symbols
}

console.log(JSON.stringify(user)); // {"name":"John", "age": 40}
console.log(Object.keys(user)); // ["name", "age"]

console.log( "User Id: " + user[id] ); // Direct access by the symbol works

//2. Unique constants
const logLevels = {
  DEBUG: Symbol('debug'),
  INFO: Symbol('info'),
  WARN: Symbol('warn'),
  ERROR: Symbol('error'),
};
console.log(logLevels.DEBUG, 'debug message');
console.log(logLevels.INFO, 'info message');

//3. Equality Checks

console.log(Symbol('foo') === Symbol('foo'));  // false
console.log(Symbol.for('foo') === Symbol.for('foo'));  // true
```

Proxy

A Proxy objektum egy proxy létrehozására szolgál egy másik objektum számára, amely képes elfogni és újradefiniálni az adott objektum alapvető műveleteit, például a tulajdonságok hozzárendelését, felsorolását, függvényhívást stb. Ezeket számos könyvtár és néhány böngésző keretrendszer használja.

A proxyobjektum:

```
let proxy = new Proxy(target, handler)
```

- target: Objektum, amelyet proxy-ként szeretnénk
- handler: Olyan objektum, amely meghatározza, hogy mely műveletek lesznek elkapottak és hogyan definiálja azokat újra.

A felhasználó által proxyzott objektum keresési viselkedése az alábbiak szerint alakul:

```
onst target = {
  name: "John",
  age: 3
};

const handler = {
  get: function(target, prop) {
    return prop in target ?
        target[prop] :
        `${prop} does not exist';
  }
};

const user = new Proxy(target, handler);
console.log(user.name); // John
console.log(user.age); // John
console.log(user.gender); // gender does not exist
```

Ezek a proxy-k az értékellenőrzéseket is kikövetelhetnek. Vegyünk egy példát a set handlerre:

```
let ageValidator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('Invalid age');
      }
    }

    obj[prop] = value; // The default behavior to store the value

    return true; // Indicate success
  }
};

const person = new Proxy({}, validator);

person.age = 30;
console.log(person.age); // 30
person.age = 'old';    // Throws an exception
person.age = 200;        // Throws an exception
```

Promise

A Promise amely egy aszinkron művelet esetleges befejezését vagy meghiúsulását jelenti.

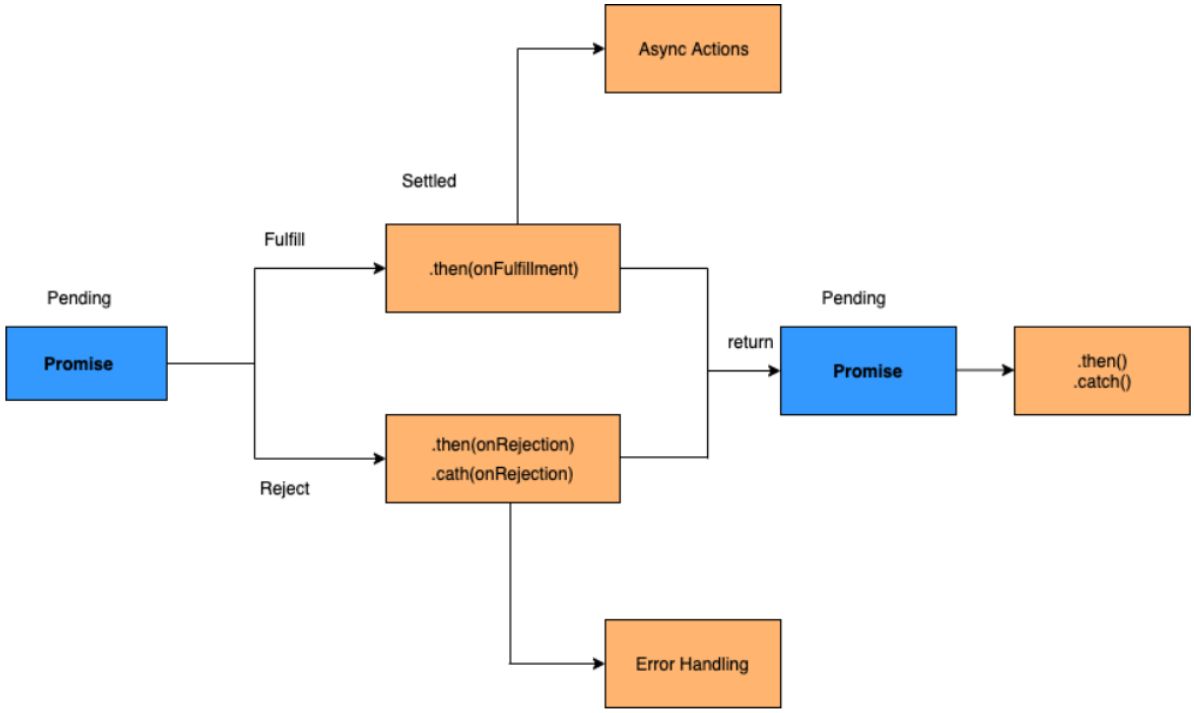
A következő állapotok egyikében van:

pending: A kezdeti állapotot képviseli, nem fulfilled és nem rejected.

fulfilled: Azt jelzi, hogy a művelet sikeresen befejeződött.

rejected: A művelet sikertelenségét jelzi.

Azt mondják, hogy egy Primise teljesül (settled), ha fulfilled vagy rejected, de nem pending. Az promise.then(), promise.catch(), and promise.finally() példánymetódusok arra szolgálnak, hogy a további cselekvéseket egy promise-hoz társítsák, amely teljesült (settled). És ezek a módszerek egy újonnan létrehozott promise objektumot is visszaküldenek, amely opcionálisan felhasználható a láncoláshoz.



promise chaining structure:

```
const promise = new Promise(function(resolve, reject) {
                setTimeout(() => resolve(1), 1000);
            });

promise.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;

    }).catch(function(error){
       console.log(error);
    });
```

Reflect

A Reflect egy, a kód azon képessége, hogy futás közben megvizsgálja és manipulálja az objektumok változóit, tulajdonságait és metódusait. A JavaScript már klasszikus reflexiós funkcióként biztosítja az Object.keys (), az Object.getOwnPropertyDescriptor () és az Array.isArray () metódusokat. Az ES6-

ban hivatalosan a Reflect objektum révén biztosított. A Reflect egy új globális objektum, amelyet metódusok meghívására, objektumok szerkesztésére, tulajdonságok lekérésére és beállítására, tulajdonságok manipulálására és kiterjesztésére használnak.

A legtöbb globális objektumtól eltérően a Reflect nem konstruktor. azaz nem használhatja a Reflect-et a new operátorral, és nem hívhatja meg a Reflect-et függvényként. Hasonló a Math és a JSON objektumokhoz, amelyekben az objektum összes metódusa statikus.
Nézzük meg a Reflect API használatát az alábbi példákkal:

1.) Objektumok létrehozása a Reflect.construct () használatával;

A construct () metódus úgy viselkedik, mint a normál new operátor, de függvényként. Ez egyenértékű az new target(...args) hívásával egy másik prototípus megadásának lehetőségével. A szintaxis az alábbiak szerint néz ki:

```
Reflect.construct(target, args [, newTarget]);
```

Paraméterei:

- target: A hívandó célfüggvény.
- argumentList: tömbszerű objektum, amely meghatározza azokat az argumentumokat, amelyekkel a target-et meg kell hívni.
- newTarget: A konstruktor, amelynek prototípusát fel kell használni. Ez egy opcionális paraméter. vagyis ha az newTarget nincs jelen, akkor az értéke alapértelmezés szerint a target.

```
class User {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }
};

let args = ['John', 'Emma'];

let john = Reflect.construct(
    User,
    args
);

console.log(john instanceof User);
console.log(john.fullName); // John Doe
```

2.) Függvény meghívása a Reflect.apply () használatával: Az ES6 előtt a Function.prototype.apply () metódus segítségével meghívhat egy függvényt egy megadott értékkel és argumentumokkal.

Például meghívhatja a Math-nak a max() statikus metódust:

```
const max = Function.prototype.apply.call(Math.max, Math, [100, 200, 300]);
console.log(max);
```

Az ES6-ban a Reflect.apply () ugyanazokat a szolgáltatásokat nyújtja, mint a Function.prototype.apply (), de kevésbé bonyolult szintaxissal.

```
const max = Reflect.apply(Math.max, Math, [100, 200, 300]);
console.log(max);
```

3.) Property meghatározása a Reflect.defineProperty () használatával: A Reflect.defineProperty () metódus hasonló az Object.defineProperty () metódushoz, de egy logikai értéket ad vissza, jelezve, hogy a property-t sikeresen definiálták-e.

A metódus szintaxisa az alábbiak szerint néz ki,

```
Reflect.defineProperty(target, propertyName, propertyDescriptor)
```

Határozzuk meg a age tulajdonságot a user objektumon,

```
class User {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    get fullName() {
        return `${this.firstName} ${this.lastName}`;
    }
};

let john = new User('John', 'Resig');

if (Reflect.defineProperty(john, 'age', {
        writable: true,
        configurable: true,
        enumerable: false,
        value: 33,
    })) {
    console.log(john.age);
} else {
    console.log('Cannot define the age property on the user object.');

}
```

4.) Property törlése a Reflect.deleteProperty () használatával:

A Reflect.deleteProperty () metódust olyan property-k törlésére használják, mint a delete operátor, de ez függvény. Logikai értéket ad vissza, jelezve, hogy a property-t sikeresen törölték-e.

```
const user = {
  name: 'John',
  age: 33
};

console.log(Reflect.deleteProperty(user, 'age')); // true
console.log(user.age); // undefined
```

5.) Objektum tulajdonságának megszerzése a Reflect.get () használatával: A Reflect.get metódust arra használjuk, hogy olyan tulajdonságokat kapjunk egy objektumon, mint a tulajdonság-hozzáférés szintaxisa, de függvényként.

```
const user = {
```

```
  name: 'John',
  age: 33
};

console.log(Reflect.get(user, 'age')); // 33
```

Bináris és oktális

Az ES5 numerikus literálokat adott oktális (0 előtag), tizedes (előtag nélküli) és hexadecimális (0x) ábrázolással. Az ES6 támogatást adott a bináris literálokhoz és az oktális literálok fejlesztéseihez.

1. Bináris literálok:

Az ES5 előtt a JavaScript nem biztosította bináris számok szó szerinti formáját. Tehát bináris karakterláncot kell használnia a parseInt () segítségével

```
const num = parseInt('110',2);
console.log(num); // 6
```

Míg az ES6 hozzáadta a bináris literálok támogatását a 0b előtaggal, amelyet bináris számok szekvenciája követett (azaz 0 és 1).

```
const num = 0b110;
console.log(num); // 6
```

2. Oktális literálok:

Az ES5-ben egy oktális literál ábrázolásához használja a nulla előtagot (0), amelyet oktális számjegyek sora követ (0 és 7 között).

```
const num = 055;
console.log(num); // 45

let invalidNum = 058;
console.log(invalidNum); // treated as decimal 58
```

Míg az ES6 az oktális literált jelenti, a 0o előtag használatával, amelyet 0 és 7 közötti oktális számjegyek követnek.

```
const num = 055;
console.log(num); // 45

const invalidNum = 028;
console.log(invalidNum); // treated as decimal 28
```

Ne feledje, ha érvénytelen számot használ az oktális literálban, a JavaScript egy SyntaxError-t dob az alábbiak szerint,

```
const invalidNum = 028;
console.log(invalidNum); // SyntaxError
```

Proper Tail Calls

A Proper Tail Calls (PTC) egy olyan technika, ahol a program vagy a kód nem hoz létre további veremkereteket a rekurzióhoz, ha a függvényhívás Proper Tail Call.

Például a tényleges függvény alábbi klasszikus rekurziója az egyes lépések veremére támaszkodik. Minden lépést n * factorial(n - 1) kell feldolgozni

```
function factorial(n) {
    if (n === 0) {
      return 1
    }
    return n * factorial(n - 1)
  }
  console.log(factorial(5)); //120
```

De ha a Tail rekurziós funkciókat használja, akkor a rekurzióban folyamatosan továbbítják az összes szükséges adatot anélkül, hogy a veremre támaszkodnának.

```
function factorial(n, acc = 1) {
    if (n === 0) {
      return acc
    }
    return factorial(n - 1, n * acc)
  }
  console.log(factorial(5)); //120
```

A fenti minta ugyanazt a kimenetet adja vissza, mint az első. De nyomon követi az összeget argumentumként anélkül, hogy rekurzív hívásoknál verem memóriát használna.

A PTC-t támogató böngészők nem generálnak verem túlcsordulást

```
 console.log(factorial(10));
 console.log(factorial(100));
 console.log(factorial(1000));
 console.log(factorial(10000));
```

ES2016 Or ES7

Az ES2015 / ES6 hatalmas újdonságokat mutatott be. De az ECMAScript 2016 vagy az ES7 csak két új funkciót vezetett be:

- Az Array.prototype.includes ()
- Hatványozási operator

Array Includes

Az ES7 előtt az indexOf metódust kell használnia, és össze kell hasonlítania az eredményt a -1 értékkel annak ellenőrzésére, hogy egy tömb elem tartalmaz-e egy adott elemet vagy sem.

```
const array = [1,2,3,4,5,6];
 if(array.indexOf(5) > -1 ){
 console.log("Found an element");
 }
```

Míg az ES7-ben az array.prototype.includes () módszert vezetik be közvetlen megközelítésként annak meghatározására, hogy egy tömb tartalmaz-e egy bizonyos értéket a bejegyzései között, vagy sem.

```
const array = [1,2,3,4,5,6];
if(array.includes(5)){
console.log("Found an element");
}
```

Ezen felül az Array.prototype.includes () jobban kezeli a NaN és a undefined értékeket, mint az Array.prototype.indexOf () metódus. azaz ha a tömb tartalmaz NaN és undefined értékeket, akkor az indexOf () nem adja vissza a megfelelő indexet, miközben NaN és Undefined keresést végez.

```
let numbers = [1, 2, 3, 4, NaN, ,];
console.log(numbers.indexOf(NaN)); // -1
console.log(numbers.indexOf(undefined)); // -1
```

Másrészt, az includes() módszer képes megtalálni ezeket az elemeket

```
let numbers = [1, 2, 3, 4, NaN, ,];
  console.log(numbers.includes(NaN)); // true
  console.log(numbers.includes(undefined)); // true
```

Hatványozási operátor

A javascript régebbi verziói a Math.pow függvény segítségével találják meg az adott számok hatványozását. Az ECMAScript 2016 bevezette az ** hatványozási operátort (hasonlóan más nyelvekhez, mint a Python vagy az F #).

```
//Prior ES7
const cube = x => Math.pow(x, 3);
console.log(cube(3)); // 27

//Using ES7
const cube1 = x => x ** 3;
console.log(cube1(3)); // 27
```

ES2017 Or ES8

Async függvények

Az ES6-ban bevezették a Promise-t a callback hell probléma megoldására. Ha egymásba ágyazott aszinkron függvényeket kell végrehajtani, az callback hell-hez vezet

```
function task() {
  task1((response1) => {
    task2(response1, (response2) => {
      task3(response2, (response3) => {
        // etc...
      };
    });
  });
}
```

De a chained promise-ok összetett folyamatot hoznak létre az aszinkron kód számára.

Az aszinkron függvényeket promise-ok és generátorok kombinációjaként vezették be, hogy lehetővé tegyék számunkra az aszinkron művelet szinkron módon történő írását. azaz ezt a függvényt az async kulcsszóval deklaráljuk, amelyek lehetővé teszik az aszinkron, promise alapú viselkedés tisztább stílusú megírását az promise láncolások elkerülése révén. 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();
```

Objektumértékek

Hasonlóan az Object.keys-hez, amely a JavaScript objektum kulcsain keresztül ismétlődik, az Object.values ugyanazt fogja tenni az értékeken. vagyis az Object.values () metódus bevezetésével egy adott objektum saját felsorolható tulajdonságértékeinek tömbjét adja vissza ugyanabban a sorrendben, mint a for...in ciklusban.

```
const countries = {
   IN: 'India',
   SG: 'Singapore',
 }
 Object.values(countries) // ['India', 'Singapore']
```

Egyébként a nem objektum argumentum kényszerítve lesz egy objektumra:

```
console.log(Object.values(['India', 'Singapore'])); // ['India', 'Singapore']
 console.log(Object.values('India')); // ['I', 'n', 'd', 'i', 'a']
```

Object entries

Az Object.entries () metódus bevezetésével visszavezet egy adott objektum saját felsorolható karakterlánc-kulcsú tulajdonságának [kulcs, érték] tömbjét ugyanazon sorrendben, mint a for...in ciklus.

```
const countries = {
      IN: 'India',
      SG: 'Singapore',
    }
    Object.entries(countries) // [["IN", "India"], ["SG", "Singapore"]]
```

Egyébként a nem objektum argumentum kényszerítve lesz egy objektumra

```
   const countriesArr = ['India', 'Singapore'];
   console.log(Object.entries(countriesArr)); // [ ['0', 'India'], ['1', 'Singapore']]

   const country = 'India';
   console.log(Object.entries(country)); // [["0", "I"], ["1", "n"], ["2", "d"], ["3", "i"], ["4",
"a"]]

   console.log(Object.entries(100)); // [], an empty array for any primitive type because it won't
have any own properties
```

Az objektum tulajdonságainak leírói

A tulajdonságleírók egy tulajdonság attribútumait írják le. Az Object.getOwnPropertyDescriptors () metódus egy adott objektum összes saját tulajdonságleíróját adja vissza.

Az alábbi attribútumokat nyújtja:

- value: A tulajdonsághoz társított érték (csak adatleírók).
- writable: akkor és csak akkor igaz, ha a tulajdonsághoz társított érték megváltoztatható
- get: Az ingatlan getter függvénye.
- set: Az ingatlan setter függvénye
- configurable: igaz akkor és csak akkor, ha a tulajdonságleíró típusa megváltoztatható vagy törölhető.
- enumerable: akkor és csak akkor igaz, ha ez a tulajdonság megjelenik a tulajdonság felsorolása során.

A tulajdonságleírók megtalálása bármely property esetében az alábbiak szerint történik:

```
const profile = {
   age: 42
 };

 const descriptors = Object.getOwnPropertyDescriptors(profile);
 console.log(descriptors); //  {age: {configurable: true, enumerable: true, writable: true }}
```

String padding

Bizonyos karakterláncokat és számokat (pénz, dátum, időzítők stb.) egy adott formátumban kell megjeleníteni. Mindkét padStart () és padEnd () metódust bevezették, hogy egy sztringet egy másik karaktersorozattal kitöltenek, amíg az eredmény nem éri el a megadott hosszt.

1.) padStart (): Ezzel a módszerrel a padding-et a karakterlánc bal vagy kezdő oldalára alkalmazzák.

Például biztonsági okokból érdemes csak a hitelkártya számának négy utolsó számjegyét megjeleníteni,

```
const cardNumber = '01234567891234';
const lastFourDigits = cardNumber.slice(-4);
const maskedCardNumber = lastFourDigits.padStart(cardNumber.length, '*');
console.log(maskedCardNumber); // expected output: "**********1234"
```

2.) padEnd (): Ezzel a módszerrel a padding-et a karakterlánc jobb vagy befejező oldalára alkalmazzák.

Például a profiladatok a címke és az értékek szerint vannak kitöltve, az alábbiak szerint:

```
const label1 = "Name";
const label2 = "Phone Number";
const value1 = "John"
const value2 = "(222)-333-3456";

console.log((label1 + ': ').padEnd(20, ' ') + value1);
```

```
console.log(label2 + ": " + value2); // Name:                John
                                     // Phone Number: (222)-333-3456
```

Shared memory and atomics

Az Atomics egy globális objektum, amely statikus módszerként végrehajtandó atomi műveleteket biztosít. SharedArrayBuffer (fix hosszúságú bináris adatpuffer) objektumokkal használják őket. E módszerek fő felhasználási esetei a következők:

1.) atomi műveletek: Ha a memória osztott, akkor több szál olvashatja és írhatja ugyanazt az adatot a memóriába. Tehát fennáll az adatok elvesztésének esélye. De az atomi műveletek biztosítják, hogy ha értékeket írják és olvassák, a műveletek befejeződjenek a következő művelet megkezdése előtt, és hogy a műveletek ne szakadjanak meg.

Olyan statikus metódusokat biztosít, mint például az add, or, and, xor, load, store, isLockFree stb.:

```
const sharedMemory = new SharedArrayBuffer(1024);
const sharedArray = new Uint8Array(sharedMemory);
sharedArray[0] = 10;

Atomics.add(sharedArray, 0, 20);
console.log(Atomics.load(sharedArray, 0)); // 30

Atomics.sub(sharedArray, 0, 10);
console.log(Atomics.load(sharedArray, 0)); // 20

Atomics.and(sharedArray, 0, 5);
console.log(Atomics.load(sharedArray, 0));  // 4

Atomics.or(sharedArray, 0, 1);
console.log(Atomics.load(sharedArray, 0));  // 5

Atomics.xor(sharedArray, 0, 1);
console.log(Atomics.load(sharedArray, 0)); // 4

Atomics.store(sharedArray, 0, 10); // 10

Atomics.compareExchange(sharedArray, 0, 5, 10);
console.log(Atomics.load(sharedArray, 0)); // 10

Atomics.exchange(sharedArray, 0, 10);
console.log(Atomics.load(sharedArray, 0)); //10

Atomics.isLockFree(1); // true
```

2.) waiting to be notified: Mind a wait (), mind az notify () metódus lehetőséget nyújt arra, hogy várjon, amíg egy bizonyos feltétel igaz lesz, és általában blokkoló konstrukciókként használják őket.

Bemutassuk ezt a funkciót olvasási és írási szálakkal.

Először határozzon meg egy osztott memóriát és tömböt

```
const sharedMemory = new SharedArrayBuffer(1024);
const sharedArray = new Int32Array(sharedMemory);
```

Az olvasási szál alszik és vár a 0 helyen, amely várhatóan 10 lesz. Az író szál által felülírt érték után más értéket is megfigyelhet.

```
Atomics.wait(sharedArray, 0, 10);
```

```
console.log(sharedArray[0]); // 100
```

Most egy író szál új értéket tárol (pl. 100), és értesíti a várakozó szálat,

```
Atomics.store(sharedArray, 0, 100);
Atomics.notify(sharedArray, 0, 1);
```

Trailing commas

A paraméterek meghatározásában és a függvényhívásokban a vesszők megengedettek:

```
function func(a,b,) { // declaration
  console.log(a, b);
}
func(1,2,); // invocation
```

De ha a függvényparaméter-meghatározás vagy a függvényhívás csak vesszőt tartalmaz, akkor szintaktikai hiba lép fel

```
function func1(,) {  // SyntaxError: missing formal parameter
  console.log('no args');
};
func1(,); // SyntaxError: expected expression, got ','
```

Megjegyzés: Nem használhatók a rest paraméterek és a JSON-ban.

ES2018 Or ES9

Aszinkron iterátorok (Async iterators)

Az ECMAScript 6 beépített támogatást nyújt az adatok szinkron ismétléséhez az iterátorok segítségével. A karakterláncok és a gyűjteményobjektumok, például a Set, a Map és az Array, egy Symbol.iterator tulajdonsággal rendelkeznek, amely iterálhatóvá teszi őket.

```
const arr = ['a', 'b', 'c', 'd'];
 const syncIterator = arr[Symbol.iterator]();

 console.log(syncIterator.next());    //{value: a, done: false}
 console.log(syncIterator.next());    //{value: b, done: false}
 console.log(syncIterator.next());    //{value: c, done: false}
 console.log(syncIterator.next());    //{value: d, done: false}
 console.log(syncIterator.next());    //{value: undefined, done: true}
```

De ezek az iterátorok csak szinkron adatforrások ábrázolására alkalmasak.

Az aszinkron adatforrásokhoz való hozzáférés érdekében az ES2018 bevezette az AsyncIterator interfészt, egy aszinkron iterációs utasítást (for-await-of) és aszinkron generátor függvényeket.

Object rest and spread operators

Az ES2015 vagy az ES6 mind a rest paramétert, mind a spread operátorokat bevezette az argumentumok tömbre és fordítva konvertálására hárompontos (...) jelöléssel.

1.) A rest paraméterekkel a függvény argumentumait tömbbé konvertálhatjuk:

```
function myfunc(p1, p2, ...p3) {
   console.log(p1, p2, p3); // 1, 2, [3, 4, 5, 6]
 }
myfunc(1, 2, 3, 4, 5, 6);
```

2.) A spread operátor éppen ellenkezõleg mûködik, ha egy tömböt külön argumentummá alakít át annak érdekében, hogy átadja egy függvénynek:

```
const myArray = [10, 5, 25, -100, 200, -200];
 console.log( Math.max(...myArray) ); // 200
```

Az ES2018 lehetővé teszi ezt a rest/spread viselkedést az objektumok számára is.

1.) Az objektumot átadhatja egy függvénynek:

```
function myfunc1({ a, ...x }) {
     console.log(a, x); // 1, { b: 2, c: 3, d:4 }
 }
 myfunc1({
   a: 1,
   b: 2,
   c: 3,
   d: 4
 });
```

2.) A spread operátor más objektumokon belül is használható:

```
const myObject = { a: 1, b: 2, c: 3, d:4 };
const myNewObject = { ...myObject, e: 5 }; // { a: 1, b: 2, c: 3, d: 4, e: 5 }
```

Promise finally

Előfordulhat, hogy el kell kerülnie a duplikált kódot az then () és catch () metódusokban.

```
myPromise
    .then(result => {
        // process the result and then clean up the resources
    })
    .catch(error => {
        // handle the error and then clean up the resources
    });
```

A finally () metódus akkor hasznos, ha valamilyen feldolgozást vagy erőforrás-tisztítást szeretne végrehajtani, amint a Promise teljesül (settled) (vagyis fulfilled vagy rejected).

```
let isLoading = true;
  fetch('http://somesite.com/users')
     .then(data => data.json())
     .catch(err => console.error(err))
```

```
    .finally(() => {
        isLoading = false;
        console.log('Finished loading!!');
      })
```

ES2019 Or ES10

Array flat and flatMap

Az ES2019 előtt reduce () vagy concat () módszereket kell használnia egy flat array megszerzéséhez.

```
function flatten(arr) {
  const flat = [].concat(...arr);
  return flat.some(Array.isArray) ? flatten(flat) : flat;
}
flatten([ [1, 2, 3], ['one', 'two', 'three', [22, 33] ], ['a', 'b', 'c'] ]);
```

Az ES2019-ben a flat () függvényt vezetik be a beágyazott tömbök „simítására (flattens)" a legfelső szintű tömbbe. A függvény funkcionalitása hasonló a Lodash _.flattenDepth () függvényéhez. Ez a függvény elfogad egy opcionális argumentumot, amely meghatározza a beágyazott tömb elsimított (flatten) szintjeinek számát, és az alapértelmezett beágyazott szint 1.

Megjegyzés: Ha a tömbben vannak üres helyek, azokat eldobja.

```
const numberArray = [[1, 2], [[3], 4], [5, 6]];
 const charArray = ['a', , 'b', , , ['c', 'd'], 'e'];
 const flattenedArrOneLevel = numberArray.flat(1);
 const flattenedArrTwoLevel = numberArray.flat(2);
 const flattenedCharArrOneLevel = charArray.flat(1);

 console.log(flattenedArrOneLevel);    // [1, 2, [3], 4, 5, 6]
 console.log(flattenedArrTwoLevel);    // [1, 2, 3, 4, 5, 6]
 console.log(flattenedCharArrOneLevel);    // ['a', 'b', 'c', 'd', 'e']
```

Míg a flatMap () metódus a map () és a flat () metódust egyesíti egyetlen módszerben. Először létrehoz egy új tömböt egy adott függvény visszatérési értékével, majd összefűzi a tömb összes résztömb elemét.

```
const numberArray1 = [[1], [2], [3], [4], [5]];

console.log(numberArray1.flatMap(value => [value * 10]));  // [10, 20, 30, 40, 50]
```

Object fromEntries

A JavaScript-ben nagyon általános az adatok egy formátumból történő átalakítása. Az ES2017 bevezette az Object.entries () metódust az objektumokba tömbökbe.

1.) Object to Array:

```
    const obj = {'a': '1', 'b': '2', 'c': '3' };
    const arr = Object.entries(obj);
    console.log(obj); // [ ['a', '1'], ['b', '2'], ['c', '3'] ]
```

De ha vissza akarod szerezni az objektumot egy tömbből, akkor iterálnod kell és konvertálnod kell az alábbiak szerint:

```
const arr = [ ['a', '1'], ['b', '2'], ['c', '3'] ];
let obj = {}
for (let [key, val] of arr) {
    obj[key] = val;
}
console.log(obj);
```

Egyszerű módra van szükségünk az iteráció elkerülésére. Az ES2019-ben bevezetik az Object.fromEntries () módszert, amely az Object.entries () viselkedés fordítottját hajtja végre. A fenti ciklus könnyen elkerülhető, ahogy fent láttuk.

2.) Iterable(e.g Array or Map) to Object

```
const arr = [ ['a', '1'], ['b', '2'], ['c', '3'] ];
 const obj = Object.fromEntries(arr);
 console.log(obj); // { a: "1", b: "2", c: "3" }
```

A módszer használatának egyik leggyakoribb esete az URL lekérdezési paramétereinek kezelése,

```
const paramsString = 'param1=foo&param2=baz';
 const searchParams = new URLSearchParams(paramsString);

 Object.fromEntries(searchParams);    // => {param1: "foo", param2: "baz"}
```

String trimStart és trimEnd

A padStart / padEnd konzisztencia érdekében az ES2019 biztosította a trimStart és trimEnd elnevezésű standard függvényeket a karakterlánc elején és végén lévő white space levágására.

```
//Prior ES2019
 let messageOne = "   Hello World!!    ";
 console.log(messageOne.trimLeft()); //Hello World!!
 console.log(messageOne.trimRight()); //   Hello World!!

 //With ES2019
 let messageTwo = "   Hello World!!    ";
 console.log(messageTwo.trimStart()); //Hello World!!
 console.log(messageTwo.trimEnd()); //   Hello World!!
```

Szimbólum leírása

A szimbólumok létrehozása közben hibakeresés céljából leírást is hozzáadhat hozzá. Az ES2019 előtt azonban nem volt módszer a leírás közvetlen elérésére. Ezt figyelembe véve az ES2019 csak olvasható leírás tulajdonságot vezetett be a szimbólum leírását tartalmazó karaktersorozat lekérésére.

Ez lehetővé teszi a szimbólumleírás hozzáférését a Symbol objektumok különböző változataihoz:

```
console.log(Symbol('one').description); // one

console.log(Symbol.for('one').description); // "one"

console.log(Symbol('').description); // ''
```

```
console.log(Symbol().description); // unefined

console.log(Symbol.iterator.description); // "Symbol.iterator"
```

Optional catch binding

Az ES9 előtt, ha nincs szüksége hibaváltozóra és kihagyja ugyanazt a változót, akkor a catch () záradék nem lesz meghívva. A probléma elkerülése érdekében bevezetésre kerül az optional catcha binding, hogy a binding paraméter opcionális legyen a catch záradékban. Ha teljesen figyelmen kívül akarja hagyni a hibát, vagy már ismeri a hibát, de csak reagálni szeretne arra, akkor ez a funkció hasznos lesz.

Lássuk az alábbi szintaxis különbséget a verziók között,

```
// With binding parameter(<ES9)
 try {
     ...
 } catch (error) {
     ...
 }
 // Without binding parameter(ES9)
 try {
     ...
 } catch {
     ...
 }
```

Például a böngészőben a feature észlelése az egyik leggyakoribb eset:

```
let isTheFeatureImplemented = false;
 try {
   if(isFeatureSupported()) {
    isTheFeatureImplemented = true;
    }
 } catch (unused) {}
```

JSON fejlesztések

A JSON-t könnyű formátumként (lightweight) használják az adatcserére (olvasásra és elemzésre). Az ECMAScript specifikáció részeként javult a JSON használata. Alapvetően 2 fontos változás kapcsolódik a JSON-hoz.

1.) JSON Superset

Az ES2019 előtt az ECMAScript az, hogy JSON a JSON.parse részhalmaza, ez nem igaz. Mivel az ECMAScript karaktersorozat-literálok a JSON-karakterláncokkal ellentétben nem tartalmazhatták az U + 2028 (LINE SEPARATOR) és az U + 2029 (PARAGRAPH SEPARATOR) karaktereket. Ha továbbra is ezeket a karaktereket használja, akkor szintaktikai hiba lép fel. Kerülő megoldásként egy escape sequence-t kellett használnia, hogy karakterláncba helyezze őket.

```
eval('"\u2028"'); // SyntaxError
```

Míg a JSON karakterláncok tartalmazhatnak U + 2028 és U + 2029 anélkül, hogy hibákat eredményeznek.

```
console.log(JSON.parse('"\u2028"')); // ''
```

Ezt a korlátozást eltávolították az ES2019-ből. Ez egyszerűsíti a specifikációt anélkül, hogy külön szabályokra lenne szükség az ECMAScript sztring literálókhoz és a JSON string literálokhoz.

2.) Well Formed JSON.Stringify (): Az ES2019 előtt a JSON.stringify metódust a nem formázott Unicode karakterláncok (rosszul formázott Unicode karakterláncok) visszaadására használják.

```
console.log(JSON.stringify("\uD800")); // '"�"'
```

Míg az ES2019-ben a JSON.stringify kimenetek érvényessé válik Unicode-ként és reprezentálható az UTF-8-ban.

```
console.log(JSON.stringify("\uD800")); // '"\ud800"'
```

Tömbös rendezés

A tömbök rendezési módszere stabil az ES2020-ban. azaz, ha rendelkezik egy tömb objektummal, és rendezi őket egy adott kulcson, akkor a lista elemei megtartják pozíciójukat a többi objektumhoz képest ugyanazzal a kulccsal. Most a tömb a stabil TimSort algoritmust használja 10 feletti tömbökhöz az instabil QuickSort helyett.

Lássunk egy példát :

```
const users = [
    { name: "Albert",  age: 30 },
    { name: "Bravo",   age: 30 },
    { name: "Colin",   age: 30 },
    { name: "Rock",    age: 50 },
    { name: "Sunny",   age: 50 },
    { name: "Talor",    age: 50 },
    { name: "John",   age: 25 },
    { name: "Kindo",  age: 25 },
    { name: "Lary",   age: 25 },
    { name: "Minjie",   age: 25 },
    { name: "Nova",    age: 25 }
]
users.sort((a, b) => a.age - b.age);
```

Function.toString ()

A függvényeknek van egy toString () nevű példánymódszere, amely egy karakterláncot ad vissza a függvénykódot képviselve. Az ECMAScript korábbi verziói eltávolítják a szóközöket, az új sorokat és megjegyzéseket a függvény kódból, de az ES2020-ban az eredetit adják vissza.

```
function sayHello(message) {
    let msg = message;
    //Print message
    console.log(`Hello, ${msg}`);
}

console.log(sayHello.toString());
```

```
// function sayHello(message) {
//       let msg = message;
//       //Print message
//       console.log(`Hello, ${msg}`);
//   }
```

Privát osztály változók

Az ES6-ban az osztályokat újrafelhasználható modulok létrehozására vezetik be, és a változókat a clousure-ban deklarálják, hogy azok privátok legyenek. Ahol az ES2020-hoz hasonlóan, a privát osztály változókat vezetik be, hogy csak az osztályban használt változókat engedélyezzék. Ha csak egy egyszerű hash szimbólumot ad hozzá a változónk vagy a függvényünk elé, akkor teljes egészében fenntarthatja őket az osztály belső használatára.

```
class User {
  #message = "Welcome to ES2020"

  login() { console.log(this.#message) }
}

const user = new User()

user.login() // Welcome to ES2020
console.log(user.#message) // Uncaught SyntaxError: Private field '#
```

Megjegyzés: Amint azt a fenti kód mutatja, Ha mégis megpróbálja elérni a változót közvetlenül az objektumból, akkor szintaktikai hibát fog kapni.

ES2020 Or ES11

Az ES2020 az ECMAScript aktuális újabb verziója, amely megfelel a 2020-as évnek. Ez az ECMAScript nyelvi specifikáció tizenegyedik kiadása. Annak ellenére, hogy ez a kiadás nem hoz annyi funkciót, mint az ES6, valóban hasznos funkciókat tartalmazott.

Ezen funkciók többségét néhány böngésző már támogatja, és a nem támogatott funkciók babel elemzővel meg tudjuk oldani. Ezt a kiadást az ECMA 2020. júniusi közgyűlése véglegesen jóváhagyja. Az ECMAScript 2020 (ES2020) nyelvi specifikáció már készen áll.

BigInt

A korábbi JavaScript-verzióban korlátozás van érvényben a Számtípus használatára. vagyis nem képviselhetjük biztonságosan az egész (Érték primitív) értékeket, amelyek nagyobbak, mint a pow (2, 53). ES2020-ban a BigInt a 7. primitív típusként jelenik meg, amely a pow (2, 53) - 1 (vagy 9007199254740991 vagy a Number.MAX_SAFE_INTEGER) méretnél nagyobb egész számokat (tetszőleges pontosságú egész számokat) ábrázolja. Ez úgy jött létre, hogy n-t egész szám literál végéhez fűzünk, vagy a BigInt () függvényt hívjuk meg.

```
// 1. Current number system
const max = Number.MAX_SAFE_INTEGER;
console.log(max + 1) // 9007199254740992
console.log(max + 2) // 9007199254740992

// 2. BigInt representation
const bigInt = 9007199254740991n;
const bigIntConstructorRep = BigInt(9007199254740991); // 9007199254740991n
const bigIntStringRep = BigInt("9007199254740991"); // 9007199254740991n

// 3. Typeof usage

console.log(typeof 1)// number
console.log(typeof 1n)// bigint
console.log(typeof BigInt('1'))// bigint

// 4. Operators

const previousMaxNum = BigInt(Number.MAX_SAFE_INTEGER);
console.log(previousMaxNum + 2n); //9007199254740993n (this was not possible before)
console.log(previousMaxNum -2n); //9007199254740990n
console.log(previousMaxNum * 2n); //18014398509481982n
console.log(previousMaxNum % 2n); //1n
console.log(previousMaxNum / 2n); // 4503599627370495n

// 5. comparison
console.log(1n === 1); // false
console.log(1n === BigInt(1)); // true
console.log(1n == 1); // true
```

Dinamikus importálás

A statikus import néhány olyan fontos felhasználási esetet támogat, mint a statikus elemzés, a bundling tools. Kívánatos továbbá, hogy futás közben dinamikusan be lehessen tölteni egy JavaScript-alkalmazás egy részét.

Az új szolgáltatás dinamikus importálásával bevezetik a modult feltételesen vagy igény szerint. Mivel promise-sal tér vissza modul névtérobjektumára, a modul resolved lehet, vagy az import már hozzárendelhető egy változóhoz az async / await használatával, az alábbiak szerint:

```
<script>
const moduleSpecifier = './message.js';
import(moduleSpecifier)
    .then((module) => {
        module.default(); // Hello, default export
        module.sayGoodBye(); //Bye, named export
    })
    .catch( err => console.log('loading error'));
</script>
<script>
(async function() {
    const moduleSpecifier = './message.js';
    const messageModule = await import(moduleSpecifier);
    messageModule.default(); // Hello, default export
    messageModule.sayGoodBye(); //Bye, named export
})();
</script>
```

és az importált modul megjelenik alapértelmezett és megnevezett exportálással is

```
export default () => {
   return "Hello, default export";
}
export const sayGoodBye = () => {
   return "Bye, named export"
```

```
}
```

Megjegyzés: A dinamikus importáláshoz nincs szükség a type = "module" script-re.

Nullish Coalescing operátor

A nullish coalescing operator (??) egy logikai operátor, amely a jobb oldali operandusát adja vissza, amikor a bal oldali operandus null vagy undefined, és egyébként a bal oldali operandusát adja vissza. Ez az operátor a || adja meg az alapértelmezett értékeket, ha az üres értéket vagy a '', 0 és NaN értéket érvényes értékként kezeli. Ennek oka, hogy a logikai OR (||) operátor (üres értéket vagy '', 0 és NaN) hamis értékként kezeli, és a megfelelő operandusértéket adja vissza, amely ebben az esetben helytelen. Ennélfogva ez az operátor valóban hamis értékek helyett nullás értékeket keres.

```
let vehicle = {
    car: {
        name: "",
        speed: 0
    }
};

console.log(vehicle.car.name || "Unknown"); // Unknown
console.log(vehicle.car.speed || 90); // 90

console.log(vehicle.car.name ?? "Unknown"); // ""(empty is valid case for name)
console.log(vehicle.car.speed ?? 90); // 0(zero is valid case for speed)
```

Rövid megjegyzésben a nulliss operátor nem null értéket ad vissza, és || operátor igaz értékeket ad vissza.

String matchAll

Van String # egyezési módszer, amellyel a karakterlánc összes találatát megkapja egy reguláris kifejezéssel, ez egyes match iterálásával. Ez a módszer azonban megadja az egyező alszövegeket.

A String # matchAll () egy új metódus, amelyet a String prototípushoz adtak, amely az összes olyan eredmény iterátorát adja vissza, amely megfelel egy karakterláncnak egy reguláris kifejezéssel szemben.

```
const regex = /t(e)(st(\d?))/g;
const string = 'test1test2';
const matchesIterator = string.matchAll(regex);
Array.from(matchesIterator, result => console.log(result));
```

Amikor ezt a kódot a böngésző konzolban adja meg, a matches iterátora tömböt állít elő amiben minden illeszkedő elem benne van és néhány extra tulajdonság is.

```
["test1", "e", "st1", "1", index: 0, input: "test1test2", groups: undefined]
["test2", "e", "st2", "2", index: 5, input: "test1test2", groups: undefined]
```

Opcionális láncolás

A JavaScriptben a property hozzáférések hosszú láncolatai meglehetősen hibára hajlamosak, ha bármelyikük null-ra vagy undefined értékre értékel. Ezenkívül nem jó ötlet minden elemnél ellenőrizni a property-k létezését, ami viszont mélyen beágyazott strukturált if utasításokhoz vezet.

Az opcionális láncolás egy olyan új szolgáltatás, amely a JavaScript-kódot tisztábbá és robusztusabbá teheti azáltal, hogy hozzáfűzi (?) Operátort, hogy leállítsa az értékelést, és undefined-altérjen vissza, ha az elem undefined vagy null. Egyébként ez az operátor felhasználható a nullable coalescing operátorral együtt az alapértelmezett értékek megadásához:

```
let vehicle = {
};

let vehicle1 = {
    car: {
        name: 'ABC',
        speed: 90
    }
};

console.log(vehicle.car?.name); // TypeError: Cannot read property 'name' of undefined

console.log(vehicle.car?.name); // Undefined
console.log(vehicle.car?.speed); // Undefined

console.log(vehicle1.car?.name); // ABC
console.log(vehicle1.car?.speed); // 90

console.log(vehicle.car?.name ?? "Unknown"); // Unknown
console.log(vehicle.car?.speed ?? 90); // 90
```

Promise.allSettled

Nagyon hasznos az egyes promise-okról naplózni (különösen a hibakereséshez), ha több promise-t kezel. A Promise.allSettled () metódus egy új promise-t ad vissza, amely resolved, miután az összes promise fulfilled vagy rejected, és az objektumok tömbjével leírja az egyes promise-ok eredményét.

```
const promise1 = new Promise((resolve, reject) => setTimeout(() => resolve(100), 1000));

const promise2 = new Promise((resolve, reject) => setTimeout(reject, 1000));

Promise.allSettled([promise1, promise2]).then(data => console.log(data)); // [
                                                                                Object { status:
"fulfilled", value: 100},
                                                                                Object { status:
"rejected", reason: undefined}
                                                                                ]
```

globalThis

Az ES2020 előtt különböző szintaxist kell írni különböző JavaScript környezetekben (cross-platformok), csak a globális objektum eléréséhez. Ez valóban nehéz a fejlesztők számára, mert a böngésző oldalán az window, self, or frames kell használni, a nodejs-ben pedig a self-et.

Másrészt this kulcsszó használható a függvények belsejében a non-stric mode-ban, de strict módban undefined-ot ad. Ha a Function ('return this') () megoldásra gondol a fenti környezetekre, akkor a CSP-t támogató környezeteknél (ahol az eval () le van tiltva).

A régebbi verziókban az es6-shim-et használhatja az alábbiak szerint:

```
var getGlobal = function () {
  if (typeof self !== 'undefined') { return self; }
  if (typeof window !== 'undefined') { return window; }
  if (typeof global !== 'undefined') { return global; }
  throw new Error('unable to locate global object');
};

var globals = getGlobal();

if (typeof globals.setTimeout !== 'function') {
  console.log('no setTimeout in this environment or runtime');
}
```

Az ES2020-ban a globalThis tulajdonság azzal a céllal került bevezetésre, hogy szabványos módon biztosítsa az érték globális elérését környezetekben.

```
if (typeof globalThis.setTimeout !== 'function') {
  console.log('no setTimeout in this environment or runtime');
}
```

import.meta

Az import.meta objektumot az ECMAScript implementáció hozta létre null prototípussal, hogy kontextus-specifikus metaadatokat kapjon egy JavaScript-modulról. Tegyük fel, hogy megpróbálja betölteni a modulomat egy szkriptből,

```
<script type="module" src="my-module.js"></script>
```

Most az import.meta objektummal férhet hozzá a modul metainformációihoz (a modul alap URL-jéhez)

```
console.log(import.meta); // { url: "file:///home/user/my-module.js" }
```

A fenti URL lehet vagy URL, ahonnan a szkriptet beszerezték (külső szkriptekhez), vagy a tartalmazó dokumentum URL-je (inline szkriptekhez).

Megjegyzés: Ne feledje, hogy az import valójában nem objektum, hanem az import.meta objektum, amely kiterjeszthető, és tulajdonságai írhatók, konfigurálhatók és felsorolhatók.

for..in order

Az ES2020 előtt a specifikációk nem határozták meg, hogy az (a in b) sorrendnek melyik sorrendben kell futnia. Annak ellenére, hogy a javascript motorok / böngészők többsége az objektum tulajdonságait a definiálásuk sorrendjében iterálja, ez nem minden esetben fordul elő. Ezt hivatalosan szabványosították az ES2020-ban.

```
var object = {
  'a': 2,
  'b': 3,
  'c': 4
```

```
}


for(let key in object) {
  console.log(key); // a b c
}
```