JavaScript Nyílfüggvények

JavaScript Függvénykifejezések, Nyílfüggvények, Callback Függvények és Tömb Metódusok

Függvény Deklaráció (Function Declaration)

A függvény deklaráció egy olyan módszer, amely egy nevesített függvényt hoz létre. A függvény deklarációi a kód végrehajtása előtt kerülnek létrehozásra, ami azt jelenti, hogy a kódban bárhol meghívhatók, még azelőtt is, hogy definiálva lennének.

Szintaxis:

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

greet(); // Output: Hello!
```

Jellemzők:

- **Hoisting**: A függvény deklarációk "felemelődnek" a kód tetejére. Ez azt jelenti, hogy a függvény deklarációkat bárhol meghívhatod a kódban, még azelőtt is, hogy azok definiálva lennének.
- **Nevesített függvények**: A függvény deklarációk mindig nevesítettek (a függvénynek van neve).

Példa:

```
console.log(add(2, 3)); // Output: 5

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

Függvény Kifejezés (Function Expression)

A függvény kifejezés egy olyan módszer, amely egy függvényt hoz létre egy kifejezés részeként. A függvény kifejezések lehetnek névtelenek vagy nevesítettek, és gyakran változókhoz rendeljük őket. A függvény kifejezések nem kerülnek hoisting alá, tehát csak azután lehet őket meghívni, hogy definiálva lettek.

Szintaxis:

```
const greet = function() {
    console.log("Hello!");
};
```

```
greet(); // Output: Hello!
```

Jellemzők:

- **Nincs hoisting**: A függvény kifejezések nem kerülnek hoisting alá, ami azt jelenti, hogy csak azután hívhatók meg, hogy azok definiálva lettek.
- **Nevesített vagy névtelen függvények**: A függvény kifejezések lehetnek névtelenek (a függvénynek nincs neve) vagy nevesítettek.

Példa:

```
console.log(add(2, 3)); // Error: add is not defined

const add = function(a, b) {
    return a + b;
};

console.log(add(2, 3)); // Output: 5
```

Összegzés

Hoisting:

- **Függvény Deklaráció**: A függvény deklarációk hoisting alá kerülnek, ami azt jelenti, hogy a függvényeket a deklarációik előtt is meg lehet hívni.

  ```
  greet(); // Output: Hello!

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

- **Függvény Kifejezés**: A függvény kifejezések nem kerülnek hoisting alá, így csak a deklaráció után lehet őket meghívni.

  ```
  greet(); // Error: greet is not defined

  const greet = function() {
      console.log("Hello!");
  };

  greet(); // Output: Hello!
  ```

Szintaxis:

- **Függvény Deklaráció**:

```
function name(parameters) {
    // function body
}
```

- **Függvény Kifejezés**:

```
const name = function(parameters) {
    // function body
};
```

Nevezetes vagy Névtelen:

- **Függvény Deklaráció**: Mindig nevesített.

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

- **Függvény Kifejezés**: Lehet névtelen vagy nevesített.

```
const greet = function() {
    console.log("Hello!");
};

const namedGreet = function greet() {
    console.log("Hello!");
};
```

Mindkét módszer hasznos és gyakran használt JavaScript-ben. A függvény deklarációkat gyakran használják egyszerűbb szkriptek és kódok esetében, míg a függvény kifejezéseket gyakran használják a callbackek és az inline függvények esetében, valamint amikor a függvényeknek nincs szüksége saját `this` kötésre (nyílfüggvények esetén).

Nyílfüggvények

A nyílfüggvények rövidebb szintaxist biztosítanak a függvények írásához. Nem rendelkeznek saját `this`, `arguments`, `super` vagy `new.target` kötésekkel (ezekről később lesz szó), és gyakran használjuk őket callbackek és inline függvények esetében.

Szintaxis:

```
const greet = () => {
    console.log("Hello!");
};

greet(); // Output: Hello!
```

Egyszerűsített Nyílfüggvények Ha a függvény törzse egyetlen kifejezésből áll, akkor elhagyhatjuk a kapcsos zárójeleket és a `return` kulcsszót is. Ilyenkor a függvényben lévő utasítás lesz egyben a függvény visszatérési értéke is.

Példa:

```
const add = (a, b) => a + b;

console.log(add(2, 3)); // Output: 5
```

Paraméterek Nyílfüggvényekben

- **Nincs paraméter**: Üres zárójelek használata. `javascript const noParam = () => console.log("No parameter"); noParam(); // Output: No parameter`
- **Egy paraméter**: Zárójelek elhagyhatók.

  ```
  const oneParam = param => console.log(param);
  oneParam("Hello"); // Output: Hello
  ```

- **Több paraméter**: Zárójelek használata kötelező. `javascript const multipleParams = (param1, param2) => console.log(param1, param2); multipleParams("Hello", "World"); // Output: Hello World`

Tömb Metódusok

Most nézzük meg részletesen a fontosabb tömb metódusokat, és hogy mi történik egy-egy iteráció során.

forEach A `forEach` metódus végrehajt egy adott függvényt a tömb minden elemén egyszer.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: `undefined`

Példa:

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach((number, index) => {
    console.log(`Index: ${index}, Value: ${number}`);
```

```
});
// Output:
// Index: 0, Value: 1
// Index: 1, Value: 2
// Index: 2, Value: 3
// Index: 3, Value: 4
// Index: 4, Value: 5
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paraméterei a jelenlegi elem és annak indexe.

map A `map` metódus létrehoz egy új tömböt, amely a megadott függvény által visszaadott eredményeket tartalmazza a tömb minden elemén végrehajtva.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: Új tömb, amely a callback függvény által visszaadott értékeket tartalmazza.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
// Output: [2, 4, 6, 8, 10]
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.
- Az eredmény hozzáadódik az új tömbhöz.

filter A `filter` metódus létrehoz egy új tömböt, amely csak azokat az elemeket tartalmazza, amelyek megfelelnek a megadott feltételnek.

Specifikáció:

- **Paraméterek**:

  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    * `currentValue`: Az aktuális elem értéke.
    * `index` (opcionális): Az aktuális elem indexe.
    * `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: Új tömb, amely csak azokat az elemeket tartalmazza, amelyekre a callback függvény `true` értéket adott vissza.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter((number) => number % 2 === 0);
console.log(evenNumbers);
// Output: [2, 4]
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.
- Ha a callback `true` értéket ad vissza, az elem bekerül az új tömbbe.

find A `find` metódus visszaadja az első olyan elemet a tömbből, amely megfelel a megadott feltételnek.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    * `currentValue`: Az aktuális elem értéke.
    * `index` (opcionális): Az aktuális elem indexe.
    * `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: Az első olyan elem, amelyre a callback függvény `true` értéket ad vissza, különben `undefined`.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const found = numbers.find((number) => number > 3);
console.log(found);
// Output: 4
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.

- Amint a callback `true` értéket ad vissza, az aktuális elem lesz a visszatérési érték és a keresés leáll.

findIndex A `findIndex` metódus visszaadja az első olyan elem indexét a tömbben, amely megfelel a megadott feltételnek.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: Az első olyan elem indexe, amelyre a callback függvény `true` értéket ad vissza, különben `-1`.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const index = numbers.findIndex((number) => number > 3);
console.log(index);
// Output: 3
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.
- Amint a callback `true` értéket ad vissza, az aktuális elem indexe lesz a visszatérési érték és a keresés leáll.

some A `some` metódus visszaadja, hogy legalább egy elem megfelel-e a megadott feltételnek.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: `true`, ha legalább egy elem megfelel a feltételnek, különben `false`.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const hasEven = numbers.some((number) => number % 2 === 0);
console.log(hasEven);
// Output: true
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.
- Amint a callback `true` értéket ad vissza, a `some` metódus `true` értéket ad vissza és a keresés leáll.

every Az `every` metódus visszaadja, hogy minden elem megfelel-e a megadott feltételnek.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
- **Visszatérési érték**: `true`, ha minden elem megfelel a feltételnek, különben `false`.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const allPositive = numbers.every((number) => number > 0);
console.log(allPositive);
// Output: true
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paramétere a jelenlegi elem.
- Amint a callback `false` értéket ad vissza, az `every` metódus `false` értéket ad vissza és a keresés leáll.

reduce A `reduce` metódus egyetlen értéket állít elő a tömb elemeinek egyesítéséből, a megadott függvény alkalmazásával.

Specifikáció:

- **Paraméterek**:
  - `callback` (kötelező): A függvény, amelyet minden egyes elemre végrehajt.
    - `accumulator`: Az akkumulált érték, amely az előző callback hívás visszatérési értéke.
    - `currentValue`: Az aktuális elem értéke.
    - `index` (opcionális): Az aktuális elem indexe.
    - `array` (opcionális): Az aktuális tömb.
  - `initialValue` (opcionális): Kezdőérték az akkumulátor számára.
- **Visszatérési érték**: Egyetlen érték, amely az akkumulátor végső értéke.

Példa:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, number) => accumulator + number, 0);
console.log(sum);
// Output: 15
```

Mi történik egy-egy iteráció során?

- Az adott függvényt (callback) meghívjuk minden egyes elemre a tömbben.
- A callback paraméterei: az akkumulátor (összesített érték) és a jelenlegi elem.
- Az akkumulátor kezdőértéke az opcionálisan megadott második paraméter (`0`).
- Minden iteráció során a callback visszatérési értéke lesz az új akkumulátor értéke.
- A `reduce` metódus végül az akkumulátor végső értékét adja vissza.

Spread Operátor Magyarázata

A spread operátor (`...`) egy új szintaktikai jellemző, amelyet az ES6 (ECMAScript 2015) vezetett be. Lehetővé teszi, hogy egy iterálható objektum (pl. tömb, string) elemeit különálló elemekként terjesszük ki. A spread operátor számos helyzetben hasznos, például tömbök és objektumok másolásakor, egyesítéskor és függvényhívásoknál.

Tömbök Másolása és Egyesítése

A spread operátorral könnyen másolhatunk vagy egyesíthetünk tömböket.

Tömb Másolása A spread operátor segítségével egy tömb másolása egyszerű és olvasható:

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];

console.log(copiedArray);
// Output: [1, 2, 3]
```

Tömb Egyesítése Két vagy több tömb egyesítése a spread operátor használatával:

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [...array1, ...array2];

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

Függvényhívásoknál Történő Használat

A spread operátort használhatjuk egy tömb elemeinek különálló argumentumként való átadására egy függvényhívás során.

Példa:

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

const numbers = [1, 2, 3];
const result = add(...numbers);

console.log(result);
// Output: 6
```

Stringek Terjesztése

A spread operátor stringeken is működik, és különálló karakterekké terjeszti ki a stringet.

Példa:

```
const string = "hello";
const characters = [...string];

console.log(characters);
// Output: ['h', 'e', 'l', 'l', 'o']
```

Objektumok Másolása és Egyesítése

Az ES2018 (ES9) bevezetésével a spread operátor már objektumokkal is használható, lehetővé téve az objektumok másolását és egyesítését.

Objektum Másolása Az objektumok másolása a spread operátor segítségével:

```
const originalObject = { a: 1, b: 2 };
const copiedObject = { ...originalObject };

console.log(copiedObject);
// Output: { a: 1, b: 2 }
```

Objektum Egyesítése Két vagy több objektum egyesítése a spread operátor használatával:

```
const object1 = { a: 1, b: 2 };
const object2 = { c: 3, d: 4 };
const mergedObject = { ...object1, ...object2 };

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

További Példák és Alkalmazások

Új Elemmel Bővítés A spread operátor használatával könnyen hozzáadhatunk új elemeket egy tömb elejére vagy végére:

```
const numbers = [2, 3, 4];
const newNumbers = [1, ...numbers, 5];

console.log(newNumbers);
// Output: [1, 2, 3, 4, 5]
```

Tömb Elválasztása Első és Maradék Elemeire A spread operátor használható a destruktúrálás során is, hogy az első elemet különválasszuk a maradéktól:

```
const numbers = [1, 2, 3, 4, 5];
const [first, ...rest] = numbers;

console.log(first); // Output: 1
console.log(rest); // Output: [2, 3, 4, 5]
```

JavaScript `sort` Függvény

A JavaScript `sort` függvénye a tömbök elemeit rendezi. Alapértelmezés szerint a `sort` függvény a tömb elemeit stringként hasonlítja össze, és az Unicode kódszámaik alapján rendezi azokat. Ha egyedi rendezési feltételt szeretnél megadni, használhatsz egy összehasonlító (compare) függvényt.

Szintaxis:

```
array.sort([compareFunction]);
```

- **compareFunction** (opcionális): Egy függvény, amely meghatározza az elemek rendezési sorrendjét. Ha nincs megadva, a `sort` függvény az elemeket stringként hasonlítja össze.

Visszatérési érték: A `sort` függvény visszatérési értéke a rendezett tömb. A rendezés az eredeti tömböt módosítja.

Alapértelmezett Rendezés

Alapértelmezés szerint a `sort` függvény az elemeket stringként rendezi.

Példa:

```
const fruits = ["banana", "apple", "cherry"];
fruits.sort();
console.log(fruits);
// Output: ["apple", "banana", "cherry"]
```

Számok Rendezése Alapértelmezett Rendezéssel: Mivel a `sort` függvény alapértelmezés szerint stringként rendezi az elemeket, a számok rendezése nem mindig adja a várt eredményt.

Példa:

```
const numbers = [10, 5, 1, 20, 25];
numbers.sort();
```

```
console.log(numbers);
// Output: [1, 10, 20, 25, 5]
```

Egyedi Rendezési Feltétel

Egyedi rendezési feltételt adhatunk meg egy összehasonlító (compare) függvény használatával. Az összehasonlító függvény két paramétert kap, és az alábbiak szerint kell visszaadnia egy értéket:

- Ha a `compareFunction(a, b)` visszatérési értéke kisebb, mint 0, akkor `a` az `b` előtt lesz a rendezett tömbben.
- Ha a visszatérési érték 0, akkor `a` és `b` változatlan sorrendben maradnak.
- Ha a visszatérési érték nagyobb, mint 0, akkor `b` az `a` előtt lesz a rendezett tömbben.

Példa: Számok Rendezése Növekvő Sorrendben

```
const numbers = [10, 5, 1, 20, 25];
numbers.sort((a, b) => a - b);
console.log(numbers);
// Output: [1, 5, 10, 20, 25]
```

Példa: Számok Rendezése Csökkenő Sorrendben

```
const numbers = [10, 5, 1, 20, 25];
numbers.sort((a, b) => b - a);
console.log(numbers);
// Output: [25, 20, 10, 5, 1]
```

Objektumok Rendezése

A `sort` függvényt objektumok tömbjének rendezésére is használhatjuk, egyedi feltétel szerint.

Példa: Objektumok Rendezése Egy Tulajdonság Alapján

```
const items = [
    { name: "apple", price: 50 },
    { name: "banana", price: 30 },
    { name: "cherry", price: 60 },
];

items.sort((a, b) => a.price - b.price);
console.log(items);
// Output:
// [
//   { name: "banana", price: 30 },
//   { name: "apple", price: 50 },
```

```
//   { name: "cherry", price: 60 }
// ]
```

Speciális Rendezési Szempontok

Case-Insensitive Rendezés A stringek rendezésénél figyelmen kívül hagyhatjuk a kis- és nagybetűk közötti különbségeket, ha mindkét stringet kis- vagy nagybetűssé alakítjuk az összehasonlítás előtt.

Példa:

```
const fruits = ["Banana", "apple", "Cherry"];
fruits.sort((a, b) => a.toLowerCase().localeCompare(b.toLowerCase()));
console.log(fruits);
// Output: ["apple", "Banana", "Cherry"]
```

A for..in és for..of ciklusok magyarázata

for..in Ciklus

A `for..in` ciklus iterál egy objektum tulajdonságainak kulcsain. Leggyakrabban objektumok iterálására használják, de használható tömbökön is, bár nem ajánlott, mivel a tömbök esetén az `Array` prototípushoz hozzáadott nem számozott tulajdonságok is bejárásra kerülnek.

Szintaxis:

```
for (variable in object) {
    // code block to be executed
}
```

- **variable**: A változó, amely az iteráció során az objektum tulajdonságainak kulcsait veszi fel.
- **object**: Az objektum, amelynek tulajdonságain iterálni szeretnénk.

Példa:

```
const person = { firstName: "John", lastName: "Doe", age: 25 };

for (let key in person) {
    console.log(key + ": " + person[key]);
}
// Output:
// firstName: John
// lastName: Doe
// age: 25
```

Tömb iterálása for..in ciklussal: Noha használható tömbök iterálására, nem ajánlott, mivel nem garantálja az elemek sorrendjét és a prototípus lánc bővítései is megjelenhetnek.

```
const array = [1, 2, 3, 4];

for (let index in array) {
    console.log(index + ": " + array[index]);
}
// Output:
// 0: 1
// 1: 2
// 2: 3
// 3: 4
```

for..of Ciklus

A `for..of` ciklus bevezetésre került az ES6-ban (ECMAScript 2015), és iterál iterálható objektumokon (pl. tömbök, stringek, Map-ek, Set-ek stb.). Ez a ciklus kizárólag az iterálható objektumok elemein megy végig, és nem iterál objektumok tulajdonságain.

Szintaxis:

```
for (variable of iterable) {
    // code block to be executed
}
```

- **variable**: A változó, amely az iteráció során az iterálható objektum elemeit veszi fel.
- **iterable**: Az iterálható objektum, amelyen iterálni szeretnénk.

Példa: Tömb Iterálása

```
const array = [1, 2, 3, 4];

for (let value of array) {
    console.log(value);
}
// Output:
// 1
// 2
// 3
// 4
```

Példa: String Iterálása

```javascript
const string = "hello";

for (let char of string) {
    console.log(char);
}
// Output:
// h
// e
// l
// l
// o
```

Összegzés

- **for..in**: Objektumok tulajdonságainak kulcsain iterál. Használható tömbökön is, de nem ajánlott a fent említett okok miatt.
  - Leginkább objektumok kulcsainak bejárására használjuk.
  - Nem garantálja az iterálás sorrendjét.
- **for..of**: Iterálható objektumok (pl. tömbök, stringek, Map-ek, Set-ek stb.) elemein iterál.
  - Kizárólag iterálható objektumok elemein megy végig.
  - Garantálja az iterálás sorrendjét.

Mindkét ciklus hasznos különböző helyzetekben, és érdemes tudni, mikor melyiket használjuk a megfelelő hatékonyság és kód tisztaság érdekében.

Rest Paraméter Magyarázata

A rest paraméter (`...`) egy új szintaktikai jellemző, amelyet az ES6 (ECMAScript 2015) vezetett be. Lehetővé teszi, hogy egy függvényben a változó számú argumentumokat tömbként kezeljük. A rest paraméter segítségével könnyen kezelhetjük azokat az eseteket, amikor nem tudjuk előre, hány argumentumot kap a függvény.

Szintaxis:

A rest paramétert mindig a függvény paramétereinek végén használjuk, és három ponttal (`...`) jelöljük.

```
function myFunction(a, b, ...rest) {
    // code block
}
```

- **a, b**: Normál paraméterek.
- **...rest**: A rest paraméter, amely egy tömböt tartalmaz, ami az összes maradék argumentumot tartalmazza.

Példa: Rest Paraméter Használata

Összes Argumentum Összege

A következő függvény tetszőleges számú számot fogad el argumentumként, és visszaadja azok összegét.

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3));       // Output: 6
console.log(sum(1, 2, 3, 4, 5)); // Output: 15
console.log(sum(10, 20));        // Output: 30
```

Több Fix Paraméter és Rest Paraméter

A következő példa egy függvényt mutat be, amely két fix paramétert (`a` és `b`) fogad el, valamint egy rest paramétert (`rest`), amely a maradék argumentumokat tartalmazza.

```
function multiplyAndSum(a, b, ...rest) {
    const product = a * b;
    const sumOfRest = rest.reduce((total, num) => total + num, 0);
    return product + sumOfRest;
}

console.log(multiplyAndSum(2, 3, 4, 5));   // Output: 11 (2*3 + 4 + 5)
console.log(multiplyAndSum(1, 2, 3, 4, 5)); // Output: 15 (1*2 + 3 + 4 + 5)
console.log(multiplyAndSum(5, 10));         // Output: 50 (5*10 + 0)
```

Rest Paraméter vs. Arguments Objektum

A rest paraméter modern alternatívája az `arguments` objektumnak. Bár az `arguments` objektum hasonló funkcionalitást biztosít, több hátránya is van:

- Az `arguments` nem egy valódi tömb, hanem egy tömbszerű objektum, így nincs hozzáférés a tömb metódusokhoz (pl. `map`, `filter`, `reduce`).
- Az `arguments` objektum nem működik nyílfüggvényekben (`arrow functions`).

Példa: Rest Paraméter és Arguments Összehasonlítása

```
function sumWithArguments() {
    return Array.from(arguments).reduce((total, num) => total + num, 0);
}

function sumWithRest(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sumWithArguments(1, 2, 3)); // Output: 6
console.log(sumWithRest(1, 2, 3));      // Output: 6
```

JavaScript Closure Magyarázata

Mi az a Closure?

A closure (lezárás) egy olyan funkcionális programozási fogalom, amely lehetővé teszi, hogy egy függvény "emlékezzen" az őt körülvevő környezet változóira még azután is, hogy a külső függvény végrehajtása befejeződött. Másképpen fogalmazva, a closure egy függvény, amely hozzáfér az őt körülvevő scope-hoz (környezethez), még akkor is, ha a külső scope már nem létezik.

Hogyan Működik a Closure?

Amikor egy függvény létrejön egy másik függvény belsejében, a belső függvény "lezárja" a külső függvény környezetét, megőrizve annak változóit és paramétereit. Ez lehetővé teszi, hogy a belső függvény később is hozzáférjen ezekhez a változókhoz.

Példa:

```
function outerFunction() {
    let outerVariable = 'I am outside!';

    function innerFunction() {
        console.log(outerVariable);
    }

    return innerFunction;
}

const closureFunction = outerFunction();
closureFunction(); // Output: 'I am outside!'
```

Részletes Magyarázat

Létrehozás

- A `outerFunction` létrehoz egy `outerVariable` nevű változót, és egy `innerFunction` nevű függvényt definiál.
- A `innerFunction` hozzáfér a `outerVariable` változóhoz.
- A `outerFunction` visszaadja a `innerFunction` referenciáját.

Végrehajtás

- Amikor a `closureFunction`-t meghívjuk, amely a `outerFunction` által visszaadott `innerFunction`, akkor a `innerFunction` kiírja a `outerVariable` értékét, mivel az továbbra is "emlékszik" rá.

További Példák

Számláló Függvény

Egy gyakori példa a closure-re egy számláló függvény létrehozása:

```
function createCounter() {
    let count = 0;

    return function() {
        count += 1;
        return count;
    };
}

const counter = createCounter();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
console.log(counter()); // Output: 3
```

Ebben a példában a `createCounter` függvény létrehoz egy `count` változót és visszaad egy névtelen függvényt. Ez a névtelen függvény képes hozzáférni és módosítani a `count` változót a `createCounter` scope-ján kívül is.

Privát Adatok Elrejtése

A closure-k használhatók arra is, hogy privát adatokat rejtsenek el a külső hozzáférés elől:

```
function createPerson(name) {
    let privateName = name;

    return {
        getName: function() {
            return privateName;
        },
        setName: function(newName) {
            privateName = newName;
        }
    };
}

const person = createPerson('John');
console.log(person.getName()); // Output: John
person.setName('Jane');
console.log(person.getName()); // Output: Jane
```

Ebben a példában a `createPerson` függvény visszaad egy objektumot, amely két metódust tartalmaz: `getName` és `setName`. Ezek a metódusok hozzáférhetnek és módosíthatják a `privateName` változót, de kívülről közvetlenül nem lehet hozzáférni ehhez a változóhoz. (Metódusokról hamarosan tanulunk)

Closure Használata a Gyakorlati JavaScript Programozásban

Eseménykezelők

Closure-k gyakran használatosak eseménykezelőkben, hogy megőrizzék az állapotot egy esemény bekövetkezésekor (ezekről hamarosan tanulunk).

```
function addClickHandler(buttonId) {
    let count = 0;

    document.getElementById(buttonId).addEventListener('click', function() {
        count += 1;
        console.log(`Button ${buttonId} clicked ${count} times`);
    });
}

addClickHandler('myButton');
```

Modulként Használva

A closure-k segítségével modulokat hozhatunk létre, amelyek lehetővé teszik a privát változók és függvények használatát.

```
const counterModule = (function() {
    let count = 0;

    return {
        increment: function() {
            count += 1;
            return count;
        },
        reset: function() {
            count = 0;
        }
    };
})();

console.log(counterModule.increment()); // Output: 1
console.log(counterModule.increment()); // Output: 2
counterModule.reset();
console.log(counterModule.increment()); // Output: 1
```

JavaScript Hibakezelés Magyarázata

A hibakezelés fontos része minden programozási nyelvnek, beleértve a JavaScriptet is. Lehetővé teszi, hogy a programozók kezeljék a váratlan helyzeteket, például a felhasználói hibákat, a hálózati hibákat vagy a programozási hibákat. A JavaScript hibakezelési mechanizmusa a `try...catch` blokkot használja.

Hibakezelési Mechanizmus

try...catch Blokk

A `try...catch` blokk két részből áll:

- **try blokk**: Itt helyezzük el azt a kódot, amely esetleg hibát okozhat.
- **catch blokk**: Ez a blokk fut le, ha a `try` blokkban hiba történik. Itt kezeljük a hibát.

Szintaxis:

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
}
```

Példa: Egyszerű Hibakezelés

```
try {
    let result = riskyOperation();
    console.log(result);
} catch (error) {
    console.error("An error occurred:", error.message);
}
```

Kiegészítő `finally` Blokk

A `finally` blokk egy opcionális harmadik rész a hibakezelési struktúrában, amely akkor is végrehajtódik, ha hiba történt, és akkor is, ha nem. A finally akkor is lefut, ha a try blockban return történt.

Szintaxis:

```
try {
    // Code that may throw an error
} catch (error) {
    // Code to handle the error
} finally {
    // Code that will always run, regardless of an error
}
```

Példa: try...catch...finally

```
try {
    let result = riskyOperation();
    console.log(result);
} catch (error) {
    console.error("An error occurred:", error.message);
} finally {
    console.log("This will always run, regardless of an error.");
}
```

Hibák Dobása (Throwing Errors)

A `throw` kulcsszó használatával saját hibákat is dobhatunk. Ez különösen hasznos, ha egy függvényben egyedi hibákat szeretnénk kezelni.

Szintaxis:

```
throw new Error("Something went wrong");
```

Példa: Hibák Dobása

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed");
    }
    return a / b;
}

try {
    let result = divide(10, 0);
    console.log(result);
} catch (error) {
    console.error("An error occurred:", error.message);
}
```

Egyedi Hibatípusok

A JavaScript rendelkezik néhány beépített hibatípussal, mint például `Error`, `TypeError`, `RangeError`, `SyntaxError`, stb. Ezeket a hibatípusokat használhatjuk egyedi hibák dobására.

Példa: Egyedi Hibatípusok Használata

```
function checkArray(arr) {
    if (!Array.isArray(arr)) {
        throw new TypeError("Expected an array");
    }
    if (arr.length === 0) {
        throw new RangeError("Array cannot be empty");
    }
    return arr.length;
}

try {
    let result = checkArray(123);
    console.log(result);
} catch (error) {
    console.error(`${error.name}: ${error.message}`);
}
```

Böngésző mint Host Environment

Böngésző mint Host Environment

A JavaScript nem csak a böngészőben futtatható, de a böngésző az egyik leggyakoribb környezet, ahol használják. A böngésző, mint "host environment" (gazda környezet), a JavaScript motor számára biztosít egy környezetet, amelyben a kód futtatható. Ez a környezet számos beépített objektumot és függvényt tartalmaz, amelyek segítségével a JavaScript hozzáférhet a weboldal elemeihez és a böngésző funkcióihoz.

BOM (Browser Object Model)

Mi az a BOM?

A BOM (Browser Object Model) egy gyűjteménye azoknak az objektumoknak és függvényeknek, amelyeket a böngésző biztosít a JavaScript számára a böngészőablak és a weblap közötti interakció kezeléséhez. A BOM nem része a szabványosított JavaScript-nek, de a legtöbb böngésző támogatja.

A BOM objektumok lehetővé teszik például:

- Az ablak tulajdonságainak és méretének lekérdezését és módosítását.
- Az aktuális URL lekérdezését és módosítását.
- A böngésző történetének kezelését.
- Információk lekérdezését a böngészőről és a felhasználó képernyőjéről.

Példa a BOM használatára

```
// Az ablak szélességének és magasságának lekérése
let width = window.innerWidth;
let height = window.innerHeight;

console.log(`Szélesség: ${width}, Magasság: ${height}`);
```

Window Object

Mi az a Window Object?

A `window` objektum a BOM központi objektuma, amely a böngészőablakot reprezentálja. Minden, a BOM-ban lévő objektum, függvény és változó a `window` objektum része. Ezért a `window` objektum nélkülözhetetlen része a böngésző környezetének, és a JavaScript számára egy globális kontextust biztosít.

A `window` Objektum Tulajdonságai és Metódusai

- **`window.document`**: A jelenlegi HTML dokumentumot reprezentálja, és hozzáférést biztosít a DOM-hoz.
- **`window.location`**: Az aktuális URL-t kezeli, és lehetővé teszi annak módosítását.
- **`window.navigator`**: Információkat ad a böngészőről és az operációs rendszerről.
- **`window.history`**: A böngésző történetét kezeli, lehetővé téve az előre és hátra navigációt.
- **`window.screen`**: Információkat ad a felhasználó képernyőjéről.
- **`window.innerWidth` és `window.innerHeight`**: Az ablak belső szélessége és magassága.
- **`alert()`, `confirm()`, `prompt()`**: Felugró ablakokat jelenít meg.

Példa a `window` Objektum Használatára

```
// URL módosítása
window.location.href = "https://www.example.com";

// Információk lekérése a böngészőről
console.log(window.navigator.userAgent);

// Felugró ablak megjelenítése
window.alert("Hello, World!");

// Ablak méreteinek lekérése
let width = window.innerWidth;
let height = window.innerHeight;

console.log(`Szélesség: ${width}, Magasság: ${height}`);
```

A DOM (Document Object Model)

Mi az a DOM?

A DOM (Document Object Model) egy programozási interfész a HTML és XML dokumentumokhoz. Lehetővé teszi, hogy a dokumentumok szerkezete fára épülő módon legyen reprezentálva, ahol minden csomópont egy dokumentum része, mint például elemek, attribútumok, szövegek, stb.

A DOM segítségével a JavaScript:

- Hozzáférhet a HTML dokumentum struktúrájához és tartalmához.
- Módosíthatja a HTML dokumentum szerkezetét, stílusait és tartalmát.
- Eseménykezelőket adhat hozzá az elemekhez, például kattintások, billentyűleütések kezelése.

DOM Fa Struktúra

A DOM fa egy hierarchikus szerkezet, amely csomópontokat (node) tartalmaz. Minden csomópont egy elemet, attribútumot, szöveget vagy más dokumentumot reprezentál. A csomópontok típusai a következők:

- **Elem csomópontok**: HTML elemeket reprezentálnak (pl. `<div>`, `<p>`, `<a>`).
- **Attribútum csomópontok**: Az elemek attribútumait reprezentálják (pl. `id`, `class`).
- **Szöveg csomópontok**: Az elemek belső szövegét reprezentálják.
- **Komment csomópontok**: HTML kommenteket reprezentálnak (pl. `<!-- Comment -->`).
- **Dokumentum csomópontok**: A teljes dokumentumot reprezentálják.

DOM Fa Példa

Vegyünk egy egyszerű HTML dokumentumot, és nézzük meg, hogyan néz ki a DOM fa struktúrája.

HTML Dokumentum

```
<!DOCTYPE html>
<html>
<head>
    <title>Example</title>
</head>
<body>
    <h1 id="main-title">Hello, World!</h1>
    <p>This is a paragraph.</p>
</body>
</html>
```

DOM Fa Struktúra

```
Document
 └── html
      ├── head
      │    └── title
      │         └── "Example"
      └── body
           ├── h1 (id="main-title")
           │    └── "Hello, World!"
           └── p
                └── "This is a paragraph."
```

DOM Csomópontok

Dokumentum Csomópont (Document Node)

A legfelső szintű csomópont, amely az egész HTML vagy XML dokumentumot reprezentálja.

Elem Csomópont (Element Node)

Minden HTML elem egy elem csomópont. Például a `<body>`, `<h1>`, és `<p>` elemek mind elem csomópontok.

Attribútum Csomópont (Attribute Node)

Az elem csomópontok attribútumai. Például az `id="main-title"` attribútum egy attribútum csomópont.

Szöveg Csomópont (Text Node)

Az elem csomópontok szöveges tartalmát reprezentálják. Például a `<h1>` elemben a "Hello, World!" egy szöveg csomópont.

DOM Fa Navigáció

A DOM fa csomópontjain különböző módokon navigálhatunk.

Navigációs Tulajdonságok

- **`parentNode`**: A szülő csomópont.
- **`parentElement`**: A szülő elem.
- **`childNodes`**: Az összes gyermek csomópont collection-je.
- **`children`**: Az összes gyermek elem collection-je.
- **`firstChild`**: Az első gyermek csomópont.
- **`firstElementChild`**: Az első gyermek elem.
- **`lastChild`**: Az utolsó gyermek csomópont.
- **`lastElementChild`**: Az utolsó gyermek elem.
- **`previousSibling`**: Az előző testvér csomópont.
- **`previousElementSibling`**: Az előző testvér elem.
- **`nextSibling`**: A következő testvér csomópont.
- **`nextElementSibling`**: A következő testvér elem.

Példa: DOM Navigáció

```
// Hozzáférés a <body> szülő eleméhez (ami a <html>)
let bodyParent = document.body.parentNode;
console.log(bodyParent.nodeName); // Output: "HTML"

// Hozzáférés a <body> első gyermekéhez
let firstChild = document.body.firstChild;
console.log(firstChild.nodeName); // Output: (valószínűleg text node, ha van whitespace)
```

DOM Kiválasztó és Navigációs Metódusok Összefoglaló

getElementById

A `getElementById` metódus egy olyan elemet keres a DOM-ban, amelynek az `id` attribútuma megegyezik a megadott sztringgel.

Szintaxis:

```
let element = document.getElementById("myId");
```

Jellemzők:

- Csak egy elemet ad vissza, mivel az `id` attribútumnak egyedinek kell lennie az oldalon.
- Ha nincs ilyen `id` attribútumú elem, `null` értéket ad vissza.

querySelectorAll

A `querySelectorAll` metódus egy NodeList-et ad vissza azokról az elemekről, amelyek megfelelnek a megadott CSS szelektornak.

Szintaxis:

```
let elements = document.querySelectorAll(".myClass");
```

Jellemzők:

- NodeList-et ad vissza, amely statikus gyűjtemény, tehát nem frissül automatikusan, ha a DOM változik.
- Tetszőleges CSS szelektort használhatunk.

querySelector

A `querySelector` metódus az első olyan elemet adja vissza, amely megfelel a megadott CSS szelektornak.

Szintaxis:

```
let element = document.querySelector(".myClass");
```

Jellemzők:

- Csak az első megfelelő elemet adja vissza.
- Tetszőleges CSS szelektort használhatunk.

matches

A `matches` metódus ellenőrzi, hogy egy adott elem megfelel-e a megadott CSS szelektornak.

Szintaxis:

```
let isMatch = element.matches(".myClass");
```

Jellemzők:

- `true` értéket ad vissza, ha az elem megfelel a szelektornak, különben `false`.

closest

A `closest` metódus az aktuális elemből kiindulva keres felfelé a DOM fa hierarchiájában, és az első olyan elemet adja vissza, amely megfelel a megadott CSS szelektornak.

Szintaxis:

```
let closestElement = element.closest(".myClass");
```

Jellemzők:

- Ha nincs megfelelő elem, `null` értéket ad vissza.

getElementsByClassName

A `getElementsByClassName` metódus egy élő HTMLCollection-t ad vissza azokról az elemekről, amelyek az adott osztálynévvel rendelkeznek.

Szintaxis:

```
let elements = document.getElementsByClassName("myClass");
```

Jellemzők:

- Élő gyűjteményt ad vissza, amely automatikusan frissül, ha a DOM változik.
- Több osztálynevet is megadhatunk szóközzel elválasztva.

getElementsByTagName

A `getElementsByTagName` metódus egy élő HTMLCollection-t ad vissza az adott címkével rendelkező összes elemmel.

Szintaxis:

```
let elements = document.getElementsByTagName("div");
```

Jellemzők:

- Élő gyűjteményt ad vissza, amely automatikusan frissül, ha a DOM változik.

getElementsByName

A `getElementsByName` metódus egy élő NodeList-et ad vissza azokról az elemekről, amelyek az adott névvel rendelkeznek.

Szintaxis:

```
let elements = document.getElementsByName("myName");
```

Jellemzők:

- Különösen hasznos form elemek esetén, ahol a `name` attribútum gyakran használt.

Live Collections (Élő Gyűjtemények)

Az élő gyűjtemények automatikusan frissülnek, amikor a DOM változik. Ez azt jelenti, hogy ha elemeket hozzáadunk vagy eltávolítunk a DOM-ból, az élő gyűjtemények (mint a `getElementsByClassName`, `getElementsByTagName`, és `getElementsByName` által visszaadott gyűjtemények) azonnal tükrözik ezeket a változásokat.

Példa:

```
let elements = document.getElementsByTagName("div");
console.log(elements.length); // Kiírja a <div> elemek számát

// Hozzáadunk egy új <div> elemet
let newDiv = document.createElement("div");
document.body.appendChild(newDiv);

console.log(elements.length); // Az új <div> hozzáadása után is kiírja a <div> elemek frissített számát
```

Összegzés

Metódus	Visszatérési típus	Élő gyűjtemény?	Leírás
`getElementById`	Element	Nem	Az első elem, amelynek az `id` attribútuma megfelel.
`querySelectorAll`	NodeList	Nem	Az összes elem, amely megfelel a CSS szelektornak.
`querySelector`	Element	Nem	Az első elem, amely megfelel a CSS szelektornak.
`matches`	Boolean	N/A	Igaz, ha az elem megfelel a CSS szelektornak.
`closest`	Element	N/A	Az első felmenő elem, amely megfelel a CSS szelektornak.
`getElementsByClassName`	HTMLCollection	Igen	Az összes elem, amely az adott osztálynévvel rendelkezik.
`getElementsByTagName`	HTMLCollection	Igen	Az összes elem, amely az adott címkével rendelkezik.
`getElementsByName`	NodeList	Igen	Az összes elem, amely az adott névvel rendelkezik.

DOM Propertyk és HTML Attribútumok Összefoglaló

HTML Attribútumok

A HTML attribútumok további információkat adnak az elemekhez. Az attribútumok mindig a kezdő tagban vannak megadva, és általában kulcs-érték párokként szerepelnek.

Példák:

```
<img src="image.jpg" alt="Kép leírása" />
<a href="https://www.example.com" target="_blank">Link</a>
<input type="text" value="Alapértelmezett szöveg" />
```

DOM Propertyk

A DOM (Document Object Model) egy programozási interfész, amely lehetővé teszi a HTML és XML dokumentumok fára épülő reprezentációját. A DOM propertyk a JavaScript segítségével érhetők el és módosíthatók, hogy dinamikusan változtassuk a dokumentumot.

Különbségek a DOM Propertyk és HTML Attribútumok Között

- **HTML attribútumok**: Az elem alapértelmezett értékei, amelyek a HTML kódban vannak megadva.
- **DOM propertyk**: Az elem aktuális értékei, amelyek a JavaScript segítségével érhetők el és módosíthatók.

Példa: Eredeti Attribútum és DOM Property Értékek

```
<input id="myInput" type="text" value="Alapértelmezett érték" />
```

```
let input = document.getElementById("myInput");

// HTML attribútum érték lekérdezése
console.log(input.getAttribute("value")); // Output: "Alapértelmezett érték"

// DOM property érték lekérdezése
console.log(input.value); // Output: "Alapértelmezett érték"

// DOM property módosítása
input.value = "Új érték";

// HTML attribútum továbbra is az eredeti értéket mutatja
console.log(input.getAttribute("value")); // Output: "Alapértelmezett érték"
```

Gyakran Használt HTML Attribútumok és DOM Propertyk

`class` és `className`

- **HTML attribútum**: `class`
- **DOM property**: `className`

```
<div class="myClass"></div>
```

```
let div = document.querySelector("div");
console.log(div.className); // Output: "myClass"
div.className = "newClass";
console.log(div.className); // Output: "newClass"
```

`for` és `htmlFor`

- **HTML attribútum**: `for`
- **DOM property**: `htmlFor`

```
<label for="myInput">Input Label</label> <input id="myInput" type="text" />
```

```
let label = document.querySelector("label");
console.log(label.htmlFor); // Output: "myInput"
label.htmlFor = "newInput";
console.log(label.htmlFor); // Output: "newInput"
```

`style`

- **HTML attribútum**: Inline CSS stílusok.
- **DOM property**: `style` objektum, amely CSS stílus tulajdonságokat tartalmaz.

```
<div style="color: red; background-color: yellow;"></div>
```

```
let div = document.querySelector("div");
console.log(div.style.color); // Output: "red"
div.style.color = "blue";
console.log(div.style.color); // Output: "blue"
```

`id`

- **HTML attribútum**: `id`
- **DOM property**: `id`

```
<div id="myId"></div>
```

```
let div = document.querySelector("div");
console.log(div.id); // Output: "myId"
div.id = "newId";
console.log(div.id); // Output: "newId"
```

value

- **HTML attribútum**: `value`
- **DOM property**: `value`

```
<input type="text" value="Alapértelmezett érték" />
```

```
let input = document.querySelector("input");
console.log(input.value); // Output: "Alapértelmezett érték"
input.value = "Új érték";
console.log(input.value); // Output: "Új érték"
```

href

- **HTML attribútum**: `href`
- **DOM property**: `href`

```
<a href="https://www.example.com">Link</a>
```

```
let link = document.querySelector("a");
console.log(link.href); // Output: "https://www.example.com"
link.href = "https://www.newexample.com";
console.log(link.href); // Output: "https://www.newexample.com"
```

Speciális Esetek

Adat-Attribútumok

Az adat-attribútumok egyedi adatokat tárolnak a HTML elemekben, amelyek a `data-` prefix-szel kezdődnek. Ezek JavaScript-ben a `dataset` property segítségével érhetők el.

```
<div data-user-id="12345"></div>
```

```
let div = document.querySelector("div");
console.log(div.dataset.userId); // Output: "12345"
div.dataset.userId = "67890";
console.log(div.dataset.userId); // Output: "67890"
```

Összegzés

- **HTML Attribútumok**: Az elem alapértelmezett értékei, amelyeket a HTML kódban adunk meg. Ezek a `getAttribute` és `setAttribute` metódusokkal érhetők el és módosíthatók.

- **DOM Propertyk**: Az elem aktuális értékei, amelyeket JavaScript-ből érhetünk el és módosíthatunk. Ezek a propertyk közvetlenül az elem objektumán érhetők el.

A DOM propertyk és HTML attribútumok közötti különbségek megértése alapvető fontosságú a hatékony DOM manipulációhoz és a dinamikus weboldalak létrehozásához.

HTML Attribútumok Kezelése

A HTML attribútumok kezelése a DOM API segítségével számos metódust kínál, amelyekkel ellenőrizhetjük, lekérdezhetjük, beállíthatjuk és eltávolíthatjuk az attribútumokat az elemekről. Az alábbiakban részletesen bemutatom a `hasAttribute`, `getAttribute`, `setAttribute` és `removeAttribute` metódusokat.

hasAttribute

A `hasAttribute` metódus ellenőrzi, hogy egy adott elem rendelkezik-e a megadott attribútummal. Igaz értéket ad vissza, ha az attribútum létezik, és hamis értéket, ha nem.

Szintaxis:

```
element.hasAttribute(name);
```

- **name**: Az attribútum neve (string).

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>hasAttribute Példa</title>
    </head>
    <body>
        <div id="myElement" class="myClass"></div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.hasAttribute("class")); // Output: true
            console.log(element.hasAttribute("style")); // Output: false
        </script>
    </body>
</html>
```

getAttribute

A `getAttribute` metódus visszaadja az adott attribútum értékét az elemről. Ha az attribútum nem létezik, `null` értéket ad vissza.

Szintaxis:

```
element.getAttribute(name);
```

- **name**: Az attribútum neve (string).

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>getAttribute Példa</title>
    </head>
    <body>
        <img id="myImage" src="image.jpg" alt="Kép leírása" />
        <script>
            let image = document.getElementById("myImage");
            console.log(image.getAttribute("src")); // Output: "image.jpg"
            console.log(image.getAttribute("alt")); // Output: "Kép leírása"
            console.log(image.getAttribute("title")); // Output: null
        </script>
    </body>
</html>
```

setAttribute

A `setAttribute` metódus beállítja az adott attribútum értékét az elemre. Ha az attribútum már létezik, az értékét módosítja; ha nem létezik, új attribútumot hoz létre.

Szintaxis:

```
element.setAttribute(name, value);
```

- **name**: Az attribútum neve (string).
- **value**: Az attribútum értéke (string).

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>setAttribute Példa</title>
    </head>
    <body>
        <a id="myLink" href="https://www.example.com">Link</a>
        <script>
            let link = document.getElementById("myLink");
            link.setAttribute("href", "https://www.newexample.com");
            link.setAttribute("title", "Új cím");
            console.log(link.getAttribute("href")); // Output: "https://www.newexample.com"
            console.log(link.getAttribute("title")); // Output: "Új cím"
        </script>
    </body>
</html>
```

removeAttribute

A `removeAttribute` metódus eltávolítja az adott attribútumot az elemről.

Szintaxis:

```
element.removeAttribute(name);
```

- **name**: Az attribútum neve (string).

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>removeAttribute Példa</title>
    </head>
    <body>
        <button id="myButton" disabled>Gomb</button>
        <script>
            let button = document.getElementById("myButton");
            button.removeAttribute("disabled");
            console.log(button.hasAttribute("disabled")); // Output: false
        </script>
    </body>
</html>
```

Összegzés

A `hasAttribute`, `getAttribute`, `setAttribute` és `removeAttribute` metódusok lehetővé teszik a HTML attribútumok hatékony kezelését a JavaScript segítségével. Ezek a metódusok alapvető eszközök a dinamikus webfejlesztésben, és segítségükkel könnyedén módosíthatjuk az elemek viselkedését és megjelenését.

Metódus	Leírás
`hasAttribute(name)`	Ellenőrzi, hogy az elem rendelkezik-e a megadott attribútummal.
`getAttribute(name)`	Visszaadja az adott attribútum értékét az elemről.
`setAttribute(name, value)`	Beállítja az adott attribútum értékét az elemre.
`removeAttribute(name)`	Eltávolítja az adott attribútumot az elemről.

Ezekkel a metódusokkal rugalmasan és hatékonyan kezelhetjük a HTML attribútumokat, így dinamikus és interaktív webalkalmazásokat hozhatunk létre.

DOM Propertyk Magyarázata

nodeType

A `nodeType` property visszaadja a csomópont típusát. Ez egy numerikus érték, amely az alábbiak közül egy lehet:

- **1**: Element (elem csomópont)
- **3**: Text (szöveg csomópont)
- **8**: Comment (komment csomópont)
- **9**: Document (dokumentum csomópont)

Szintaxis:

```
let type = node.nodeType;
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>nodeType Példa</title>
    </head>
    <body>
        <div id="myElement">Szöveg</div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.nodeType); // Output: 1 (Element)
            console.log(element.firstChild.nodeType); // Output: 3 (Text)
        </script>
    </body>
</html>
```

nodeName

A `nodeName` property visszaadja a csomópont nevét. Ez lehet a tag neve (pl. `DIV`, `SPAN`) vagy más típusú csomópontok esetén speciális érték (pl. `#text` szöveg csomópontok esetén).

Szintaxis:

```
let name = node.nodeName;
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>nodeName Példa</title>
    </head>
    <body>
        <div id="myElement">Szöveg</div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.nodeName); // Output: "DIV"
            console.log(element.firstChild.nodeName); // Output: "#text"
        </script>
    </body>
</html>
```

tagName

A `tagName` property visszaadja az elem címkéjének nevét. Ez mindig nagybetűs formában van visszaadva.

Szintaxis:

```
let tag = element.tagName;
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>tagName Példa</title>
    </head>
    <body>
        <div id="myElement">Szöveg</div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.tagName); // Output: "DIV"
        </script>
    </body>
</html>
```

innerHTML

Az `innerHTML` property lehetővé teszi az elem belső HTML tartalmának lekérdezését vagy beállítását. Ez az elem összes gyermekének HTML kódját tartalmazza.

Szintaxis:

```
let html = element.innerHTML;
element.innerHTML = "<p>Új tartalom</p>";
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>innerHTML Példa</title>
    </head>
    <body>
        <div id="myElement">Eredeti <strong>tartalom</strong></div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.innerHTML); // Output: "Eredeti <strong>tartalom</strong>"
            element.innerHTML = "<p>Új tartalom</p>";
        </script>
    </body>
</html>
```

outerHTML

Az `outerHTML` property lehetővé teszi az elem és annak teljes tartalmának HTML kódjának lekérdezését vagy beállítását. Ez magában foglalja az elemet magát és annak minden gyermekét.

Szintaxis:

```
let html = element.outerHTML;
element.outerHTML = '<div id="newElement">Új tartalom</div>';
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>outerHTML Példa</title>
    </head>
    <body>
        <div id="myElement">Eredeti <strong>tartalom</strong></div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.outerHTML); // Output: '<div id="myElement">Eredeti <strong>tartalom</strong></div>'
            element.outerHTML = '<div id="newElement">Új tartalom</div>';
        </script>
    </body>
</html>
```

textContent

A `textContent` property lehetővé teszi az elem szöveges tartalmának lekérdezését vagy beállítását. Ez az elem és annak összes gyermekének szövegét adja vissza, figyelmen kívül hagyva a HTML tageket.

Szintaxis:

```
let text = element.textContent;
element.textContent = "Új szöveg";
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>textContent Példa</title>
    </head>
    <body>
        <div id="myElement">Eredeti <strong>tartalom</strong></div>
        <script>
            let element = document.getElementById("myElement");
            console.log(element.textContent); // Output: "Eredeti tartalom"
            element.textContent = "Új szöveg";
        </script>
    </body>
</html>
```

hidden

A `hidden` property egy boolean érték, amely azt határozza meg, hogy az elem látható-e vagy sem. Ha az értéke `true`, az elem el van rejtve, ha `false`, akkor látható.

Szintaxis:

```
let isHidden = element.hidden;
element.hidden = true;
```

Példa:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>hidden Példa</title>
    </head>
    <body>
        <div id="myElement">Ez az elem el lesz rejtve</div>
        <script>
            let element = document.getElementById("myElement");
            element.hidden = true;
            console.log(element.hidden); // Output: true
        </script>
    </body>
</html>
```

Összegzés

Property	Leírás
`nodeType`	Visszaadja a csomópont típusát numerikus értékként.
`nodeName`	Visszaadja a csomópont nevét.
`tagName`	Visszaadja az elem címkéjének nevét.
`innerHTML`	Lekérdezi vagy beállítja az elem belső HTML tartalmát.
`outerHTML`	Lekérdezi vagy beállítja az elem teljes HTML kódját.
`textContent`	Lekérdezi vagy beállítja az elem szöveges tartalmát.
`hidden`	Boolean érték, amely meghatározza, hogy az elem látható-e.

Ezek a propertyk alapvető eszközök a DOM manipulációban, amelyek lehetővé teszik az elemek tartalmának és láthatóságának dinamikus kezelését a JavaScript segítségével.

DOM Manipulációs Metódusok Magyarázata

createElement

A `createElement` metódus egy új HTML elemet hoz létre. A létrehozott elem még nincs a DOM-ban, így azt hozzá kell adni egy meglévő elemhez.

Szintaxis:

```
let newElement = document.createElement(tagName);
```

- **tagName**: A létrehozandó HTML elem típusa (pl. `div`, `p`, `span`).

Példa:

```
let newDiv = document.createElement("div");
newDiv.textContent = "Ez egy új div elem";
document.body.appendChild(newDiv);
```

createTextNode

A `createTextNode` metódus egy új szövegcsomópontot hoz létre. A szövegcsomópontot hozzá kell adni egy meglévő elemhez, hogy megjelenjen a DOM-ban.

Szintaxis:

```
let newText = document.createTextNode(data);
```

- **data**: A létrehozandó szöveg.

Példa:

```
let newText = document.createTextNode("Ez egy új szövegcsomópont");
let newParagraph = document.createElement("p");
newParagraph.appendChild(newText);
document.body.appendChild(newParagraph);
```

append

Az `append` metódus új csomópontokat vagy szövegeket ad hozzá egy elem belsejének végéhez. Több csomópontot vagy szöveget is hozzáadhat egyszerre.

Szintaxis:

```
parentElement.append(nodesOrDOMStrings);
```

- **nodesOrDOMStrings**: Az új csomópontok vagy szövegek, amelyeket hozzá szeretnénk adni.

Példa:

```
let parentDiv = document.getElementById("parentDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
parentDiv.append(newDiv, " és szöveg");
```

prepend

A `prepend` metódus új csomópontokat vagy szövegeket ad hozzá egy elem belsejének elejéhez. Több csomópontot vagy szöveget is hozzáadhat egyszerre.

Szintaxis:

```
parentElement.prepend(nodesOrDOMStrings);
```

- **nodesOrDOMStrings**: Az új csomópontok vagy szövegek, amelyeket hozzá szeretnénk adni.

Példa:

```
let parentDiv = document.getElementById("parentDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
parentDiv.prepend(newDiv, " és szöveg");
```

before

A `before` metódus új csomópontokat vagy szövegeket ad hozzá egy elem előtt. Több csomópontot vagy szöveget is hozzáadhat egyszerre.

Szintaxis:

```
referenceNode.before(nodesOrDOMStrings);
```

- **nodesOrDOMStrings**: Az új csomópontok vagy szövegek, amelyeket hozzá szeretnénk adni.

Példa:

```
let referenceDiv = document.getElementById("referenceDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
referenceDiv.before(newDiv, " és szöveg");
```

after

Az `after` metódus új csomópontokat vagy szövegeket ad hozzá egy elem után. Több csomópontot vagy szöveget is hozzáadhat egyszerre.

Szintaxis:

```
referenceNode.after(nodesOrDOMStrings);
```

- **nodesOrDOMStrings**: Az új csomópontok vagy szövegek, amelyeket hozzá szeretnénk adni.

Példa:

```
let referenceDiv = document.getElementById("referenceDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
referenceDiv.after(newDiv, " és szöveg");
```

replaceWith

A `replaceWith` metódus kicseréli az elemet egy vagy több új csomópontra vagy szövegre.

Szintaxis:

```
oldNode.replaceWith(nodesOrDOMStrings);
```

- **nodesOrDOMStrings**: Az új csomópontok vagy szövegek, amelyekkel az elemet kicseréljük.

Példa:

```
let oldDiv = document.getElementById("oldDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
oldDiv.replaceWith(newDiv, " és szöveg");
```

insertAdjacentHTML

Az `insertAdjacentHTML` metódus HTML szöveget illeszt be az elem egy megadott pozíciójába. Az illesztett szöveg HTML-ként értelmeződik, és a megfelelő elemek létrejönnek.

Szintaxis:

```
element.insertAdjacentHTML(position, text);
```

- **position**: Az illesztés helye (`'beforebegin'`, `'afterbegin'`, `'beforeend'`, `'afterend'`).

- **text**: A beszúrandó HTML szöveg.

Példa:

```
let referenceDiv = document.getElementById("referenceDiv");
referenceDiv.insertAdjacentHTML("beforebegin", "<div>Új elem</div>");
referenceDiv.insertAdjacentHTML("afterend", "<div>Új elem</div>");
```

insertAdjacentText

Az `insertAdjacentText` metódus szöveget illeszt be az elem egy megadott pozíciójába. Az illesztett szöveg nem HTML-ként, hanem egyszerű szövegként kerül beillesztésre.

Szintaxis:

```
element.insertAdjacentText(position, text);
```

- **position**: Az illesztés helye (`'beforebegin'`, `'afterbegin'`, `'beforeend'`, `'afterend'`).
- **text**: A beszúrandó szöveg.

Példa:

```
let referenceDiv = document.getElementById("referenceDiv");
referenceDiv.insertAdjacentText("beforebegin", "Új szöveg");
referenceDiv.insertAdjacentText("afterend", "Új szöveg");
```

insertAdjacentElement

Az `insertAdjacentElement` metódus egy meglévő DOM elemet illeszt be az elem egy megadott pozíciójába.

Szintaxis:

```
element.insertAdjacentElement(position, newElement);
```

- **position**: Az illesztés helye (`'beforebegin'`, `'afterbegin'`, `'beforeend'`, `'afterend'`).
- **newElement**: A beszúrandó DOM elem.

Példa:

```
let referenceDiv = document.getElementById("referenceDiv");
let newDiv = document.createElement("div");
newDiv.textContent = "Új elem";
referenceDiv.insertAdjacentElement("beforebegin", newDiv);
```

remove

A `remove` metódus eltávolítja az elemet a DOM-ból.

Szintaxis:

```
element.remove();
```

Példa:

```
let element = document.getElementById("elementToRemove");
element.remove();
```

Összegzés

Metódus	Leírás
`createElement`	Új HTML elemet hoz létre.
`createTextNode`	Új szövegcsomópontot hoz létre.
`append`	Új csomópontokat vagy szövegeket ad az elem végéhez.
`prepend`	Új csomópontokat vagy szövegeket ad az elem elejéhez.
`before`	Új csomópontokat vagy szövegeket ad az elem elé.
`after`	Új csomópontokat vagy szövegeket ad az elem után.
`replaceWith`	Kicseréli az elemet egy vagy több új csomópontra vagy szövegre.
`insertAdjacentHTML`	HTML szöveget illeszt be az elem egy megadott pozíciójába.
`insertAdjacentText`	Szöveget illeszt be az elem egy megadott pozíciójába.
`insertAdjacentElement`	Egy meglévő DOM elemet illeszt be az elem egy megadott pozíciójába.
`remove`	Eltávolítja az elemet a DOM-ból.

Ezek a metódusok alapvető eszközök a DOM manipulációban, amelyek lehetővé teszik az elemek dinamikus létrehozását, módosítását, beillesztését és eltávolítását.

JavaScript Események Magyarázata

Mi az Esemény?

Az esemény egy olyan jelenség, amely valamilyen művelet eredményeként történik, például egy felhasználói művelet (kattintás, billentyűleütés) vagy egy rendszer által generált esemény (pl. oldal betöltődése). Az események lehetővé teszik a weboldalak számára, hogy interaktívak és dinamikusak legyenek.

Események Kezelése

Az eseménykezelés lényege, hogy figyeljük az eseményeket és reagáljunk rájuk valamilyen módon. Az események kezeléséhez JavaScript-ben eseménykezelőket használunk.

Események Hozzáadása

Eseménykezelőket többféleképpen adhatunk hozzá az elemekhez:

1. HTML attribútum használatával

Az eseménykezelőt közvetlenül a HTML elem attribútumában definiálhatjuk.

Példa:

```
<button onclick="alert('Gombra kattintottál!')">Kattints rám!</button>
```

2. DOM Property használatával

Az eseménykezelőt JavaScript-ben is definiálhatjuk, közvetlenül az elem property-jéhez rendelve.

Példa:

```
<button id="myButton">Kattints rám!</button>
<script>
    document.getElementById("myButton").onclick = function () {
        alert("Gombra kattintottál!");
    };
</script>
```

3. addEventListener metódus használatával

Az `addEventListener` metódus a leggyakoribb és legjobb módszer az eseménykezelők hozzáadására, mivel lehetővé teszi több eseménykezelő hozzáadását ugyanahhoz az eseményhez.

Példa:

```
<button id="myButton">Kattints rám!</button>
<script>
    document.getElementById("myButton").addEventListener("click", function () {
        alert("Gombra kattintottál!");
    });
</script>
```

Eseménytípusok

Számos különböző eseménytípus létezik, amelyek különböző felhasználói műveletekre vagy rendszereseményekre reagálnak:

1. Egér Események

- **`click`**: A felhasználó kattint egy elemre.
- **`dblclick`**: A felhasználó duplán kattint egy elemre.
- **`mousedown`**: Az egérgomb lenyomásra kerül egy elem felett.
- **`mouseup`**: Az egérgomb felengedésre kerül egy elem felett.
- **`mousemove`**: Az egér mozog egy elem felett.
- **`mouseenter`**: Az egér belép egy elem fölé.
- **`mouseleave`**: Az egér elhagy egy elemet.

2. Billentyű Események

- **`keydown`**: A felhasználó lenyom egy billentyűt.
- **`keypress`**: A felhasználó lenyom egy karakter billentyűt.
- **`keyup`**: A felhasználó felenged egy billentyűt.

3. Form Események

- **`submit`**: Egy űrlap beküldésre kerül.
- **`focus`**: Egy elem fókuszba kerül.
- **`blur`**: Egy elem elveszti a fókuszt.
- **`change`**: Egy űrlap elem értéke megváltozik.

4. Dokumentum Események

- **`DOMContentLoaded`**: A DOM teljesen betöltődött és feldolgozott.
- **`load`**: Az oldal teljesen betöltődött, beleértve az összes erőforrást (képek, stíluslapok stb.).
- **`resize`**: Az ablak mérete megváltozik.
- **`scroll`**: Az oldal vagy egy elem el van görgetve.

Eseményobjektum

Amikor egy esemény bekövetkezik, az eseménykezelő függvény egy eseményobjektumot kap paraméterként, amely információkat tartalmaz az eseményről.

Példa:

```
<button id="myButton">Kattints rám!</button>
<script>
    document
        .getElementById("myButton")
        .addEventListener("click", function (event) {
            console.log(event.type); // Az esemény típusa (pl. "click")
            console.log(event.target); // Az elem, amelyre kattintottak
        });
</script>
```

Eseménykezelők Eltávolítása

Az `addEventListener`-rel hozzáadott eseménykezelők eltávolíthatók az `removeEventListener` metódussal.

Példa:

```
<button id="myButton">Kattints rám!</button>
<script>
    function handleClick() {
        alert("Gombra kattintottál!");
    }

    let button = document.getElementById("myButton");
    button.addEventListener("click", handleClick);

    // Eseménykezelő eltávolítása
    button.removeEventListener("click", handleClick);
</script>
```

Feladatok Eseményekkel

Feladat 1: Szöveg Színének Megváltoztatása

Cél: Hozz létre egy gombot, amelyre kattintva megváltozik egy szöveg színe.

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Szöveg Színének Megváltoztatása</title>
    </head>
    <body>
        <p id="myText">Ez a szöveg színe megváltozik.</p>
        <button id="myButton">Változtasd meg a színt!</button>
        <script>
            document
                .getElementById("myButton")
                .addEventListener("click", function () {
                    document.getElementById("myText").style.color = "red";
                });
        </script>
    </body>
</html>
```

Feladat 2: Szöveg Megjelenítése Eseménykor

Cél: Hozz létre egy szövegmezőt és egy gombot. Amikor a gombra kattintasz, jelenítsd meg a szövegmező tartalmát egy `p` elemben.

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Szöveg Megjelenítése</title>
    </head>
    <body>
        <input type="text" id="myInput" placeholder="Írj valamit..." />
        <button id="myButton">Mutasd meg a szöveget!</button>
        <p id="output"></p>
        <script>
            document
                .getElementById("myButton")
                .addEventListener("click", function () {
                    let inputText = document.getElementById("myInput").value;
                    document.getElementById("output").textContent = inputText;
                });
        </script>
    </body>
</html>
```

Feladat 3: Egér Pozíciójának Követése

Cél: Hozz létre egy dobozt, amely követi az egér pozícióját és megjeleníti a koordinátákat.

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Egér Pozíciója</title>
        <style>
            #myBox {
                width: 200px;
                height: 200px;
                border: 1px solid black;
                position: relative;
            }
        </style>
    </head>
    <body>
        <div id="myBox"></div>
        <p id="coords"></p>
        <script>
            document
                .getElementById("myBox")
                .addEventListener("mousemove", function (event) {
                    let x = event.clientX;
                    let y = event.clientY;
                    document.getElementById(
                        "coords"
                    ).textContent = `X: ${x}, Y: ${y}`;
                });
        </script>
    </body>
</html>
```

07.08.

Aszinkron JavaScript

Mi az az Aszinkron JavaScript?

Az aszinkron programozás lehetővé teszi, hogy a JavaScript kód blokkolás nélkül végezzen el műveleteket. Ez különösen hasznos, amikor hosszabb időt igénylő műveleteket, például hálózati kéréseket vagy időigényes számításokat kell végezni, mivel lehetővé teszi, hogy a program más műveleteket is végrehajtson ahelyett, hogy várna ezeknek a műveleteknek a befejezésére.

Callback Hell és Pyramid of Doom

Callback Hell

A "callback hell" egy olyan helyzet, amikor mélyen egymásba ágyazott callback-eket használunk, amelyek miatt a kód olvashatatlanná és karbantarthatatlanná válik.

Példa Callback Hell-re:

```
fs.readFile("file1.txt", "utf8", function (err, data1) {
    if (err) {
        throw err;
    }
    fs.readFile("file2.txt", "utf8", function (err, data2) {
        if (err) {
            throw err;
        }
        fs.readFile("file3.txt", "utf8", function (err, data3) {
            if (err) {
                throw err;
            }
            console.log(data1, data2, data3);
        });
    });
});
```

Pyramid of Doom

A "pyramid of doom" kifejezés arra a helyzetre utal, amikor a kódunk egymásba ágyazott callback-ek miatt piramis alakot vesz fel, ami megnehezíti az olvasást és a hibakeresést.

Példa Pyramid of Doom-ra:

```
doSomething(function (result) {
    doSomethingElse(result, function (newResult) {
        doAnotherThing(newResult, function (finalResult) {
            console.log(finalResult);
        });
    });
});
```

Promise-ok

Promise Szintaxis

A Promise egy objektum, amely egy aszinkron művelet végkimenetelét reprezentálja. Egy Promise három állapotban lehet: pending, fulfilled, rejected.

Promise Létrehozása

```
let myPromise = new Promise(function(resolve, reject) {
    // Aszinkron művelet
    if (/* sikeres */) {
        resolve("Siker!");
    } else {
        reject("Hiba történt");
    }
});
```

Promise Állapotok

- **Pending**: A művelet folyamatban van.
- **Fulfilled**: A művelet sikeresen befejeződött, és egy értéket adott vissza.
- **Rejected**: A művelet sikertelen volt, és egy hibát adott vissza.

resolve és reject

- **resolve**: Meghívásakor a Promise állapota `fulfilled` lesz, és a megadott értéket adja vissza.
- **reject**: Meghívásakor a Promise állapota `rejected` lesz, és a megadott hibát adja vissza.

Példa resolve és reject használatára:

```
let myPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        let success = true; // Ez egy példa, valós helyzetben itt történne valami aszinkron művelet
        if (success) {
            resolve("Siker!");
        } else {
            reject("Hiba történt");
        }
    }, 2000);
});
```

then, catch, finally

- **then**: A Promise sikeres teljesülésekor (fulfilled) fut le, és az eredményt adja vissza.
- **catch**: A Promise elutasítása (rejected) esetén fut le, és a hibát adja vissza.
- **finally**: Függetlenül attól, hogy a Promise sikeresen teljesült vagy elutasításra került, mindig lefut.

Példa then, catch, finally használatára:

```
myPromise
    .then(function (value) {
        console.log("Siker:", value);
    })
    .catch(function (error) {
        console.log("Hiba:", error);
    })
    .finally(function () {
        console.log("Ez mindig lefut, függetlenül a Promise állapotától.");
    });
```

Promise-ok Láncolása

A Promise-ok láncolása lehetővé teszi több aszinkron művelet sorozatos végrehajtását, mindegyik a korábbi művelet eredményére építve. A `then` metódus mindig új Promise-t ad vissza, amely lehetővé teszi a láncolást.

Példa Promise-ok Láncolására:

```
let firstPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("Első művelet kész!");
    }, 1000);
});

firstPromise
    .then(function (value) {
        console.log(value);
        return new Promise(function (resolve, reject) {
            setTimeout(function () {
                resolve("Második művelet kész!");
            }, 1000);
        });
    })
    .then(function (value) {
        console.log(value);
        return new Promise(function (resolve, reject) {
            setTimeout(function () {
                resolve("Harmadik művelet kész!");
            }, 1000);
        });
    })
    .then(function (value) {
        console.log(value);
    })
    .catch(function (error) {
        console.error(error);
    });
```

Statikus Promise metódusok

Promise.all

A `Promise.all` metódus egyetlen Promise-t ad vissza, amely akkor teljesül, ha az összes megadott Promise teljesül, vagy elutasításra kerül, ha bármelyik Promise elutasításra kerül. Az eredmény egy tömb lesz, amely az összes Promise értékeit tartalmazza.

Szintaxis:

```
Promise.all([promise1, promise2, ...]);
```

Példa:

```
let promise1 = Promise.resolve(3);
let promise2 = 42;
let promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, "foo");
});

Promise.all([promise1, promise2, promise3])
    .then((values) => {
        console.log(values); // [3, 42, "foo"]
    })
    .catch((error) => {
        console.error(error);
    });
```

Promise.allSettled

A `Promise.allSettled` metódus egyetlen Promise-t ad vissza, amely akkor teljesül, amikor az összes megadott Promise teljesül vagy elutasításra kerül. Az eredmény egy tömb lesz, amely minden Promise állapotát és értékét vagy hibáját tartalmazza.

Szintaxis:

```
Promise.allSettled([promise1, promise2, ...]);
```

Példa:

```
let promise1 = Promise.resolve(3);
let promise2 = new Promise((resolve, reject) => {
    setTimeout(reject, 100, "foo");
});

Promise.allSettled([promise1, promise2]).then((results) => {
    results.forEach((result) =>
        console.log(result.status, result.value || result.reason)
    );
    // "fulfilled" 3
    // "rejected" "foo"
});
```

Promise.race

A `Promise.race` metódus egyetlen Promise-t ad vissza, amely akkor teljesül vagy kerül elutasításra, amikor az első megadott Promise teljesül vagy elutasításra kerül. Az eredmény az első befejezett Promise értéke vagy hibája lesz.

Szintaxis:

```
Promise.race([promise1, promise2, ...]);
```

Példa:

```
let promise1 = new Promise((resolve, reject) => {
    setTimeout(resolve, 500, "one");
});
let promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 100, "two");
});

Promise.race([promise1, promise2])
    .then((value) => {
        console.log(value); // "two"
    })
    .catch((error) => {
        console.error(error);
    });
```

Promise.any

A `Promise.any` metódus egyetlen Promise-t ad vissza, amely akkor teljesül, ha az első Promise teljesül. Ha az összes megadott Promise elutasításra kerül, akkor az eredmény egy aggregált hiba lesz.

Szintaxis:

```
Promise.any([promise1, promise2, ...]);
```

Példa:

```
let promise1 = new Promise((resolve, reject) => {
    setTimeout(reject, 100, "foo");
});
let promise2 = new Promise((resolve, reject) => {
    setTimeout(resolve, 500, "bar");
});
let promise3 = new Promise((resolve, reject) => {
    setTimeout(resolve, 1000, "baz");
});

Promise.any([promise1, promise2, promise3])
    .then((value) => {
        console.log(value); // "bar"
    })
    .catch((error) => {
        console.error(error);
    });
```

Promise.resolve

A `Promise.resolve` metódus egy új, teljesült Promise-t ad vissza a megadott értékkel. Ha az érték egy Promise, akkor az új Promise az eredeti Promise állapotát fogja tükrözni.

Szintaxis:

```
Promise.resolve(value);
```

Példa:

```
let resolvedPromise = Promise.resolve(42);

resolvedPromise.then((value) => {
    console.log(value); // 42
});
```

Promise.reject

A `Promise.reject` metódus egy új, elutasított Promise-t ad vissza a megadott hibával vagy okkal.

Szintaxis:

```
Promise.reject(reason);
```

Példa:

```
let rejectedPromise = Promise.reject(new Error("Hiba történt"));

rejectedPromise.catch((error) => {
    console.error(error); // Error: Hiba történt
});
```

Hálózati Kérés (Network Request)

Mi az a Hálózati Kérés?

A hálózati kérés egy olyan folyamat, amely során egy számítógép (kliens) adatokat kér egy másik számítógéptől (szerver) a hálózaton keresztül. A webfejlesztésben a hálózati kéréseket gyakran használják adatok lekérésére vagy küldésére egy szerverre az interneten keresztül.

Példa Hálózati Kérésre:

Amikor egy böngésző megnyit egy weboldalt, egy hálózati kérést küld a weboldal szerverére, amely visszaküldi a weboldal HTML kódját. Ez a HTML kód letöltésre kerül, és a böngésző megjeleníti azt.

API (Application Programming Interface)

Mi az az API?

Az API (Application Programming Interface) egy olyan interfész, amely lehetővé teszi, hogy két különböző szoftveralkalmazás kommunikáljon egymással. Az API meghatározza a szabályokat és protokollokat, amelyeken keresztül az alkalmazások adatokat cserélhetnek.

Példa API-ra:

Egy időjárás API lehetővé teszi, hogy egy alkalmazás lekérje az aktuális időjárási adatokat egy adott helyszínről. Az alkalmazás HTTP kéréseket küld az API-nak, és az API visszaküldi az időjárási adatokat JSON formátumban.

HTTP Metódusok

Mi az a HTTP?

A HTTP (HyperText Transfer Protocol) az alapvető protokoll, amelyet a weben használnak az adatok átvitelére. A HTTP metódusok meghatározzák, hogy milyen műveleteket lehet végrehajtani az erőforrásokon.

Gyakori HTTP Metódusok

- **GET**: Adatok lekérése egy szerverről. A GET kéréseknek nincs mellékhatása, azaz nem módosítják a szerveren lévő adatokat.
- **POST**: Adatok küldése a szerverre. A POST kéréseket gyakran használják adatok beküldésére, például űrlapok küldésére.
- **PUT**: Egy meglévő erőforrás módosítása vagy egy új erőforrás létrehozása a szerveren.
- **DELETE**: Egy erőforrás törlése a szerverről.
- **PATCH**: Egy meglévő erőforrás részleges módosítása a szerveren.

Példa GET Kérésre:

```
GET /users HTTP/1.1
Host: api.example.com
```

JSON (JavaScript Object Notation)

Mi az a JSON?

A JSON (JavaScript Object Notation) egy könnyű adatcsere formátum, amelyet könnyű ember által olvasni és írni, valamint gépek számára elemezni és generálni. A JSON-t gyakran használják API-kban az adatok cseréjére.

JSON Példa:

```
{
    "name": "John Doe",
    "age": 30,
    "city": "New York"
}
```

fetch Használata

Mi az a fetch?

A `fetch` egy modern JavaScript API, amely lehetővé teszi hálózati kérések egyszerű végrehajtását. A `fetch` segítségével aszinkron HTTP kéréseket küldhetünk és fogadhatunk adatokat a szerverről.

fetch Szintaxis

Alapvető GET Kérés:

```
fetch("https://api.example.com/data")
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log(data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

POST Kérés:

```
fetch("https://api.example.com/data", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({ name: "John Doe", age: 30 }),
})
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log(data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

fetch Paraméterek

- **URL**: A kért erőforrás URL-je.
- **options**: Egy opcionális objektum, amely tartalmazza a kéréssel kapcsolatos beállításokat (pl. metódus, fejlécek, törzs).

fetch Válasz Objektum

A `fetch` által visszaadott válasz objektum tartalmazza a kéréssel kapcsolatos információkat, például a státuszkódot és a válasz törzsét.

Válasz Objektum Példája:

```
fetch("https://api.example.com/data")
    .then((response) => {
        console.log(response.status); // Státuszkód
        console.log(response.statusText); // Státusz szöveg
        return response.json(); // Válasz törzsének JSON formátumra alakítása
    })
    .then((data) => {
        console.log(data);
    });
```

Összegzés

Hálózati Kérés (Network Request)

A hálózati kérések lehetővé teszik, hogy a kliens adatokhoz férjen hozzá egy szerveren keresztül. Az API-k és HTTP metódusok használatával különböző műveleteket hajthatunk végre, mint például adatok lekérése vagy küldése.

API (Application Programming Interface)

Az API-k lehetővé teszik különböző alkalmazások közötti kommunikációt és adatcserét. Az API-k meghatározzák a kommunikációs szabályokat és protokollokat.

HTTP Metódusok

A HTTP metódusok meghatározzák, hogy milyen műveleteket hajthatunk végre egy erőforráson:

- **GET**: Adatok lekérése.
- **POST**: Adatok küldése.
- **PUT**: Adatok módosítása vagy létrehozása.
- **DELETE**: Adatok törlése.
- **PATCH**: Adatok részleges módosítása.

JSON (JavaScript Object Notation)

A JSON egy könnyű adatcsere formátum, amelyet gyakran használnak az API-kban. Könnyen olvasható és elemezhető mind emberek, mind gépek számára.

fetch Használata

A `fetch` API segítségével aszinkron HTTP kéréseket küldhetünk és fogadhatunk adatokat a szerverről. A `fetch` használata egyszerű és modern módja a hálózati kérések kezelésének JavaScript-ben.

A fenti példák és magyarázatok segítségével könnyebben megérthetjük és alkalmazhatjuk a hálózati kéréseket, API-kat, HTTP metódusokat, JSON formátumot és a `fetch` API-t a JavaScript alkalmazásokban.

HTTP Metódusok és Kérések

POST, PUT, PATCH és DELETE Kérések

A HTTP protokollban többféle módszer létezik az adatok küldésére és kezelésére a szerverrel való kommunikáció során. A leggyakrabban használt módszerek a POST, PUT, PATCH és DELETE, amelyek különböző típusú műveletekhez alkalmazhatók. Ezek a metódusok az adatok kezelésére és manipulálására szolgálnak a RESTful API-kban.

POST Kérés

A POST kérések adatokat küldenek a szerverre, és általában új erőforrások létrehozására használják őket. A küldött adatokat a kérés törzsében (body) kell megadni.

Példa egy POST kérésre a `fetch` API használatával:

```
fetch("https://api.example.com/resource", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({
        name: "John Doe",
        email: "john.doe@example.com",
    }),
})
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log("Sikeresen létrehozva:", data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

PUT Kérés

A PUT kérések egy meglévő erőforrás teljes módosítására vagy létrehozására szolgálnak. Ha az erőforrás létezik, akkor az új adatokkal felülíródik, ha nem létezik, akkor létrejön.

Példa egy PUT kérésre a `fetch` API használatával:

```
fetch("https://api.example.com/resource/1", {
    method: "PUT",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({
        name: "Jane Doe",
        email: "jane.doe@example.com",
    }),
})
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log("Sikeresen módosítva:", data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

PATCH Kérés

A PATCH kérések egy meglévő erőforrás részleges módosítására szolgálnak. Csak a megadott mezők frissülnek, a többi mező változatlan marad.

Példa egy PATCH kérésre a `fetch` API használatával:

```
fetch("https://api.example.com/resource/1", {
    method: "PATCH",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({
        email: "jane.doe@newdomain.com",
    }),
})
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log("Sikeresen frissítve:", data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

DELETE Kérés

A DELETE kérések egy meglévő erőforrás törlésére szolgálnak a szerverről.

Példa egy DELETE kérésre a `fetch` API használatával:

```
fetch("https://api.example.com/resource/1", {
    method: "DELETE",
    headers: {
        "Content-Type": "application/json",
    },
})
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log("Sikeresen törölve:", data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

Async-Await Magyarázat

Mi az az Async-Await?

Az async-await a JavaScript nyelv két kulcsszava, amelyeket az ECMAScript 2017 (ES8) specifikációban vezettek be. Az async-await segítségével az aszinkron műveleteket könnyebben kezelhetjük, mivel szinkron kódhoz hasonló szerkezetet biztosítanak, ezzel javítva a kód olvashatóságát és karbantarthatóságát.

Az async Kulcsszó

Az `async` kulcsszó egy függvény előtt használva azt jelzi, hogy a függvény aszinkron műveleteket tartalmaz. Egy async függvény mindig egy Promise-t ad vissza.

Szintaxis:

```
async function myAsyncFunction() {
    // kód
}
```

Az await Kulcsszó

Az `await` kulcsszó az `async` függvényen belül használható. Az `await`-et egy Promise előtt használva azt jelzi, hogy a kódnak várnia kell, amíg a Promise teljesül, mielőtt folytatná a végrehajtást.

Szintaxis:

```
let result = await somePromise;
```

Hogyan Működik az Async-Await?

Az async-await használata során az `await` kulcsszó "megállítja" az async függvény végrehajtását, amíg a Promise teljesül vagy elutasításra kerül. Ez lehetővé teszi, hogy a kód szinkron stílusban írjuk, miközben a háttérben továbbra is aszinkron módon működik.

Példa: Async-Await Használata

Hagyományos Promise Kezelés

```
function getData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Adatok");
        }, 2000);
    });
}

getData()
    .then((data) => {
        console.log(data); // "Adatok"
    })
    .catch((error) => {
        console.error(error);
    });
```

Async-Await Használata

```
async function fetchData() {
    try {
        let data = await getData();
        console.log(data); // "Adatok"
    } catch (error) {
        console.error(error);
    }
}

fetchData();

function getData() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Adatok");
        }, 2000);
    });
}
```

Fontos Pontok

1. **Async függvény mindig Promise-t ad vissza**: Ha egy async függvényen belül visszaadunk egy értéket, az érték egy teljesült Promise-ba lesz csomagolva.

   ```
   async function example() {
       return "Hello";
   }

   example().then((value) => console.log(value)); // "Hello"
   ```

2. **Await csak async függvényen belül használható**: Az `await` kulcsszó csak egy async függvényen belül használható, különben szintaxis hiba lép fel.

   ```
   async function example() {
       let result = await somePromise;
       console.log(result);
   }
   ```

3. **Hibakezelés**: Az async függvényekben a hibakezelés egyszerűen megoldható a try-catch blokkok használatával.

```
async function fetchData() {
    try {
        let data = await getData();
        console.log(data);
    } catch (error) {
        console.error(error);
    }
}
```

Gyakorlati Példa: Hálózati Kérés Async-Await Segítségével

Hagyományos Promise Kezelés Fetch API-val

```
fetch("https://api.example.com/data")
    .then((response) => {
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        return response.json();
    })
    .then((data) => {
        console.log(data);
    })
    .catch((error) => {
        console.error("Hiba:", error);
    });
```

Async-Await Használata Fetch API-val

```
async function getData() {
    try {
        let response = await fetch("https://api.example.com/data");
        if (!response.ok) {
            throw new Error("Hálózati hiba történt");
        }
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.error("Hiba:", error);
    }
}

getData();
```