## 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.