Miért Nem Lehet Tömböket == Operátorral Összehasonlítani

JavaScriptben az == és === operátorok használatával történő összehasonlításkor az alapvető különbség az, hogy az == operátor típuskonverziót hajt végre (laza egyenlőség), míg a === operátor szigorú egyenlőséget ellenőriz (típuskonverzió nélkül). Azonban egyik operátor sem működik megfelelően tömbök összehasonlítására. Az okok a következők:

1. Referencia Típusú Összehasonlítás

JavaScriptben a tömbök referencia típusok. Ez azt jelenti, hogy amikor két tömböt hasonlítunk össze az == vagy === operátorral, akkor azok memóriában lévő címét hasonlítjuk össze, nem pedig a tényleges tartalmukat.

Példa:

```
let tomb1 = [1, 2, 3];
let tomb2 = [1, 2, 3];

console.log(tomb1 == tomb2); // false
console.log(tomb1 === tomb2); // false
```

Ebben a példában a `tomb1` és `tomb2` ugyanazokat az elemeket tartalmazzák, de különböző helyen vannak tárolva a memóriában, így az összehasonlításuk `false` értéket ad vissza.

2. Objektumok és Tömbök Memóriakezelése

Mivel a tömbök és más objektumok referenciaként vannak tárolva, az összehasonlítás során a JavaScript csak azt ellenőrzi, hogy a két változó ugyanarra a referenciára mutat-e, nem pedig azt, hogy a két tömb elemei megegyeznek-e.

Példa:

```
let tomb1 = [1, 2, 3];
let tomb2 = tomb1;

console.log(tomb1 == tomb2); // true
console.log(tomb1 === tomb2); // true
```

Ebben a példában a `tomb2` változónak ugyanaz a referenciája, mint a `tomb1` változónak, tehát az összehasonlítás `true` értéket ad vissza.

JavaScript Tömb Metódusok Magyarázata

JavaScript tömbökhöz számos beépített metódus létezik, amelyek lehetővé teszik a tömbök módosítását és kezelését. Az alábbiakban részletesen bemutatom a `splice`, `slice`, `indexOf`, `lastIndexOf`, `concat` és `reverse` metódusokat.

1. `splice` Metódus

A `splice` metódus lehetővé teszi elemek hozzáadását, eltávolítását vagy cseréjét egy tömbben.

Szintaxis:

```
array.splice(start, deleteCount, item1, item2, ..., itemN);
```

- `start`: Az a pozíció, ahol a módosítás kezdődik.
- `deleteCount`: Az eltávolítandó elemek száma.
- `item1, item2, ..., itemN`: Azok az elemek, amelyeket a tömbbe kell illeszteni (opcionális).

Példa:

```
let gyumolcsok = ["alma", "banán", "cseresznye"];
gyumolcsok.splice(1, 1, "körte", "narancs"); // Az 1-es indexű elemtől kezdve 1 elemet töröl, és hozzáad két új elemet
console.log(gyumolcsok); // ["alma", "körte", "narancs", "cseresznye"]
```

2. `slice` Metódus

A `slice` metódus egy új tömböt hoz létre, amely az eredeti tömb egy részét tartalmazza. Az eredeti tömb nem változik.

Szintaxis:

```
array.slice(start, end);
```

- `start`: A kezdő index (beleértve).
- `end`: A befejező index (kizárva). Ha nincs megadva, a tömb végéig tart.

Példa:

```
let gyumolcsok = ["alma", "banán", "cseresznye", "körte"];
let reszlet = gyumolcsok.slice(1, 3); // Új tömb az 1-es és 2-es indexű elemekkel
console.log(reszlet); // ["banán", "cseresznye"]
```

3. `indexOf` Metódus

Az `indexOf` metódus megkeresi az első előfordulását egy adott elemnek a tömbben, és visszaadja annak indexét. Ha az elem nem található, `-1`-et ad vissza.

Szintaxis:

```
array.indexOf(searchElement, fromIndex);
```

- `searchElement`: Az a keresendő elem.
- `fromIndex`: Az index, amelytől kezdve a keresést elindítjuk (opcionális).

Példa:

```
let gyumolcsok = ["alma", "banán", "cseresznye"];
let index = gyumolcsok.indexOf("banán");
console.log(index); // 1
```

4. `lastIndexOf` Metódus

A `lastIndexOf` metódus megkeresi az utolsó előfordulását egy adott elemnek a tömbben, és visszaadja annak indexét. Ha az elem nem található, `-1`-et ad vissza.

Szintaxis:

```
array.lastIndexOf(searchElement, fromIndex);
```

- `searchElement`: Az a keresendő elem.
- `fromIndex`: Az index, amelytől kezdve visszafelé a keresést elindítjuk (opcionális).

Példa:

```
let gyumolcsok = ["alma", "banán", "cseresznye", "banán"];
let index = gyumolcsok.lastIndexOf("banán");
console.log(index); // 3
```

5. `concat` Metódus

A `concat` metódus összekapcsol egy vagy több tömböt az eredeti tömb másolataival, és egy új tömböt ad vissza. Az eredeti tömb nem változik.

Szintaxis:

```
let newArray = array1.concat(array2, array3, ..., arrayN);
```

- `array1, array2, ..., arrayN`: Az összekapcsolandó tömbök.

Példa:

```
let gyumolcsok = ["alma", "banán"];
let zoldsegek = ["répa", "brokkoli"];
let etelek = gyumolcsok.concat(zoldsegek);
console.log(etelek); // ["alma", "banán", "répa", "brokkoli"]
```

6. `reverse` Metódus

A `reverse` metódus megfordítja a tömb elemeinek sorrendjét. Az eredeti tömb módosul.

Szintaxis:

```
array.reverse();
```

Példa:

```
let gyumolcsok = ["alma", "banán", "cseresznye"];
gyumolcsok.reverse();
console.log(gyumolcsok); // ["cseresznye", "banán", "alma"]
```

ÉS (&&) és VAGY (||) Operátorok Működése és a Rövidzár

JavaScriptben az ÉS (&&) és a VAGY (||) operátorok logikai műveletek végrehajtására szolgálnak. Ezek az operátorok az úgynevezett "rövidzáras kiértékelést" (short-circuit evaluation) használják, ami azt jelenti, hogy a kifejezések kiértékelése során a lehető legkevesebb műveletet végzik el, és a kifejezés értékelését megállítják, amint a végeredmény egyértelművé válik.

ÉS Operátor (&&)

Az ÉS operátor két kifejezés közötti logikai ÉS kapcsolatot valósít meg. Csak akkor ad vissza truthy értéket, ha mindkét operandus igaz lenne logikai (boolean) értékké konvertálva.

Működése:

1. Kiértékeli az első operandust.
2. Ha az első operandus falsy, akkor az első operandussal tér vissza, és a második operandus kiértékelése nem történik meg (rövidzár).
3. Ha az első operandus truthy, akkor kiértékeli a második operandust, és annak értékét adja vissza.
4. Ha egyik operandus sem lett truthy akkor az utolsó operandust adja vissza

Példák:

```
1 && 1; // 1
1 && 0; // 0
null && "szöveg"; // null
undefined && null; // undefined
1 && null && undefined; // null
```

Rövidzáras Kiértékelés Példa:

```
let a = 10;
let b = 0;
a > 5 && (b = b + 1); // a > 5 igaz, tehát b = b + 1 kiértékelésre kerül, b értéke 1 lesz

let c = 0;
a < 5 && (c = c + 1); // a < 5 hamis, tehát c = c + 1 nem kerül kiértékelésre, c értéke marad 0
```

VAGY Operátor (||)

A VAGY operátor két kifejezés közötti logikai VAGY kapcsolatot valósít meg. Csak akkor ad vissza falsy értéket, ha mindkét operandus hamis falsy.

Működése:

1. Kiértékeli az első operandust.
2. Ha az első operandus trutgh, akkor az első operandussal tér vissza, és a második operandus kiértékelése nem történik meg (rövidzár).
3. Ha az első operandus falsy, akkor kiértékeli a második operandust, és annak értékét adja vissza.
4. Ha mindegyik operandus falsy volt akkor az utolsót adja vissza

Példák:

```
1 || 2; // 1
2 || null; // 2
undefined || "szöveg"; // "szöveg"
undefined || null; // null
```

Rövidzáras Kiértékelés Példa:

```
let x = 10;
let y = 0;
x > 5 || (y = y + 1); // x > 5 igaz, tehát y = y + 1 nem kerül kiértékelésre, y értéke marad 0

let z = 0;
x < 5 || (z = z + 1); // x < 5 hamis, tehát z = z + 1 kiértékelésre kerül, z értéke 1 lesz
```

Miért és Hogyan Használjuk a Rövidzárat?

Miért használjuk?

1. **Hatékonyság**: Rövidzáras kiértékelés javítja a kód hatékonyságát, mivel felesleges műveletek kiértékelését elkerüli.
2. **Biztonság**: Bizonyos műveletek elkerülhetők, amelyek hibaüzeneteket eredményeznének, ha előzőleg nem vizsgáljuk meg őket. Például objektum tulajdonságainak ellenőrzése előtt megvizsgálhatjuk, hogy az objektum létezik-e.

Hogyan használjuk?

1. **Értékadás csak szükség esetén**:

```
let isLoggedIn = true;
isLoggedIn && showWelcomeMessage(); // A showWelcomeMessage csak akkor kerül meghívásra, ha az isLoggedIn igaz
```

2. **Biztonságos hozzáférés objektum tulajdonságokhoz**: (erről majd később beszélünk)

```
let user = null;
let userName = user && user.name; // userName csak akkor kap értéket, ha user nem null
```

3. **Alapértelmezett érték megadása**:

```
let name = providedName || "Default Name"; // Ha providedName hamis érték (null, undefined, ""), akkor a name "Default Name" le
```

ÉS (&&) és VAGY (||) Operátorok Műveleti Sorrendje JavaScriptben

A logikai operátoroknak, mint az ÉS (&&) és a VAGY (||), meghatározott műveleti sorrendjük (precedenciájuk) van JavaScriptben. Ez azt jelenti, hogy amikor egy kifejezés több logikai operátort tartalmaz, a JavaScript egy adott sorrendben értékeli ki azokat.

Műveleti sorrend (precedencia)

1. **Logikai NOT (!)**: A legmagasabb precedenciával rendelkezik, tehát először kerül kiértékelésre.
2. **Logikai ÉS (&&)**: Közepes precedenciával rendelkezik, az operátorok között az ÉS kerül előbb kiértékelésre, mint a VAGY.
3. **Logikai VAGY (||)**: A legalacsonyabb precedenciával rendelkezik a három közül.

Működés és Példák

Logikai NOT (!)

A logikai NOT operátor minden más logikai operátor előtt kerül kiértékelésre.

```
let a = true;
let b = false;

!a; // false
!b; // true
```

Logikai ÉS (&&)

Az ÉS operátor precedenciája magasabb, mint a VAGY operátoré, tehát az ÉS műveletek előbb kerülnek kiértékelésre, mint a VAGY műveletek.

```
let a = true;
let b = false;
let c = true;

(a && b) || c; // (a && b) || c
// a && b -> false (mert b false)
// false || c -> true (mert c true)
```

Logikai VAGY (||)

A VAGY operátor precedenciája alacsonyabb, mint az ÉS operátoré, tehát a VAGY műveletek a legutolsók, amelyek kiértékelésre kerülnek, ha nincs zárójel használva.

```
let a = false;
let b = true;
let c = true;

a || (b && c); // a || (b && c)
// b && c -> true (mert mindkettő true)
// a || true -> true (mert a VAGY művelet eredménye true, ha bármelyik operandus true)
```

Zárójelek használata a műveleti sorrend módosításához

Zárójeleket használhatunk a műveleti sorrend megváltoztatásához és a kifejezés olvashatóságának javításához.

Példák zárójelekkel:

```
let a = false;
let b = true;
let c = false;

(a || b) && c; // Először a (a || b) kifejezés kerül kiértékelésre
// a || b -> true (mert b true)
// true && c -> false (mert c false)

a || (b && c); // Először a (b && c) kifejezés kerül kiértékelésre
// b && c -> false (mert c false)
// a || false -> false (mert mindkettő false)
```

Összefoglalás

- **Logikai NOT (!)**: A legmagasabb precedenciával rendelkezik, tehát először kerül kiértékelésre.
- **Logikai ÉS (&&)**: Közepes precedenciával rendelkezik, és előbb kerül kiértékelésre, mint a VAGY (||).
- **Logikai VAGY (||)**: A legalacsonyabb precedenciával rendelkezik, tehát utoljára kerül kiértékelésre, ha nincs zárójel használva.

A zárójelek használatával módosíthatjuk a kifejezések kiértékelésének sorrendjét, ami lehetővé teszi a logikai műveletek sorrendjének egyértelműbbé és olvashatóbbá tételét.

Függvények JavaScript-ben

Függvény Deklarálása

A függvények olyan kódrészletek, amelyeket újra és újra felhasználhatunk a programunkban. JavaScript-ben többféleképpen deklarálhatunk függvényeket.

Hagyományos Függvény Deklaráció

Szintaxis:

```
function fuggvenyNeve(param1, param2, ...) {
    // kódrészlet
}
```

Függvény Paraméterek

A függvény paraméterek olyan változók, amelyeket a függvény hívásakor adunk át a függvénynek. Ezek a paraméterek a függvény belsejében használhatók. Akármennyi paramétert megadhatunk (akár 0-t is). A paramétereket vesszővel kell elválasztani.

Példa:

```
function welcome(nev) {
    console.log("Szia, " + nev + "!");
}

koszont("Anna"); // Szia, Anna!
```

Függvény Visszatérési Értéke

A függvények visszatérhetnek egy értékkel a `return` kulcsszó segítségével. Ha a függvény eléri a `return` kulcsszót, a végrehajtás leáll, és a megadott érték visszatér a függvény hívójához.

Példa:

```
function multiply(a, b) {
    return a * b;
}

let result = szorzas(3, 4);
console.log("Az eredmény: " + result); // Az eredmény: 12
```

Változók Scope-ja Függvényekben

A scope (hatókör) azt határozza meg, hogy a változók hol érhetők el a programban. JavaScript-ben háromféle scope létezik: globális, függvény és blokkszintű.

Globális Scope

A globális változók bárhonnan elérhetők a kódban.

Példa:

```
let globalisValtozo = "Ez egy globális változó";

function mutatGlobalisValtozo() {
    console.log(globalisValtozo);
}

mutatGlobalisValtozo(); // Ez egy globális változó
```

Függvény Scope

A függvényen belül deklarált változók csak a függvényen belül érhetők el.

Példa:

```
function mutatValtozo() {
    let lokalisValtozo = "Ez egy lokális változó";
    console.log(lokalisValtozo);
}

mutatValtozo(); // Ez egy lokális változó
// console.log(lokalisValtozo); // Hiba: lokalisValtozo nem definiált
```

Blokkszintű Scope

A blokkszintű változók (let és const) csak a blokkban érhetők el, ahol deklarálták őket, például egy if vagy for ciklusban.

Példa:

```
function tesztScope() {
    if (true) {
        let blokkszintuValtozo = "Ez egy blokkszintű változó";
        console.log(blokkszintuValtozo);
    }
    // console.log(blokkszintuValtozo); // Hiba: blokkszintuValtozo nem definiált
}

tesztScope();
```

If

```
let a = true;
if(a){
    // a () között van az if "feltétele"
    // amit ide írok csak akkor fut le ha a feltétel igaz
    // ami a {} között van az az if "belseje"
    // ebben a példában az itt lévő kódok lefutnak
}
if(false){
    // ebben a példában az itt lévő kódok nem futnak le
}
```

```
let first; // true vagy false
let second; // true vagy false

// ez mindig lefut
if(first){
    // ez akkor fut le ha a "first" igaz
    if(second){
        // ez akkor fut le ha a "first" és a "second" is igaz
        //  a belső if-be lépésnek a feltétele hogy a külsőbe is belépjen
    }
    // ez akkor fut le ha a "first" igaz
}
// ez mindig lefut

if(first && second){
    // ez akkor fut le ha a "first" és a "second" is igaz
}

if(first || second){
    // ez akkor fut le ha a "first" vagy a "second" igaz
}
```

else

Ha egy `if` elágazásnak van `else` réssze akkor valamelyik ágba mindenképpen belép

```
let first = true;
let second = false;
// ez mindig lefut
if(first){
    // ez akkor fut le ha a "first" igaz
}else{
    // ez akkor fut le ha a "first" nem igaz
}
```

```
let number = 11;
if(number % 2 == 0){
    console.log("a szám páros");
}else{
    console.log("a szám páratlan");
}
```

else if

Ha egy `if` elágazásnak van `else if` réssze akkor az `else if` ágba akkor lép be ha az előtte lévő feltételek hamisak de az `else if` feltétele igaz

```
let number = 11;
if(number % 2 == 0){
    console.log("a szám páros");
}else if (number % 2 == 1){
    console.log("a szám páratlan");
}else{
    console.log("ez nem egész szám");
}
```

Mindig csak egy ágba tudunk belépni

```
if(true){
    // ez egy ág
    // ebben az esetben ez lefut
}else if (true){
    // ez egy másik ág
    // ebben az esetben ez NEM fut le
}else if(true){
    // ez egy másik ág
    // ebben az esetben ez NEM fut le
}
else{
    // ez egy másik ág
    // ebben az esetben ez NEM fut le
}
```

Természetesen írhatunk `else` ág nélküli de `elsi if` ággal rendelkező `if` elágazást

```
let number = "virág";
if(number % 2 == 0){
    console.log("a szám páros");
}else if (number % 2 == 1){
    console.log("a szám páratlan");
}
```

JS ismétlés

Csak azokat írom ide, amiket plusszban mondtam a 05.18-i órához képest

Script tag

Helye

A `<script>` taget elhelyezhetjük az `<head>` vagy a `<body>` elemben, attól függően, mikor szeretnénk, hogy a JavaScript kód betöltődjön és fusson. Általában a `<body>` végére helyezzük, hogy biztosak legyünk benne, hogy az oldal összes eleme betöltődött, mielőtt a JavaScript kód lefut.

Külső script fájl importálása

A külső JavaScript fájlok használata lehetővé teszi a kód szervezettebb tárolását és újrahasználatát. Ebben az esetben a `<script>` tag src attribútumát használjuk, hogy hivatkozzunk a külső fájlra.

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Külső JavaScript</title>
    </head>
    <body>
        <h1>Üdvözlet!</h1>
        <script src="script.js"></script>
    </body>
</html>
```

Ha van src attribútum beállítva, akkor a script tag belseje ignorálva lesz.

```
<!DOCTYPE html>
<html lang="hu">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <title>Külső JavaScript</title>
    </head>
    <body>
        <h1>Üdvözlet!</h1>
        <script src="script.js">
            console.log("Ez nem fut le");
        </script>
    </body>
</html>
```

strict mode

A "strict mode" (szigorú mód) egy JavaScript futási mód, amely szigorúbb szabályokat vezet be a kód írására és futtatására vonatkozóan. A "strict mode" bevezetése az ECMAScript 5 (ES5) szabványban történt.

A "strict mode"-ot egy speciális kifejezéssel lehet engedélyezni: "use strict";. Ezt a kifejezést vagy a teljes szkript elején, vagy egy függvény elején kell elhelyezni. Ha a teljes szkript elején helyezzük el, akkor az egész szkript szigorú módban fog futni. Ha csak egy függvény elején helyezzük el, akkor csak az a függvény fog szigorú módban futni. (Függvényekről később tanulunk majd)

Példa:

```
"use strict";

x = 3.14; // Hiba, mert x nincs deklarálva
```

Template literals

1. Template Literals

A template literals egy újfajta szintaxis, amelyet az ECMAScript 6 (ES6) vezetett be. Lehetővé teszik, hogy egyszerűbben és olvashatóbban írjunk szövegformázott stringeket és interpoláljunk változókat a szövegbe. A template literals használatához backtick (`) karaktereket használunk a szöveg körül, nem pedig a hagyományos idézőjeleket (' vagy ").

2. Változó Interpoláció

A template literals egyik legnagyobb előnye, hogy lehetővé teszi a változók és kifejezések beillesztését a szövegbe közvetlenül a `${}` szintaxis használatával. Ebben az esetben a `${age}` kifejezés interpolálja az `age` változó értékét a stringbe.

3. Példa Használat

Tegyük fel, hogy van egy változónk, amely az életkort tárolja:

```
let age = 30;
```

Amikor a fenti template literal kódot futtatjuk:

```
let age = 30;
let message = `I'm ${age} years old`;
console.log(message);
```

Ez az alábbi stringet adja eredményül:

```
I'm 30 years old
```

Boolean konverzió

A Boolean konverzió azt jelenti, hogy egy értéket logikai típusúvá alakítunk, azaz `true` vagy `false` értékké. JavaScriptben ezt a `Boolean` konstruktor vagy a logikai negálás (`!`) kétszeri használatával érhetjük el (erről nem beszéltünk órán).

Igaz (true) értékek

A következő értékek mindig `true`-vá konvertálódnak:

1. **Nem üres stringek**:
   - Például: `"hello"`, `"false"`, `"0"`, `" "`
2. **Nem nulla számok**:
   - Például: `1`, `-1`, `3.14`, `Infinity`, `-Infinity`
3. **Logikai érték true**:
   - Például: `true`

Példák:

```
Boolean("hello"); // true
Boolean(123); // true
Boolean(true); // true
```

Hamis (false) értékek

A következő értékek mindig `false`-vá konvertálódnak:

1. **Üres string**:
   - Például: `""`
2. **Nulla szám**:
   - Például: `0`, `-0`
3. **Null érték**:
   - Például: `null`
4. **Undefined érték**:
   - Például: `undefined`
5. **NaN (Not-a-Number)**:
   - Például: `NaN`
6. **Logikai érték false**:
   - Például: `false`

Példák:

```
Boolean(""); // false
Boolean(0); // false
Boolean(null); // false
Boolean(undefined); // false
Boolean(NaN); // false
Boolean(false); // false
```

További Példák és Magyarázatok

1. **Nem üres stringek**:

- Még az olyan stringek is, amelyek látszólag hamis értékek, mint például `"false"` vagy `"0"`, true-ként konvertálódnak, mert nem üresek.

```
Boolean("false"); // true
Boolean("0"); // true
```

2. **Nem nulla számok**:

   - Bármely szám, ami nem nulla, true-ra konvertálódik.

```
Boolean(1); // true
Boolean(-100); // true
Boolean(Infinity); // true
Boolean(-Infinity); // true
```

3. **Logikai negálás használata**:

   - Kétszeri logikai negálással (`!!`) bármely értéket Boolean típusúvá konvertálhatunk.

```
!!"hello"; // true
!!0; // false
!!{}; // true
!!null; // false
```

ÉS (&&) és VAGY (||) Műveletek

JavaScriptben az ÉS (&&) és a VAGY (||) logikai műveletek lehetővé teszik, hogy több feltételt kombináljunk.

ÉS Művelet (&&)

Az ÉS művelet akkor ad vissza `true` értéket, ha mindkét operandusa `true`. Ha az egyik operandus `false`, akkor a kifejezés értéke `false` lesz.

Példák:

```
true && true; // true
true && false; // false
false && true; // false
false && false; // false

// Példák nem logikai értékekkel:
let a = 10;
let b = 20;

a > 5 && b > 15; // true, mert mindkét kifejezés igaz
a > 15 && b > 15; // false, mert az első kifejezés hamis
```

VAGY Művelet (||)

A VAGY művelet akkor ad vissza `true` értéket, ha bármelyik operandusa `true`. Ha mindkét operandus `false`, akkor a kifejezés értéke `false` lesz.

Példák:

```
true || true; // true
true || false; // true
false || true; // true
false || false; // false

// Példák nem logikai értékekkel:
let x = 10;
let y = 20;

x > 15 || y > 15; // true, mert a második kifejezés igaz
x > 15 || y > 25; // false, mert mindkét kifejezés hamis
```

Ciklusok Magyarázata JavaScriptben

A ciklusok lehetővé teszik, hogy egy kódrészletet többször végrehajtsunk. Ez különösen hasznos ismétlődő feladatoknál, például egy tömb összes elemének feldolgozásánál. JavaScriptben többféle ciklus létezik:

1. `for` ciklus
2. `while` ciklus
3. `do...while` ciklus

1. `for` ciklus

A `for` ciklus a legismertebb és leggyakrabban használt ciklus. Három részből áll: inicializálás, feltétel és iterátor.

Szintaxis:

```
for (inicializálás; feltétel; iterátor) {
    // Kódblokk, amely végrehajtásra kerül minden iterációban
}
```

Példa:

```
for (let i = 0; i < 5; i++) {
    console.log(i); // Kiírja: 0, 1, 2, 3, 4
}
```

2. `while` ciklus

A `while` ciklus addig ismétli a kódrészletet, amíg a feltétel igaz (`true`).

Szintaxis:

```
while (feltétel) {
    // Kódblokk, amely végrehajtásra kerül minden iterációban, amíg a feltétel igaz
}
```

Példa:

```
let i = 0;
while (i < 5) {
    console.log(i); // Kiírja: 0, 1, 2, 3, 4
    i++;
}
```

3. `do...while` ciklus

A `do...while` ciklus hasonló a `while` ciklushoz, de ebben az esetben a kódblokk legalább egyszer végrehajtásra kerül, mielőtt a feltétel ellenőrzésre kerül.

Szintaxis:

```
do {
    // Kódblokk, amely végrehajtásra kerül
} while (feltétel);
```

Példa:

```
let i = 0;
do {
    console.log(i); // Kiírja: 0, 1, 2, 3, 4
    i++;
} while (i < 5);
```

Objektumok Magyarázata JavaScriptben

Az objektumok az egyik legfontosabb és leggyakrabban használt adatstruktúrák JavaScriptben. Az objektumok lehetővé teszik, hogy összefüggő adatokat egyetlen egységben tároljunk. Ezek az adatok kulcs-érték párokként kerülnek tárolásra, ahol a kulcs egy egyedi azonosító, az érték pedig bármilyen típusú adat lehet.

Objektum Létrehozása

Objektumliterálok

Az objektumokat leggyakrabban objektumliterálok segítségével hozzuk létre. Az objektumliterálokat kapcsos zárójelek (`{}`) határolják, és a kulcs-érték párok vesszővel vannak elválasztva.

Példa:

```
let szemely = {
    nev: "János",
    kor: 30,
    foglalkozas: "programozó",
};
```

Ebben a példában a `szemely` objektum három tulajdonságot (kulcs-érték párt) tartalmaz: `nev`, `kor` és `foglalkozas`.

Objektum Tulajdonságainak Hozzáférése

Pont Notáció

A pont notációval hozzáférhetünk az objektum tulajdonságaihoz.

Példa:

```
console.log(szemely.nev); // Kiírja: "János"
console.log(szemely.kor); // Kiírja: 30
console.log(szemely.foglalkozas); // Kiírja: "programozó"
```

Kapcsos Zárójel Notáció

A kapcsos zárójel notációval is hozzáférhetünk az objektum tulajdonságaihoz. Ez különösen hasznos, ha a tulajdonság neve dinamikusan kerül meghatározásra.

Példa:

```
console.log(szemely["nev"]); // Kiírja: "János"
console.log(szemely["kor"]); // Kiírja: 30
console.log(szemely["foglalkozas"]); // Kiírja: "programozó"
```

Objektum Tulajdonságainak Módosítása

Az objektum tulajdonságainak értékét megváltoztathatjuk a pont vagy a kapcsos zárójel notáció használatával.

Példa:

```
szemely.kor = 31; // Módosítja a 'kor' tulajdonságot
szemely["foglalkozas"] = "vezető programozó"; // Módosítja a 'foglalkozas' tulajdonságot

console.log(szemely); // Kiírja: { nev: 'János', kor: 31, foglalkozas: 'vezető programozó' }
```

Új Tulajdonság Hozzáadása

Új tulajdonságot is hozzáadhatunk egy objektumhoz.

Példa:

```
szemely.lakcim = "Budapest";
console.log(szemely); // Kiírja: { nev: 'János', kor: 31, foglalkozas: 'vezető programozó', lakcim: 'Budapest' }
```

Tulajdonság Eltávolítása

Egy tulajdonságot eltávolíthatunk a `delete` operátor segítségével.

Példa:

```
delete szemely.lakcim;
console.log(szemely); // Kiírja: { nev: 'János', kor: 31, foglalkozas: 'vezető programozó' }
```

Tömbök Magyarázata JavaScriptben

A tömbök (arrays) alapvető adatstruktúrák JavaScriptben, amelyek lehetővé teszik, hogy több értéket tároljunk egyetlen változóban. A tömbök különösen hasznosak, amikor egy halom hasonló adatot szeretnénk kezelni és rendszerezni.

Tömbök Létrehozása

Tömbök létrehozására többféle mód van JavaScriptben. A leggyakoribb módszer az, amikor a tömböt szögletes zárójelek (`[]`) segítségével hozzuk létre, és az elemeket vesszővel választjuk el egymástól.

Példák:

```
let uresTomb = []; // Üres tömb
let szamok = [1, 2, 3, 4, 5]; // Számokat tartalmazó tömb
let szovegek = ["alma", "banán", "cseresznye"]; // Sztringeket tartalmazó tömb
let vegyes = [1, "alma", true, null]; // Vegyes típusú elemeket tartalmazó tömb
```

Tömbök Hozzáférése és Módosítása

A tömb elemeihez indexeléssel férhetünk hozzá. Az indexelés 0-tól kezdődik, tehát az első elem indexe 0, a másodiké 1, és így tovább.

Példák:

```
let gyumolcsok = ["alma", "banán", "cseresznye"];
console.log(gyumolcsok[0]); // Kiírja: "alma"
console.log(gyumolcsok[1]); // Kiírja: "banán"
console.log(gyumolcsok[2]); // Kiírja: "cseresznye"

gyumolcsok[1] = "körte"; // A második elem módosítása
console.log(gyumolcsok); // Kiírja: ["alma", "körte", "cseresznye"]
```

Tömbök Hossza

A tömb hosszát a `length` tulajdonsággal érhetjük el, amely megadja, hogy hány elem található a tömbben.

Példa:

```
let szamok = [1, 2, 3, 4, 5];
console.log(szamok.length); // Kiírja: 5
```

Tömb Műveletek

JavaScriptben számos beépített metódus áll rendelkezésre a tömbökkel való munkához. Néhány alapvető művelet:

- **Elem hozzáadása a végéhez**: `push`

  ```
  let szamok = [1, 2, 3];
  szamok.push(4); // Hozzáadja a 4-et a tömb végéhez
  console.log(szamok); // Kiírja: [1, 2, 3, 4]
  ```

- **Elem eltávolítása a végéről**: `pop`

  ```
  let szamok = [1, 2, 3];
  let utolsoElem = szamok.pop(); // Eltávolítja az utolsó elemet és visszaadja azt
  console.log(utolsoElem); // Kiírja: 3
  console.log(szamok); // Kiírja: [1, 2]
  ```

- **Elem hozzáadása az elejéhez**: `unshift`

  ```
  let szamok = [1, 2, 3];
  szamok.unshift(0); // Hozzáadja a 0-t a tömb elejéhez
  console.log(szamok); // Kiírja: [0, 1, 2, 3]
  ```

- **Elem eltávolítása az elejéről**: `shift`

  ```
  let szamok = [1, 2, 3];
  let elsoElem = szamok.shift(); // Eltávolítja az első elemet és visszaadja azt
  console.log(elsoElem); // Kiírja: 1
  console.log(szamok); // Kiírja: [2, 3]
  ```

Operátorok

Logikai Operátorok

A logikai operátorok egy logikai értéket adnak vissza azaz `true` vagy `false`

< és > Operátor

```
2 > 1; // érték vizsgálat (olvasuk 2 nagyobb-e 1)
//eredménye true
2 < 1; // érték vizsgálat (olvasuk 2 kisebb-e 1)
//eredménye false
```

<= és >= Operátor

```
2 >= 1; // érték vizsgálat (olvasuk 2 nagyobb vagy egyenlő 1)
//eredménye true
2 <= 1; // érték vizsgálat (olvasuk 2 kisebb vagy egyenlő 1)
//eredménye false
```

Egyenlőség vizsgálat == megengedő

```
let a;
a = 1; // érték adás (olvasuk a legyen egyenlő 1)
2 == 1; // érték vizsgálat (olvasuk 2 egyenlő-e 1)
// értéke false
```

Egyenlőség vizsgálat === szigorú

```
"1" == 1; // értéke true
"1" === 1; // értéke false mert vizsgálja a tipusát is
```

Negáció

```
1 == 1; // értéke true
1 != 1; // értéke false
1 !== 1; // értéke false
let boolean = !(10 > 1); // értéke false
!true; // értéke false
```

`&&` logikai `és`

```
true && false; // false
false && true; // false
false && false; // false
true && true; // true
```

`||` logikai `vagy`

```
true || false; // true
false || true; // true
true || true; // true
false || false; // false
```

Ezeket lehet `"kombózni"` is

```
!(false || true && false); // true
```

% Operátor

```
11 % 2; // visza adja az osztás utáni maradékot
// értéke 1
```

```
11 % 2 == 0; // megvizsgálja hogy a 11 osztható e 2-vel
// false
12 % 3 == 0; // megvizsgálja hogy a 12 osztható e 3-vel
// true
```

+=, -=, /=, %=, Operátor

```
let a = 0;
// a kövi 2 sor egy és ugyanaz
a = a + 3;
a += 3;

// a kövi 2 sor egy és ugyanaz
a = a - 4;
a -= 4;

// a kövi 2 sor egy és ugyanaz
a = a / 2;
a /= 2;

// a kövi 2 sor egy és ugyanaz
a = a % 3;
a %= 3;
```

++ Operátor

```
let a = 0;
// a kövi 3 sor egy és ugyanaz
a = a + 1;
a += 1;
a++;
```

-- Operátor

```
let a = 0;
// a kövi 3 sor egy és ugyanaz
a = a - 1;
a -= 1;
a--;
```

Logikai értékkel visszatérő metódus

isNan()

```
let a = Number("number");
console.log(a); // NaN
console.log(isNaN(a)) //true
console.log(isNaN(2)) //false
```

confirm()

Ez egy felugró ablak ami logikai értéket add vissza

```
const okay = confirm("Nekem adod a veséd?");
console.log(okay);
```

Többsoros Stringek és a `\n` Karakter

JavaScript-ben többsoros stringek létrehozásához használhatjuk a backtick (`) karaktereket, amelyeket sablon stringeknek is nevezünk. Ez lehetővé teszi, hogy egy string több sorban legyen, anélkül, hogy speciális karaktereket kellene használnunk. Azonban használhatjuk a `\n` karaktert is, amely új sort jelent a stringben.

Példa backtick karakterekkel:

```
let tobbSorosString = `Ez egy többsoros
string, amely
több sorban van.`;
console.log(tobbSorosString);
```

Példa `\n` karakterekkel:

```
let tobbSorosString = "Ez egy többsoros\nstring, amely\ntöbb sorban van.";
console.log(tobbSorosString);
```

Mindkét példa ugyanazt az eredményt adja, de a backtick karakterek használata általában olvashatóbb és karbantarthatóbb.

Stringek Indexelése

JavaScript-ben a stringek indexelése 0-tól kezdődik. Az egyes karakterekhez az indexükkel férhetünk hozzá.

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg[0]); // "J"
console.log(szoveg[4]); // "S"
```

`toUpperCase` és `toLowerCase` Metódusok

A `toUpperCase` metódus egy stringet nagybetűssé alakít, a `toLowerCase` metódus pedig kisbetűssé.

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.toUpperCase()); // "JAVASCRIPT"
console.log(szoveg.toLowerCase()); // "javascript"
```

`indexOf` Metódus

Az `indexOf` metódus megkeresi egy adott alstring első előfordulását a stringben, és visszaadja annak indexét. Ha az alstring nem található, `-1`-et ad vissza.

Szintaxis:

```
string.indexOf(searchValue, fromIndex);
```

- `searchValue`: A keresendő alstring.
- `fromIndex`: Az index, amelytől kezdve a keresést elindítjuk (opcionális).

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.indexOf("Script")); // 4
console.log(szoveg.indexOf("Java")); // 0
console.log(szoveg.indexOf("Python")); // -1
```

`startsWith` Metódus

A `startsWith` metódus ellenőrzi, hogy egy string egy adott alstringgel kezdődik-e, és `true` vagy `false` értéket ad vissza.

Szintaxis:

```
string.startsWith(searchString, position);
```

- `searchString`: A keresendő alstring.
- `position`: Az index, amelytől kezdve a keresést elindítjuk (opcionális).

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.startsWith("Java")); // true
console.log(szoveg.startsWith("Script")); // false
console.log(szoveg.startsWith("Script", 4)); // true
```

`endsWith` Metódus

Az `endsWith` metódus ellenőrzi, hogy egy string egy adott alstringgel végződik-e, és `true` vagy `false` értéket ad vissza.

Szintaxis:

```
string.endsWith(searchString, length);
```

- `searchString`: A keresendő alstring.
- `length`: A string hossza, amelyig a keresést elvégezzük (opcionális).

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.endsWith("Script")); // true
console.log(szoveg.endsWith("Java")); // false
console.log(szoveg.endsWith("Java", 4)); // true
```

`slice` Metódus

A `slice` metódus egy új stringet hoz létre az eredeti string egy részéből, amely a megadott kezdő és záró index közötti karaktereket tartalmazza.

Szintaxis:

```
string.slice(start, end);
```

- `start`: A kezdő index (beleértve).
- `end`: A záró index (kizárva).

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.slice(0, 4)); // "Java"
console.log(szoveg.slice(4)); // "Script"
```

`substring` Metódus

A `substring` metódus egy új stringet hoz létre az eredeti string egy részéből, amely a megadott kezdő és záró index közötti karaktereket tartalmazza.

Szintaxis:

```
string.substring(start, end);
```

- `start`: A kezdő index (beleértve).
- `end`: A záró index (kizárva).

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.substring(0, 4)); // "Java"
console.log(szoveg.substring(4)); // "Script"
```

`substr` Metódus

A `substr` metódus egy új stringet hoz létre az eredeti string egy részéből, amely a megadott kezdő indexnél kezdődik, és a megadott hosszúságú.

Szintaxis:

```
string.substr(start, length);
```

- `start`: A kezdő index (beleértve).
- `length`: A kivágott rész hosszúsága.

Példa:

```
let szoveg = "JavaScript";
console.log(szoveg.substr(0, 4)); // "Java"
console.log(szoveg.substr(4, 6)); // "Script"
```

Stringek Összehasonlítása

A stringek összehasonlítására a ==, ===, < és > operátorokat használhatjuk. Az egyenlőség operátorok a stringek értékét hasonlítják össze, míg a < és > operátorok lexikografikus (ábécé) sorrendben hasonlítják össze őket.

Példa:

```
let szoveg1 = "apple";
let szoveg2 = "banana";
let szoveg3 = "apple";

console.log(szoveg1 == szoveg2); // false
console.log(szoveg1 === szoveg3); // true
console.log(szoveg1 < szoveg2); // true (mert "apple" előbb jön, mint "banana")
console.log(szoveg2 > szoveg3); // true (mert "banana" később jön, mint "apple")
```

`toLocaleCompare` Metódus

A `toLocaleCompare` metódus lehetővé teszi két string lokális összehasonlítását, tehát az ékezetes betűket helyesen kezelve és háromféle értéket ad vissza:

- `-1` ha az első string kisebb
- `1` ha az első string nagyobb
- `0` ha a két string egyenlő

Szintaxis:

```
string1.toLocaleCompare(string2);
```

Példa:

```
let szoveg1 = "apple";
let szoveg2 = "banana";
let szoveg3 = "apple";

console.log(szoveg1.toLocaleCompare(szoveg2)); // -1
console.log(szoveg2.toLocaleCompare(szoveg1)); // 1
console.log(szoveg1.toLocaleCompare(szoveg3)); // 0
```

JS

A kód lehetséges helye egy html fájlban és az első kódunk:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script>
        console.log('Hello Word!')
    </script>
</head>
<body>

</body>
</html>
```

Kiiratás és alert

Kiiratás konzolra

Szöveg más néven string kiiratása:

```
console.log('Hello')
vagy
console.log("Hello")
```

Szám kiiratása:

```
console.log(4)
vagy
console.log(10.4)
```

Logikai érték kiiratása:

```
console.log(true)
```

Alert

Hasonlóképen működik mint a console.log() csak vár egy megerősítést hogy oké vettem

```
alert(true)
```

Komment

```
console.log(true) //inenn kezdve van kommentelve a sor eleje lefut
console.log(true)
// console.log(true) ez a sor komentelve van
console.log(true)
/* innen
ezen
keresztül
komentelve van idáig */
console.log(true)
```

Szövegek összefűzése azaz konkatenáció és sorendiség

```
alert('Hello' + ' ' +'Word') //eredmény: Hello Word
alert(5 + 5 + 'Hello') //eredmény: 10Hello
alert('Hello' + 5 + 5) //eredmény: Hello55
alert('Hello' + (5 + 5)) //eredmény: Hello10
```

ezeknek az eredménye egy string

Tört számok és felsorolás

```
console.log(3.14) //ez tört szám
console.log(3,14) //ez két érték a 3 és a 14
console.log('Hello', 5 + 5) //ennek az eredménye 2 érték Hello szöveg és 10 szám
```

Sorrendiség

a sorrendiség alapvetően balról jobbra van ezt befolyásolhatjuk `()` -kel

```
console.log("Andris vagyok, én is " + 20 + 6 + " éves") // eredmény: "Andris vagyok, én is 206 éves"
console.log("Andris vagyok, én is " + (20 + 6) + " éves") // eredmény: "Andris vagyok, én is 26 éves"
console.log("Andris vagyok, én is", 20 + 6, "éves") // eredmény: "Andris vagyok, én is 26 éves"
```

- Az első pédában amit össze rakunk az egy `string + szám + szám + string`
  mivel egy `string + szám = string` így csak össze fűzi
  - `"Andris vagyok, én is " + 20 = "Andris vagyok, én is 20"`
  - `"Andris vagyok, én is 20" + 6 = "Andris vagyok, én is 206"`
  - `"Andris vagyok, én is 206" + " éves" = "Andris vagyok, én is 206 éves"`
- A második pédában `()` segítségével meg változtatjuk a sorrendiséget
  így először a két számot adjuk össze aminek az eredménye szám

    - `20 + 6 = 26`
    - `"Andris vagyok, én is " + 26 = "Andris vagyok, én is 26"`
    - `"Andris vagyok, én is 26" + " éves" = "Andris vagyok, én is 26 éves"`
- A harmadik pédában nem fűzük össze a szöveget így a szám szám ként viselkedik
  és elvégződik az össze adás
    - `20 + 6 = 26`
    - az eredményünk egy `string` egy `szám` és egy `string`

Változók

Adat tárolására alkalmas.

```
var firstVar; //változó létrehozása azaz deklarálása
firstVar = 1; //változó érték adása azaz inicalizálása
var otherVar = 3; // van lehetőségünk egy sorban/utasításban deklarálni és inicalizálni is
console.log(otherVar) //kiiratódik a 3 mivel a változó értékét iratjuk ki
```

3 féle lehetőségünk van rá js -ben

```
var oldOne;
let bestOne; //ezt használjuk
const notBadButWeDontUse;
```

A változó neve mint az előző pédában is látszik kisbetűvel kezdem majd minden új szó esetén nagy betűt használok egybeírva és angol főneveket. Ez az úgynevezet CamelCase.

```
//HIBÁS KÓD
let bestOne;
let bestOne;
console.log(bestOne);
```

- A változó neveinek egyedinek kell lennie.
- Számmal nem kezdődhet van pár megengedett karakter pl.: $ de kerüljük kezdjük latinbetűvel

Változók viselkedése

```
let myVar = 3;
console.log(myVar);
myVar = "Töltöttkáposzta";
alert(myVar);
//a konzolra iratáskor a 3 iratódik ki, de az alert már az Töltöttkáposzta
//ugyanaz a változó tárolhat számot stringet vagy logikai értéket is
```

Változókkal való műveletek

Mikor hivatkozok a változómra a benne tárolt érték ként viselkedik

```
let myFirstNumber = 3;
let mySecondNumber = 8;
console.log(myFirstNumber + mySecondNumber);
// ez eredményben ugyan az mint a
console.log(3 + 8);
let lastNumber = myFirstNumber + mySecondNumber;
// it a last number értéke 3 + 8 azaz 11 lesz
```

Undefined & NaN

```
let myUndefined;
console.log(myUndefined); // nincs értéke a változómnak (undefined)
console.log(myUndefined + 5) // NaN (Not a Number)
console.log(myUndefined + "Móka") // undefinedMóka
```

Adat bekérése

Adat bekérésére a `prompt` -ot használjuk ami szöveget olvas be

```
let firstNumber = prompt("Adj meg egy számot", 5); //megadjuk az 5-öt
let secondNumber = prompt("Adj meg egy másik", 8); //megadjuk az 8-at
console.log(firstNumber + firstNumber); //az eredmény 58
```

String át alakítása számmá a `Number()` segítségével

```
let firstNumber = prompt("Adj meg egy számot"); //megadjuk az 5-öt
let secondNumber = prompt("Adj meg egy másik"); //megadjuk az 8-at
console.log(Number(firstNumber) + Number(secondNumber)); //az eredmény 13
```

```
let number = "10";
console.log(Number(number)); //az eredmény NaN
```