Tartalom

TypeScript elem típusának megadása

Generikus típusmegadás (*generic type parameter*)

```
const inputElem = document.querySelector<HTMLInputElement>('#szamInput');
```

- Ez a **biztonságosabb és ajánlottabb** módszer.
- A querySelector metódus TypeScript generikus függvényként van deklarálva, és a **<HTMLInputElement>** rész azt jelenti, hogy **te explicit megadod, milyen típusú elemet vársz**.
- Ha nem a megfelelő típus jön vissza (pl. nem egy <input>), a szerkesztő (VS Code, WebStorm stb.) **már fordításkor jelezheti a problémát**.

Ismertebb neve: Generikus típusparaméter (generic type annotation)

Típus-kényszerítés (*type assertion / casting*)

```
const inputElem = document.querySelector("#szamInput") as HTMLInputElement;
```

- Ezzel azt **mondod a fordítónak**, hogy *„bízom benne, ez biztosan **HTMLInputElement**”*, akkor is, ha ezt nem tudja garantálni.
- Nem ellenőrzi, hogy valóban **<input>** típusú elemről van-e szó.
- Ha nem az, akkor **futásidőben** hibát okozhat (pl. **value** tulajdonság nem létezik azon az elemen).

Ismertebb neve: Típus assertálás (*type assertion* vagy *cast*)

Összefoglalva

Módszer	Típusmegadás neve	Biztonság	Mikor használd
<HTMLInputElement>	Generikus típusparaméter	**Biztonságosabb**	Ha a típus *valószínűsíthetően stimmel*
as HTMLInputElement	Típusassertálás (cast)	**Kockázatosabb**	Ha **biztos vagy benne**, hogy az elem típusát nem kell ellenőrizni

Tipp

Használj querySelector<>()-t, ahol lehet, mert:

- IntelliSense működik
- Statikus típusellenőrzés már a szerkesztőben segít

- Kevesebb futásidőbeli hiba

Általánosabb típusok a DOM-ban

Típusnév	Mire használjuk?
HTMLElement	Bármely HTML elem
Element	Bármely DOM elem (beleértve SVG-t is)
Node	Bármilyen DOM-csomópont (elem, szöveg, komment stb.)
EventTarget	Minden, amire esemény figyelőt lehet rakni

Miért ne használjunk általános típust, ha nem muszáj?

```
//Általános típus megadás:
const input = document.querySelector('#email') as HTMLElement;
console.log(input.value); //Property 'value' does not exist on type
'HTMLElement'
```

```
//Specifikus típus megadás:
const input = document.querySelector<HTMLInputElement>('#email');
console.log(input?.value); // Típusosan működik, IntelliSense is
segít
```

Mert az általános típus **nem ismeri az elem speciális tulajdonságait**, így:

- Nem tudsz közvetlenül hozzáférni pl. .value, .checked, .files, stb.
- A TypeScript **fordítási hibát dob** vagy kényszerít, hogy előbb castolj (as HTMLInputElement)
- Elveszíted az IDE (pl. VSCode) intelligens kiegészítéseit (IntelliSense)
- Később hibához vezethet, ha rosszul feltételezed az elérhető tulajdonságokat

Miért érdemes specifikus típust megadni?

Előny	Részletezés
Típusbiztonság	A TypeScript már fordítási időben ellenőrzi, hogy léteznek-e a használt mezők/metódusok
IntelliSense támogatás	Az IDE javaslatokat ad (pl. value, checked, selectedIndex)
Hibák megelőzése	Elkerülheted a futásidőbeli undefined.property hibákat
Jobban olvasható kód	Egyértelmű lesz másnak (és később neked is), milyen típussal dolgozol
Jobb fejlesztői élmény	Könnyebb tesztelni, refaktorálni és karbantartani a kódot

A specifikus típusok használata TypeScriptben biztosítja, hogy a DOM-elemekhez tartozó tulajdonságok (mint value, checked, stb.) elérhetőek és típusbiztosak legyenek.

Ha általános típust használunk, ezekhez a mezőkhöz nem férünk hozzá közvetlenül, vagy külön castolni kell.

A pontos típusmegadás ezért növeli a fejlesztési biztonságot, segíti a hibakeresést, és gyorsabbá teszi a munkát a fejlesztői környezetben.

Tipp

Ha nem vagy biztos egy elem típusában, használhatod az általános HTMLElement típust, de **ha lehet, mindig pontos típusra castolj** (pl. HTMLInputElement), hogy elérd a típus-specifikus mezőket, mint pl. .value, .checked, stb.

Legfontosabb HTML elemtípusok TypeScript-ben

HTML elem	TypeScript típusa
<input>	HTMLInputElement
<form>	HTMLFormElement
<select>	HTMLSelectElement
<textarea>	HTMLTextAreaElement
<button>	HTMLButtonElement
<a> (link)	HTMLAnchorElement
<img>	HTMLImageElement
<video>	HTMLVideoElement
<audio>	HTMLAudioElement
<label>	HTMLLabelElement
<table>	HTMLTableElement
<tr>	HTMLTableRowElement
<td> / <th>	HTMLTableCellElement
** **	HTMLDivElement
****	HTMLSpanElement
<canvas>	HTMLCanvasElement
** / **	HTMLUListElement / HTMLOListElement
** **	HTMLLIElement

További fontos, de ritkábban használt HTML típusok

HTML elem	TypeScript típusa
<fieldset>	HTMLFieldSetElement
<progress>	HTMLProgressElement
<meter>	HTMLMeterElement
<details>	HTMLDetailsElement
<summary>	HTMLElement (nincs külön típus)

<iframe>	HTMLIFrameElement
<script>	HTMLScriptElement
<style>	HTMLStyleElement
<link> (pl. CSS)	HTMLLinkElement
<source>	HTMLSourceElement
<track>	HTMLTrackElement
<map>	HTMLMapElement
<area>	HTMLAreaElement
<object>	HTMLObjectElement
<embed>	HTMLEmbedElement

Elemek létezésének ellenőrzése (? és a ! elemek)

elem_neve? → Optional chaining (opcionális láncolás)

```
gomb?.addEventListener('click', () => { ... });
```

Mit jelent?

- Csak akkor hívja meg az addEventListener-t, **ha a gomb nem null vagy undefined**.
- Ha a gomb nem létezik, akkor sem dob hibát – egyszerűen kihagyja a műveletet.

Biztonságos
Nem garantálja, hogy az eseménykezelő mindig hozzárendelődik.

elem_neve! → Non-null assertion operator (nem-null állítás)

```
gomb!.addEventListener('click', () => { ... });
```

Mit jelent?

- Azt **állítod a TypeScript felé**, hogy *„tudom, hogy ez nem null vagy undefined, bízz bennem!”*
- Nem ad fordítási hibát akkor sem, ha querySelector elvileg null-t is visszaadhat.

Futásidőben hibát dob, ha a gomb mégis null.
Akkor használd, ha *garantált*, hogy az elem mindig ott lesz (pl. statikus HTML esetén).

if (elem) / if (!elem) – Manuális null-ellenőrzés

```
if (!gomb) {
    console.error("A futtató gomb hiányzik a projektből");
    return;
}
```

Mit jelent?

- **Legbiztonságosabb forma**: a kód csak akkor fut tovább, ha az elem valóban létezik.
- Átlátható, jól tanítható, hiba megelőzésre ideális.
- Alkalmas hibakezelésre (console.warn, console.error return stb.).

A return ilyenkor megszakítja a további kód futását, így:

1. **Nem próbálod elérni a gomb elem tulajdonságait**
2. **Elkerülöd a futásidőbeni hibát**
3. **Jelezed, hogy itt probléma van, a program nem folytatható ebben az állapotban**

Mikor használd?

- Olyan kódrészlet elején, ahol a DOM-elemek létfontosságúak
- Mielőtt egy eseményt, tulajdonságot vagy értéket kezelnél
- Hibakezelés mellé (console.error, console.warn, stb.)

Kifejezés	Jelentés	Mikor használd?
gomb?	Ha van, akkor használd	Ha az elem opcionális vagy lehet, hogy nem létezik
gomb!	Biztos, hogy van → használd	Ha 100% biztos vagy benne, hogy az elem a DOM-ban van
if (gomb)	Manuális ellenőrzés	Legbiztonságosabb, teljes kontrollt ad

Események kezelése TypeScript-ben

Események típusának megadása

A DOM események (events) kezelése TypeScriptben ugyanolyan fontos, mint JavaScriptben, **de itt pontos típusokat is rendelhetünk hozzájuk** – így már fordításkor kiderülhet, ha valami hibás.

```
const gomb = document.querySelector<HTMLButtonElement>('#gomb');

gomb?.addEventListener('click', (event: MouseEvent) => {
    console.log('Rákattintottál!');
});
```

Esemény létrehozás példa Arrow Function nélkül

Lehetőségünk van természetesen arrow function nélkül is eseményt létrehozni, ebben az esetben azt több elemhez is hozzárendelhetjük ahogy azt korábban már addEventListener esetén is megtettük.

```
const gomb = document.querySelector<HTMLButtonElement>('#kuldesGomb');

function kattintasKezelo(event: MouseEvent): void {
    alert('Kattintottál a gombra!');
}

// esemény regisztrálása
gomb?.addEventListener('click', kattintasKezelo);
```

Paraméter és Event típus együttes használata:

Amennyiben az eventet is szeretnénk jelölni és persze paramétert is adni, ami ugye addEventListenernél nem igazán lehetséges, akkor a korábban JS DOM-ban már használt arrow functionos megoldással lehetséges.

```
function gombKezelo(event: MouseEvent, uzenet: string): void {
    console.log("Esemény típusa:", event.type);
    alert(uzenet);
}

const gomb = document.querySelector<HTMLButtonElement>('#kuldesGomb');

// Burkoló arrow function, ami átadja az eseményt és egy plusz paramétert
gomb?.addEventListener('click', (event) => gombKezelo(event, "Sikeres
kattintás!"));
```

Események típusai

1. **Egérrel kapcsolatos események** (MouseEvent)

- click – kattintás
- dblclick – dupla kattintás
- mousedown – egérgomb lenyomása
- mouseup – egérgomb felengedése
- mousemove – egér mozgatása
- mouseover – egér rámegy egy elemre
- mouseout – egér elhagyja az elemet
- contextmenu – jobb klikk (helyi menü)

---

2. **Billentyűzettel kapcsolatos események** (KeyboardEvent)

- keydown – billentyű lenyomása
- keyup – billentyű felengedése
- keypress – (elavult, inkább ne használd)

---

3. **Űrlap események** (SubmitEvent vagy Event)

- submit – űrlap elküldése
- reset – űrlap visszaállítása
- change – beviteli elem értékének módosítása
- input – beírás input mezőbe (valós időben)

---

4. **Fókusz események** (FocusEvent)

- focus – elem fókuszt kap (pl. belekattintanak inputba)
- blur – elem fókuszt veszít

---

5. **UI események** (UIEvent, Event)

- resize – ablakméret változik
- scroll – elem vagy oldal görgetése
- load – oldal vagy elem betöltődött
- unload – oldal elhagyása

---

6. **Húzás-eldobás** (drag and drop) (DragEvent)

- dragstart – húzás megkezdése
- drag – húzás közben
- dragenter – elem fölé húzás
- dragover – elem fölött marad
- dragleave – kilép az elemről
- drop – ledobás történik
- dragend – húzás vége

---

7. **Érintőképernyős események** (TouchEvent)

- touchstart – ujj érintése az elemhez
- touchmove – ujj mozgatása az elem fölött

- touchend – érintés vége
- touchcancel – érintés megszakad

---

8. **Egyéb események**

- animationstart, animationend, animationiteration – animációs események
- transitionend – CSS átmenet vége
- visibilitychange – ha az oldal láthatósága változik (pl. tab váltás)
- pointerdown, pointerup, pointermove – egységes pointer-események egérre, tollra, érintésre

Előnyök, ha jelöljük az esemény típusát

Előny	Miért számít?
Autocomplete	Az IDE (pl. VS Code) javasolja, milyen tulajdonságokat érhetsz el (event.key, event.clientX, stb.)
Hibamegelőzés	TS már fordításkor szól, ha pl. olyan mezőt próbálsz elérni, ami nem létezik (event.value nem jó)
Könnyebb tanulás/fejlesztés	Látszik, milyen típusú esemény érkezik, mit várhatsz el
Kód átláthatósága	Később visszatérve vagy más fejlesztőnek is egyértelmű, milyen eseménnyel dolgozik a handler
Fordítási biztonság	A típusok segítenek elkerülni a futásidőbeli hibákat (pl. undefined.property)

Mi történik, ha nem jelöljük?

- TypeScript automatikusan az event típusát **any vagy Event**-ként kezeli.
- Ez **kikapcsolja a típusellenőrzést** → bármire hivatkozhatsz benne, akkor is ha hibás.
- **Futásidőben** jöhet a hiba (pl. event.clientX is undefined).
- Az event.target típusa csak EventTarget, amin nincs .value, .files, stb.

Mire jó az event.preventDefault() metódus?

Amikor egy űrlapot (<form>) a felhasználó elküld (pl. egy **submit** gomb megnyomásával), a böngésző **alapértelmezett viselkedése**, hogy:

- újra tölti az oldalt, vagy
- az űrlap action attribútuma alapján továbbítja az adatokat egy másik oldalra.

Ez viszont **megakadályozza**, hogy az adatokat előbb JavaScript/TypeScript kóddal **ellenőrizzük, validáljuk vagy feldolgozzuk** anélkül, hogy az oldal újra töltődne.

Hogyan használjuk?

Használjuk az event.preventDefault() metódust az eseménykezelőben, hogy letiltsuk az alapértelmezett működést, és teljesen kézben tarthassuk a logikát.

```
const form =
document.querySelector<HTMLFormElement>('#kapcsolatForm');

form?.addEventListener('submit', (event: SubmitEvent) => {
    event.preventDefault(); // Megakadályozza az oldal újratöltését

    const nevInput =
document.querySelector<HTMLInputElement>('#nev');
    const emailInput =
document.querySelector<HTMLInputElement>('#email');

    if (!nevInput?.value || !emailInput?.value) {
        alert("Kérlek, töltsd ki az összes mezőt!");
    } else {
        alert("Űrlap sikeresen elküldve!\nNév: " + nevInput.value +
"\nEmail: " + emailInput.value);
        // Itt jöhetne például egy fetch() vagy AJAX kérés
(Adatbázis műveletek)
    }
});
```

Az event.preventDefault() metódus megakadályozza az űrlap automatikus elküldését (és oldalfrissítést), így lehetővé teszi az adatok **validálását, manipulálását vagy aszinkron továbbítását** JavaScript/TypeScript segítségével.

Ez különösen fontos akkor, ha nem szeretnénk, hogy a felhasználói élmény megszakadjon vagy feleslegesen újra töltődjön az oldal.