

Webtechnológia és webalkalmazás-fejlesztés - TypeScript

TypeScript

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKNB_MSTM071

2026. március 21.

Mi a TypeScript?

- A TypeScript a JavaScript **típusos kiterjesztése**.
- Fejlesztés közben jelez hibákat, mielőtt futna a kód.
- A fordítás után sima JavaScript fut a böngészőben vagy Node.js alatt.

Hogyan gondolkodj róla kezdőként?

- A JavaScript megmondja, **hogyan** fusson a program.
- A TypeScript megmutatja, **milyen adatokkal** futhat biztonságosan.
- A típusok dokumentációként is működnek, így a következő fejlesztő gyorsabban megérti a kódot.
- A cél nem a bonyolult típusok gyártása, hanem a hibák korai kiszűrése.

Miért jó kezdőknek?

- Könnyebb megérteni, milyen adatot vár egy függvény.
- Biztonságosabb a refaktorálás.
- Jobb IDE-támogatást kapsz (autocomplete, hibajelzések).
- Kisebb az esélye a futásidejű meglepetéseknek.

Mire figyelj a tanulás elején?

- Először a JavaScript alapokat erősítsd meg: változók, függvények, objektumok, tömbök.
- Utána vezesd be a TypeScriptet kis lépésekben: előbb paraméter- és visszatérési típusokkal.
- Ne használd túl korán az any típust, mert ezzel elveszíted a TypeScript fő előnyét.

Telepítés és első projekt

```
mkdir ts-course  
cd ts-course  
npm init -y  
npm install -D typescript ts-node @types/node  
npx tsc --init
```

Egyszerű tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext",
    "strict": true,
    "rootDir": "src",
    "outDir": "dist",
    "esModuleInterop": true
  },
  "include": ["src"]
}
```

- strict: szigorúbb ellenőrzés, kevesebb rejtett hiba.
- rootDir/outDir: tisztán elválasztja a forráskódot és a fordított kódot.
- module NodeNext: modern ES-modulok Node környezetben.

Első TypeScript-kód

```
// src/index.ts
function greet(name: string): string {
  return `Hello, ${name}!`;
}
```

```
console.log(greet("TypeScript"));
```

```
npx tsc
node dist/index.js
```

- Először a tsc fordító fut le, ez ellenőrzi a típusokat.
- Ezután már JavaScript fut, vagyis a végrehajtás mindig JS-ben történik.

Primitív típusok

```
let userName: string = "Anna";  
let age: number = 20;  
let isStudent: boolean = true;  
let value: null = null;  
let missing: undefined = undefined;
```

- Ezek a legalapvetőbb építőelemek, minden összetettebb típus ezekre épül.
- A null és undefined kezelése kulcsfontosságú a stabil alkalmazásokhoz.

Típusinferencia

```
let city = "Gyor"; // string
let points = 10;    // number
const role = "admin"; // "admin" literal
```

- Nem kell mindig kiírni a típust.
- Akkor érdemes explicit típust adni, ha nő tőle az olvashatóság.
- Jó egyensúly: ahol egyértelmű, hagyd inferálni; ahol kétértelmű, írd explicit típust.

Tömbök és tuple

```
const nums: number[] = [1, 2, 3];  
const tags: Array<string> = ["ts", "js"];  
  
const point: [number, number] = [10, 20];  
const personRow: [id: number, name: string] = [1, "Bela"];
```

- Tömb: tetszőleges elemszám, azonos elemtípus.
- Tuple: fix elemszám, pozícióként meghatározott típus.

Objektumtípusok

```
const user: { id: number; name: string; active: boolean } = {  
  id: 1,  
  name: "Kata",  
  active: true,  
};
```

- Objektumoknál a típus gyorsan hosszúvá válhat, ezért a következő lépés a type/interface használata.

Tipikus kezdő hibák típusoknál

- Összekeverni a number és string értékeket (pl. "10" + 5).
- Elfelejtteni, hogy a const literál-típust is adhat.
- Nem kezelni a null lehetőségét külső adatoknál.

Függvénytípusok

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

```
const multiply = (a: number, b: number): number => a * b;
```

- Mindig típusold a bemenetet, és lehetőség szerint a visszatérési értéket is.
- Ettől egyértelműbb lesz a függvény szerződése.

Opcionális és alapértelmezett paraméter

```
function greet(name: string, title?: string): string {  
    return title ? `${title} ${name}` : name;  
}
```

```
function power(base: number, exp = 2): number {  
    return base ** exp;  
}
```

- Opcionális paraméter csak a paraméterlista végén legyen.
- Az alapértelmezett paraméter csökkenti a hívó kód bonyolultságát.

Union típusok

```
let id: string | number;  
id = "A12";  
id = 12;
```

```
type Direction = "up" | "down" | "left" | "right";  
let move: Direction = "left";
```

- A union azt jelenti, hogy az adat többféle típusból érkezhethet.
- Literál unionnal szűk, előre ismert értékészletet adhatsz meg.

Típuszűkítés (narrowing)

```
function printId(value: string | number) {  
  if (typeof value === "string") {  
    console.log(value.toUpperCase());  
  } else {  
    console.log(value.toFixed(0));  
  }  
}
```

- Narrowing nélkül a fordító nem engedné string- vagy number-specifikus metódusok hívását.

Gondolkodási minta union esetén

- 1. lépés: sorold fel, milyen típusok jöhetnek.
- 2. lépés: minden esetre írd meg megfelelő ellenőrzést.
- 3. lépés: minden ágban csak az adott típushoz illő műveletet végezz.

Type alias

```
type User = {  
  id: number;  
  name: string;  
  email?: string;  
};  
  
const u: User = { id: 1, name: "Dora" };
```

- A type kulcsszó jó választás, ha később unionöket vagy utility típusokat is használsz.

Interface

```
interface Product {  
  sku: string;  
  price: number;  
  discount?: number;  
}
```

```
const p: Product = { sku: "A-1", price: 100 };
```

- Az interface különösen hasznos API-szerződéseknel és osztályoknál.

Type vs interface (gyakorlat)

- Interface: objektumszerződésekhez.
- Type: unionokhoz, metszetekhez, utility típusokhoz.
- Mindkettő használható objektumokhoz, a csapatdöntés a fontos.
- A legfontosabb: kódbázison belül maradj következetes.

Readonly és opcionálisság

```
type Settings = {  
  readonly appName: string;  
  theme?: "light" | "dark";  
};
```

- readonly: a property létrehozás után nem módosítható.
- ?: az adott mezőt nem kötelező minden esetben megadni.

Mikor válts type aliasról interface-re?

- Ha objektumszerződést publikus API-ként szeretnél kommunikálni.
- Ha osztály implementálja a szerződést.
- Ha a csapatod konvenciója ezt preferálja.

Generic alap

```
function identity<T>(value: T): T {  
    return value;  
}  
  
const a = identity(42);           // number  
const b = identity("hello");     // string
```

- A generic lehetővé teszi, hogy ugyanaz a függvény több típusra is működjön.
- A T helyére a fordító a hívás alapján következteti a típust.

Generic constraint

```
function firstItem<T extends any[]>(items: T) {  
    return items[0];  
}
```

```
const first = firstItem([10, 20, 30]);
```

- A constraint (extends) korlátozza, milyen típus adható a generic paraméternek.

Utility típusok

```
type User = { id: number; name: string; email: string };  
  
type UserPatch = Partial<User>;  
type UserPublic = Pick<User, "id" | "name">;  
type UserWithoutEmail = Omit<User, "email">;
```

- Utility típusokkal gyorsan előállíthatsz új típusokat manuális másolás nélkül.

Record es Readonly

```
type ScoreByName = Record<string, number>;
```

```
type Config = Readonly<{  
  apiUrl: string;  
  retries: number;  

```

- Record: kulcs-érték struktúra típusosan.
- Readonly: megakadályozza a véletlen módosításokat.

A genericek tanulási sorrendje

- 1 Először használj kész generic API-kat (pl. `Promise<T>`, `Array<T>`).
- 2 Utána írd egyszerű saját generic függvényt.
- 3 Végül vezesd be a constraintet és utility típusokat.

Modulok: export/import

```
// math.ts
export function add(a: number, b: number) {
  return a + b;
}
```

```
// app.ts
import { add } from "./math.js";
console.log(add(2, 3));
```

- Modulokra bontással átláthatóbb és újrafelhasználhatóbb kódot kapsz.
- NodeNext módban importnál gyakran .js kiterjesztést használsz.

Async/Await típusokkal

```
type User = { id: number; name: string };

async function fetchUser(id: number): Promise<User> {
  const res = await fetch(`/api/users/${id}`);
  if (!res.ok) throw new Error("Request failed");
  return res.json();
}
```

- A `Promise<User>` egyértelműen jelzi a hívó kódnak a várt eredményt.
- Hibakezelés nélkül API-hívásnál nehezen diagnosztizálható hibák jelennek meg.

Kulso csomagok tipizalasa

```
npm install lodash  
npm install -D @types/lodash
```

- Sok csomag már tartalmaz beépített típusokat.
- Ha nem, akkor @types csomagot kell telepíteni.
- Típusok nélkül külső csomag használata gyorsan any-hoz vezet.

Axios pelda válasz-tipussal

```
import axios from "axios";

type Todo = { id: number; title: string; done: boolean };

async function getTodo(id: number): Promise<Todo> {
  const r = await axios.get<Todo>(`/api/todos/${id}`);
  return r.data;
}
```

- A `<Todo>` generic paraméterrel a válasz struktúrája már a hívás helyén ellenőrizhető.

Gyakorlati minta API-khoz

- Definiálj külön típusokat a kérésekhez: Request DTO, Response DTO.
- A hálózati réteget különítsd el service függvényekbe.
- A komponensek csak a már tipizált adatot kapják.

any vs unknown

```
let loose: any = 123;  
loose = "brmi"; // nincs ellenrz  
  
let safe: unknown = JSON.parse('{ "id": 1 }');  
if (typeof safe === "object" && safe !== null) {  
  // itt mr szkthet  
}
```

- Az any kikapcsolja a típusellenőrzést, ezért csak átmeneti esetben használd.
- Az unknown biztonságosabb, mert használat előtt kötelező ellenőrizni.

Futásidejű validáció Type Guarddal

```
function isUser(v: unknown): v is { id: number; name: string } {  
  return (  
    typeof v === "object" &&  
    v !== null &&  
    "id" in v &&  
    "name" in v  
  );  
}
```

- A TypeScript nem helyettesíti a futásidejű validációt.
- Külső adatnál (API, localStorage, URL-paraméter) mindig ellenőrizz.

Strict mód bekapcsolása

```
{  
  "compilerOptions": {  
    "strict": true,  
    "noUncheckedIndexedAccess": true,  
    "exactOptionalPropertyTypes": true  
  }  
}
```

- Ezek a beállítások kezdetben több hibát mutatnak, de hosszabb távon stabilabb kódot adnak.

Gyakori hibák

- Túl sok any használata.
- Null/undefined esetek figyelmen kívül hagyása.
- Túl bonyolult típusok írása stabil alapok nélkül.
- Típusok másolása utility típusok használata helyett.

Bevált gyakorlat röviden

- Alapértelmezésben legyen bekapcsolva a strict mód.
- Külső adatot mindig validálj.
- Bonyolult típus helyett használj több kicsi, beszédes típust.
- Legyen következetes kódstílus és linterbeállítás.

4 hetes tanulási terv

- 1 1. hét: típusok, tömbök, objektumok, függvények.
 - 2 2. hét: unionök, narrowing, aliasok, interfészek.
 - 3 3. hét: generics, utility típusok, modulok.
 - 4 4. hét: aszinkron kód, API-típusok, futásidejű validáció.
- Napi 45-60 perc folyamatos gyakorlás már jól látható fejlődést ad.

Projektötletek kezdőknek

- Teendőlista-alkalmazás (CRUD) típusosan.
- Egyszerű REST API-kliens.
- Jegyzetelő CLI fájlba mentéssel.
- Cél: az alaptípusok és a függvényszerződések rutinszerű használata.

Projektötletek középhaladóknak

- Modulokra bontott mini backend.
- Validációs réteggel rendelkező API-kliens.
- Kis NPM-csomag saját típusdefiníciókkal.
- Cél: valós, projekt-szerű struktúrában gondolkodni.

Mit érdemes mindenképp megtanulni?

- A típusinferencia értéke.
- A union + narrowing magabiztos használata.
- A generics és utility típusok alapjai.
- Futásidejű validáció külső adat esetén.

Önellenőrző kérdések minden modul után

- Meg tudom magyarázni, mit vár és mit ad vissza a függvényem?
- Használtam-e any-t ott, ahol az unknown biztonságosabb lenne?
- Tudok-e példát mondani a union + narrowing együtt használatára?
- Validáltam-e a külső adatot, mielőtt használtam?

keyof és indexed access

```
type User = {  
  id: number;  
  name: string;  
  active: boolean;  
};  
  
type UserKey = keyof User;    // "id" | "name" | "active"  
type NameType = User["name"]; // string
```

- A keyof segítségével dinamikusan hivatkozhatok egy típus kulcsaira.
- Az indexed access típus pontosan kiolvassa egy mező típusát.

Mapped type alapminta

```
type Optional<T> = {  
  [K in keyof T]?: T[K];  
};  
  
type Readonly<T> = {  
  readonly [K in keyof T]: T[K];  
};
```

- Mapped type-pal meglévő típusból generálhatsz új, következetes szerkezetet.

Conditional type és infer

```
type ElementType<T> = T extends (infer U)[] ? U : T;
```

```
type A = ElementType<string[]>; // string
```

```
type B = ElementType<number>;    // number
```

- A conditional type típus-szintű "if" logikát ad.
- Az infer kulcsszóval résztípusokat tudsz kinyerni.

Mikor használd ezeket?

- Ha sok hasonló DTO típust kell karbantartanod.
- Ha könyvtárszintű, újrafelhasználható típus-API-t építesz.
- Ha ismétlődő kézi típusmásolást látsz a projektben.

Szabály

Haladó típust csak akkor vezess be, ha egyszerűbbé teszi a használatot.

Osztályok TypeScriptben

```
class Account {  
  constructor(  
    public owner: string,  
    private balance: number  
  ) {}  
  
  deposit(amount: number): void {  
    this.balance += amount;  
  }  
  
  getBalance(): number {  
    return this.balance;  
  }  
}
```


Interface + osztály együtt

```
interface Logger {  
    log(message: string): void;  
}  
  
class ConsoleLogger implements Logger {  
    log(message: string): void {  
        console.log(message);  
    }  
}
```

- Az interface szerződés, az osztály ennek konkrét megvalósítása.

Rétegzett felépítés röviden

- domain: üzleti logika és modellek
- service: alkalmazási műveletek
- infrastructure: API, adatbázis, külső kapcsolatok
- ui/controller: bemenet-kimenet kezelése

Előny

A jól rétegzett struktúra könnyebben tesztelhető és bővíthető.

Gyakori tervezési hiba

- Minden logika egyetlen fájlban van.
- Az API-hívás közvetlenül a komponensben történik típusos határ nélkül.
- A validáció és üzleti szabályok keverednek.
- Megoldás: rétegek szétválasztása és tiszta típushatárok.

Hogyan olvasd a TypeScript hibákat?

- Először a hiba kódját nézd (pl. TS2322).
- Utána nézd meg: "mit kaptam" és "mit vár a típus".
- A hiba gyakran a hívási pontnál jelenik meg, de az ok lehet korábban.

Tipikus hiba: TS2322

```
let count: number;  
count = "10"; // TS2322
```

- Jelentés: nem kompatibilis típus-hozzárendelés.
- Javítás: konvertálj (`Number("10")`) vagy javítsd az adatforrást.

Tipikus hiba: TS2532

```
type User = { name?: string };  
const u: User = {};  
  
console.log(u.name.toUpperCase()); // TS2532
```

- Jelentés: az érték lehet undefined.
- Javítás: ellenőrzés (if), optional chaining (u.name?.toUpperCase()).

Gyakorlati hibakeresési rutin

- 1 Szűkítsd a hibát a legkisebb reprodukálható példára.
- 2 Ellenőrizd a bemenet tényleges futásidejű értékét.
- 3 Egyeztesd a runtime adatot a deklarált típussal.
- 4 Írj tesztet, hogy a hiba ne térjen vissza.

Feladat 1: Típusos szűrés

Feladat

Készíts függvényt, amely egy vegyes (string | number) tömbből csak a számokat adja vissza.

```
function onlyNumbers(values: Array<string | number>): number[] {  
    return values.filter((v): v is number => typeof v === "number");  
}
```


Feladat 2: Generic API válasz

Feladat

Definiálj újrafelhasználható API-válasz típust data, ok és error mezőkkel.

```
type ApiResponse<T> = {  
  data: T;  
  ok: boolean;  
  error?: string;  
};
```

Feladat 3: Runtime validáció

Feladat

Írj type guardot, amely ellenőrzi, hogy az adat Todo-e.

```
type Todo = { id: number; title: string; done: boolean };

function isTodo(v: unknown): v is Todo {
  return (
    typeof v === "object" && v !== null &&
    "id" in v && "title" in v && "done" in v
  );
}
```

Értékelési szempontok

- Típusbiztonság: nincs felesleges any.
- Olvashatóság: beszédes típus- és függvénynév.
- Hibatűrés: külső adat validálása.
- Bővíthetőség: új mező esetén is tiszta marad a kód.