

TypeScript

Kiegészítő tananyag – a JavaScript alapoktól a haladó típusokig

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKLB_INTM131.git

2026. szeptember 8.



Miről szól ez az anyag?

Ez egy **önállóan is végigkövethető, kiegészítő** tananyag a TypeScriptről. Nem feltételezi, hogy láttad a 2. hét TypeScript diáit – a legelejétől indulunk, majd jóval mélyebbre megyünk, mint amit egy heti óra kerete engedne.

- Ha még sosem láttál TypeScript kódot, nyugodtan kezd itt
- Ha a 2. héten már megismerted az alapokat, néhány dia ismétlés lesz – utána viszont új anyag következik
- A cél: hogy magabiztosan tudj TypeScript kódot olvasni és írni, ne csak felismerni

Nem kötelező anyag

Ez a diasor *nem* számonkérendő tananyag – annak szól, aki jobban meg szeretné érteni a TypeScript-et, mint amire a heti órák ideje engedne.

Mi az a TypeScript, és miért érdemes használni?

A JavaScript **gyengén típusos** nyelv: egy változó bármikor bármilyen típusú értéket felvehet, és a legtöbb hiba csak futás közben derül ki. A **TypeScript** (Microsoft, 2012) a JavaScript egy kiterjesztése, ami *statikus típusrendszert* ad hozzá – a kód végül mindig sima JavaScript-té fordul le.

- A típushibák nagy része már íráskor / fordításkor kiderül, nem futás közben, esetleg egy élő felhasználónál
- Az IDE (VS Code, stb.) pontos kódkiegészítést, azonnali hibajelzést tud adni
- A kód önmagát dokumentálja: egy függvény szignatúrájából látszik, mit vár és mit ad vissza
- Nagyobb csapatban, nagyobb, hosszú távon karbantartott kódbázisban különösen megtérül

Fontos

A TypeScript *opcionális, fokozatosan bevezethető* réteg – egy meglévő JavaScript projektbe fájlanként is átállhatunk rá.

Telepítés és a tsconfig.json

A TypeScript fordítót (tsc) npm csomagként telepítjük, majd egy konfigurációs fájlt hozunk létre, ami megmondja a fordítónak, hogyan viselkedjen.

Telepítés

```
npm install -D typescript  
npx tsc --init
```

A tsconfig.json legfontosabb mezői

- `target` – milyen JavaScript verzióra fordítson (pl. "ES2020")
- `module` – modulrendszer ("commonjs" vagy "esnext")
- `outDir` / `rootDir` – hova kerüljön a lefordított kód, honnan olvasson forrást
- `strict` – bekapcsolja az összes szigorú típusellenőrzést (*erősen ajánlott true*)

Futtatás fejlesztés közben

A `.ts` fájlokat sem a böngésző, sem a Node.js nem futtatja közvetlenül – előbb JavaScript-té kell fordítani. Fejlesztés közben ezt kényelmesebb eszközökkel oldjuk meg, mint kézzel fordítani minden mentés után.

Klasszikus fordítás

```
npx tsc          # egyszeri fordítás
npx tsc --watch  # automatikusan ujrafordít mentés után
```

Közvetlen futtatás (nincs külön fordítási lépés)

```
npx tsx index.ts
```

A `tsx` (és a régebbi `ts-node`) memóriában fordítja és rögtön futtatja is a kódot – ez a legkényelmesebb fejlesztés közben.

Az első .ts fájl

Nézzük meg, hogyan néz ki egy egyszerű, típusozott TypeScript fájl – ez lesz az alapja a következő diáknak, ahol részletesen végigvesszük az egyes típusokat.

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

```
const result = add(2, 3);  
console.log('Az eredmény: ${result}');
```

```
// add('2', 3); -- ez fordítási hibát adna!
```

Amit érdemes megfigyelni

A paraméterek és a visszatérési érték típusát kettősponttal adjuk meg. Ha rossz típusú értékkel próbálnánk hívni a függvényt, a fordító *azonnal* hibát jelezne – még mielőtt a kód lefutna.

Primitív típusok

A TypeScript primitív típusai megegyeznek a JavaScript primitív értékeivel – csak most a fordító is számon tartja, melyik változó melyik típusú.

```
let name: string = 'Anna';
let age: number = 25;
let isActive: boolean = true;
let id: bigint = 123n;
let nothing: null = null;
let notSet: undefined = undefined;
```

- string, number, boolean – a leggyakrabban használt három típus
- null és undefined önálló típusként is léteznek, de gyakrabban más típusokkal együtt (pl. string | null) fordulnak elő
- bigint és symbol ritkábban használt, speciális típusok

Tömbök és tuple-ök

A tömböknél meg kell adni, milyen típusú elemeket tartalmaznak. A **tuple** egy speciális, rögzített hosszúságú és típusosrendű tömb.

```
let names: string[] = ['Anna', 'Bela'];  
let scores: Array<number> = [10, 20, 30];  
  
// Tuple: pontosan két elem, előre rögzített típussal  
let point: [number, number] = [3, 5];  
let entry: [string, number] = ['age', 25];
```

Miért hasznos a tuple?

Egy `[string, number]` tuple azonnal jelzi, hogy pontosan két elemet várunk, meghatározott sorrendben – ellentétben egy sima `(string | number)[]` tömbbel, ami bármilyen hosszú és sorrendű keveréket engedne.

any, unknown és never

Három speciális típus, amit érdemes megkülönböztetni – könnyű összekeverni őket, de nagyon más a szerepük.

- **any**: teljesen kikapcsolja a típusellenőrzést erre a változóra – gyakorlatilag "visszatérünk sima JavaScript-hez". Kerülendő, csak végső esetben használjuk!
- **unknown**: szintén "bármilyen lehet", de *használat előtt* típusellenőrzést (narrowing) kényszerít ki – ez a biztonságos alternatívája az **any**-nek
- **never**: olyan érték típusa, ami *sosem* fordulhat elő – pl. egy függvény, ami mindig hibát dob, vagy egy végtelen ciklus visszatérési típusa

Ökölszabály

Ha típust nem tudsz vagy nem akarsz megadni, használj **unknown**-t **any** helyett – így a fordító legalább kikényszeríti, hogy ellenőrizd az értéket, mielőtt használnád.

Type inference – amikor nem kell kiírni a típust

A TypeScript sok esetben *magától kikövetkezteti* (infer) a típust a kezdőértékből – nem kell mindenhol explicit típust írunk.

```
let name = 'Anna';    // TypeScript tudja: ez string
let age = 25;         // TypeScript tudja: ez number

name = 42;            // Hiba! nem lehet number-t string-nek adni
```

Mikor írjunk mégis explicit típust?

Függvényparamétereknél szinte mindig érdemes (a fordító nem tudja kitalálni, mit fogunk hívni), de egyszerű, azonnal értékadott lokális változóknál nyugodtan hagyatkozhatunk az inferálásra.

Type alias – saját név egy típusnak

A `type` kulcsszóval nevet adhatunk egy típusnak – ez különösen hasznos, ha egy összetettebb típust (pl. union típust, amit a 4. fejezetben veszünk) többször is használunk.

```
type UserId = number;
type Username = string;

function getUser(id: UserId): Username {
  return 'Anna';
}
```

Ez még nem sokkal több, mint egy elnevezés – de amint a következő fejezetekben összetettebb típusokat (union, interfész) ismerünk meg, a `type` igazán hasznossá válik.

Összefoglalás – Alaptípusok

Alaptípusok

- Primitívek: `string`, `number`, `boolean`, `null`, `undefined`, `bigint`
- Tömbök: `string[]`; Tuple: rögzített hosszú/sorrendű tömb, pl. `[string, number]`
- `any` (kerülendő) vs `unknown` (biztonságos) vs `never` (sosem előforduló érték)
- Type inference: a fordító sokszor magától kitalálja a típust
- `type` – nevet adhatunk egy típusnak, újrafelhasználhatóvá téve azt

Interfész alapok

Az interface egy objektum *alakját* (mely mezők, milyen típussal) írja le. Ahol egy interfészt típusként megadunk, a fordító ellenőrzi, hogy az érték tényleg megfelel-e neki.

```
interface User {  
  id: number;  
  name: string;  
  email: string;  
  nickname?: string;    // opcionális mezo  
  readonly createdAt: Date; // csak olvashato  
}
```

```
const user: User = {  
  id: 1,  
  name: 'Anna',  
  email: 'anna@example.com',  
  createdAt: new Date(),  
};
```

Opcionális és csak olvasható mezők

Két apró, de nagyon hasznos jelölés van, amivel pontosabbá tehetjük az interfészünket: a `?` és a `readonly`.

- `mezo?: típus` – a mező elhagyható, de ha megadjuk, a megjelölt típusúnak kell lennie
- `readonly mezo: típus` – a mező az objektum létrehozása után nem módosítható

```
const user: User = { id: 1, name: 'Anna',  
  email: 'a@b.com', createdAt: new Date() };
```

```
user.createdAt = new Date(); // Hiba! readonly mezo
```

Miért hasznos?

A `readonly` segít elkerülni azokat a hibákat, amikor véletlenül módosítunk egy olyan mezőt, aminek soha nem szabadna változnia (pl. egy azonosítót).

Interfészek kiterjesztése

Egy interfész `extends`-szel örökölheti egy másik interfész mezőit – így elkerülhetjük az ismétlődést hasonló, egymásra épülő alakoknál.

```
interface Animal {  
  name: string;  
  age: number;  
}  
  
interface Dog extends Animal {  
  breed: string;  
}
```

```
const myDog: Dog = { name: 'Buxsi', age: 3, breed: 'labrador' };
```

A Dog interfész mindent örököl az Animal-tól, és hozzáad egy breed mezőt – pontosan úgy, ahogy egy osztály is örökölhet egy másiktól.

interface vs type – mikor melyiket?

A két eszköz sokban hasonlít, és a legtöbb egyszerű esetben szinte mindegy, melyiket választjuk – de van néhány különbség, ami eldöntheti a választást.

- **interface**: csak objektum alakok leírására való; több, azonos nevű interfész deklaráció automatikusan összeolvad (declaration merging); **extends**-szel bővíthető
- **type**: bármilyen típust elnevezhet – objektumot, uniót, primitívet is; nem olvad össze automatikusan, de intersection-nel (&) kombinálható

Gyakorlati ökölszabály

Objektum *alakokhoz* (pl. egy adatmodell) válaszd az **interface**-t – olvashatóbb hibaüzeneteket ad. Union típusokhoz (a következő fejezet) csak a **type** jöhet szóba.

Összefoglalás – Interface és Type

Interface és Type

- **interface**: objektum alakjának leírása – mezők, opcionális (?) és readonly mezők
- **extends**: interfészek egymásra építése, kódismétlés elkerülése
- **type**: bármilyen típusnak adhatunk nevet, nemcsak objektumoknak
- Objektum alakoknál **interface**, union típusoknál **type** – ökölszabályként

Függvények típusozása

Egy TypeScript függvénynél mind a paraméterek, mind a visszatérési érték típusát megadhatjuk – a visszatérési típust a fordító a legtöbbször ki is következteti, de explicit kiírva olvashatóbb, és véd az elgépelés ellen.

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

```
// Nincs visszatérő érték -> void  
function logMessage(message: string): void {  
    console.log(message);  
}
```

Miért érdemes kiírni a visszatérési típust?

Ha véletlenül rossz típusú értéket adnánk vissza (pl. `string`-et `number` helyett), a fordító azonnal jelezné a hibát – még mielőtt bárhol meghívnánk a függvényt.

Opcionális és alapértelmezett paraméterek

Nem minden paraméter kötelező – a JavaScript-hez hasonlóan itt is megadhatunk alapértelmezett értéket, vagy jelölhetjük egy paramétert opcionálisként.

```
function greet(name: string, greeting: string = 'Szia'): string {  
    return `${greeting}, ${name}!`;  
}
```

```
function createUser(name: string, nickname?: string) {  
    return { name, nickname: nickname ?? name };  
}
```

```
greet('Anna');           // "Szia, Anna!"  
greet('Anna', 'Hello');  // "Hello, Anna!"
```

Fontos sorrend

Az opcionális (?) és alapértelmezett paraméterek csak a kötelező paraméterek *után* szerepelhetnek a paraméterlistában.

Függvénytípusok – callback-ek típusozása

Amikor egy függvény egy másik függvényt vár paraméterként (pl. egy callback-et), magának a "függvény alaknak" is megadhatjuk a típusát.

```
type MathOperation = (a: number, b: number) => number;
```

```
function calculate(a: number, b: number, op: MathOperation): number {  
    return op(a, b);  
}
```

```
const sum: MathOperation = (a, b) => a + b;  
calculate(2, 3, sum); // 5
```

Ez különösen hasznos Express.js-ben: egy route handler is pontosan egy ilyen "függvény alakú" típus `((req, res) => void)`.

Function overloads – röviden

Néha egy függvény többféle paraméter-kombinációval is hívható, és a visszatérési típus a bemenettől függ. Erre valók a **overloads**: több szignatúra deklarálása egyetlen implementáció előtt.

- Több "fejléct" (szignatúrát) írunk le, majd egyetlen tényleges implementációt
- A hívó fél oldaláról úgy tűnik, mintha több, különböző típusú változata lenne a függvénynek
- Ritkán szükséges kezdőként – inkább unió típusokkal (következő fejezet) old meg a legtöbb esetet

Gyakorlati tanács

Ha csak most ismerkedsz a TypeScript-tel, nem kell aktívan használnod az overloads-ot – elég felismerni, ha egy könyvtár dokumentációjában találkozol vele.

Összefoglalás – Függvények

Függvények TypeScript-ben

- Paraméterek és visszatérési típus típusozása – `void`, ha nincs visszatérő érték
- Alapértelmezett érték (`= ...`) és opcionális paraméter (`?`), mindig a kötelezők után
- Függvénytípusok: `(a: number, b: number) => number` – callback-ek típusozásához
- Overloads: több szignatúra egy implementációhoz – ritkán szükséges kezdőként

Union típusok – "vagy ez, vagy az"

Egy **union** típus (| jellel) azt fejezi ki, hogy egy érték *többféle* típus egyike lehet. Ez az egyik legtöbbször használt eszköz a valós TypeScript kódban.

```
type Id = string | number;

function printId(id: Id) {
  console.log('Az azonosító: ${id}');
}

printId(123);           // OK
printId('abc-123');    // OK is
```

Gyakori példa

```
type HttpMethod = 'GET' | 'POST' | 'PUT' | 'PATCH' | 'DELETE';
```

– ilyenkor a fordító nem enged elgépelte, érvénytelen metódusnevet.

Type narrowing – a típus leszűkítése

Ha egy változó union típusú, a fordító csak azokat a műveleteket engedi rajta, amik *minden* lehetséges típusnál érvényesek – amíg nem szűkítjük le (narrowing), melyik konkrét típusról van szó.

```
function printId(id: string | number) {  
  if (typeof id === 'string') {  
    console.log(id.toUpperCase()); // itt mar biztos string  
  } else {  
    console.log(id.toFixed(2));    // itt mar biztos number  
  }  
}
```

- `typeof` – primitív típusok megkülönböztetésére (`string`, `number`, stb.)
- `instanceof` – osztálypéldányok megkülönböztetésére
- `in` – egy adott mező meglétének ellenőrzésére egy objektumon

Intersection típusok – "ez is, az is"

Az **intersection** típus (& jellel) ennek épp az ellentéte: egy érték *egyszerre* feleljen meg több típusnak is – a mezők összeadódnak.

```
interface HasId {  
    id: number;  
}  
  
interface HasName {  
    name: string;  
}  
  
type Entity = HasId & HasName;  
  
const user: Entity = { id: 1, name: 'Anna' };
```

Az Entity típusú értéknek *mindkét* interfész mezőit tartalmaznia kell – ez hasznos, ha kisebb, újrafelhasználható "építőelemekből" rakunk össze nagyobb típusokat.

Literal típusok

A TypeScript-ben nemcsak `string` vagy `number` lehet egy típus, hanem egy *konkrét érték* is – ezt hívjuk literal típusnak. Union-nal kombinálva rendkívül kifejező, önmagát dokumentáló kódot kapunk.

```
type Status = 'pending' | 'active' | 'closed';

function setStatus(status: Status) {
  // csak a három felsorolt string-et fogadja el
}

setStatus('active');    // OK
setStatus('done');      // Hiba! nincs ilyen Status
```

Összefoglalás – Union és Intersection

Union és Intersection típusok

- Union (`|`): "vagy ez, vagy az" – egy érték többféle típus egyike lehet
- Type narrowing: `typeof/instanceof/in` – leszűkítjük, melyik konkrét típusról van szó
- Intersection (`&`): "ez is, az is" – több típus mezői összeadódnak
- Literal típusok: konkrét értékek típusként, gyakran union-nal kombinálva (pl. HTTP metódusok)

A probléma, amit a generikák oldanak meg

Képzeljünk el egy függvényt, ami egy tömb első elemét adja vissza – de szeretnénk, hogy *bármilyen* típusú tömbre működjön, miközben megőrizzük a típusinformációt.

```
function firstOfNumbers(arr: number[]): number {  
    return arr[0];  
}  
function firstOfStrings(arr: string[]): string {  
    return arr[0];  
}  
// ...es így tovább minden típusra külön függvény?
```

A probléma

any-jal megoldhatnánk, de akkor elveszítenénk a típusbiztonságot: a fordító többé nem tudná, mit ad vissza a függvény.

Generikus függvények

A **generikus típus** egy "típusparamétert" (jellemzően T) vezet be, amit a híváskor a konkrét típus tölt ki – így egyetlen függvény működik bármilyen típusra, típusbiztonság elvesztése nélkül.

```
function first<T>(arr: T[]): T {  
    return arr[0];  
}
```

```
const n = first([1, 2, 3]);    // T = number  
const s = first(['a', 'b', 'c']); // T = string
```

A TypeScript minden híváskor kikövetkezteti, mi legyen T – a visszatérési érték típusa mindig pontosan illeszkedik a bemenethez.

Generikus interfészek

Nemcsak függvények, hanem interfészek (és típusok) is lehetnek generikusak – ez az alapja például annak, ahogy a legtöbb adatbázis-eredmény vagy API válasz típus felépül.

```
interface Box<T> {  
  value: T;  
}
```

```
const numberBox: Box<number> = { value: 42 };  
const stringBox: Box<string> = { value: 'hello' };
```

```
interface ApiResponse<T> {  
  data: T;  
  success: boolean;  
}
```

Egy `ApiResponse<User>` pontosan jelzi, hogy a `data` mező egy `User` objektumot fog tartalmazni – ezt fogjuk a későbbi heteken az Express válaszainknál is használni.

Constraints – a generikus típus korlátozása

Néha nem szeretnénk *bármilyen* típust elfogadni – csak olyanokat, amiknek van egy adott mezőjük vagy tulajdonságuk. Erre valók a `extends` kulcsszóval megadott **constraint**-ek.

```
interface HasLength {  
  length: number;  
}  
  
function logLength<T extends HasLength>(item: T): T {  
  console.log(item.length);  
  return item;  
}  
  
logLength('hello');           // OK - a string-nek van length-je  
logLength([1, 2, 3]);         // OK - a tömbnek is  
logLength(42);                // Hiba! a number-nek nincs length-je
```

Összefoglalás – Generikus típusok

Generikus típusok

- A generikák elkerülik a kód duplikációt anélkül, hogy fel kellene adnunk a típusbiztonságot
- `function first<T>(arr: T[]): T` – a típusparamétert a hívás tölti ki
- Interfészek/típusok is lehetnek generikusak, pl. `Box<T>`, `ApiResponse<T>`
- `T extends ...` – a generikus típus korlátozása egy adott "alakra"

Miért kellene a Utility Type-ok?

A TypeScript beépít néhány **beépített generikus segédtypust** (utility type), amikkel egy meglévő típusból egy *módosított változatot* tudunk létrehozni – anélkül, hogy a mezőket kézzel újra le kellene írjunk.

```
interface User {  
  id: number;  
  name: string;  
  email: string;  
}
```

A következő diákon ebből az egy User interfészből építünk különböző, gyakran szükséges változatokat – pl. "frissítéshez", "listázáshoz" vagy "létrehozáshoz".

Partial és Required

```
type UserUpdate = Partial<User>;  
// { id?: number; name?: string; email?: string; }  
  
type UserStrict = Required<User>;  
// minden mezo kotelezo, meg ha eredetileg ? volt is
```

- `Partial<T>`: minden mezőt opcionálissá tesz – tökéletes egy PATCH végpont bemenetéhez, ahol nem kötelező minden mezőt megadni
- `Required<T>`: minden mezőt kötelezővé tesz – a `Partial` ellentéte

Pick és Omit

Ez a két típus mezőket *válogat ki*, vagy éppen *hagy ki* egy meglévő típusból.

```
type UserPreview = Pick<User, 'id' | 'name'>;  
// { id: number; name: string; }
```

```
type UserWithoutEmail = Omit<User, 'email'>;  
// { id: number; name: string; }
```

Gyakori felhasználás

Egy adatbázisból visszakapott User rekordot gyakran *nem* akarunk teljes egészében visszaküldeni az API válaszbán (pl. egy jelszó-hash mezőt biztosan nem) – a Pick/Omit pontosan erre való.

Record és Readonly

```
type UsersById = Record<number, User>;  
// { [id: number]: User } - kulcs-érték par tipusa
```

```
type ImmutableUser = Readonly<User>;  
// minden mezo csak olvashato (readonly)
```

- `Record<K, V>`: egy objektum típusa, ahol a kulcsok `K`, az értékek `V` típusúak – gyakori egy azonosító szerint indexelt "gyorsítótárnál"
- `Readonly<T>`: minden mezőt `readonly`-vá tesz – hasznos, ha biztosítani akarjuk, hogy egy objektum ne módosuljon

Összefoglalás – Utility Types

Utility Types

- `Partial<T>` / `Required<T>` – mezők opcionálissá / kötelezővé tétele
- `Pick<T, K>` / `Omit<T, K>` – mezők kiválogatása / kihagyása
- `Record<K, V>` – kulcs-érték típusú objektum
- `Readonly<T>` – minden mező írásvédetté tétele
- Mindegyik egy meglévő típusból épít egy *módosított változatot* – kódismétlés nélkül

Enum alapok

Az enum (felsorolt típus) egy névvel ellátott, korlátozott értékkészletet definiál – hasonló célt szolgál, mint a korábban látott string literal union, de saját névtérrel és futásidejű objektummal is jár.

```
enum Role {  
    Admin,      // 0  
    Editor,     // 1  
    Viewer,     // 2  
}
```

```
enum Status {  
    Pending = 'pending',  
    Active = 'active',  
    Closed = 'closed',  
}
```

```
const myRole: Role = Role.Admin;
```

Numerikus enumnál automatikusan 0-tól számozódnak az értékek; string enumnál mi adjuk meg a konkrét szöveges értéket – ez utóbbi jobban olvasható naplókban, hibaüzenetekben.

Osztályok TypeScript-ben

A TypeScript osztályai a JavaScript ES6 osztályaira épülnek, kiegészítve a mezők típusozásával.

```
class Person {  
  name: string;  
  age: number;  
  
  constructor(name: string, age: number) {  
    this.name = name;  
    this.age = age;  
  }  
  
  greet(): string {  
    return 'Szia, ${this.name} vagyok!';  
  }  
}  
  
const p = new Person('Anna', 25);
```

Access modifierek

Az osztály mezőihöz és metódusaihoz **láthatósági szinteket** rendelhetünk – ez segít elrejteni az implementációs részleteket, és csak a szándékolt "publikus felületet" mutatni kifelé.

```
class BankAccount {  
    private balance: number = 0;  
    public readonly owner: string;  
  
    constructor(owner: string) {  
        this.owner = owner;  
    }  
  
    public deposit(amount: number): void {  
        this.balance += amount;  
    }  
}
```

- **public** (alapértelmezett): bárhonnan elérhető
- **private**: csak az osztályon belülről érhető el
- **protected**: az osztályon és leszármazottjain belül érhető el

Interfész implementálása osztállyal

Egy osztály az `implements` kulcsszóval kötelezettséget vállal arra, hogy megfelel egy adott interfésznek – a fordító ellenőrzi, hogy tényleg minden szükséges mezőt/metódust megvalósít-e.

```
interface Greetable {  
    greet(): string;  
}  
  
class Person implements Greetable {  
    constructor(private name: string) {}  
  
    greet(): string {  
        return 'Szia, ${this.name} vagyok!';  
    }  
}
```

Decorators – rövid kitekintés

A **decorator** egy speciális jelölés (`@valami`), amit osztályok, mezők vagy metódusok elé írhatunk, és ami futásidejű viselkedést csatol hozzájuk – kicsit hasonlóan ahhoz, ahogy Express-ben a middleware-ek "csomagolják be" a route handlereket.

Miért fontos ez most csak kitekintésként?

A decorators ma még kísérleti funkció TypeScript-ben, de rendkívül elterjedt az ORM-ekben (pl. a későbbi heteken megismert TypeORM) – ott `@Entity`, `@Column` jelölésekkel írjuk le az adatbázis-táblákat.

```
@Entity()  
class User {  
    @Column()  
    name: string;  
}
```

Összefoglalás – Enum és Osztályok

Enum és Osztályok

- **enum**: névvel ellátott, korlátozott értékkeszlet (numerikus vagy string alapú)
- **Osztályok**: mezők típusozása, konstruktor, metódusok – az ES6 osztályokra épül
- **Access modifier**ek: `public` / `private` / `protected` – a láthatóság szabályozása
- **implements**: egy osztály kötelezettséget vállal egy interfész megvalósítására
- **Decorators**: `@valami` jelölések, amik későbbi heteken (TypeORM) kerülnek elő

Teljes összefoglalás

Alapok

Statikus típusrendszer JavaScript fölött; `tsc` fordít, `tsx/ts-node` közvetlenül futtat; `tsconfig.json` vezérli a fordítást (`strict` ajánlott).

Típusok

Primitívek, tömbök, tuple-ök; `any` kerülendő, `unknown` biztonságos; `interface/type`; `union (|)` és `intersection (&)`; generikák (`<T>`); utility types (`Partial`, `Pick`, stb.)

Osztályok

Access modifierok (`public/private/protected`), `implements`, `enum`, `decorators` (kitekintés a `TypeORM` felé)

Hogyan tovább?

A legjobb módja a TypeScript elsajátításának a *gyakorlás* – az elmélet önmagában nem elég, próbáld ki a saját kódodban is.

- Alakítsd át egy régebbi, sima JavaScript projektedet fokozatosan TypeScript-re, fájlonként haladva
- Kapcsold be a `strict` módot a `tsconfig.json`-ban – kezdetben tűnhet szigorúnak, de hosszabb távon ez fizetődik ki igazán
- Olvasd a hivatalos dokumentációt: typescriptlang.org/docs – naprakész, sok interaktív példával
- A tárgy további heteiben (Express.js + TypeScript, majd TypeORM) a itt tanultak folyamatosan elő fognak kerülni

Emlékeztető

Ez az anyag kiegészítő, önálló tanulást szolgál – nem számonkérendő, de a mélyebb megértés a féléves munkádat is megkönnyíti.