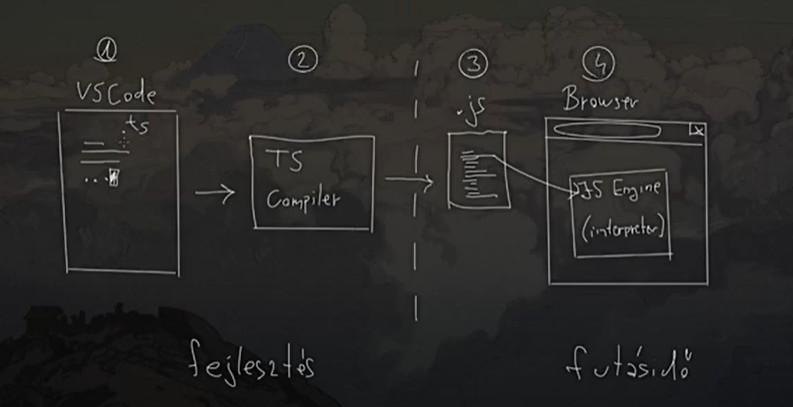
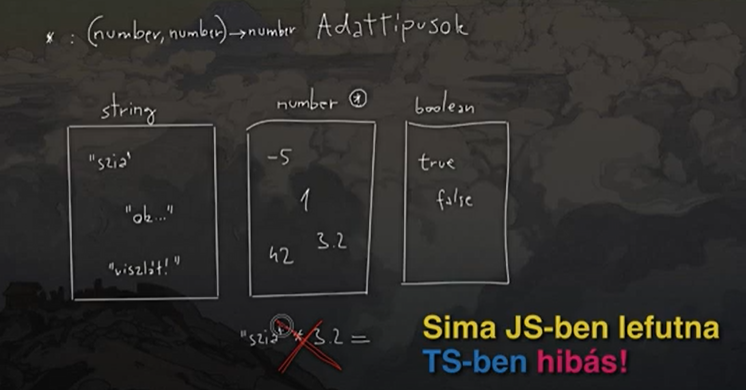
**Typescript**: A JS nyelvet egészíti ki statikus típusossággal.



* TS fájt írunk, amit egy TS compiler dolgoz fel és készít JS kódot, amit már a böngésző is értelmezni tud.
* a feldolgozás alatt megköveteli a statikus típusosságot és plusz megkötéseket alkalmaz a kóddal szemben
* ha nem felel meg ez a kód akkor ezt nem fordítja le, hanem csak kiírja a hibákat
* ezzel a módszerrel a szoftver stabilabb lesz és kevesebb hibát fog tartalmazni a statikus megkötéseknek köszönhetően

**Típusok:** típusok alatt halmazokat értünk, amelyek különböző értékeket foglalnak magukba. Azért jó. hogy az értékek be vannak csoportosítva a fajtájuk szerint mert ez előirányozza, hogy milyen műveletek hajthatóak végre rajtuk.



pl. definiálva van egy szorzás operáció, ami vár két number típusú értéket bementként és visszaad ugyancsak egy number típusú értéket. Amennyiben a szorzás egyik operandusaként egy string értéket adnánk meg akkor a TS hibát jelez és nem fut le mivel az operandus nem felel meg típus előírásoknak. Sima JS-ben lefutna.

**Telepítés:** npm i -g typescript

**Verzió:** tsc -v

**Inicializálás:** tsc –init (tsconfig.json fájlt generál)

**tsconfig beállítás:**

* készíteni egy src mappát
* "rootDir": "./src" – beállítása a config fájlban
* készíteni egy dist mappát
* "outDir": "./dist" – beállítása a config fájlban

**Elindítása és a kód figyelése**: tsc -w

**TS fájl létrehozása:** valami.ts

src mappa tartalmazza a TS fájlokat

dist mappa tartalmazza a JS fájlokat, amikre hivatkozhatunk majd a HTML-ben

**TYPE INFERENCE**

Ts compiler az értékből levezeti a típust. Nem kell explicit megadni a típust. Később ennek a változónak az értéke más típusra nem változtatható

**FÜGGVÉNYEK**

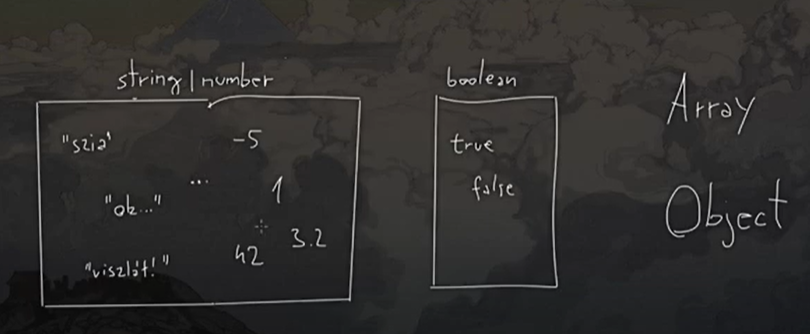
function peldaFuggveny(bemeneti: string): number {

return 2;

}

Bemenetnek meghatároztuk, hogy az csak string típusú lehet és a kiment csak number típusú

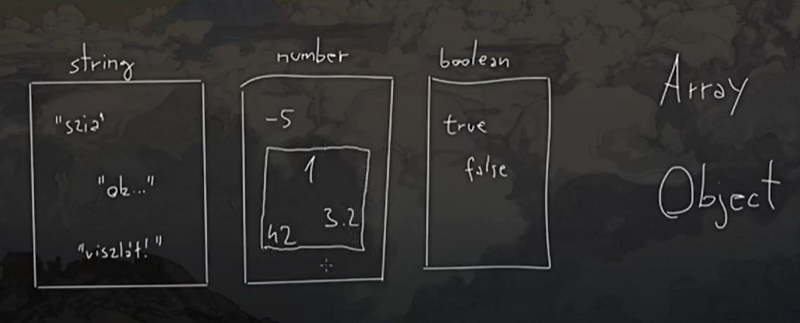
**UNION TYPE**



Saját unió létrehozása a típusokból, **halmazok összevonsása**

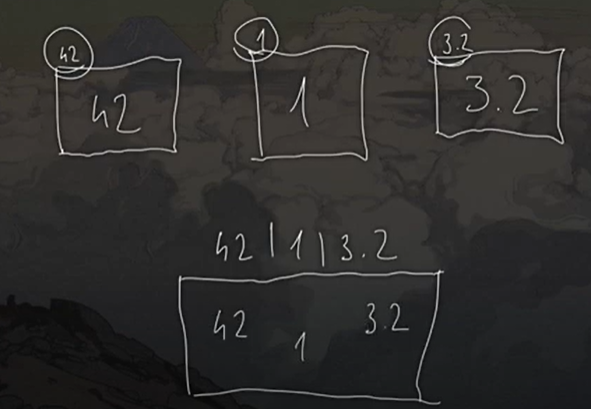
pl.: a stringből és numberből létrehozott unió - **string | number**

**LITERAL TÍPUS**

****

**halmazok szűkítése**

Van olyan eset amikor egy halmazban megadott értékek közül csak bizonyos elemekre van szükségünk. A típus definícióban kell megadni explicite azokat az értékeket amelyeket a változó felvehet



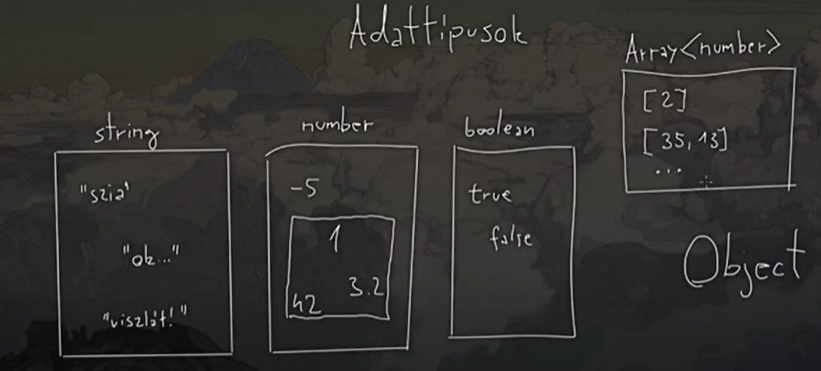
pl.: a number típusokból csak az 1,42,3.2-re van szükségünk

let ertek3: 1 | 42 | 3.2 = 42;

**TYPE ALIAS**

A type kulcsó után megadjuk a címkét majd a halmaz definícióját. Ezek után a megadott címkét használhatjuk a típusdefinícióban, változóknál és függvényeknél is bemeneti és kimeneti típusként is.

**ARRAY**



**TUPPLE**

Olyan tömb ami kevés elemet tartalmaz és az elemek a típusaiban eltérnek.

//string a 0.indexen van a number pedig az 1.indexen

let szemely: [string, number];

Címkéket is lehet használni a TUPLE definíciójában

type Szemely = [nev: string, eletkor: number];

let szemely: Szemely = ["Zolika", 45];

A TUPLE-t a distucturing révén érdemes használni

const [nev, eletkor] = szemely;

**OBJEKTUM**

Ha több értéket szeretnénk egyszerre tárolni akkor már objektumot kell használni TUPLE helyett.

let album = {

id: 1,

name: "valami",

title: "Bla Bla",

};

TYPE ALIAS használata:

type Album = {

id: number;

name: string;

title: string;

};

let album2: Album = {

id: 2,

name: "Halihó",

title: "Bla Bla",

};

**INTERFACE**

Ha egy objektum struktúráját egy általános referenciaként akarom meghatározni, mint a TYPE ALIAS-nál, akkor e helyett INTERFACE-t is használhatunk

interface Album {

id: number;

name: string;

title: string;

}

Az INTERFACE bővíthető további kulcs-érték párral a TYPE ALIAS nem.

Az INTERFACE használható a TYPE ALIAS-ban is:

interface Photo {

id: number;

title: string;

url: string;

thumbnailUrl: string;

}

type Album3 = {

id: number;

name: string;

title: string;

photos: Array<Photo>;

};