Kliensoldali technológiák

TypeScript

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

JavaScript kihívásai

- A nagy JavaScript kódbázisok nehezen karbantarthatók
- Az ECMAScript szabványosítás jó irányba ment, de nem követték elég gyorsan az implementációk

➢Készítsünk egy új programozási nyelvet, ami a JavaScript programozóknak szól és megoldást nyújt az egységes kezelésre!

Mit ad a TypeScript

- Kihasználhatjuk a statikus típusellenőrzést
  - A fordítási mechanizmus miatt **fordítási** idejű hibákat kapunk **futási** idejű hibák helyett
  - A **statikus típusosság** miatt igazi IntelliSense lehetséges
- Elrejti a JavaScript "furcsaságait"
- Használhatunk még nem szabványos JavaScript elemeket, amikből fordítás után sima JavaScript lesz
- Jelentősen javul a kód karbantarthatósága, olvashatósága
  - Ezáltal nagy kódbázis karbantartását is lehetővé teszi

Változódeklarációk (1)

- A **`var`** kulcsszóval függvényhez (nem blokkhoz) kötött változót hozhatunk létre
  - A külső függvény a globális névtér
- A **`let`** és **`const`** kulcsszavakkal blokkhoz kötött változókat hozhatunk létre, a **`const`** immutábilis
- A típust explicit megadhatjuk a változó neve után, vagy az értékadásból egyértelműen kiderül
  - Különben a típus **`any`** lesz

Enumok

- Az enum sorszámozása 0-tól növekszik, kivéve, ha explicit növeljük

```
enum DogKind { Pitbull = 1, Terrier, Corgi };

let dog = DogKind.Pitbull
console.log(dog)
// reverse enum: number -> string
console.log(DogKind[dog])
```

```
[LOG]: 1
[LOG]: "Pitbull"
```

- String alapú enumok

```
enum Direction {
  Up = "fel",
  Down = "le",
  Left = "balra",
  Right = "jobbra",
}

let dir = Direction.Up
console.log(dir)
```

```
[LOG]: "fel"
```

Osztályok áttekintése

- Támogatottak:
  - Osztályok
  - Interfészek (explicit- és implicit megvalósítás)
  - Absztrakt osztályok
  - Öröklés
  - Láthatósági módosítószók
  - Osztályszintű változók és függvények
- Nem támogatottak:
  - Valódi metódus overloading
  - Valódi többszörös öröklés
  - Típusonként több konstruktor

Modulok (1)

- A modulok az egységbezárást segítik, a logikailag összefüggő osztályok, objektumok, függvények, változók egy *logikai* fájlba helyezhetők
  - A modulokból ezek kifelé publikálhatók (export), kívülről pedig konzumálhatók (import)
  - Külső függőségek is ezt a mintát használják, így szeparálhatók a felelősségi körök

JavaScript kihívásai

- A nagy JavaScript kódbázisok nehezen karbantarthatók
- Az ECMAScript szabványosítás jó irányba ment, de nem követték elég gyorsan az implementációk

➢ **Készítsünk egy új programozási nyelvet, ami a JavaScript programozóknak szól és megoldást nyújt az egységes kezelésre!**

Anders Hejlsberg

TURBO PASCAL

Borland® Delphi™

C#

TypeScript

“You can write large programs in JavaScript.
You just can’t maintain them.”

Anders Hejlsberg

Áttekintés

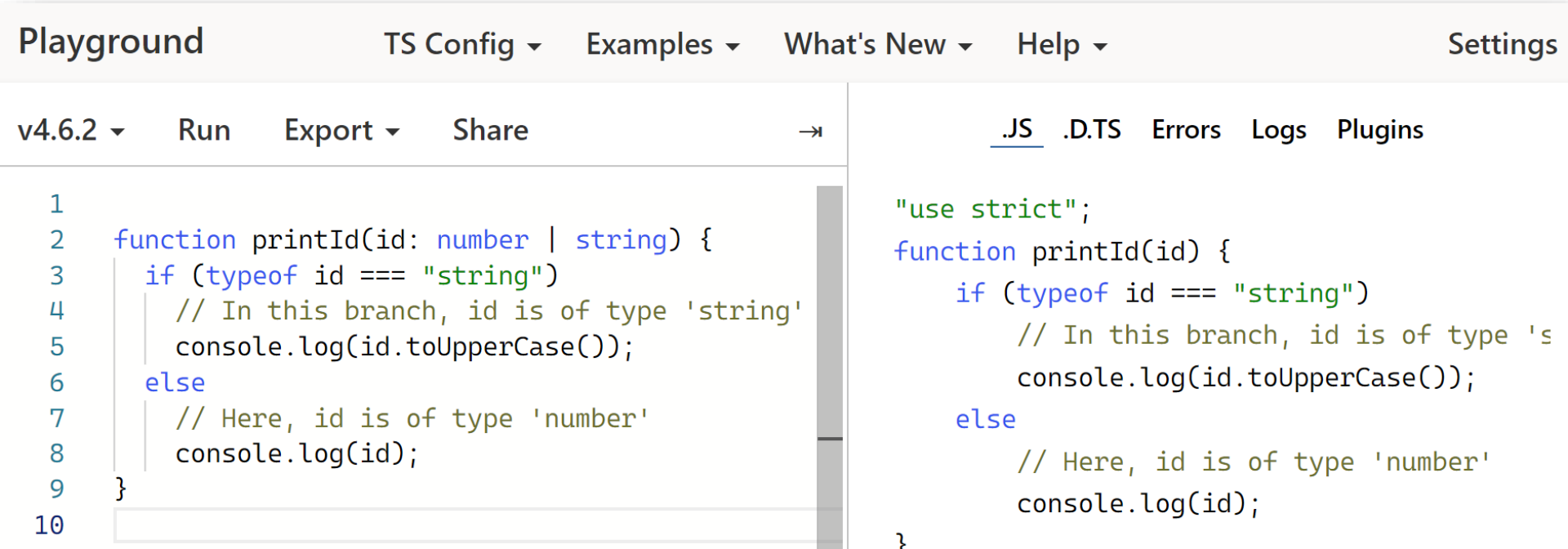
- A TypeScript a JavaScript típusos szuperszetje
  - Minden JavaScript egyben Typescript is
  - A dinamikus működés **kiegészítése** statikus típusinformációkkal
- A TypeScript első verziója 2012-ben került kiadásra
- A Microsoft ingyenes, open source terméke
  - https://github.com/Microsoft/TypeScript
  - https://www.typescriptlang.org/

Célközönség, elterjedtség

- A JavaScript programozóknak könnyű belevágni
- Objektumorientált paradigmákat erősen követi
- Erősen, okosan típusos
- JavaScriptre fordul, így mindenhol fut, ahol JavaScript is fut
  - Konfigurálható, hogy melyik ECMAScript verzióval legyen kompatibilis a lefordult kód
- A TypeScript fejlesztése erősen visszahat az ECMAScript szabványosításra

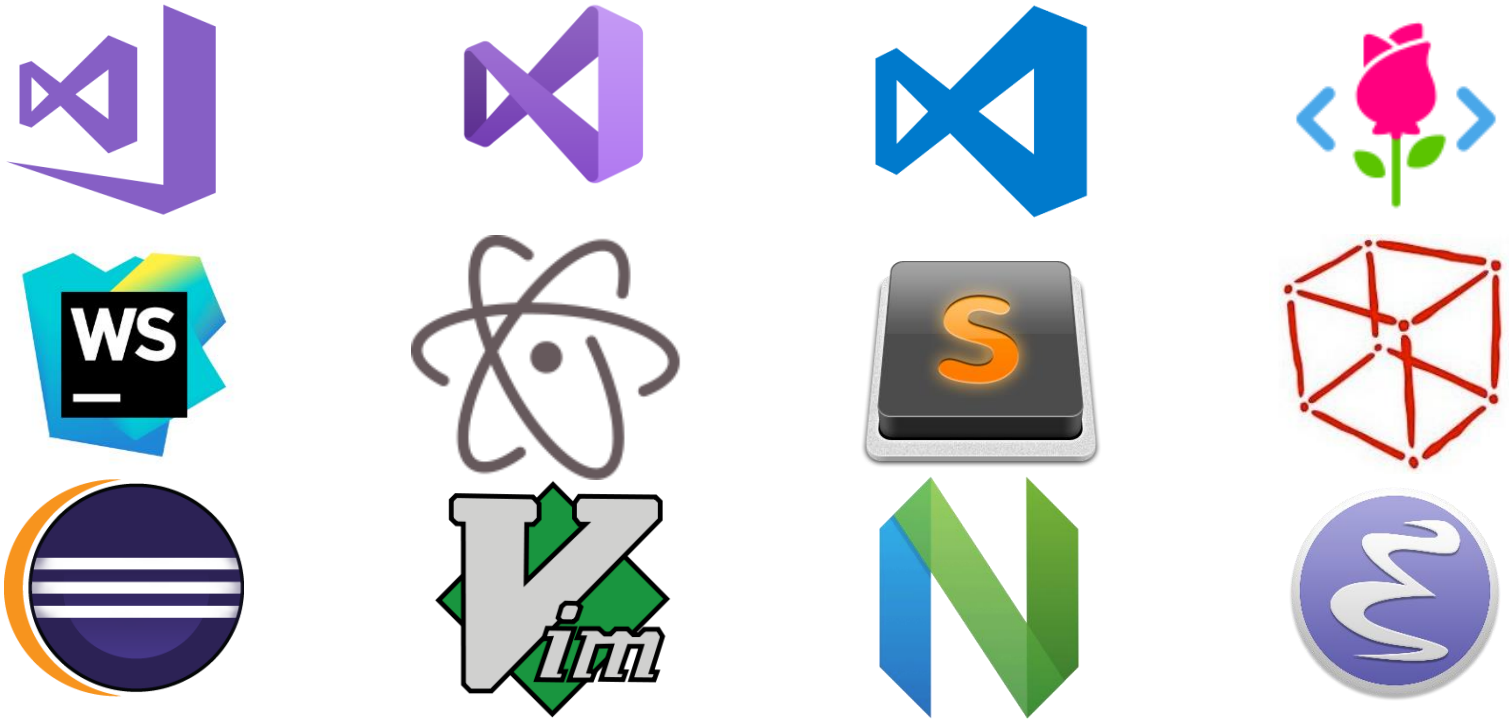
TypeScript futtatása

- A TypeScript fordító (TypeScript Compiler – **tsc**) egy **source-to-source compiler**, vagy **transpiler**
- A fordító a TypeScriptből JavaScriptet készít, az fut a böngészőben
  - > https://www.typescriptlang.org/play



Fejlesztőeszközök

- A TypeScript széles körben támogatott fejlesztőeszközök terén
- IDE támogatás:



Fejlesztéstámogatás

- Típusdefiníciók, fordító-kiegészítők, osztálykönyvtárak beszerzése
  - NPM (NodeJS Package Manager), Yarn
- Csomagolás, disztribúció-előállítás
  - Webpack, MSBuild, NuGet, Browserify, JSPM
- Automatizálási szkriptek
  - Gulp, Grunt
- TSC command-line interface (CLI)

Mit ad a TypeScript

- Kihasználhatjuk a statikus típusellenőrzést
  - A fordítási mechanizmus miatt **fordítási** idejű hibákat kapunk **futási** idejű hibák helyett
  - A **statikus típusosság** miatt igazi IntelliSense lehetséges
- Elrejti a JavaScript “furcsaságait”
- Használhatunk még nem szabványos JavaScript elemeket, amikből fordítás után sima JavaScript lesz
- Jelentősen javul a kód karbantarthatósága, olvashatósága
  - Ezáltal nagy kódbázis karbantartását is lehetővé teszi

Fordítási hibák (1)

program.js:

```
function Greeter(greeting) {
  this.greeting = greeting;
  this.greet = function () {
    return "<h1>" + this.greeting + "</h1>";
  }
};

var greeter = new Greeter("Hello, world!");
document.body.innerHTML = greeter.greet;
```

Mi a hiba?

Fordítási hibák (2)

program.ts:

```
function Greeter(greeting) {
  this.greeting = greeting;
  this.greet = function () {
    return "<h1>" + this.greeting + "</h1>";
  }
};

var greeter = new Greeter("Hello, world!");
document.body.innerHTML = greeter.greet();
```


```
[ts] Type '() => string' is not assignable to type 'string'.
(property) Element.innerHTML: string
document.body.innerHTML = greeter.greet;
```

Statikus típusosság (1)

- A megszokott magasszintű nyelvekhez hasonlóan statikus típusosságra is tudunk hagyatkozni
- A típusosság lehet **explicit** (konkrétan megadjuk a típust) vagy **implicit** (a kontextusból egyértelműen következik)
- A típushibák csak fordítási/fejlesztési időben derülnek ki, futási idejű hibáink a dinamikus környezet miatt továbbra is lehetnek

Statikus típusosság (2)

```
var num = 10;
num = 1.0;
num = "alma";
```

```
[ts] Type '"alma"' is not assignable to type 'number'.
var num: number
```

```
var num2: number;
num2 = "kutya";
```

```
[ts] Type '"kutya"' is not assignable to type 'number'.
var num2: number
```

Statikus típusosság (3)

- Megtarthatjuk a dinamikus típusosságot (any típus)

```
var something;
something = "";
something = 4;
```

- Visszatérhetünk dinamikus típusosságra

```
var num = 6;
num = <any>"kutya";
num = "kutya" as any;

var num: any = 6;
num = "kutya";
```

Statikus típusosság (4)

- A TypeScript **any** típusa a dinamikus típust reprezentálja
  - Az **any** bármilyen értéket felvehet és bárminek értékül adható
  - A jó kód nem tartalmaz **any**-t
  - Nincs IntelliSense, visszatérünk a futási időben felbukkanó típushibákhoz
  - A fordítónak megadható, hogy változó ne lehessen implicit **any** típusú (`noImplicitAny` flag)

Statikus típusosság (5)

- A TypeScript nagyon komplex típusrendszerrel rendelkezik
  - Osztályok, absztrakt osztályok, interfészek, öröklés
  - Enum típusok
  - String literálok
  - Type inference (típusok kikövetkeztetése, strukturális típusosság)
  - Genericitás, generikus típusok, függvények
  - Implicit interfészmegvalósítás
  - Unió- és metszettípusok

Strukturális típusosság (1)

- **Strukturális típusosság**: egy A objektum a B típussal strukturálisan kompatibilis, ha A megvalósítja a B által leírt **strukturális interfész**t
- **Strukturális interfész**: egy típus publikusan elérhető tagváltozóinak, függvényeinek halmaza

- A nyelv célja, hogy a script nyelvekre jellemző kreatív típus kavalkádot keret közé emelje a kifejező erő csökkentése nélkül

Strukturális típusosság (2)

```
interface Named {
    name: string;
}
class Person {
    id: string;
    name: string;
}

let john: Named = new Person();
```

Strukturális típusosság (3)

```
interface Named {
    name: string;
    shortName: string;
}
class Person {
    id: string;
    name: string;
}

let john: Named = new Person();
```

```
[ts]
Type 'Person' is not assignable to type 'Named'.
  Property 'shortName' is missing in type 'Person'.

let john: Named
```

Változódeklarációk (1)

- A **`var`** kulcsszóval függvényhez (nem blokkhoz) kötött változót hozhatunk létre
  - A külső függvény a globális névtér
- A **`let`** és **`const`** kulcsszavakkal blokkhoz kötött változókat hozhatunk létre, a **`const`** immutábilis
- A típust explicit megadhatjuk a változó neve után, vagy az értékadásból egyértelműen kiderül
  - Különben a típus **`any`** lesz

Változódeklarációk (2)

```
let var1 = "Bodri";          // string típusú változó
var1 = 6;                    // Error: Type '6' is not
                                  assignable to type 'string'.

let var2: any = "Fülöp";     // explicit any típusú változó
var2 = 8;                    // OK
const var3;                  // Error: 'const' declarations
                                  must be initialized.

const var4 = "Kókusz";       // string típusú konstans
var4 = "Banán";              // Error: Cannot assign to 'var4'
                                  because it is a constant
                                  or a read-only property.


if (Math.random() < 0.5) {
    let var5 = "Morzsi";
    var var6 = "Puffancs";
}

console.log(var5);           // Error: Cannot find name 'var5'.
console.log(var6);           // OK
```

Ismétlés, Javascript

- **undefined**
  - Inicializálatlan a változó
  - Nem létező property
  - -> NaN
- **null**
  - Üres, nem létező érték
  - Explicit be kell állítani
  - -> 0
- Mindkettő *falsy*
- Mindkettő egyben típus is
  - BUG: typeof(null) -> object

```
var a:any = {}
console.log(a.empty)

var b:any
console.log(b)
```

```
[LOG]: undefined
[LOG]: undefined
```

```
var c:any = null
console.log(c)
```

```
[LOG]: null
```

```
console.log( undefined == null )
console.log( undefined === null )
```

```
[LOG]: true
[LOG]: false
```

Null ellenőrzés nélkül

- Alapértelmezetten a változók lehetnek **null**-ok és **undefined**-ok is
  - A fordító nem szól és ez okozhat hibákat

```
let num: number;
function f(n: number) {
    console.log(n)
}
f(num);
num = 6;
f(num);
```

```
[LOG]: undefined
[LOG]: 6
```

Null ellenőrzés 1

- A fordító `strictNullChecks` flag-jének beállításával a **`null`** és **`undefined`** típusok nem lesznek többé részhalmazai az összes többi típusnak

```
let num: number;
function f(n: number) {
    // ...
}
f(num);          // Error: Variable 'num' is used before
                        being assigned.
num = null;      // Error: Type 'null' is not assignable
                        to type 'number'.
num = 6;         // OK
f(num);          // OK
```

Null ellenőrzés 2

- A fordító `strictNullChecks` flag-jének beállításával a **`null`** és **`undefined`** típusok nem lesznek többé részhalmazai az összes többi típusnak

```
let num: number | null;
function f(n: number) {
    // ...
}
f(num);          // Argument of type 'number | null' is not
                    assignable to parameter of type 'number'.
                    Type 'null' is not assignable to type 'number'
num = null;      // OK
num = 6;         // OK
f(num);          // OK
```

Típus szűkítés (*narrowing*)

- A fordító követi a típus ellenőrzéseket és ennek megfelelően végzi az ellenőrzést
  - Például itt tudja, hogy az ‚x' változó nem lehet null az else ágban és biztonságosan hívható rajta a funkció

```
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    // x is not null -> x: string
    console.log("Hello, " + x.toUpperCase());
  }
}
```

Type assertion

- “Típus bizonygatás”: nem ekvivalens a típuskonverzióval (cast), ugyanis valójában csak a compilernek szól, nem történik futási idejű típusmódosítás vagy -ellenőrzés
  - A JS dinamikusságából adódik, hogy szükség van rá
  - A futási idejű hibákat nem küszöböli ki
  - Két ekvivalens szintaxis:

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
strLength = (someValue as string).length;
```

Non-null assertion: !

- A fordítónak jelzi, hogy úgy kezelje az értéket, hogy az biztos nem null vagy undefined
  - > **Nincs futás idejű ellenőrzés**, csak a fordító működését befolyásolja!

```
function doSomething(x: string | null) {
    console.log("Hello, " + x.toUpperCase());
}
```

```
Object is possibly 'null'.
```

```
function doSomething(x: string | null) {
    // assert = we know better, this is not null for sure
    console.log("Hello, " + x!.toUpperCase());
}

doSomething(null)
```

```
[ERR]: "Executed JavaScript Failed:"
[ERR]: Cannot read properties of null
(reading 'toUpperCase')
```

Opcionális láncolás

- A ?. („Elvis”) operátorral opcionálisan érhetünk el tagokat változókon, ha a változó nem null vagy undefined értékű (különben a visszatérés undefined)

```
if (foo && foo.bar && foo.bar.baz) {
  // ...
}

if (foo?.bar?.baz) {
  // ...
}
```

Null összefűzés

- Null összefűzéssel (??, *null coalescing*) olyan kifejezéseket gyártunk, amelyek a jobb oldalt adják vissza, ha a bal oldal null vagy undefined

```
let x = (foo !== null && foo !== undefined) ?
  foo :
  bar();

let x = foo ? foo : bar();

let x = foo ?? bar();
```

Enumok

- Az enum sorszámozása 0-tól növekszik, kivéve, ha explicit növeljük

```
enum DogKind { Pitbull = 1, Terrier, Corgi };

let dog = DogKind.Pitbull
console.log(dog)
// reverse enum: number -> string
console.log(DogKind[dog])
```

```
[LOG]: 1
[LOG]: "Pitbull"
```

- String alapú enumok

```
enum Direction {
  Up = "fel",
  Down = "le",
  Left = "balra",
  Right = "jobbra",
}

let dir = Direction.Up
console.log(dir)
```

```
[LOG]: "fel"
```

Szövek literálok

- Készíthetünk típust konstans stringekből is
  - > String literálok uniója

```
function printKind( kind: "Pitbull" | "Terrier" | "Corgi" )
{
    console.log(`this dog is a ${kind}`)
}

printKind("Terrier")
printKind("corgi")
```

```
[LOG]: "this dog is a Terrier"
```

- Fordítás idejű hibát kapunk, ha ismeretlen szöveget használunk!
- Figyeljük meg: a kind paraméternek nincs nevesített típusa!

Szám literálok

- Készíthetünk típust konstans számokból is
  - Számok uniója

```
function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```

- Literálokat kapcsolhatunk más típusokhoz

```
function calculate(width: number | "auto") {
    if(width === "auto")
        return Math.random() * 100
    else
        // type narrowing: we know width is a number
        return width * width
}
calculate(42)
calculate("auto")
calculate("automatic")
```

Típus annotáció

- Strukturális típusosság: **nem a típus neve számít, hanem a benne lévő tagok!**
  - A tagokat több féle módon megadhatjuk
  - Type annotation: { } jelek között felsorolva, név nélkül

```
// The parameter's type annotation is an object type
function print(person: { name: string; age: number }) {
    console.log(`${person.name} is ${person.age} year(s) old`)
}
print({name: "Luke", age: 26})
```

Opcionális értékek: ?

- Paraméterek vagy propertyk lehetnek opcionálisak, ilyenkor **undefined** értéket vesznek fel

```
function print(name: string, age?: number)
{
    console.log(`${name} is ${age ?? "unknown"} years old`)
}

print("Yoda")
```

```
[LOG]: "Yoda is unknown years old"
```

BUG

```
function print(person: { name: string, age?: number })
{
    if( person.age )
        console.log(`${person.name} was born in ${new Date().getFullYear() - person.age}`)
    else
        console.log(`${person.name} is here forever`)
}

print({name: "Bill Gates", age: 67})
```

```
[LOG]: "Bill Gates was born in 1955"
```

Opcionális értékek unióval

- A típust explicit kiegészítjük a **undefined** értékkel
  - > A typeof is type guardként működik!

```
function print(person: { name: string, age: number | undefined })
{
    if( typeof(person.age) !== "undefined")
        // we know here that age is a number
        console.log(`${person.name} was born in ${new Date().getFullYear() - person.age}`)
    else
        console.log(`${person.name} is here forever`)
}

print({name: "Bill Gates", age: 67})
```

```
[LOG]: "Bill Gates was born in 1955"
```

Unió típusok

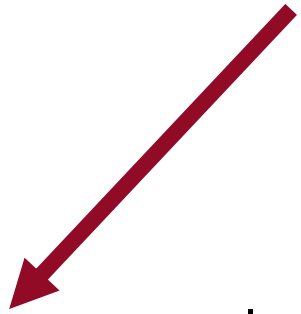
- Meglévő típusokból állíthatunk össze új típusokat
  - > Azt jelent, hogy az objektum vagy ilyen vagy olyan propertykkel rendelkezik (strukturális típusosság)

```
function printId(id: number | string) {
  if (typeof id === "string")
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  else
    // Here, id is of type 'number'
    console.log(id);
}
```

```
function welcomePeople(x: string[] | string) {
  if (Array.isArray(x)) {
    // Here: 'x' is 'string[]'
    console.log("Hello, " + x.join(" and "));
  } else {
    // Here: 'x' is 'string'
    console.log("Welcome lone traveler " + x);
  }
}
```

Unió típusok

Unió típusú paraméter



```
type sizes = 'small' | 'medium' | 'large';

function increaseSome(value: number, param: number | sizes) {
    if (typeof param === "number") {
        return value += param * Math.random();
    } else switch (param) {
        case 'small':  return value *= 1 + Math.random() * 5;
        case 'medium': return value *= 1 + Math.random() * 10;
        case 'large':  return value *= 1 + Math.random() * 50;
        default: return value;
    }
}
```

Fordítási idejű következtetés: a típust a fordító a kifejezésfából egyértelműen kikövetkezteti (ha nem number, csak sizes lehet)

Type alias

- A típus annotációknak nevet adunk

```
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
type ID = number | string;
```

```
type bool = false | true;
```

Interfészek TypeScriptben

- Interfész definiálhat:
  - Tagváltozót
  - Függvényt
  - Függvényszignatúrát
  - Konstruktorszignatúrát
  - Indexelhető típust

Interfész hasonló a type aliashoz

- Tagokat definiálunk benne névvel, típussal

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
type ID = number | string;
```

```
type bool = false | true;
```

Interfészek – strukturális típusosság

```
let myObj = { size: 10, label: "Size 10 Object" };

function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}
printLabel(myObj);

interface LabelledValue {
    label: string;
}

function printLabel2(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

printLabel2(myObj);
```

`myObj` implicit megvalósítja a `LabelledValue` interfészt

Funkciót leíró típusok

- A funkció be-kimeneti típusai meghatározzák a funkció-típust: type alias vagy interfész

```
function filter(items: string[], query: string)
{
    return items.filter(value=>value.includes(query))
}
console.log(filter(['Vader', 'Luke', 'Yoda'], 'a').join(','))
```

```
type filterFunc = (items: string[], query: string) => string []
let f1: filterFunc = filter
```

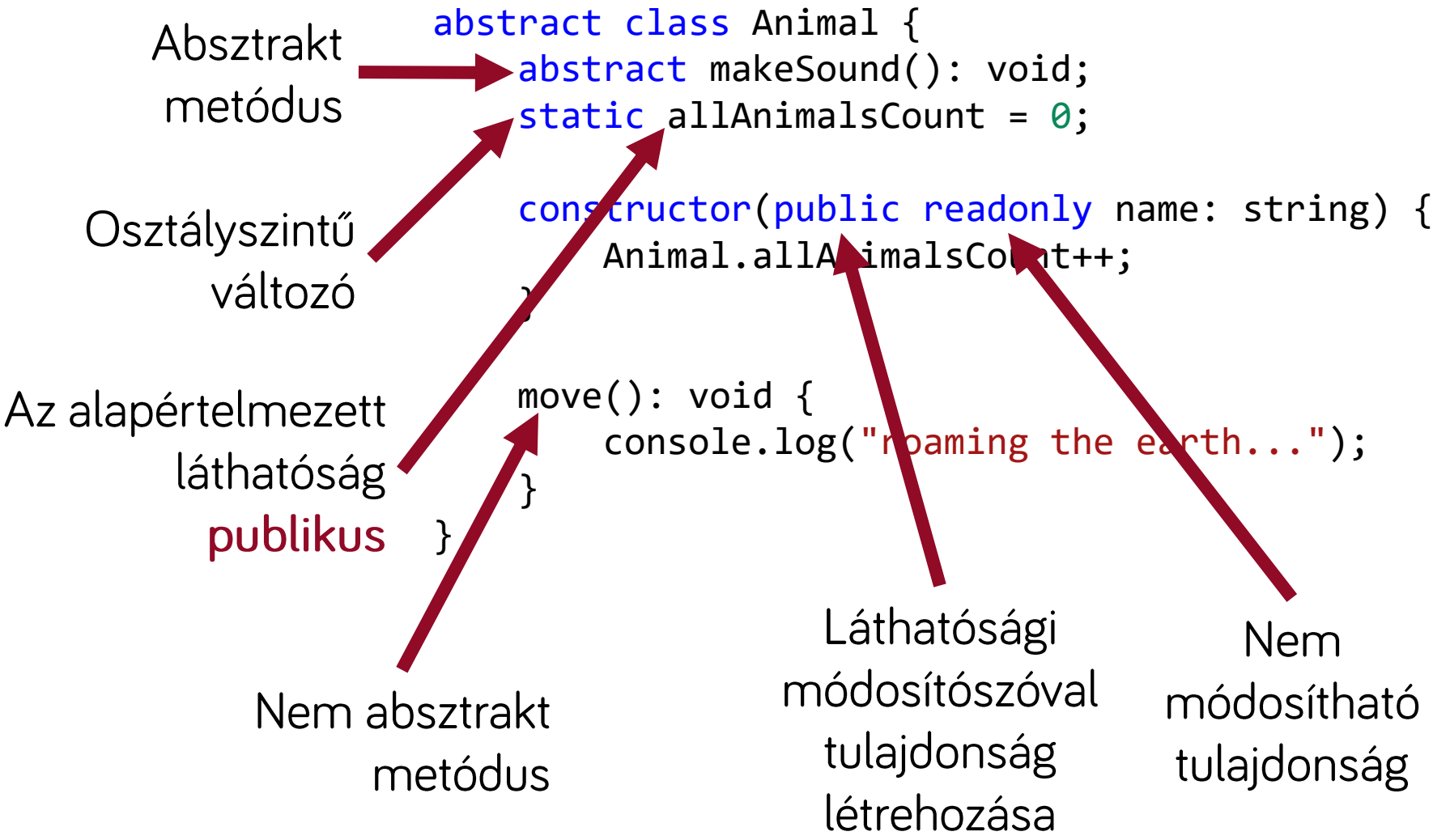
```
interface filterInt
{
    (items: string[], query: string): string [];
}
let f2: filterInt = f1
let f3: filterInt = filter

console.log(f1(['Vader', 'Luke', 'Yoda'], 'a').join(','))
```

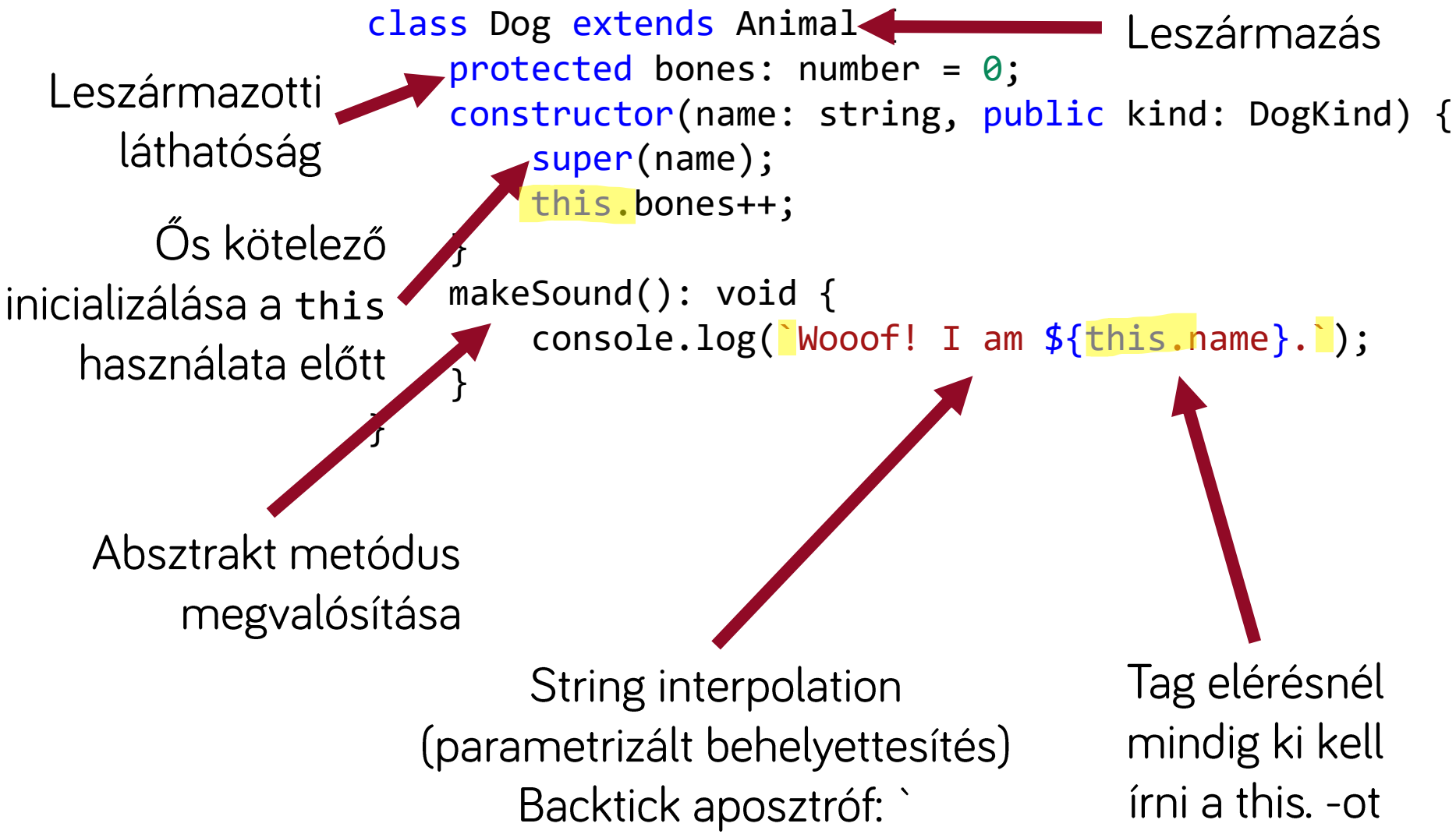
Osztályok áttekintése

- Támogatottak:
  - Osztályok
  - Interfészek (explicit- és implicit megvalósítás)
  - Absztrakt osztályok
  - Öröklés
  - Láthatósági módosítószók
  - Osztályszintű változók és függvények
- Nem támogatottak:
  - Valódi metódus overloading
  - Valódi többszörös öröklés
  - Típusonként több konstruktor

Osztályok, öröklés, tulajdonságok (1)



Osztályok, öröklés, tulajdonságok (2)



Accessors (~Java/C# property)

```
let passcode = "secret passcode";
class Employee {
    private _fullName: string;
    get fullName(): string {
        return this._fullName;
    }
    set fullName(newName: string) {
        if (passcode == "secret passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let employee = new Employee();
employee.fullName = "Bob Smith";
console.log(employee.fullName);
```

`get` és `set` kizárólag azonos láthatósággal, azonos típussal szerepelhet, de elegendő csak az egyik

A használat látszólag megegyezik a sima mező használatával

Indexerek (1)

- JavaScript hozomány, **tagok szöveges** elérése
  - Mint egy dictionary

```
let foo: any = {};
foo['Hello'] = 'World';
console.log(foo['Hello']); // World
```

- Az indexelés szöveggel történik, toStringet hív a JavaScript
  - JavaScript tetszőleges típuson hív toStringet, a TypeScript csak a **number**t engedi meg indexerként

```
let foo: any = {}
foo['1'] = 'Luke'
foo[1] = 'Darth';
console.log(foo['1']); // Darth
```

Indexerek (2)

- Explicit deklarálhatunk indexert
  - > Minden tagnak meg kell felelnie az indexer típusának!
  - > Indexer paraméter csak string (esetleg number)
  - > Nincs get/set, nem accessor!

```
interface Foo {
    [member: string] : number | string

    x: number;
    y: string; // OK
}
```

```
interface Foo {
    [member: string] : number

    x: number;
    y: string; // ERROR: Property `y` must be of type number
}
```

- Az "ismeretlen" tagok így az indexeren keresztül érhetőek el - nem típusos!

Privát mezők TypeScriptben

- Csak fordítás időben van tiltva a hozzáférés, futás időben JavaScriptben, indexelt property hozzáféréssel elérhető

```
class MySafe {
  private secretKey = 12345;
}

const s = new MySafe();

// Not allowed during type checking
console.log(s.secretKey);
// Property 'secretKey' is private and only accessible within class 'MySafe'.

// OK
console.log(s["secretKey"]);
```

Erősen védett mezők:

- JavaScriptben a # jellel hozhatók létre erősen védett privát mezők

```
class MySafe {
  #secretKey = 12345;
  print()
  {
      console.log(this.#secretKey)
  }
}

const s = new MySafe();

// Not allowed during type checking
console.log(s.#secretKey);
// Property '#secretKey' is not accessible outside class 'MySafe'
// because it has a private identifier.

console.log(s["#secretKey"]);
// Element implicitly has an 'any' type because expression of type
// '"#secretKey"' can't be used to index type 'MySafe'.
// Property '#secretKey' does not exist on type 'MySafe'.
```

Metódus overload

- JavaScriptben elhagyhatjuk a metódus bemenő paramétereit, ezek **undefined** értéket vesznek fel
- Azonos nevű metódusokból nem hozhatunk létre többet, akkor sem, ha paramétereikben különböznek -> **nincs overload**
  - Ezért sem lehet több konstruktora egy osztálynak
- Egy metódusnak lehet több overload szignatúrája
- Egyetlen implementáció tartozik hozzá
- **Az implementációs szignatúra nem hívható közvetlenül**

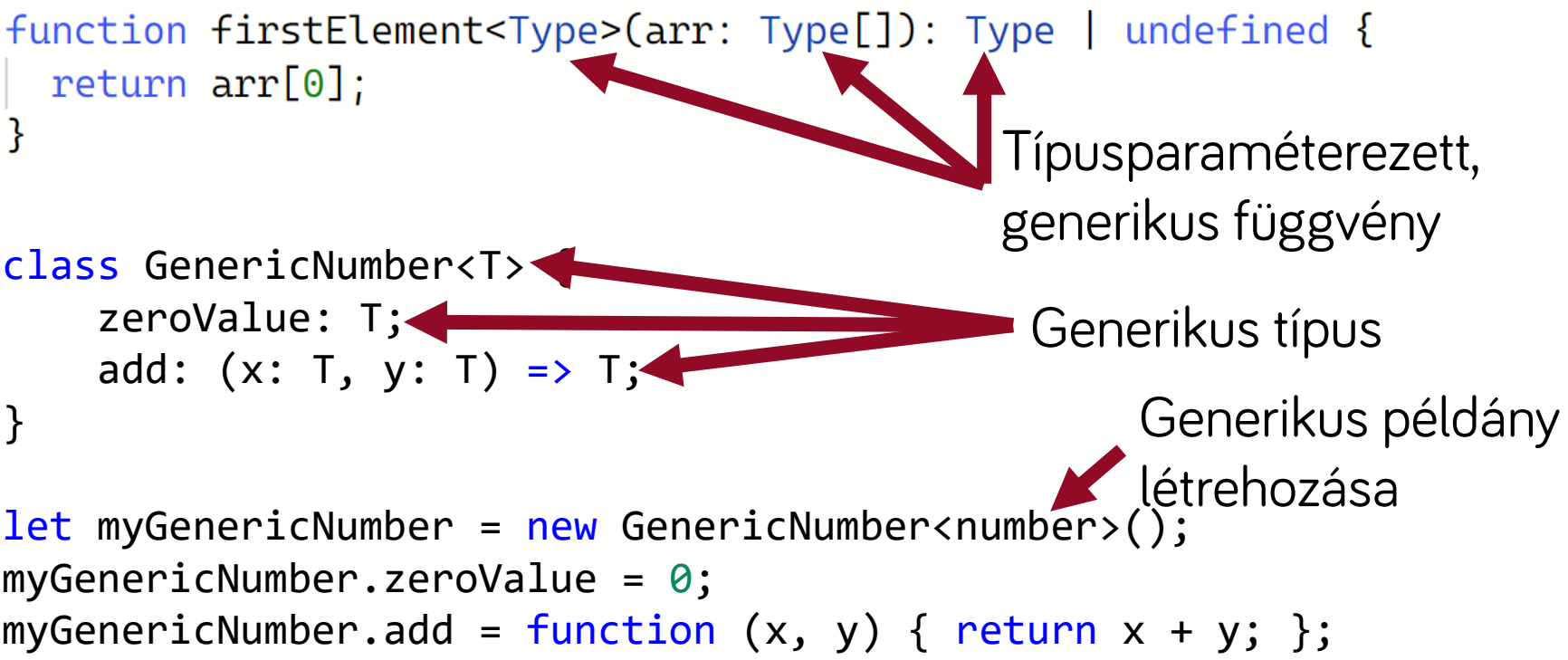
Overload szignatúrák

- A dátum létrehozása egy vagy három paraméterrel
- A paraméterek típusai kompatibilisak, lehetnek opcionálisak, akkor undefined értéket vesznek fel

```
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else {
    return new Date(mOrTimestamp);
  }
}
const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3);
// No overload expects 2 arguments, but overloads do exist that expect
// either 1 or 3 arguments.
```

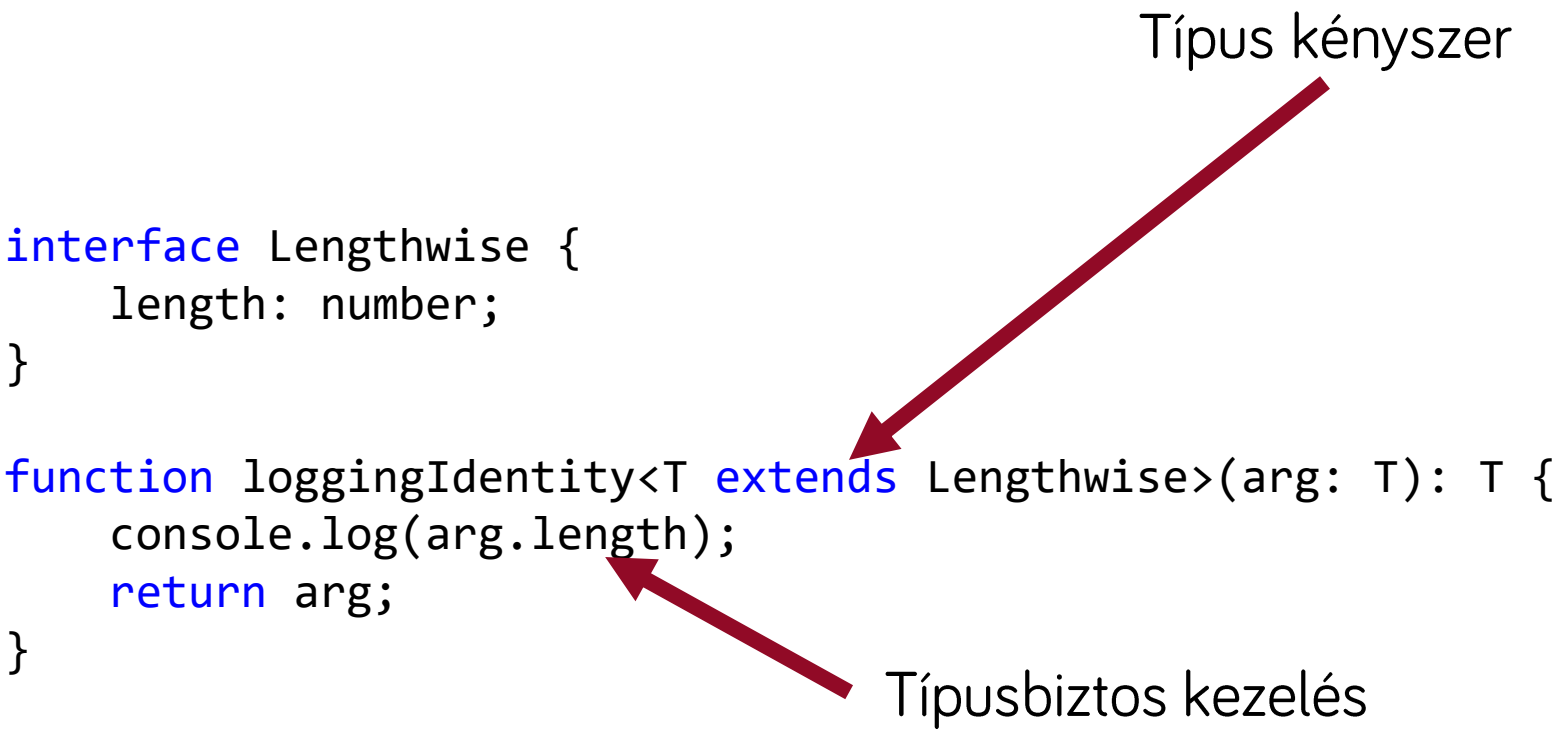
Generikus típus paraméterek

- A korábban tanultakkal analóg módon működik a generikus típusparaméterek használata



Generikus típus kényszerek

- Generikus paraméternek lehet kényszere



Metszet típusok

- Típusok bővíthetőek egymással, az új típus a régi típusok tagjait mind tartalmazza (szemben az unióval)

```
interface Sizable {
  size: number;
}
type Labeled = {
  label: string;
}

type SizableLabel = Sizable & Labeled

function print(title: SizableLabel)
{
    console.log(`label ${title.label} is size ${title.size}`)
}

print({label: "Morning", size: 10})
print({lable: "Morning", size: 10})
```

```
Argument of type '{ lable: string; size: number; }'
is not assignable to parameter of type
'SizableLabel'.
  Object literal may only specify known properties,
but 'lable' does not exist in type 'SizableLabel'.
Did you mean to write 'label'?
```

Elkenés

- Az elkenés (…, *spread*) segítségével könnyen bonthatunk szét és építhetünk össze objektumokat tagjaiból és tömböket más tömbökből

```
let defaults = { food: "spicy", price: "$$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };

let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

Dekorátorok

- A dekorátorok az aspektus-orientált programozás kellékei, más nyelvekben **attribútum**ként vagy **annotáció**ként ismert
  - Metaprogramozás: olyan kódot írunk, ami feldolgoz olyan kódot, amely felhasználói adattal dolgozik (programozás)
- A dekorátor megváltoztatja a dekorált nyelvi elem (például metódus, osztály) működését

Kiinduló kód

```
class Dog {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    bark() {
        console.log(`Woof, my name is ${this.name}.`);
    }
}


const dog = new Dog("Skip");
dog.bark();
```

```
[LOG]: "Woof, my name is Skip."
```

Dekorált kód

- Naplózzuk az ugatást!

```
class Dog {
    name: string;
    constructor(name: string) {
        this.name = name;
    }

    @loggedMethod
    bark() {
        console.log(`Woof, my name is ${this.name}.`);
    }
}



const dog = new Dog("Skip");
dog.bark();
```

dekorátor alkalmazása

```
[LOG]: "LOG: Entering method."
[LOG]: "Woof, my name is Skip."
[LOG]: "LOG: Exiting method."
```

Metódus naplózó dekorátor

- Az eredeti metódus *helyett* az alábbi metódus hívódik meg, kiegészítve a kontextussal

```
function loggedMethod(originalMethod: any, context: ClassMethodDecoratorContext) {
    const methodName = String(context.name);

    function replacementMethod(this: any, ...args: any[]) {
        console.log("LOG: Entering method {methodName}.")
        const result = originalMethod.call(this, ...args);
        console.log("LOG: Exiting method {methodName}.")
        return result;
    }

    return replacementMethod;
}
```

Dekorátor paraméterek

- A függvény egy dekorátor függvényt ad vissza

```
@loggedMethod("Dog")
bark() {
    console.log(`Woof, my name is ${this.name}.`);
}
```

```
[LOG]: "Dog - Entering method 'bark'."
[LOG]: "Woof, my name is Skip."
[LOG]: "Dog - Exiting method 'bark'."
```

```
function loggedMethod(headMessage = "LOG:") {
    return function actualDecorator(originalMethod: any, context: ClassMethodDecoratorContext) {
        const methodName = String(context.name);

        function replacementMethod(this: any, ...args: any[]) {
            console.log(`${headMessage} - Entering method '${methodName}'.`)
            const result = originalMethod.call(this, ...args);
            console.log(`${headMessage} - Exiting method '${methodName}'.`)
            return result;
        }

        return replacementMethod;
    }
}
```

A dekorátor egy függvény

- Metódus, osztály, mező, setter, getter megváltoztatásához
- Lecserélhetjük a dekorált elemet
- Beleszólhatunk az inicializálásba

```
type Decorator = (
  value: DecoratedValue, // only fields differ
  context: {
    kind: string;
    name: string | symbol;
    addInitializer(initializer: () => void): void;

    // Don't always exist:
    static: boolean;
    private: boolean;
    access: {get: () => unknown, set: (value: unknown) => void};
  }
) => void | ReplacementValue; // only fields differ
```

Objektumok létrehozása

```
interface IPerson {
    name: string
    age: number
}

let person: IPerson = { name: "Darth Vader", age: 43 }
let person2: {name:string, age:number} = { name: "Darth Vader", age: 43 }

person = person2

class Person
{
    name: string
    age: number

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }
}

person = new Person("Luke Skywalker", 24)
```

Tömbök létrehozása

```
let people: IPerson [] = [
    { name: "Darth Vader", age: 43 },
    { name: "Luke Stywalker", age: 24 } ]

let people2: Array<Person> = [
    { name: "Darth Vader", age: 43 },
    { name: "Luke Stywalker", age: 24 } ]

let people3: Array<{name:string, age:number}> = [
    { name: "Darth Vader", age: 43 },
    { name: "Luke Stywalker", age: 24 } ]

let people4: Array<typeof person> = [
    { name: "Darth Vader", age: 43 },
    { name: "Luke Stywalker", age: 24 } ]
```

Egyéb konstukciók

- Típusmanipuláció…
- Szimbólumok
- Iterátorok és generátorok
- Névterek
- Mixinek
- Segédtípusok (Partial<T, U>, NonNullable<T>, …)
- Egyéb speciális típusok, például never
- … és még nagyon sok más

Modulok (1)

- A modulok az egységbezárást segítik, a logikailag összefüggő osztályok, objektumok, függvények, változók egy *logikai* fájlba helyezhetők
  - A modulokból ezek kifelé publikálhatók (export), kívülről pedig konzumálhatók (import)
  - Külső függőségek is ezt a mintát használják, így szeparálhatók a felelősségi körök

Modulok (2)

```
//components.ts
export class Processor { }
export class Memory { }
export class PC {
    constructor(public memory: Memory, public cpu: Processor) { }
}
export let composePC = (memory: Memory, cpu: Processor) =>
    new PC(memory, cpu);
```

A modulból exportált tagok

```
//office.ts
import * as Components from './components';
import PC from './components';
import { Memory, Processor } from './components';
import Rx from 'rx';

let obs = Rx.Observable.create<Components.PC>((obs) => obs.onNext(
    Components.composePC(new Memory(), new Processor())));
```

Saját importok

Külső függőség importja

A fordítás eredménye

- A fordítás eredményeképp előálló JavaScript **nem tartalmazza** a TypeScript által használt típusinformációkat
  - A lefordított JavaScript továbbra is dinamikusan típusos
- A meglevő JavaScript forrásainkat a hiányzó típusinformációkkal kiegészítve használhatunk külső JavaScript forrásokat is TypeScript kódban
  - A statikus típusrendszert megtartva

Típusdeklarációs fájlok

- A fordítónak megadható, hogy a JavaScript forrás mellett a típusinformációkat is exportálja
  - **.d.ts** kiterjesztésű fájlok
  - Kizárólag típusinformációt hordoznak
  - Nem hajthatók végre
  - Analóg a *C/C++ header fájlokkal*
- Külső JavaScript forrásokat használhatunk **típusosan**, ha rendelkezünk a .d.ts fájllal
  - TypeScriptből generálható
  - Kézzel is elkészíthető (a legtöbbet kézzel írják)

Generált típusdeklarációs fájl

kutya.ts:

```
class Kutya {
  constructor(private name: string) { }
  ugat() { console.log(`${name}: Woof!`)}
}
```

kutya.js

kutya.d.ts

Csak típusinformáció, nincs futtatható kód

```
declare class Kutya {
  private name;
  constructor(name: string);
  ugat(): void;
}
```

```
var Kutya = /** @class */ (function () {
  function Kutya(name) {
    this.name = name;
  }
  Kutya.prototype.ugat = function () { console.log(name + ": Woof!"); };
  return Kutya;
}());
```

Csak futtatható kód, nincs típusinformáció

Típusdeklarációs fájl beszerzése

- Hogyan szerezzük be a típusdeklarációs fájlokat?
  - Beépített támogatás van a *@types* npm csomagokra, a fordítónak megadható, mire van szükségünk:

```
// tsconfig.json:
{
    "compilerOptions": { /* ... */ },
    "typeAcquisition": {
        "enable": true,
        "include": [ "jquery", "react", "lodash" ]
    }
}
```

Típusdeklarációs fájl letöltése

- Hogyan szerezzük be a típusdeklarációs fájlokat?
  - Letölthetjük npm-ből a megfelelő *@types* scoped package-eket:

```
npm install --save @types/jquery
```

  - Így a fordító a node_modules/@types/jquery mappában találja a jQuery típusdeklarációt az osztálykönyvtár használatához

Példa: típusdeklarációk JQueryhez

- JQuery – jquery.d.ts típusdeklaráció (részlet):

```
// Type definitions for jQuery 1.10.x / 2.0.x
interface JQueryStatic {
    ajax(settings: JQueryAjaxSettings): JQueryXHR;
    ajax(url: string, settings?: JQueryAjaxSettings): JQueryXHR;
    (selector: string, context?: Element|JQuery): JQuery;
    (): JQuery;
}

interface JQuery {
    addClass(className: string): JQuery;
    attr(attributeName: string): string;
    attr(attributeName: string, value: string|number|null): JQuery;
    attr(attributes: Object): JQuery;
}
declare var jQuery: JQueryStatic;
declare var $: JQueryStatic;
```

IntelliSense típusdeklaráció alapján

- Így tehát a külső, JS-ben íródott (és használt) osztálykönyvtárakhoz is kapunk IntelliSense-t és fordítási idejű hibákat: