Multiplatform szoftverfejlesztés

TypeScript

JavaScript

JavaScript támogatás

- Van szabvány, és megkésve követjük is
- Fordítót használunk
  - Babel
  - TypeScript
  - Tipikusan ES6-ra fordítunk
- Ez meglepően jól működik
  - Ellentétben a HTML és CSS problémákkal

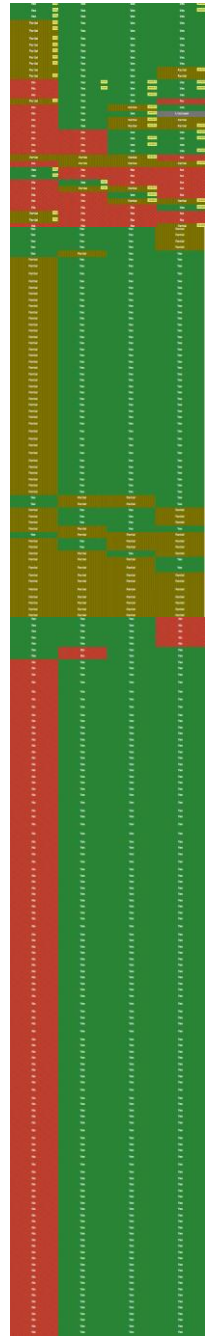
JavaScript támogatás

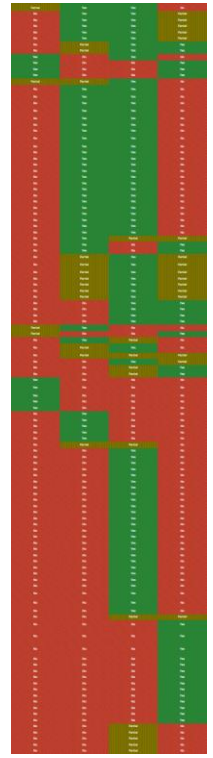
- Polyfillt használunk a nem implementált funkcionalitáshoz

```
if ( !String.prototype.startsWith )
{
    Object.defineProperty( String.prototype, 'startsWith', {
        value: function ( search, rawPos )
        {
            var pos = rawPos > 0 ? rawPos | 0 : 0;
            return this.substring( pos, pos + search.length ) === search;
        }
    } );
}
```

Jelenlegi támogatás

- Report: caniuse 2022 április
  - Csak JS
- Oszlopok
  - IE 11
  - Firefox
  - Chrome
  - iOS Safari
- Az jobb alsó részben JS API-k vannak
  - Web Bluetooth, WebUSB, stb.
- Firefox és Chrome (Edge) elöl





Gyengén típusos

- Vannak típusok
  - number, string, boolean, Object, Symbol, function, bigint, null, undefined
  - Csak futásidőben kerülnek ellenőrzésre
  - A runtime megpróbál mindenhol konvertálni
    - Csak végső esetben dob hibát
- De nem kell/lehet kiírni őket
  - Rövid, jól átlátható kódot eredményez
    - Amíg kicsi a program

Gyengén típusos

- Közepes és nagy szoftvereknél probléma
  - Nincs fordításidejű ellenőrzés, sokkal többet kell tesztelni
  - Problémás a tooling
    - Kódkiegészítés: nehéz javaslatokat adni a fejlesztőnek, hogy mit tud beírni
    - Ellenőrzés: kevesebb hibát lehet kielemezni
  - Nem látni a kódból, hogyan kell használni
    - Ez megoldható dokumentációval – /**…*/ széleskörben támogatott
- Mi csak a közepes és nagy szoftverekre koncentrálunk

TypeScript

Általában

Mi a TypeScript?

- Típusos JavaScript, a típusok opcionálisak
  - Minden JS egyben TS is
  - Amint beírunk típust valahova, az már csak TS
- Típust a változó neve után írjuk

```
function A( a, b ){ // .js, vagy .ts fájl is lehet
{
  return a + b;
}
function A( a: number, b: number ) // csak .ts fájlban lehet
{
  return a + b;
}
```

Tudásban TypeScript=JavaScript

- TS nem tud többet
- Ha kivesszük a típusokat, akkor JS-t kapunk
  - Ugyanúgy fog viselkedni a kód futásidőben
  - typeof és társai is csak a JS szintet hozzák
- Az egyetlen különbség, hogy van egy fordítási lépés
  - Ez nagyon fontos, akkor is használjuk, ha JS-t írunk (babeljs segítségével)
  - A cél, hogy átkódoljuk az új szabvány szerint megírt kódot a célplatformra (pl. ES5 IE11 miatt)
  - TS esetében ez a lépés kiveszi a típusokat is

Miért fontosak a típusok

- Kezdetleges dokumentáció
  - Sokszor lehet következtetni, hogy mit csinál
    - Névből – JS/TS
    - Paraméterek nevéből – JS/TS
    - Paraméterek típusából – csak TS
- Tooling
  - Kódkiegészítés
    - Kontextusfüggő: típus korlátozza a listát
  - Linter
- Fordításidejű kódellenőrzés
  - Hasonló linterhez, de sokkal hatékonyabb

Miért fontosak a típusok

- Tesztelés segítése
  - Típusok esetén a tesztelés költsége jelentősen csökken (akár felére)
    - A hibák jelentős része nem kerül bele a kódba
- Összességében a típusok csökkentik a költséget

OO paradigma

- Fontos: a típusok használata nem segít az objektum orientáltságon
  - Függetlenek egymástól
- JS támogatja az OO irányelveket
  - Vannak osztályok, egységbe zárás (encapsulation)
  - Belső működés elrejtése – absztrakció
    - Private (#) csak ES2019-től, TS-ben volt/van
  - Öröklés
  - Polimorfizmus – ez nincs JS-ben, sem TS-ben
    - Egy függvény viszont több típussal is tud működni
    - Az eredeti célt el tudja érni

OOP TS-ben

- Támogatottak
  - Osztályok
  - Interfészek (explicit- és implicit megvalósítás)
  - Absztrakt osztályok
  - Öröklés
  - Láthatósági módosítók (public, private, protected)
  - Osztályszintű változók és függvények
  - Enum típusok, string literálok, unió- és metszettípusok
- Nem támogatottak
  - Valódi metódus overloading
  - Valódi többszörös öröklés
  - Típusonként több konstruktor/azonos nevű függvény

Osztályok és öröklés

- A legtöbb keretrendszer nem osztály alapú
  - Régen nem voltak osztályok
  - Kezdő programozóknak egy akadály
  - this probléma nem segít
  - Komponens alapú fejlesztés
    - Kompozíciót használunk, nem öröklést
  - Egységbe zárást a komponens valósítja meg
    - Ami vagy osztály, vagy nem
- TS-től a típusosságot kérjük
  - Osztályokkal külön nem foglalkozunk
  - Ettől még sok kód osztályt fog használni

TypeScript

Típusok

Alaptípusok

- Az alaptípusok a JS alaptípusok, plusz

```
let a: number[]; // tömb
let b: [ number, string ]; // tuple
enum Color { Red, Green, Blue };
let c: Color; // enum
let d: any; // nincs ellenőrzés
let e: "red" | "green" | "blue"; // string literal
```

Összetett típusok

- Unió: string | number
  - Vagy egyik, vagy másik
  - Nagyon sokat használjuk
    - Mert azonos neve nem lehet függvényeknek
      - Például polimorfizmus megoldására
    - Nem a fordító dönti el, hogy melyiket kell hívni
      - Függvényen belül if-elünk
- Metszet: ObjA & ObjB
  - Minden A-ból és B-ből

Függvények, röviden

- Default és opcionális paraméterek

```
function fd( a: string = "hello", b?: string )
{
}
```

- undefined-ot kapunk, ha nincs megadva
  - Vagy kézzel azt adtak át
  - Tehát a default paraméter is lehet undefined
- Azonos a működés JS-sel
  - Nem fordul le, ha nem adunk meg egy kötelező paramétert – minden az, ami nem opcionális/default

Osztályok, röviden

- Konstruktorban tulajdonság

```
class C
{
  constructor( public name: string ) { }
}
```

- public, protected, private működik
  - De csak fordítás időben
  - #field is működik, ez futásidőben is
- readonly, static, abstract kulcsszavak
- Accessors: get és set

this

- TS nem oldja meg teljesen a this problémát
  - De segít rajta
- Nekünk kell megoldani
  - Minden callback-nél használjunk arrow function szintaktikát

```
setInterval( () =>
{
  // itt a this azonos a külsővel
}, 1000 );
```

Type guards

- Fordító követi a kód logikáját

```
function toS( x: string | number )
{
  if ( typeof x === "string" )
    return x;
  else
    return x.toFixed();
}
```

- Működik instanceof esetén is

Paraméteres típusok – Generics

- Használhatunk előre nem ismert típusokat

```
function concat<T>( a: T, b: T )
{
  return a.toString() + b.toString();
}
concat( 1, "2" );  // Error
```

- Osztályok és függvények is
- Több paraméter is lehetséges
- A fordító látja, hogy mivel hívjuk, nem kell megadni – mint C#, vagy C++

Paraméteres típusok – Generics

- Kényszerekkel

```
interface HasLength
{
  length: number;
}
function getTotalLength<T extends HasLength>( a: T, b: T )
{
  return a.length + b.length;
}
```

Interfészek – interface kulcsszó

- Objektum tulajdonság
- Objektum függvény
- Objektum konstruktor függvény
- Függvény
- Indexer

- Ezeket mind meg lehet adni interface nélkül is

```
interface HasLength<T>
{
  new(): T;
  length: number;
  getLength(): number;
}
interface Indexable
{
  [ key: string ]: string;
}
interface Action<T>
{
  ( param1: T );
}
var x: Action<string> =
  s => console.log( s );
```

Struktúrálisan típusos

- Két változó akkor azonos típusú, ha struktúrálisan azonos a típusuk
- Például

```
type SoN = string | number;
function FA()
{
  let a: SoN = 1;
  let b: number | string = a;
}
```

- A típus neve nem számít

Struktúrálisan típusos

- Ez igaz interfészekre és osztályokra is

```
interface IA
{
  a: string;
}
interface IB
{
  a: string;
}
```

- És minden más típusra
  - Ha kompatibilis, akkor fordul

Struktúrálisan típusos

- Függvények is követik a kompatibilitás elvét
- Még trükkös esetekben is

```
let x = ( a: string ) => { };
let y = ( a: string, b: string ) => { };
y = x; // OK
x = y; // Error
```

  - JS-ben mindenhol átadhatok kevesebb paraméterrel rendelkező függvényt

Modulok

És egyéb nyelvi elemek

Névterek (ritkán használt)

- Egy fordítási egységen belül

```
namespace NS
{
   export class C
   {
   }
}
```

- Használata: /// <reference path="x.ts"/>
- Kód darabolása a cél
  - Nagyon hasonló az osztály egységbe záráshoz

Modulok

```
module M
{
  export class C{}
}
```

- Ezt csak modul betöltővel lehet használni
  - import { MyClass } from 'my-class';
- Fordításnál állíthatjuk, hogy milyen kódot generáljon
  - CommonJS (Node.js)
  - RequireJS (AMD)
  - ...

Típusdeklarációs fájlok .d.ts

- Külső könyvtárakhoz van típusokat leíró fájl
- Fel kell tenni
  - npm i -D @types/jquery
  - Vagy letölteni kézzel
- Majd megmondani a fordítónak
  - /// <reference path="jquery/jquery.d.ts" />
  - Ezt a .ts fájlunkban, ahol használjuk
- Óriási gyűjtemény
  - https://definitelytyped.org/

Típusdeklarációs fájlok .d.ts

- Ezek sima .ts fájlok
  - De tipikusan nincs bennük olyan kód, ami benne marad fordítás után
  - Csak típusok leírása van bennük
- Tipikusan: type, declare, interface

```
type StringOrNumber = string | number;
declare class A
{
  private name: string;
}
```

Típusdeklarációs fájlok .d.ts

- Mi magunk is írhatunk .d.ts fájlt
- Célok
  - Könyvtárat írunk
    - Más fel fogja használni
    - JS-ként adjuk át, így a típusok eltűnnek belőle
  - Más nyelven készítettük a szervert
    - C# szerver típusait célszerű deklarálni .d.ts fájlban
    - Lehet automatikus folyamat
    - Protocol Buffer megoldás .d.ts fájlt is generálhat

Aszinkron programozás

async, await

Promise

- Egyre több API használ Promise-t
  - Ez egy osztály, ami támogatja
    - Több feliratkozót
    - Hívás-válasz mintát – mint egy függvényhívás
      - De például ismétlődő eseményekre nem alkalmas – nem egy esemény
    - Egységes hibakezelést
      - Van benne try-catch, nem kell kézzel beletenni
    - Láncolást: .then(valami).then(más)
- Felhasználása .then(callback)
  - Vagy .catch(callback)

Promise

- A sima callbackhez képest kényelmesebb
  - Mindennek azonos az interfésze
    - Nem kell tudni, hogy melyik paraméter is a callback
  - Azonos a hibakezelés is
- Nem tökéletes
  - A kód még mindig callbackekben van
- ES6-tól van
  - ES5-re fordításkor belefordítja a kódját

Promise – delay

- Egy példa a setTimeout Promise-ra alakítására

```
function delay( ms: number )
{
  return new Promise( ( resolve, reject ) =>
    setTimeout( resolve, ms ) );
}
```

- Visszadunk egy Promise-t
- Elindítunk egy timert
- Amikor lejár, meghívjuk a resolve-ot
  - Ami meghív minden .then-t, ami rá van téve

async, await

- Ha egy függvény Promise-t ad vissza
  - Beírhatunk elé egy awaitet
  - Feltéve, hogy async függvényben vagyunk

```
async function fa()
{
  await delay( 500 );
  console.log( "hello" );
}
```

- Az await utáni kód a .then-be kerül fordításkor
  - Minden await ponton elvágja a kódot a fordító

async

- Promise-t ad vissza (csak nem látszik)
- Akkor hívja meg a resolve-ot, amikor az utolsó sor is lefutott
- Meghívja a reject-et, ha kivétel keletkezik

```
async function fa()
{
  await delay( 500 );
  console.log( "hello" );
}
```

```
function fa()
{
  return new Promise( ( resolve, reject ) =>
  {
    delay( 500 ).then( () =>
    {
      console.log( "hello" );
      resolve();
    } );
  } );
}
```

Szálak

- JS-ben csak egy szál van, azon megy az egész
  - Ha szinkron minden, akkor megszakítás nélkül
  - Ha olyan API-t hívunk, ami később hív vissza, akkor kiütemezi a szálat
    - És folytatja ugyanazon a szálon, amikor visszatér
- Más nyelvekben (pl. C#) kontextus van
  - Azonos kontextusban kapjuk vissza a vezérlést
  - A fő szál, ami a UI-t futtatja egy külön kontextusban van egyedül
    - A fő szálon való aszinkron programozás egyszálú – nem kell szinkronizálni
  - Háttérszálak – például szerver kódban – egy kontextusban vannak együtt
    - Alapban többszálú, az aszinkron programozás nem változtat ezen

Kérdések?