TypeScript Úvod

kurz Vývoj progresívnych webových aplikácií

Eduard Kuric

STU FIIT

TypeScript

https://www.typescriptlang.org/

- Open-source statický typový programovací jazyk
  - stavia na JavaScripte, vyvíjaný a udržiavaný Microsoftom
- Používa JavaScript syntax
  - pridáva novú syntax a údajové typy
- Transpiluje sa do JavaScriptu
  - vo webovom prehliadači sa vykonáva JavaScript, nie TypeScript

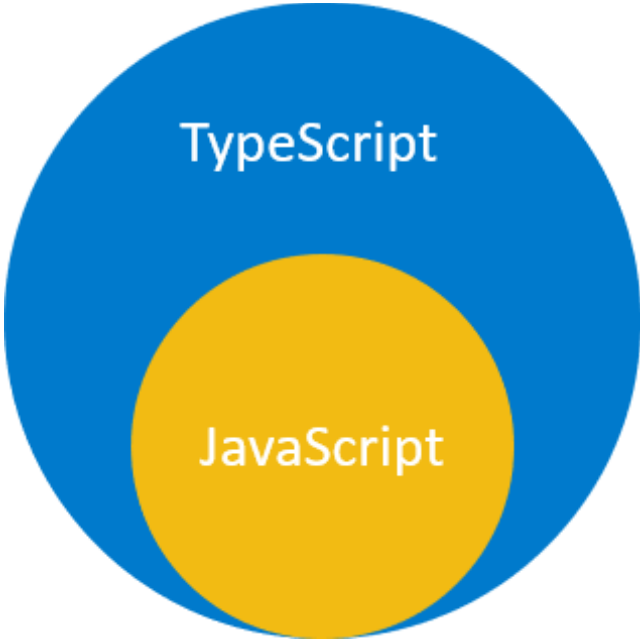
TypeScript playground

- **Online transpilátor:**
- https://www.typescriptlang.org/play

- Kompilátor vs transpilátor
- **Transpilátor** – kompilátor, ktorý tranformuje/konvertuje kód napísany v jednom jazyku do kódu v inom jazyku, ktorý má podobnú úroveň abstrakcie (alebo inej verzie rovnakého jazyka), transpilátory sú podmnožina kompilátorov,
- **Kompilátor** - prekladá kód z jedného jazyka do iného jazyka, spravidla na nižšej úrovni (človekom čitateľný kód na strojové inštrukcie).

JS kód = TS kód

- JavaScript kód (bez syntaktických chýb) je zároveň TypeScript kódom
  - všetky JavaScript programy sú zároveň TypeScript programy



Zvyšuje produktivitu…

- … a pomáha predchádzať chybám
  - **pomocou typov môžeme zachytiť chyby už v čase transpilácie** (nie až za behu, čo je temnou stránkou JS)
- Majme funkciu:

```
function add(x, y) { return x + y; }
```

- Získaním hodnôt z HTML input elementov a odovzdaním ich funkcii `add` môžeme získať neočakávaný výsledok:

```
let result = add(input1.value, input2.value);
console.log(result);
// konkatenacia retazcov, vysledok je 1020
// input1.value a input2.value su retazce,
// nie cisla
```

Špecifikujme typy explicitne …

- TypeScript transpilátor vyhodí chybu pri transpilácii TS kódu na JS kód

```
function add(x: number, y: number) {
   return x + y;
}
let result = add(input1.value,
input2.value);
```

Statický typový systém

- Oproti dynamickému typovému systému (akým je JavaScript), v TypeScripte majú premenné dátový typ.
  - Ak premenná nemá explictne uvedený typ
    - transpilátor nájde prvé použitie premennej v bloku kódu,
    - určí typ podľa typu hodnoty, ktorá jej je priradená,
    - a bude uvažovať daný typ pre predmetnú premennú vo zvyšku bloku kódu.

```
var num = 2; // number
console.log("value of num" + num);
num = "12";
console.log(num);
```

error TS2011: Cannot convert 'string' to 'number'.

Budúci JavaScript už dnes

- TypeScript implementuje plánované funkcie budúceho JavaScriptu (štandard ECMAScript)
  - Každý rok TC39 (Technical Committee number 39) prináša nové vlastnosti JavaScriptu
  - Aktuálna verzia ES2023 (10. 9. 2022)
  - Návrh novej špecifikácie zvyčajne prechádza 5 fázami (Stage 0: Strawperson, Stage 1: Proposal, Stage 2: Draft, Stage 3: Candidate, Stage 4: Finished)
- Nové vlastnosti JavaScriptu môžeme používať pri programovaní skôr, ako sú podporované webovými prehliadačmi (alebo inými prostrediami)
  - TypeScript obsahuje podporu nových vlastnosti už zvyčajne vo faze 3
- Prehľad histórie TypeScriptu

Problémy s dynamickými typmi

- Predpokladajme funkciu

```
function getProduct(id){
  return {
    id: id,
    name: `Awesome Gadget ${id}`,
    price: 99.5
  }
}

const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
// The product undefined costs $99.5
```

Problémy s dynamickými typmi

- Predpokladajme funkciu

```
function getProduct(id){
  return {
    id: id,
    name: `Awesome Gadget
    price: 99.5
  }
}
```

Problém je, že product objekt nemá atribút Name. Má atribút name.

Žiaľ, toto vieme zistiť iba keď vykonáme skript.

Odkazovanie na atribút, ktorý v objekte neexistuje, je bežným problémom pri práci v JavaScripte.

```
const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
// The product undefined costs $99.5
```

Problémy s dynamickými typmi /2

```
const showProduct = (name, price)  => {
  console.log(`The product ${name} costs ${price}$.`);
};

const product = getProduct(1);
showProduct(product.price, product.name);

// The product 99.5 costs $Awesome Gadget 1
```

Problémy s dynamickými typmi /2

```
const showProduct = (
  console.log(`The pr
${price}$.`);
};

const product = getPr
showProduct(product.p
;

// The product 99.5 costs $Awesome Gadget 1
```

Odovzdali sme **argumenty v opačnom poradí**.

Toto je ukážka **ďalšieho bežného problém**u pri práci v JavaScripte.

Riešenie v TypeScripte

```
interface Product {
    id: number,
    name: string,
    price: number
};

function getProduct(id) : Product {
  return {
    id: id,
    name: `Awesome Gadget ${id}`,
    price: 99.5
  }
}
```

Riešenie v TypeScripte

```
interface Product {
    id: number,
    name: string,
    price: number
};

function getProduct(id) : Pr
  return {
    id: id,
    name: `Awesome Gadget ${id}`,
    price: 99.5
  }
}
```

Explicitne sme definovali, že
getProduct vráti typ Product

Riešenie v TypeScripte

```
const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
```

Keď budeme referencovať na atribút, ktorý neexistuje, **editor zvýrazní problémový atribút**

```
const product = getProduct(1);
console.log(`The product ${product.Name} costs $${product.price}`);
```

any

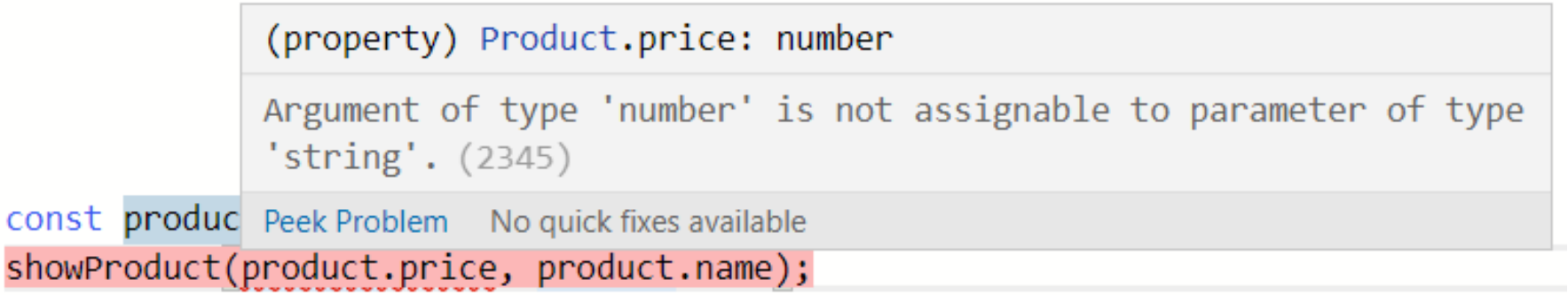
Property 'Name' does not exist on type 'Product'. Did you mean 'name'? (2551)

input.tsx(3, 5): 'name' is declared here.

Peek Problem Quick Fix...

Riešenie v TypeScripte /2

```
const showProduct = (name:string, price:number)  =>
{
  console.log(`The product ${name} costs ${price}$.`);
};
const product = getProduct(1);
showProduct(product.price, product.name);
```



Hello world v Quasare

- Pri vytváraní **Quasar** projektu vyberme v **sprievodcovi**, že chceme **podporu TypeScriptu**
- Všetky súbory budú mať v projekte **extenziu „`.ts`“** (nie `.js`)
  - skontrolujte napr. `src/store/...`
- Všimnime si, že element `script` obsahuje ponovom atribút `lang`:

```
<script lang="ts">
```

- Všetky skripty v Quasar App budú vykonávané ako TypeScript
- Upravme komponent `src/pages/`**`IndexPage.vue`**

Index.vue - template

```
<template>
  <q-page class="row items-center justify-evenly">
    <q-card class="q-pa-md">
      <q-card-section>
        <div class="text-h4">Hello TypeScript</div>
      </q-card-section>
      <q-card-section>
        <q-field bottom-slots dense>
          <q-input v-model="inputText" />
          <template v-slot:hint>
            Count: {{ count }}
          </template>
        </q-field>
      </q-card-section>
      <q-card-actions>
        <q-btn @click="reset()">Reset</q-btn>
      </q-card-actions>
    </q-card>
  </q-page>
</template>
```

Index.vue - script

```
<script lang="ts">
import { defineComponent } from 'vue'
export default defineComponent({
  name: 'PageIndex',
  data: () => {
    return { inputText: '' }
  },
  methods: {
    reset () {
      this.inputText = ''
    }
  },
  computed: {
    count () {
      return this.inputText.length
    }
  }
})
</script>
```

Jednoduchá aplikácia so vstupným textovým polom, výstupom “count” a tlačidlom reset.

Kěd používateľ napíše reťazec do vstupného poľa, uvidí počet znakov daného reťazca pod polom.

Reset tlačidlo vymaže vstupné pole.

Pridajme typ – rozhranie `State`

- Cieľom je zabezpečiť aby všetky údaje prichádzajúce zo skriptu <script> do šablóny <template> boli **staticky typové** (statically typed)
  - Inými slovami striktne zadefinujeme typ premennej `inputText`
- Zadefinujme rozhranie

```
import { defineComponent } from 'vue'
interface State {
  inputText: string;
}
export default defineComponent({
…
```

Upravme skript

- Použime anotáciu v modeli data, ktorou určíme typ návratovej hodnoty – cez rozhranie **`State`**

```
export default defineComponent({
  name: 'App',
  data: (): State => {
    return { inputText: '' }
  },
  methods: {
...
```

Preverme kontrolu na typ

- Ak by sme sa pokúsili inicializovať hodnotu premennej `inputText` na číslo `1`

…

```
data: (): State => {
   return { inputText: 1 }
},
```

…

Dostaneme chybu:

- TS2322: Type 'number' is not assignable to type 'string'.

Nový store module - TypeScript

- Nový store module v Quasare s jazykom TypeScript vytvoríme cez CLI s prepínačom **`-f ts`**:

```
quasar new store -f ts <module-name>
```

- Ak používate **eslint**, preset `standard`, ten má nastavené podľa dokumentácie pravidlo:

```
"semi": [2, "never"] //alt. "always"
```

- Vo vygenerovaných súboroch v store module sú použité "extra semicolons". Tieto treba zo súborov odobrať, fixneme to týmto CLI príkazom:

```
npm run lint -- --fix
```