TypeScript

- **Sources**
  - https://www.typescriptlang.org/play
  - https://www.typescriptlang.org/docs/handbook/intro.html
  - https://react-typescript-cheatsheet.netlify.app/
- **The Basics**
  - Spouštění příkazem `tsc hello.ts`.
  - Příkaz `tsc --noEmitOnError hello.ts` - chceme spustit příkaz a nechceme, aby TypeScript reportoval chyby.
  - Spouštět vždy s nastavením `strict: true`, který zapne striknější kontroly.
  - Volba `-noImplicitAny` - pokud není typ uveden a TypeScript ho nezná, v defaultní podobě se přiřadí `any`. Touhle volbou se přiřazování `any` zakáže.
  - Volba `-strictNullChecks` - standardně je možné undefined a `null` přiřadit k proměnné, touhle volbou se dané chování zakáže a je nutné explicitně říct, že je možné `null` či `undefined` použít.
- **Everyday Types**
  - Označení pole: `number[]` nebo `Array<number>`.
  - Oddělení parametrů ve funkci pomocí (`,`) nebo (`;`).
  - Volitelný parametr pomocí `?`.
  - Union typ pomocí `|`.
  - `Typ type Typ = {}` je vlastně alias. Používá se pro struktury, abychom se nemuseli pořád opakovat.
  - Dědičnost typů pomocí `&`: `type A & { test: 1 }`
  - `Interface` se mohou používat jen pro objekty, typy mohou být i pro primitivní hodnoty.
  - `Interface` by měly být preferovány, pokud nepotřebuji `type`.
  - Někdy je potřeba deklarovat typ typ explicitně pomocí `as`:
    - `const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement.`
    - `const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas")` (pouze pro tsx soubory).
  - Asertace pomocí `as` se používá při převodu na více či méně specifičtější verzi (není možný převod třeba z čísla na string ap., to by bylo nutné použít klasické přetypování pomocí javascriptu).
  - Převod může být někdy příliš striktní, takže nejprve převést na `any` a pak na druhý typ: `(expr as any) as T.`
  - U literar typu je hodnota jasná: `const x = 'a'` - kompilátor řekne, že hodnota je `x string 'a'`, nikoliv jen `string`.
  - Možnost deklarovat literar type takto: `let a: "muj string" = "muj string".`
  - Deklarace jako `as const`: `const req = { url: "https://example.com", method: "GET" } as const` - nelze měnit obsah, vše konstanta.
  - Zápisem (`!`) za proměnnou TypeScriptu říkáme, že ten výraz je určitě existuje (nebude to `null` ani `undefined`).
  - Další použitelné typy jsou `bigint` či `symbol`.
- **Narrowing**
  - Zužování. Když se použije např. blok `typeof`, TypeScript ví, že daná hodnota hodnota dále je daného typu. Ví to o mnoha dalších typů a konstrukcí.
  - Predikáty:
    - Predikáty mají vždy jeden parametr a vrací typ boolean.

        - Používá se operátor `is:`
        - `function isFish(pet: Fish | Bird): pet is Fish { // boolean }.`
        - Když zavolám `isFish(arg)`, tak podle vrácené hodnoty boolean kompilátor ví, zda zacházet s hodnotou v `arg` jako s instancí `Fish` nebo jako s instancí `Bird`.
    - Typ `never`. Třeba u `switch` v `default` části, ta se nemá nikdy vykonat. Vyčerpali jsme všechny možnosti, a tak TypeScript přiřadí typ `never`. Podobně třeba funkce, která vyhodí výjimku.
- **More on Functions**
    - Zápis parametru funkce: `(a: string) => void`
    - Zápis funkce s vlastnostmi: `type T { () => string; prop: text }`
    - Zápis konstruktoru: `new (s: string): T;`
    - Zápis generiky: `function fn<Type>(arr: Type[]): Type.`
        - Type se nikde nedefinuje, když ho zavoláme jako string, tak se tak bude chovat, když jako number, tak se bude chovat takhle ap.
        - `function map<In, Out>(arr: In[], fn: (arg: In) => Out): Out[]`
    - Použití omezení pomocí `extends`, kdy se řekne, že chceme sice použít generiku, ale ta má mít nějaký subset vlastností:
        - `type P = { length: number }`
        - `function longest<Type extends P>(a: Type, b: Type): Type`
    - Specifikovat typ generiky se dá i při volání funkce:
        - `function combine<Type>(arr1: Type[], arr2: Type[]): Type[]`
        - `const arr = combine<string | number>([1, 2, 3], ["hello"]`
        - Typescript nyní ví, že `arr` může být `string` i `number`. Bez tohoto určení union typu při volání by funkce nefungovala, musel by být použit jen jeden typ.
    - Přetížení funkcí:
        - `function fn (a: string): string;`
        - `function fn (a: number): number;`
        - `function fn (a: string | number) { return a }`
        - Preferovat raději zápis s union typy v parametrech.
    - Někdy je potřeba upřesnit, co přesně je `this`:
        - `function fn (this: User) { this.a = 1 }`
    - Další typy:
        - `void`
        - `object`
            - jakýkoliv typ, který není primitivní, není to to samé jako `{}` nebo `Object`
        - `unknown`
            - podobné jako any, ale je bezpečnější, protože s any není možné provádět žádné operace
            - `const fn = (a: string): unknown => JSON.parse(a)`
        - `never`
        - `Function`
    - Rest parametry: `multiply(n: number, ...m: number[])`
- **Object types**
    - Anotaci není možné dávat do object descructing, protože `const x = {a: Type}` je validní js syntaxe, vytvoří se proměnná `Type`.
    - Používat `readonly` vlastnosti, je to ekvivalent `const` pro vlastnosti objektů.
    - Deklarace objektu s indexy: `{ [index: number]: string; }`
    - Kombinace typů: `type ColorfulCircle = Colorful & Circle;`

- Generické typy: `interface Box<Type>` a pak `let boxA: Box<string>`, podobně to funguje i pro typy.
- Typ `ReadonlyArray` použít, když chci říct, že se pole bude jen číst.
- Tuple type:
  - `type MyTuple = [string, number].`
  - Hodí se na věci typu `option = ["uppercase", true]`

- **Type manipulation**
  - Možnost určit, že generika je nějakého specifičtějšího typu přes <Type extends Xyz>.
  - Možno používat i pro třídy, rozhraní, typy.
  - Factory funkce:
    - `function f<A extends B>(c: new () => A): A { return new c(); }`
  - Operátor `keyof` vrací union typ:
    - `keyof { x: number; y: number } // 'x' | 'y'`
    - použití s `extends: <Type extends keyof { x: number; y: number }>`
  - Operátor `typeof` je možné použít i pro zjistění typu:
    - `let n = typeof Type`
  - Možno vytvořit nový typ z jiného:
    - `type Age = Person["age"];`
    - `type T = Person["age" | "name"];`
    - `type Age = typeof MyArray[number]["age"];`
    - `type key = "age"; type Age = Person[key];`
  - Podmiňovací typy pomocí `extends`:
    - `type A = B extends C ? number : string;`
    - `type A<T extends number | string> = T extends number ? B : C;`
  - Odvozování pomocí `infer`:
    - Klíčové slovo `infer` říká, že vyvozujeme něco z něčeho jiného. Musí být vždy uvnitř `extends`.
      - `OK: type MyType1<T> = T extends infer R ? R : never;`
        - Bude fungovat, říká, že `R` bude vyvozeno z `T`.
        - `type T1 = MyType1<{b: string}> // T1 je { b: string; }`
      - `NOT OK: type MyType2<T> = T extends R ? R : never;`
        - Nebude fungovat, protože `R` je neznámé.
    - `type Flatten<T> = T extends Array<infer I> ? I : T;`
      - `type A = Flatten<string> // A je string`
      - `type B = Flatten<string[]> // A je opět string`
  - Distributivní podmíněné typy:
    - 1) základní chování, kdy chci, aby byl výsledek jen jeden typ:
      - `type ToArray<T> = T extends any ? T[] : never;`
      - `type a = ToArray<string | number>; // type a = string[] | number[]`
    - 2) když chci, aby to byl mix, použít []
      - `type ToArray<T> = [T] extends [any] ? T[] : never;`
      - `type a = ToArray<string | number>; // type a = (string | number)[]`
  - Mapped types:
    - Vytvoření dynamického typu na základě klíčů:
      - `type A<T> = { [P in keyof T]: boolean; };`
      - `type Features = { darkMode: () => void }`
      - `type Options = A<Features> // { darkMode: boolean }`

        - Možno používat operátory `readonly` a `?` a doplňovat je nebo odebírat pomocí `+` nebo `-`.
            - `-readonly [P in keyof T]: T[P];`
            - `[P in keyof T]-?: T[P];`
        - Remapování pomocí operátoru as:
            - `[P in keyof T as NewType]: T[P]`
    - Template Literal Types:
        - Vytváření typů:
            - `type W = "world";`
            - `type G = \`hello ${W}\`; // type G = "hello world"`
        - V případě unionů se vytváří všechny kombinace:
            - `type L = "en" | "ja" | "pt";`
            - `type H = \`hello ${L}\`; // H = hello_en, hello_ja, hello_pt`
        - Různé pomocné metody typu `Uppercase<StringType>` ap.
- **Classes**
    - Inicializaci provést v konstruktoru, popř. dát za deklaraci vlastnosti vykřičník (!).
    - Vlastnosti označené jako `readonly` je možné přiřadit jen v konstruktoru.
    - Je možné implementovat jen signaturu metody bez obsahu, potomek ji pak musí implementovat.
    - Při dědění built-in tříd (hlavně výjimek) je nutné zavolat super a nastavit `prototype` (viz handbook).
    - Typescript má jak svojí deklaraci privátní vlastnosti, která funguje jen během kompilace, tak podporuje i nativní privátní proměnné prefixované křížkem (`#`).
    - Podpora statických bloků a abstraktních tříd.
    - Při deklaraci metod používat raději arrow funkce, protože se tím vyloučí potenciální problémy s `this`.
    - Jako typ v parametru je možné použít this a odkázat se tak na instanci aktuální třídy:
        - `class A { method (obj: this) {} }`
    - V deklaraci konstruktoru je možné rovnou uvést `private`, `protected`, `public`, `readonly` a ušetřit si tak čas.
- **Reference**
    - Utility types
    - **Decorators**
        - Lze je používat jen pro třídy.
    - **Declaration merging**
        - V typescriptu mohu deklarovat některé konstrukce pod stejným jménem a ty se pak mergují. Především to platí pro interfaces a namespaces.
        - `interface A { id: number }`
        - `interface A { name: string } // A má nyní id i name`
    - **Enums**
        - Není potřeba zadávat hodnoty, pak se začnou číslovat od nuly
            - `enum A { X, Y } // X = 0`
        - Nemixovat typy.
        - Když mají typ literal (třeba nějaký string), stanou se i typem.
        - Nepoužívat `const ENUM`, jsou kolem toho problémy a nepředpokládané chování.
    - **Iterators a generators**
    - JSX
        - .tsx
    - Mixins
    - **Moduly**:
        - Reexport z jiného modulu: `export * from "module"`.
        - Použití `import type from …` je lepší v tom, že kompilátor ví, že načítáme jen typy a ten soubor pak do výsledného buildu nepřidá.

        - Když se načítá nějaká externí knihovna, která nebyla napsaná v typescriptu, popíše se její rozhraní do souboru s koncovkou `.d.ts`.
        - Deklarace modulu: `declare module "url" { export cost a = 'xxx' }`
        - Import deklarace: `/// <reference path="node.d.ts"/>`
        - Nepoužívat namespaces, tady je dána struktura pomocí filesystému, a tak nejsou potřeba.
    - Module resolution
    - Namespaces
    - Namespaces and modules
    - Symbols
    - Triple slash directive
    - Type compatibility
    - Type interference
    - Variable declaration
- **Declaration Files**
    - TODO
- **Javascript**
    - TODO
- **Project configuration**
    - TODO
- **React TypeScript cheatsheet**
    - TODO