TypeScript a OOP

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

12.11.2025

Úvod

Co je TypeScript?

TypeScript je nadstavba JavaScriptu, která přidává **statickou typovou kontrolu**

- Vytvořen a udržován Microsoftem
- Open-source projekt
- Transpiluje se do JavaScriptu
- Kompatibilní se všemi prostředími, kde běží JavaScript

Co je TypeScript?

TypeScript = JavaScript + **typy**

```
// JavaScript
function add(a, b) {
  return a + b;
}

// TypeScript
function add(a: number, b: number): number {
  return a + b;
}
```

Proč TypeScript?

Jaké jsou podle vás výhody použití TypeScriptu?

- Odhalení chyb již při **kompilaci**, ne až za běhu
- Lepší **IntelliSense** a automatické doplňování v editoru
- **Refaktoring** kódu je bezpečnější a jednodušší
- Dokumentace pomocí typů
- Lepší **škálovatelnost** pro velké projekty
- Podpora **moderních JavaScript** funkcí

Proč TypeScript?

Příklad problému, který TypeScript řeší:

```
// JavaScript - runtime error
function getUser(id) {
  return { name: "John", age: 30 };
}

const user = getUser(1);
console.log(user.nmae);
// undefined - typo!
```

Proč TypeScript?

```typescript
// TypeScript - compile-time error
function getUser(id: number): { name: string; age: number } {
  return { name: "John", age: 30 };
}

const user = getUser(1);
console.log(user.nmae);
// Error: Property 'nmae' does not exist on type ...
```

Instalace a nastavení

TypeScript můžete nainstalovat pomocí npm:

```
npm install -g typescript
```

Ověření instalace:

```
tsc --version
```

Instalace a nastavení

Vytvoření TypeScript souboru:

```
echo 'console.log("Hello TypeScript");' > hello.ts
```

Kompilace do JavaScriptu:

```
tsc hello.ts
```

Výsledkem je soubor `hello.js`, který lze spustit:

```
node hello.js
```

Instalace a nastavení

Inicializace TypeScript projektu:

```
tsc --init
```

Vytvoří se soubor `tsconfig.json` s konfigurací kompilátoru

Základní možnosti konfigurace:

- `target` - verze JavaScriptu (ES5, ES6, ...)
- `module` - systém modulů (CommonJS, ES6, ...)
- `strict` - přísná kontrola typů
- `outDir` - adresář pro zkompilované soubory

Základy TypeScriptu

Typový systém

TypeScript používá **statický typový systém**

Co to znamená?

- Typy proměnných jsou známy **v době kompilace**
- Kompilátor kontroluje, zda používáte správné typy
- Chyby jsou odhaleny **před spuštěním** programu

Typový systém

TypeScript má dva způsoby určení typu:

1. **Explicitní anotace typů**

```
let age: number = 30;
let name: string = "John";
```

2. **Inference typů** (odvození)

```
let age = 30;       // TypeScript infere: number
let name = "John"; // TypeScript infere: string
```

Základní typy

TypeScript obsahuje stejné základní typy jako JavaScript:

- `number` - čísla (celá i desetinná)
- `string` - řetězce
- `boolean` - logické hodnoty
- `null` a `undefined`
- `object` - objekty
- A přidává nové: `any`, `unknown`, `never`, `void`

Základní typy

Number

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
let big: number = 1_000_000;
```

Základní typy

String

```
let color: string = "blue";
let fullName: string = `John Doe`;
let greeting: string = `Hello, ${fullName}`;
```

Boolean

```
let isDone: boolean = false;
let isActive: boolean = true;
```

Základní typy

Array

```
// Two ways to write array types
let list: number[] = [1, 2, 3];
let list2: Array<number> = [1, 2, 3];

let names: string[] = ["John", "Jane"];
```

Základní typy

Tuple - pole s pevným počtem prvků různých typů

```
let tuple: [string, number];
tuple = ["hello", 10]; // OK
tuple = [10, "hello"]; // Error

// Accessing tuple elements
console.log(tuple[0]); // "hello"
console.log(tuple[1]); // 10
```

Základní typy

Enum - pojmenované konstanty

```
enum Color {
  Red,
  Green,
  Blue
}

let c: Color = Color.Green;
console.log(c); // 1 (index)
```

Základní typy

Enum lze použít i s vlastními hodnotami

```
// With custom values
enum Status {
  Active = "ACTIVE",
  Inactive = "INACTIVE"
}
```

Základní typy

Any - vypnutí typové kontroly

```
let notSure: any = 4;
notSure = "maybe a string";
notSure = false; // OK

// Use sparingly!
```

Základní typy

Unknown - bezpečnější alternativa k any

```
let value: unknown = 4;
value = "string"; // OK

// But you must check type before using
if (typeof value === "string") {
  console.log(value.toUpperCase());
}
```

Základní typy

Void - absence návratové hodnoty

```
function logMessage(message: string): void {
  console.log(message);
  // no return statement
}
```

Základní typy

Never - funkce nikdy nevrací hodnotu

```
function error(message: string): never {
  throw new Error(message);
}

function infiniteLoop(): never {
  while (true) {}
}
```

Funkce a typy funkcí

Funkce v TypeScriptu můžeme typovat několika způsoby:

```
// Named function with types
function add(a: number, b: number): number {
  return a + b;
}

// Arrow function with types
const subtract = (a: number, b: number): number => {
  return a - b;
};
```

Funkce a typy funkcí

Volitelné parametry

```
function greet(name: string, greeting?: string): string {
  if (greeting) {
    return `${greeting}, ${name}`;
  }
  return `Hello, ${name}`;
}

greet("John");                  // "Hello, John"
greet("John", "Good day");  // "Good day, John"
```

Funkce a typy funkcí

Otazník za parametrem označuje, že parametr je volitelný a jedná se o „syntax sugar“ pro ekvivalentní definici například
`string | undefined`

Funkce a typy funkcí

Výchozí hodnoty parametrů

```
function greet(
  name: string,
  greeting: string = "Hello"
): string {
  return `${greeting}, ${name}`;
}

greet("John");              // "Hello, John"
greet("John", "Hi");        // "Hi, John"
```

Funkce a typy funkcí

Rest parametry

```
function sum(...numbers: number[]): number {
  return numbers.reduce((total, n) => total + n, 0);
}

sum(1, 2, 3);         // 6
sum(1, 2, 3, 4, 5); // 15
```

Funkce a typy funkcí

Typ funkce

```
// Function type
let myAdd: (a: number, b: number) => number;

myAdd = function(a: number, b: number): number {
  return a + b;
};
```

Funkce a typy funkcí

```
function calculate(
  a: number,
  b: number,
  operation: (x: number, y: number) => number
): number {
  return operation(a, b);
}

calculate(2, 3, (x, y) => x + y); // 5
calculate(2, 3, (x, y) => x * y); // 6
```

Union a Intersection typy

Union typy umožňují proměnné mít více povolených typů

```
let value: string | number;

value = "hello";  // OK
value = 42;       // OK
value = true;     // Error: boolean is not assignable
```

Union a Intersection typy

Použití union typů s funkcemi:

```
function printId(id: string | number) {
  console.log(`ID: ${id}`);
}

printId(101);      // OK
printId("202");    // OK
```

Union a Intersection typy

Intersection typy kombinují více typů do jednoho

```
type Person = {
  name: string;
  age: number;
};
type Employee = {
  employeeId: number;
  department: string;
};
type Staff = Person & Employee;
```

Union a Intersection typy

```
const john: Staff = {
  name: "John",
  age: 30,
  employeeId: 1234,
  department: "IT"
};
```

Type Guards - zúžení typu

Type guards jsou techniky pro zúžení (narrowing) typu proměnné v určitém kontextu

TypeScript umí automaticky infereovat konkrétnější typ na základě kontextu

Type Guards - zúžení typu

typeof - pro primitivní typy

```
function processValue(value: string | number) {
  if (typeof value === "string") {
    // TypeScript knows that value is string
    return value.toUpperCase();
  } else {
    // TypeScript knows that value is number
    return value.toFixed(2);
  }
}
```

Type Guards - zúžení typu

instanceof - pro třídy a objekty

```
class Dog { bark() { console.log("Woof!"); } }
class Cat { meow() { console.log("Meow!"); } }

function makeSound(animal: Dog | Cat) {
  if (animal instanceof Dog) {
    animal.bark(); // TypeScript knows it's a Dog
  } else {
    animal.meow(); // TypeScript knows it's a Cat
  }
}
```

Type Guards - zúžení typu

in operator - kontrola existence vlastnosti

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    animal.swim(); // TypeScript knows it's a Fish
  } else {
    animal.fly();  // TypeScript knows it's a Bird
  }
}
```

Type Guards - zúžení typu

Custom type guard - vlastní funkce pro kontrolu typu

```
interface Cat { meow: () => void; }
interface Dog { bark: () => void; }

function isCat(animal: Cat | Dog): animal is Cat {
  return (animal as Cat).meow !== undefined;
}
```

Type Guards - zúžení typu

```
function makeSound(animal: Cat | Dog) {
  if (isCat(animal)) {
    animal.meow(); // TypeScript knows it's a Cat
  } else {
    animal.bark(); // TypeScript knows it's a Dog
  }
}
```

Klíčové slovo `is` v návratovém typu říká TypeScriptu, že funkce je type guard

Discriminated Unions (Tagged Unions)

Discriminated Union je pattern, kde každá varianta union typu má společnou vlastnost (**diskriminátor**)

Velmi užitečné pro reprezentaci stavů nebo variant

Discriminated Unions (Tagged Unions)

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "square"; size: number }
  | { kind: "rectangle"; width: number; height: number };
```

Vlastnost `kind` je **diskriminátor** - jednoznačně určuje typ

Discriminated Unions (Tagged Unions)

```
function getArea(shape: Shape): number {
  switch (shape.kind) {
    case "circle":
      return Math.PI * shape.radius ** 2;
    case "square":
      return shape.size ** 2;
    case "rectangle":
      return shape.width * shape.height;
  }
}
```

TypeScript v každé větvi `switch` ví přesný typ!

Discriminated Unions (Tagged Unions)

Praktický příklad - API response:

```
type ApiResponse<T> =
  | { status: "success"; data: T }
  | { status: "error"; error: string }
  | { status: "loading" };
```

Discriminated Unions (Tagged Unions)

```
function handleResponse(response: ApiResponse<User>) {
  switch (response.status) {
    case "success":
      console.log(response.data.name); // OK
      break;
    case "error":
      console.error(response.error); // OK
      break;
    case "loading":
      console.log("Loading..."); // OK
      break;
```

```
    }
}
```

Discriminated Unions (Tagged Unions)

Výhody Discriminated Unions:

- **Exhaustive checking** - TypeScript kontroluje, že jste ošetřili všechny případy
- **Type safety** - přesná typová kontrola v každé větvi
- **Čitelnost** - explicitní reprezentace stavů
- Skvělé pro **state management**

Type aliases a Type assertions

Type alias vytvoří nové jméno pro typ

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

const point: Point = { x: 10, y: 20 };
```

Type aliases a Type assertions

```
// Type alias for union
type ID = string | number;
function printId(id: ID) {
  console.log(id);
}

// Type alias for function
type GreetFunction = (name: string) => string;
const greet: GreetFunction = (name) => {
  return `Hello, ${name}`;
};
```

Type aliases a Type assertions

Type assertion řekne kompilátoru, že znáte typ lépe

```
// Two syntaxes
let someValue: unknown = "this is a string";

let strLength1: number = (someValue as string).length;
let strLength2: number = (<string>someValue).length;
```

Pozor! Type assertion **neprovádí konverzi**, pouze říká kompilátoru „věř mi, vím co dělám“

Type aliases a Type assertions

```
// Practical example with DOM
const input =
  document.getElementById("myInput") as HTMLInputElement;
input.value = "Hello";
```

Bez type assertion by TypeScript nevěděl, že element má vlastnost `value`

Objektově orientované programování

Základy OOP

Objektově orientované programování (OOP) je programovací paradigma založené na konceptu **objektů**

Objekty obsahují:

- **Data** (vlastnosti, fieldy, atributy)
- **Chování** (metody, funkce)

Základy OOP

Základní principy OOP:

1. **Zapouzdření** (Encapsulation)
2. **Dědičnost** (Inheritance)
3. **Polymorfismus** (Polymorphism)
4. **Abstrakce** (Abstraction)

Základy OOP

Proč používat OOP?

- Lepší **organizace** kódu
- **Znovupoužitelnost** kódu
- Snadnější **údržba** a rozšiřování
- Modelování **reálného světa**
- Lépe škálovatelné pro **velké projekty**

Třídy v TypeScriptu

Třída je šablona (blueprint) pro vytváření objektů

Třídy v TypeScriptu

```
class Person {
  name: string;
  age: number;

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

  greet(): string { return `Hello, I am ${this.name}`; }
}
```

Třídy v TypeScriptu

Vytvoření instance třídy:

```
const john = new Person("John", 30);
console.log(john.greet()); // "Hello, my name is John"
console.log(john.age);     // 30
```

Třídy v TypeScriptu

Rozdíl oproti JavaScript ES6 třídám?

TypeScript přidává:

- **Typovou kontrolu** vlastností a parametrů
- **Modifikátory přístupu** (public, private, protected)
- **Abstraktní třídy** a metody
- Lepší podporu pro **rozhraní**

Konstruktory a vlastnosti

Konstruktor je speciální metoda volaná při vytvoření instance

Konstruktory a vlastnosti

```
class Rectangle {
  width: number;
  height: number;

  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }

  area(): number { return this.width * this.height; }
}
```

Konstruktory a vlastnosti

TypeScript nabízí **zkrácenou syntaxi** pro vlastnosti:

```
// Long way
class Rectangle {
  width: number;
  height: number;

  constructor(width: number, height: number) {
    this.width = width;
    this.height = height;
  }
}
```

Konstruktory a vlastnosti

```
// Short way (parameter properties)
class Rectangle {
  constructor(
    public width: number,
    public height: number
  ) {}
}
```

Obě verze dělají totéž, ale zkrácená syntaxe je čitelnější!

Konstruktory a vlastnosti

Výchozí hodnoty vlastností:

```
class Counter {
  count: number = 0;

  increment(): void {
    this.count++;
  }
}

const counter = new Counter();
console.log(counter.count); // 0
```

Statické vlastnosti a metody

Statické členy patří třídě samotné, ne instancím

Označují se klíčovým slovem `static`

Přistupuje se k nim přes název třídy, ne přes instanci

Statické vlastnosti a metody

```
class MathHelper {
  static PI = 3.14159;

  static circleArea(radius: number): number {
    return this.PI * radius ** 2;
  }

  static circleCircumference(radius: number): number {
    return 2 * this.PI * radius;
  }
}
```

Statické vlastnosti a metody

Použití statických členů:

```
// Access through class name
console.log(MathHelper.PI);                 // 3.14159
console.log(MathHelper.circleArea(5));      // 78.53975

// CANNOT access through instance
const helper = new MathHelper();
console.log(helper.PI);  // Error: Property 'PI' is static
```

Statické vlastnosti a metody

Kdy používat statické členy?

- **Utility funkce** související s třídou
- **Konstanty** sdílené všemi instancemi
- **Factory metody** pro vytváření instancí
- **Počítadla** instancí

Factory pomocí statické metody

```
class User {
  constructor(
    public name: string,
    public email: string
  ) {}

  static fromJSON(json: string): User {
    const data = JSON.parse(json);
    return new User(data.name, data.email);
  }
}
```

Factory pomocí statické metody

```
const jsonString =
  '{"name": "John", "email": "john@example.com"}';
const user = User.fromJSON(jsonString);
console.log(user.name);  // "John"
console.log(user.email); // "john@example.com"
```

Počítadlo instancí

```
class Database {
  private static instanceCount = 0;

  constructor(public name: string) {
    Database.instanceCount++;
  }

  static getInstanceCount(): number {
    return Database.instanceCount;
  }
}
```

Počítadlo instancí

```
const db1 = new Database("users");
const db2 = new Database("products");

console.log(Database.getInstanceCount()); // 2
```

Zapouzdření

Modifikátory přístupu

Zapouzdření znamená skrytí interních detailů objektu a poskytnutí veřejného rozhraní

TypeScript nabízí tři modifikátory přístupu:

- `public` - přístupné odkudkoliv (výchozí)
- `private` - přístupné pouze uvnitř třídy
- `protected` - přístupné uvnitř třídy a jejích potomků

Modifikátory přístupu

Public (výchozí)

```
class Person {
  public name: string; // explicitly public
  age: number;         // implicitly public

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

Modifikátory přístupu

```
const person = new Person("John", 30);
console.log(person.name); // OK
console.log(person.age);  // OK
```

Modifikátory přístupu

Private

```
class BankAccount {
  private balance: number = 0;

  deposit(amount: number): void {
    this.balance += amount;
  }
  getBalance(): number {
    return this.balance;
  }
}
```

Modifikátory přístupu

```
const account = new BankAccount();
account.deposit(100);
console.log(account.getBalance()); // 100
console.log(account.balance);
// Error: Property 'balance' is private
```

Private vlastnosti jsou přístupné pouze uvnitř třídy!

Modifikátory přístupu

Protected

```
class Animal {
  protected name: string;

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

Modifikátory přístupu

```
class Dog extends Animal {
  bark(): void {
    console.log(`${this.name} says woof!`); // OK - protected
  }
}
```

Modifikátory přístupu

```
const dog = new Dog("Rex");
dog.bark(); // "Rex says woof!"
console.log(dog.name);
// Error: Property 'name' is protected
```

Gettery a settery

Gettery a **settery** poskytují kontrolovaný přístup k vlastnostem

```
class Person {
  private _age: number = 0;

  get age(): number {
    return this._age;
  }
```

Gettery a settery

```
  set age(value: number) {
    if (value < 0) {
      throw new Error("Age cannot be negative");
    }
    this._age = value;
  }
}
```

Gettery a settery

Použití getterů a setterů:

```
const person = new Person();
person.age = 30;           // calls setter
console.log(person.age); // calls getter, output: 30

person.age = -5;           // Error: Age cannot be negative
```

Přístup vypadá jako k běžné vlastnosti, ale máme kontrolu!

Gettery a settery

Praktický příklad - validace emailu:

```
class User {
  private _email: string = "";

  get email(): string {
    return this._email;
  }
```

Gettery a settery

```
  set email(value: string) {
    if (!value.includes("@")) {
      throw new Error("Invalid email");
    }
    this._email = value;
  }
}
```

Readonly vlastnosti

Readonly vlastnosti lze nastavit pouze v konstruktoru

```
class Person {
  readonly id: number;
  readonly birthDate: Date;

  constructor(id: number, birthDate: Date) {
    this.id = id;
    this.birthDate = birthDate;
  }
}
```

Readonly vlastnosti

```
const person = new Person(1, new Date("1990-01-01"));
console.log(person.id); // 1

person.id = 2;
// Error: Cannot assign to 'id' because it is a read-only
property
```

Užitečné pro vlastnosti, které by se neměly měnit po vytvoření objektu!

Readonly vlastnosti

Readonly s parameter properties:

```
class Person {
  constructor(
    public readonly id: number,
    public name: string
  ) {}
}

const person = new Person(1, "John");
person.name = "Jane"; // OK
person.id = 2;        // Error: readonly
```

Dědičnost

Extends klíčové slovo

Dědičnost umožňuje vytvořit novou třídu na základě existující

Nová třída (**potomek**) zdědí vlastnosti a metody z **rodičovské třídy**

Používá se klíčové slovo `extends`

Extends klíčové slovo

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

  move(distance: number): void {
    console.log(`${this.name} moved ${distance}m`);
  }
}
```

Extends klíčové slovo

```
class Dog extends Animal {
  bark(): void {
    console.log(`${this.name} says woof!`);
  }
}
```

Extends klíčové slovo

```
const dog = new Dog("Rex");
dog.bark();             // "Rex says woof!"
dog.move(10);           // "Rex moved 10m" - inherited method
console.log(dog.name); // "Rex" - inherited property
```

Extends klíčové slovo

Výhody dědičnosti:

- **Znovupoužití** kódu z rodičovské třídy
- **Rozšíření** funkcionality rodičovské třídy
- Vytvoření **hierarchie** tříd
- **DRY princip** (Don’t Repeat Yourself)

Super a přístup k rodičovské třídě

Klíčové slovo `super` umožňuje přístup k rodičovské třídě

Používá se:

- V konstruktoru pro volání rodičovského konstruktoru
- Pro volání rodičovských metod

Super a přístup k rodičovské třídě

Super v konstruktoru

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

class Dog extends Animal {
  constructor(name: string, public breed: string) {
    super(name); // calls parent constructor
    // must be first in constructor!
  }
}
```

Super a přístup k rodičovské třídě

```
const dog = new Dog("Rex", "German Shepherd");
console.log(dog.name);  // "Rex"
console.log(dog.breed); // "German Shepherd"
```

Bez volání `super()` by došlo k chybě!

Super pro volání metod

```
class Animal {
  move(distance: number): void {
    console.log(`Animal moved ${distance}m`);
  }
}
class Dog extends Animal {
  move(distance: number): void {
    super.move(distance); // calls parent method
    console.log("with enthusiasm!");
  }
}
```

Super pro volání metod

```
const dog = new Dog("Rex");
dog.move(10);
// "Animal moved 10m"
// "with enthusiasm!"
```

Přepisování metod

Přepisování metod (method overriding) znamená definovat metodu v potomkovi se stejným jménem jako v rodiči

Potomek pak používá svou vlastní implementaci

Přepisování metod

```
class Animal {
  makeSound(): void {
    console.log("Some generic animal sound");
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log("Woof!");
  }
}
```

```
class Cat extends Animal {
  makeSound(): void {
    console.log("Meow!");
  }
}
```

Přepisování metod

```
const dog = new Dog();
const cat = new Cat();

dog.makeSound(); // "Woof!"
cat.makeSound(); // "Meow!"
```

Každá třída má svou vlastní implementaci!

Přepisování metod

TypeScript 4.3+ umožňuje explicitně označit přepisování pomocí klíčového slova `override`:

```
class Dog extends Animal {
  override makeSound(): void {
    console.log("Woof!");
  }
}
```

Kompilátor pak kontroluje, že metoda v rodiči existuje. Pomáhá při refaktoringu a prevenci chyb

Polymorfismus

Polymorfismus

Poly morfismus znamená „mnoho forem“

Objekt může mít více forem a chovat se různě podle kontextu

V OOP: stejné rozhraní, různá implementace

Příklad polymorfismu

```
class Animal {
  makeSound(): void {
    console.log("Some sound");
  }
}

class Dog extends Animal {
  makeSound(): void {
    console.log("Woof!");
  }
}
```

Příklad polymorfismu

```
class Cat extends Animal {
  makeSound(): void {
    console.log("Meow!");
  }
}

function makeAnimalSound(animal: Animal): void {
  animal.makeSound();
}
```

Příklad polymorfismu

```
const dog = new Dog();
const cat = new Cat();

makeAnimalSound(dog); // "Woof!"
makeAnimalSound(cat); // "Meow!"
```

Funkce pracuje s typem `Animal`, ale volá správnou metodu podle skutečného typu!

Příklad polymorfismu

```
abstract class PaymentMethod {
  abstract processPayment(amount: number): void;
}
```

Příklad polymorfismu

```
class CreditCard extends PaymentMethod {
  processPayment(amount: number): void {
    console.log(`Paid ${amount} by credit card`);
  }
}

class PayPal extends PaymentMethod {
  processPayment(amount: number): void {
    console.log(`Paid ${amount} via PayPal`);
  }
}
```

Příklad polymorfismu

```
function checkout(
  payment: PaymentMethod,
  amount: number
): void {
  payment.processPayment(amount);
}

checkout(new CreditCard(), 100); // "Paid 100 by credit card"
checkout(new PayPal(), 50);      // "Paid 50 via PayPal"
```

Přetěžování metod

Přetěžování metod umožňuje definovat více verzí stejné funkce s různými parametry

TypeScript podporuje přetěžování pomocí **signatury funkcí**

Přetěžování metod

```
class Calculator {
  // Overload signatures
  add(a: number, b: number): number;
  add(a: string, b: string): string;
  add(a: number[], b: number[]): number[];

  // Implementation
  add(a: any, b: any): any {
```

Přetěžování metod

```
    if (typeof a === "number" && typeof b === "number") {
      return a + b;
    }
    if (typeof a === "string" && typeof b === "string") {
      return a + b;
    }
    if (Array.isArray(a) && Array.isArray(b)) {
      return [...a, ...b];
    }
  }
}
```

Přetěžování metod

```
const calc = new Calculator();

console.log(calc.add(1, 2));           // 3
console.log(calc.add("Hello", " World")); // "Hello World"
console.log(calc.add([1, 2], [3, 4])); // [1, 2, 3, 4]
```

Přetěžování s volitelnými parametry

```
class DateFormatter {
  format(date: Date): string;
  format(date: Date, format: string): string;
  format(date: Date, format?: string): string {
    if (format) {
      // Custom formatting
      return date.toLocaleDateString();
    }
    return date.toString();
  }
}
```

Polymorfismus s rozhraními

```
interface Shape {
  area(): number;
  perimeter(): number;
}
```

Polymorfismus s rozhraními

```
class Circle implements Shape {
  constructor(
    private radius: number
  ) {}

  area(): number { return Math.PI * this.radius ** 2; }
  perimeter(): number {
    return 2 * Math.PI * this.radius;
  }
}
```

Polymorfismus s rozhraními

```
class Rectangle implements Shape {
  constructor(
    private width: number,
    private height: number
  ) {}
  area(): number { return this.width * this.height; }
  perimeter(): number {
    return 2 * (this.width + this.height);
  }
}
```

Polymorfismus s rozhraními

```
function printShapeInfo(shape: Shape): void {
  console.log(`Area: ${shape.area()}`);
  console.log(`Perimeter: ${shape.perimeter()}`);
}

const circle = new Circle(5);
const rectangle = new Rectangle(4, 6);

printShapeInfo(circle);
printShapeInfo(rectangle);
```

Abstrakce a rozhraní

Abstraktní třídy

Abstraktní třída je třída, kterou nelze přímo instancovat

Slouží jako **základní šablona** pro jiné třídy

Používá se klíčové slovo `abstract`

Abstraktní třídy

```
abstract class Animal {
  constructor(public name: string) {}

  // Abstract method - must be implemented by child
  abstract makeSound(): void;

  // Concrete method - shared implementation
  move(): void {
    console.log(`${this.name} is moving`);
  }
}
```

Abstraktní třídy

```
// Cannot instantiate abstract class
const animal = new Animal("Generic"); // Error!
```

Abstraktní třídy

```
// Must extend and implement abstract methods
class Dog extends Animal {
  makeSound(): void {
    console.log("Woof!");
  }
}

const dog = new Dog("Rex"); // OK
dog.makeSound();            // "Woof!"
dog.move();                 // "Rex is moving"
```

Abstraktní třídy

Kdy používat abstraktní třídy?

- Chcete **sdílet implementaci** mezi třídami
- Chcete **vynutit určité metody** v potomcích
- Chcete definovat **společné chování**
- Máte **hierarchii** souvisejících tříd

Rozhraní

Rozhraní (interface) definuje **kontrakt** - strukturu objektu bez implementace

```
interface Person {
  name: string;
  age: number;
  greet(): string;
}
```

Rozhraní říká „co“, ne „jak“

Rozhraní

Použití rozhraní:

```
const john: Person = {
  name: "John",
  age: 30,
  greet() {
    return `Hello, I'm ${this.name}`;
  }
};

console.log(john.greet()); // "Hello, I'm John"
```

Rozhraní s volitelnými vlastnostmi

```
interface User {
  id: number;
  name: string;
  email?: string; // optional
  phone?: string; // optional
}

const user1: User = { id: 1, name: "John" }; // OK
const user2: User = {
  id: 2, name: "Jane", email: "jane@example.com"
}; // OK
```

Rozhraní s readonly vlastnostmi

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

const point: Point = { x: 10, y: 20 };
point.x = 5; // Error: Cannot assign to 'x' because it is a
read-only property
```

Rozhraní s readonly vlastnostmi

Rozšiřování rozhraní:

```
interface Person {
  name: string;
  age: number;
}

interface Employee extends Person {
  employeeId: number;
  department: string;
}
```

Rozhraní s readonly vlastnostmi

```
const emp: Employee = {
  name: "John",
  age: 30,
  employeeId: 1234,
  department: "IT"
};
```

Index Signatures - dynamické klíče

Index signatures umožňují definovat typ pro dynamické klíče objektu

Užitečné pro objekty, které fungují jako slovníky nebo mapy

Index Signatures - dynamické klíče

```
interface StringMap {
  [key: string]: string;
}

const translations: StringMap = {
  hello: "Ahoj",
  goodbye: "Sbohem",
  yes: "Ano"
};
```

Index Signatures - dynamické klíče

```
console.log(translations["hello"]); // "Ahoj"
translations["no"] = "Ne"; // OK
```

Pozor! JavaScript interně převádí čísla na řetězce, takže je možné použít i čísla jako klíče. TypeScript toto chování respektuje

```
translations[1] = "One"; // OK
console.log(translations[1]); // "One"
```

Index Signatures - dynamické klíče

Index signature s číselnými klíči:

```
interface NumberDictionary {
  [index: number]: string;
}

const names: NumberDictionary = {
  0: "Alice",
  1: "Bob",
  2: "Charlie"
};
```

Index Signatures - dynamické klíče

Kombinace fixních a dynamických vlastností:

```
interface Config {
  version: string;                    // fixed property
  [key: string]: string | number; // dynamic properties
}

const config: Config = {
  version: "1.0.0",
  port: 3000,
  host: "localhost"
};
```

Index Signatures - dynamické klíče

```
interface Config {
  version: string;
  [key: string]: string | number | Config;
}

const config: Config = {
  version: "1.0.0",
  port: 3000,
  host: "localhost",
  db: { version: "0.2.0", host: "localhost", port: 3306 }
};
```

Index Signatures - dynamické klíče

```
interface Cache<T> {
  [key: string]: T;
}

const userCache: Cache<{ name: string; age: number }> = {
  "user1": { name: "John", age: 30 },
  "user2": { name: "Jane", age: 25 }
};
```

Tento příklad využíval takzvaný **generický typ** (typový parametr) o kterém se ještě budeme bavit.

Rozdíl mezi třídou a rozhraním

Kdy použít **rozhraní** a kdy **abstraktní třídu**?

Rozhraní:

- Pouze **definice** bez implementace
- Třída může implementovat **více rozhraní**
- **Čistý kontrakt** - co objekt umí
- Interface se používá pro **duck typing**
  - „Pokud to chodí jako kachna a kváká jako kachna, pak to musí být kachna“
- Hierarchie „má“ vztahů (has-a) / „umí“ (can-do)

Rozdíl mezi třídou a rozhraním

Abstraktní třída:

- Může obsahovat **implementaci**
- Třída může dědit pouze z **jedné** třídy
- **Sdílená funkcionalita** mezi potomky
- Hierarchie „je“ vztahů (is-a)

```
// Can implement multiple interfaces
class User implements Person, Serializable { }
// Can extend only one class
class Dog extends Animal { }
```

Rozdíl mezi třídou a rozhraním

```
// Interface - contract for functionality
interface Flyable {
  fly(): void;
}
```

Rozdíl mezi třídou a rozhraním

```
// Abstract class - shared behavior
abstract class Bird {
  constructor(public name: string) {}

  abstract makeSound(): void;

  eat(): void {
    console.log(`${this.name} is eating`);
  }
}
```

Rozdíl mezi třídou a rozhraním

```
// Class implementing interface and extending abstract class
class Sparrow extends Bird implements Flyable {
  makeSound(): void {
    console.log("Chirp!");
  }

  fly(): void {
    console.log(`${this.name} is flying`);
  }
}
```

Rozdíl mezi třídou a rozhraním

```
const sparrow = new Sparrow("Tweety");
sparrow.makeSound(); // "Chirp!"
sparrow.fly();       // "Tweety is flying"
sparrow.eat();       // "Tweety is eating"
```

Implementace rozhraní

Klíčové slovo `implements` se používá pro implementaci rozhraní

```
interface Logger {
  log(message: string): void;
  error(message: string): void;
}
```

Implementace rozhraní

```
class ConsoleLogger implements Logger {
  log(message: string): void {
    console.log(`[LOG] ${message}`);
  }

  error(message: string): void {
    console.error(`[ERROR] ${message}`);
  }
}
```

Implementace více rozhraní

```
interface Printable {
  print(): void;
}

interface Saveable {
  save(): void;
}
```

Implementace více rozhraní

```
class Document implements Printable, Saveable {
  print(): void {
    console.log("Printing document");
  }

  save(): void {
    console.log("Saving document");
  }
}
```

Rozhraní pro funkce

```
interface SearchFunc {
  (source: string, substring: string): boolean;
}

const mySearch: SearchFunc = function(src, sub) {
  return src.includes(sub);
};

console.log(mySearch("Hello World", "World")); // true
```

SOLID principy

Úvod do SOLID

SOLID je akronym pěti základních principů objektově orientovaného designu

Vytvořil je Robert C. Martin (Uncle Bob)

Cílem je vytvářet **udržovatelný**, **škálovatelný** a **flexibilní** kód

Úvod do SOLID

SOLID principy:

- **S** - Single Responsibility Principle
- **O** - Open/Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Single Responsibility Principle (SRP)

Každá třída by měla mít pouze jednu odpovědnost (jeden důvod ke změně)

Třída by měla dělat pouze jednu věc a dělat ji dobře

Single Responsibility Principle (SRP)

❌ **Špatně** - třída má více odpovědností:

```
class User {
  constructor(
    public name: string,
    public email: string
  ) {}

  save() { }
  sendEmail() { }
}
```

Single Responsibility Principle (SRP)

✅ **Dobře** - každá třída má jednu odpovědnost:

```
class User {
  constructor(public name: string, public email: string) {}
}

const user = new User("John", "john@example.com");
```

Single Responsibility Principle (SRP)

```
class UserRepository {
  save(user: User) {
    console.log("Saving to database...");
  }
}

const repo = new UserRepository();
repo.save(user);
```

Single Responsibility Principle (SRP)

```
class EmailService {
  send(to: string, message: string) {
    console.log("Sending email...");
  }
}

const emailService = new EmailService();
emailService.send(user.email, "Welcome!");
```

Open/Closed Principle (OCP)

Třídy by měly být otevřené pro rozšíření, ale **uzavřené pro modifikaci**

Měli bychom být schopni přidat novou funkcionalitu bez změny existujícího kódu

Open/Closed Principle (OCP)

❌ **Špatně** - při přidání nového typu musíme modifikovat funkci:

```
class Rectangle {
  constructor(public width: number, public height: number) {}
}
class Circle {
  constructor(public radius: number) {}
}
```

Open/Closed Principle (OCP)

```
function calculateArea(shape: any): number {
  if (shape instanceof Rectangle) {
    return shape.width * shape.height;
  } else if (shape instanceof Circle) {
    return Math.PI * shape.radius ** 2;
  }
  return 0;
}
```

Co když přidáme trojúhelník? Musíme modifikovat `calculateArea`!

Open/Closed Principle (OCP)

✅ **Dobře** - použijeme rozhraní:

```
interface Shape {
  area(): number;
}
```

Open/Closed Principle (OCP)

```
class Rectangle implements Shape {
  constructor(
    private width: number,
    private height: number
  ) {}

  area(): number {
    return this.width * this.height;
  }
}
```

Open/Closed Principle (OCP)

```
class Circle implements Shape {
  constructor(
    private radius: number
  ) {}

  area(): number {
    return Math.PI * this.radius ** 2;
  }
}
```

Open/Closed Principle (OCP)

```
class Triangle implements Shape {
  constructor(
    private base: number,
    private height: number
  ) {}

  area(): number {
    return (this.base * this.height) / 2;
  }
}
```

Open/Closed Principle (OCP)

```
function calculateArea(shape: Shape): number {
  return shape.area();
}

// Can add new shapes without changing the calculateArea
const rect = new Rectangle(10, 5);
const circle = new Circle(5);
const triangle = new Triangle(10, 8);
```

Open/Closed Principle (OCP)

```
console.log(calculateArea(rect));      // 50
console.log(calculateArea(circle));    // 78.54
console.log(calculateArea(triangle)); // 40
```

Liskov Substitution Principle (LSP)

Objekty rodičovské třídy by měly být nahraditelné objekty potomků bez změny funkčnosti

Potomci musí dodržovat „smlouvu“ (kontrakt) rodičovské třídy

Liskov Substitution Principle (LSP)

❌ **Špatně** - porušení LSP:

```
class Bird {
  fly(): void { console.log("Flying..."); }
}

class Penguin extends Bird {
  fly(): void { throw new Error("Penguins can't fly!"); }
}
```

`Penguin` nelze použít místo `Bird` - porušuje očekávané chování!

Liskov Substitution Principle (LSP)

✅ **Dobře** - správná hierarchie:

```
abstract class Bird {
  abstract move(): void;
}
class FlyingBird extends Bird {
  move(): void { console.log("Flying..."); }
}
class Penguin extends Bird {
  move(): void { console.log("Swimming..."); }
}
```

Liskov Substitution Principle (LSP)

```
function makeBirdMove(bird: Bird): void {
  bird.move();
}

const sparrow = new FlyingBird();
const penguin = new Penguin();

makeBirdMove(sparrow); // "Flying..."
makeBirdMove(penguin); // "Swimming..."
```

Oba potomci fungují správně místo rodiče!

Interface Segregation Principle (ISP)

Klienti by neměli záviset na rozhraních, která nepoužívají

Lepší je mít více specifických rozhraní než jedno velké

Interface Segregation Principle (ISP)

❌ **Špatně** - velké rozhraní nutí implementovat nepotřebné metody:

```
interface Worker {
  work(): void;
  eat(): void;
  sleep(): void;
}
```

Interface Segregation Principle (ISP)

```
class Human implements Worker {
  work() { console.log("Working..."); }
  eat() { console.log("Eating..."); }
  sleep() { console.log("Sleeping..."); }
}
```

Interface Segregation Principle (ISP)

```
class Robot implements Worker {
  work() { console.log("Working..."); }
  eat() { /* Robot doesn't eat! */ }
  sleep() { /* Robot doesn't sleep! */ }
}
```

Robot musí implementovat metody, které nedávají smysl!

Interface Segregation Principle (ISP)

✅ **Dobře** - rozdělíme na menší rozhraní:

```
interface CanWork {
  work(): void;
}
interface CanEat {
  eat(): void;
}
interface CanSleep {
  sleep(): void;
}
```

Interface Segregation Principle (ISP)

```
class Human implements CanWork, CanEat, CanSleep {
  work() { console.log("Working..."); }
  eat() { console.log("Eating..."); }
  sleep() { console.log("Sleeping..."); }
}

class Robot implements CanWork {
  work() { console.log("Working..."); }
}
```

Každá třída implementuje jen to, co potřebuje!

Dependency Inversion Principle (DIP)

Závislosti by měly směřovat k abstrakcím, ne ke konkrétním implementacím

- High-level moduly by neměly záviset na low-level modulech
- Oba by měly záviset na abstrakcích (rozhraních)

Dependency Inversion Principle (DIP)

❌ **Špatně** - závislost na konkrétní implementaci:

```
class MySQLDatabase {
  save(data: string): void {
    console.log("Saving to MySQL:", data);
  }
}
```

Dependency Inversion Principle (DIP)

```
class UserService {
  private db = new MySQLDatabase();

  saveUser(name: string): void {
    this.db.save(name);
  }
}
```

Co když chceme použít PostgreSQL? Musíme měnit `UserService`!

Dependency Inversion Principle (DIP)

✅ **Dobře** - závislost na abstrakci:

```
interface Database {
  save(data: string): void;
}
```

Dependency Inversion Principle (DIP)

```
class MySQLDatabase implements Database {
  save(data: string): void {
    console.log("Saving to MySQL:", data);
  }
}

class PostgreSQLDatabase implements Database {
  save(data: string): void {
    console.log("Saving to PostgreSQL:", data);
  }
}
```

Dependency Inversion Principle (DIP)

```
class UserService {
  constructor(
    private db: Database
  ) {}

  saveUser(name: string): void {
    this.db.save(name);
  }
}
```

Dependency Inversion Principle (DIP)

```
// We can simply change the implementation
const mysqlService = new UserService(
  new MySQLDatabase()
);

const postgresService = new UserService(
  new PostgreSQLDatabase()
);
```

Dependency Injection - závislosti předáváme zvenčí!

Shrnutí SOLID

SOLID principy pomáhají vytvářet:

- **Udržovatelný** kód - snadná údržba a změny
- **Testovatelný** kód - snazší psaní testů
- **Flexibilní** kód - snadné rozšiřování
- **Znovupoužitelný** kód - komponenty lze použít jinde
- **Škálovatelný** kód - růst projektu bez problémů

Shrnutí SOLID

Klíčové zásady:

- **SRP**: Jedna třída = jedna odpovědnost
- **OCP**: Rozšiřuj, nemodifikuj
- **LSP**: Potomci musí být nahraditelní
- **ISP**: Malá rozhraní jsou lepší
- **DIP**: Závislosti na abstrakcích, ne implementacích

Generika

Úvod do generik

Generika umožňují psát kód, který funguje s různými typy

Místo konkrétního typu použijeme **typový parametr**

Označuje se <T> nebo jiným písmenem

Úvod do generik

Proč generika?

Bez generik:

```
function identity(arg: number): number {
  return arg;
}
```

S generikami:

```
function identity<T>(arg: T): T {
  return arg;
}
```

Úvod do generik

Jedna funkce pro všechny typy!

Úvod do generik

Použití generické funkce:

```
const num = identity<number>(42);        // T=number
const str = identity<string>("hello");   // T=string

// Type inference
const num2 = identity(42);
// TypeScript inferred: T=number

const str2 = identity("hello");
// TypeScript inferred: T=string
```

Generické funkce

Generické funkce s polem:

```
function firstElement<T>(arr: T[]): T | undefined {
  return arr[0];
}

const numbers = [1, 2, 3];
const first = firstElement(numbers); // number | undefined

const strings = ["a", "b", "c"];
const firstStr = firstElement(strings); // string | undefined
```

Generické funkce

Kombinace typů donutí TypeScript infereovat nejlepší společný typ

```
const arr = [1, "2", 3];
const first = firstElement(arr); // T=(number | string)
```

Generické funkce s více typovými parametry

```
function pair<T, U>(first: T, second: U): [T, U] {
  return [first, second];
}

const p1 = pair("hello", 42);       // [string, number]
const p2 = pair(true, "world");     // [boolean, string]
```

Generické funkce s více typovými parametry

Praktický příklad - swap:

```
function swap<T, U>(tuple: [T, U]): [U, T] {
  return [tuple[1], tuple[0]];
}

const original: [string, number] = ["age", 30];
const swapped = swap(original); // [number, string]
console.log(swapped); // [30, "age"]
```

Generické třídy

Generické třídy fungují stejně jako generické funkce

```
class Box<T> {
  private content: T;

  constructor(value: T) {
    this.content = value;
  }

  getValue(): T {
    return this.content;
  }
```

Generické třídy

```
  setValue(value: T): void {
    this.content = value;
  }
}
```

Generické třídy

```
const numberBox = new Box<number>(123);
console.log(numberBox.getValue()); // 123
numberBox.setValue(456);

const stringBox = new Box<string>("hello");
console.log(stringBox.getValue()); // "hello"
```

Praktický příklad - generický Stack

```
class Stack<T> {
  private items: T[] = [];

  push(item: T): void {
    this.items.push(item);
  }

  pop(): T | undefined {
    return this.items.pop();
  }
```

Praktický příklad - generický Stack

```
  peek(): T | undefined {
    return this.items[this.items.length - 1];
  }
}
```

Praktický příklad - generický Stack

```
const numberStack = new Stack<number>();
numberStack.push(1);
numberStack.push(2);
console.log(numberStack.pop()); // 2

const stringStack = new Stack<string>();
stringStack.push("hello");
stringStack.push("world");
console.log(stringStack.pop()); // "world"
```

Generická omezení (constraints)

Constraints omezují, jaké typy můžeme použít

Používá se klíčové slovo `extends`

```
function getLength<
  T extends { length: number }
>(arg: T): number {
  return arg.length;
}
```

Generická omezení (constraints)

```
getLength("hello");   // OK - string has length
getLength([1, 2, 3]); // OK - array has length
getLength(123);       // Error - number doesn't have length
```

Generická omezení (constraints)

Constraint s rozhraním:

```
interface HasId {
  id: number;
}

function printId<T extends HasId>(obj: T): void {
  console.log(obj.id);
}

printId({ id: 1, name: "John" }); // OK
printId({ name: "Jane" });        // Error: missing 'id'
```

Generická omezení (constraints)

```
interface Nameable {
  name: string;
}

interface Ageable {
  age: number;
}
```

Generická omezení (constraints)

```
function printPerson<
  T extends Nameable & Ageable
>(person: T): void {
  console.log(`${person.name}, ${person.age} years old`);
}

printPerson({ name: "John", age: 30 }); // OK
printPerson({ name: "Jane" }); // Error: missing 'age'
```

Generická omezení (constraints)

Constraint pro klíče objektu:

```
function getProperty<
  T, K extends keyof T
>(obj: T, key: K): T[K] {
  return obj[key];
}
```

- `T` je typ objektu
- `K` je omezen na klíče objektu `T`

Neztrácíme typy při práci s vlastnostmi generického objektu

Generická omezení (constraints)

```
const person = { name: "John", age: 30 };
const name = getProperty(person, "name"); // string
const age = getProperty(person, "age");   // number
getProperty(person, "email"); // Error: "email" not in person
```

Utility typy

TypeScript obsahuje **vestavěné utility typy**

Usnadňují práci s typy bez nutnosti psát vlastní

Utility typy

Partial<T> - všechny vlastnosti volitelné

```
interface User {
  id: number;
  name: string;
  email: string;
}

function updateUser(user: User, updates: Partial<User>): User
{
  return { ...user, ...updates };
}
```

Utility typy

```
const user: User = {
  id: 1,
  name: "John",
  email: "john@example.com"
};

const updated = updateUser(user, { name: "Jane" }); // OK
```

Utility typy

Required - všechny vlastnosti povinné

```
interface Config {
  host?: string;
  port?: number;
}
function connect(config: Required<Config>): void {
  console.log(`Connecting to ${config.host}:${config.port}`);
}

connect({ host: "localhost", port: 3000 }); // OK
connect({ host: "localhost" }); // Error: missing 'port'
```

Utility typy

Readonly<T> - všechny vlastnosti readonly

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

const point: Readonly<Point> = { x: 10, y: 20 };
point.x = 5; // Error: Cannot assign to 'x'
```

Utility typy

Pick<T, K> - vyber pouze určité vlastnosti

```
interface User {
  id: number;
  name: string;
  email: string;
  password: string;
}

type PublicUser = Pick<User, "id" | "name" | "email">;
// { id: number; name: string; email: string; }
```

Utility typy

Omit<T, K> - vynech určité vlastnosti

```
type UserWithoutPassword = Omit<User, "password">;
// { id: number; name: string; email: string; }
```

Utility typy

Record<K, T> - vytvoř objekt s danými klíči a typem hodnot

```
type Role = "admin" | "user" | "guest";

const permissions: Record<Role, string[]> = {
  admin: ["read", "write", "delete"],
  user: ["read", "write"],
  guest: ["read"]
};
```

Pokročilé koncepty

Dekorátory

Dekorátor je speciální druh deklarace, kterou lze připojit k třídě, metodě, vlastnosti nebo parametru

Označuje se symbolem `@`

Umožňuje přidat metadata nebo změnit chování

Dekorátory

Pro použití dekorátorů v TypeScriptu <5.0 musíte povolit experimentální funkci v `tsconfig.json`:

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

Nebo použít TypeScript 5.0+ s novými dekorátory (bez experimentalDecorators)

Typy dekorátorů

TypeScript podporuje několik typů dekorátorů:

1. **Class decorators** - pro třídy
2. **Method decorators** - pro metody
3. **Property decorators** - pro vlastnosti
4. **Parameter decorators** - pro parametry metod

Typy dekorátorů

Class decorator

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
}
```

`Object.seal` zabraňuje změnám na třídě a jejím prototypu

Typy dekorátorů

```
@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(title: string) {
    this.title = title;
  }
}
```

Typy dekorátorů

Method decorator

```
function log(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;
```

Typy dekorátorů

```
  descriptor.value = function(...args: any[]) {
    console.log(`Calling ${propertyKey} with`, args);
    const result = originalMethod.apply(this, args);
    console.log(`Result:`, result);
    return result;
  };
}
```

Typy dekorátorů

```
class Calculator {
  @log
  add(a: number, b: number): number {
    return a + b;
  }
}

const calc = new Calculator();
calc.add(2, 3);
// Calling add with [2, 3]
// Result: 5
```

Praktické příklady

Readonly decorator - znemožní změnu metody

```
function readonly(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  descriptor.writable = false;
}
```

Praktické příklady

```
class Person {
  @readonly
  getName() {
    return "John";
  }
}
```

Pokud je funkce označena jako readonly, nemůže být přepsána

```
const person = new Person();
person.getName(); // "John"
person.getName = function() { return "Jane"; }; // Error
```

Praktické příklady

Validation decorator - validace parametrů

```
function validate(
  target: any,
  propertyKey: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;
```

Praktické příklady

```
  descriptor.value = function(...args: any[]) {
    if (args.some(arg => arg < 0)) {
      throw new Error("Arguments must be positive");
    }
    return originalMethod.apply(this, args);
  };
}
```

Praktické příklady

```
class MathOperations {
  @validate
  divide(a: number, b: number): number {
    return a / b;
  }
}

const math = new MathOperations();
math.divide(10, 2);  // OK
math.divide(-10, 2); // Error: Arguments must be positive
```

Decorator factories

Decorator factory je funkce, která vrací dekorátor

Umožňuje předávat parametry do dekorátoru

```
function component(prefix: string) {
  return function(constructor: Function) {
    console.log(`${prefix}: ${constructor.name}`);
  };
}
```

Decorator factories

```
@component("APP")
class UserComponent {
  // ...
}
// Output: "APP: UserComponent"
```

Decorator factories

Parametrizovaný **method decorator**

```
function delay(milliseconds: number) {
  return function(
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const originalMethod = descriptor.value;
```

Decorator factories

```
    descriptor.value = async function(...args: any[]) {
      await new Promise(
        resolve => setTimeout(resolve, milliseconds)
      );
      return originalMethod.apply(this, args);
    };
  };
}
```

Decorator factories

```
class API {
  @delay(1000)
  async fetchData() {
    return "Data loaded";
  }
}

const api = new API();
await api.fetchData(); // Wait 1s before calling
```

Decorator factories

Kombinace více parametrů

```
function logWithLevel(level: "info" | "warn" | "error") {
  return function(
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const originalMethod = descriptor.value;
```

Decorator factories

```
    descriptor.value = function(...args: any[]) {
      console[level](
        `[${level.toUpperCase()}] ${propertyKey}`,
        args
      );
      return originalMethod.apply(this, args);
    };
  };
}
```

Decorator factories

```
class Service {
  @logWithLevel("info")
  start() { }

  @logWithLevel("error")
  handleError(err: Error) { }
}
```

Property decorators

Property decorator se aplikuje na vlastnosti třídy

Přijímá dva parametry:

- `target` - prototyp třídy (nebo konstruktor pro statické vlastnosti)
- `propertyKey` - název vlastnosti

```
function format(
  target: any,
  propertyKey: string
) {
  // Dekorátor pro vlastnost
}
```

Property decorators

Příklad: **uppercase** vlastnost

```
function uppercase(target: any, propertyKey: string) {
  let value: string;

  const getter = function() {
    return value;
  };
```

Property decorators

```
  const setter = function(newVal: string) {
    value = newVal.toUpperCase();
  };

  Object.defineProperty(target, propertyKey, {
    get: getter,
    set: setter,
    enumerable: true,
    configurable: true
  });
}
```

Property decorators

```
class User {
  @uppercase
  name: string;

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

const user = new User("john");
console.log(user.name); // "JOHN"
```

Property decorators

Required field validator

```
const requiredFields: Map<any, string[]> = new Map();

function required(target: any, propertyKey: string) {
  const constructor = target.constructor;
  const fields = requiredFields.get(constructor) || [];
  fields.push(propertyKey);
  requiredFields.set(constructor, fields);
}
```

Property decorators

```
function validate(instance: any): boolean {
  const fields = requiredFields.get(
    instance.constructor
  ) || [];

  return fields.every(field => {
    return instance[field] !== undefined
      && instance[field] !== null;
  });
}
```

Property decorators

```
class Form {
  @required
  username: string;

  @required
  email: string;

  age?: number;
}
```

Property decorators

```
const form = new Form();
form.username = "john";
validate(form); // false - chybí email
```

Dekorátory v praxi

Reálné frameworky TypeScriptu intenzivně využívají dekorátory

- **NestJS** - backend framework
- **TypeORM** - ORM knihovna
- **Angular** - frontend framework
- **MobX** - state management

Dekorátory v praxi

NestJS - HTTP controller

```
import { Controller, Get, Post, Body } from '@nestjs/common';

@Controller('users')
export class UsersController {
  @Get()
  findAll() {
    return 'This returns all users';
  }
```

Dekorátory v praxi

```
  @Get(':id')
  findOne(@Param('id') id: string) {
    return `This returns user #${id}`;
  }

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return 'This creates a new user';
  }
}
```

Dekorátory v praxi

NestJS - Dependency Injection

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  private users = [];

  findAll() {
    return this.users;
  }
```

Dekorátory v praxi

```
  create(user: any) {
    this.users.push(user);
  }
}
```

Dekorátory v praxi

```
@Controller('users')
export class UsersController {
  constructor(
    private usersService: UsersService
  ) {}

  @Get()
  findAll() {
    return this.usersService.findAll();
  }
}
```

Dekorátory v praxi

TypeORM - Entity definice

```
// Import decorators from TypeORM
import {
  Entity,
  PrimaryGeneratedColumn,
  Column
} from 'typeorm';
```

Dekorátory v praxi

```
@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  firstName: string;

  @Column()
  lastName: string;
```

Dekorátory v praxi

```
  @Column({ unique: true })
  email: string;

  @Column({ default: true })
  isActive: boolean;
}
```

Dekorátory v praxi

TypeORM - Vztahy mezi entitami

```
import {
  Entity,
  Column,
  OneToMany,
  ManyToOne
} from 'typeorm';
```

Dekorátory v praxi

```
@Entity()
export class Post {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  title: string;

  @ManyToOne(() => User, user => user.posts)
  author: User;
}
```

Dekorátory v praxi

```
@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @OneToMany(() => Post, post => post.author)
  posts: Post[];
}
```

Dekorátory v praxi

TypeORM - Validace a transformace

```
import {
  Entity,
  Column,
  BeforeInsert
} from 'typeorm';
import * as bcrypt from 'bcrypt';
```

Dekorátory v praxi

```
@Entity()
export class User {
  @Column()
  password: string;
```

Dekorátory v praxi

```
  @BeforeInsert()
  async hashPassword() {
    this.password = await bcrypt.hash(
      this.password,
      10
    );
  }
}
```

Dekorátor `@BeforeInsert()` zajistí, že heslo bude **zahashováno** před uložením entity do databáze

Mixiny

Co jsou mixiny?

Mixin je vzor, který umožňuje kombinovat chování z více tříd

TypeScript nepodporuje vícenásobnou dědičnost, ale mixiny jsou alternativou

Mixiny přidávají funkcionalitu do třídy bez použití dědičnosti

Co jsou mixiny?

Proč používat mixiny?

- **Kombinace funkcionality** z více zdrojů
- **Znovupoužitelnost** kódu
- Obejití omezení **jednoduché dědičnosti**
- **Kompozice** místo dědičnosti

Implementace mixinů v TypeScriptu

Základní pattern pro mixiny:

```
// Mixin type
type Constructor<T = {}> = new (...args: any[]) => T;
```

Implementace mixinů v TypeScriptu

```
// Mixin function
function Timestamped<
  TBase extends Constructor
>(Base: TBase) {
  return class extends Base {
    timestamp = new Date();
    getTimestamp() {
      return this.timestamp;
    }
  };
}
```

Implementace mixinů v TypeScriptu

```
// Base class
class User {
  constructor(public name: string) {}
}

// Apply mixin
const TimestampedUser = Timestamped(User);

const user = new TimestampedUser("John");
console.log(user.name);                // "John"
console.log(user.getTimestamp()); // current date
```

Implementace mixinů v TypeScriptu

```
function Activatable<
  TBase extends Constructor
>(Base: TBase) {
  return class extends Base {
    isActive = false;
    activate() { this.isActive = true; }
    deactivate() { this.isActive = false; }
  };
}
```

Implementace mixinů v TypeScriptu

```
// Combining multiple mixins
class User {
  constructor(public name: string) {}
}

const EnhancedUser = Timestamped(Activatable(User));
```

Implementace mixinů v TypeScriptu

```
const user = new EnhancedUser("John");
console.log(user.name);           // "John"
console.log(user.getTimestamp()); // current date
user.activate();
console.log(user.isActive);       // true
```

Disposable pattern

```
function Disposable<
  TBase extends Constructor
>(Base: TBase) {
  return class extends Base {
    isDisposed = false;
    dispose() { this.isDisposed = true; }
  };
}
```

Disposable pattern

```
class Resource {
  constructor(public id: number) {}
}

const DisposableResource = Disposable(Resource);
```

Composition over Inheritance

Prefer composition over inheritance je důležitý princip v OOP

Co to znamená?

Místo dědičnosti („je“) preferujeme kompozici („má“)

Composition over Inheritance

Problémy s dědičností:

- **Těsné vazby** - potomci závisí na rodiči
- **Křehkost** - změna v rodiči může rozbít potomky
- **Omezení** - můžeme dědit jen z jedné třídy
- **Neflexibilita** - složité změnit hierarchii později

Composition over Inheritance

❌ **Špatně** - nadměrné použití dědičnosti:

```
class Vehicle {
  move() { console.log("Moving..."); }
}
class FlyingVehicle extends Vehicle {
  fly() { console.log("Flying..."); }
}
class SwimmingVehicle extends Vehicle {
  swim() { console.log("Swimming..."); }
}
```

Composition over Inheritance

Co ale s futuristickou lodí, která umí plout i létat? 🤔

Composition over Inheritance

✅ **Dobře** - použití kompozice:

```
interface Movement {
  execute(): void;
}
```

Composition over Inheritance

```
class FlyMovement implements Movement {
  execute() { console.log("Flying..."); }
}

class SwimMovement implements Movement {
  execute() { console.log("Swimming..."); }
}

class DriveMovement implements Movement {
  execute() { console.log("Driving..."); }
}
```

Composition over Inheritance

```
class Vehicle {
  constructor(
    private movements: Movement[]
  ) {}

  move() {
    this.movements.forEach(m => m.execute());
  }
}
```

Composition over Inheritance

```
// Flexible functionality composition
const airplane = new Vehicle([
  new FlyMovement(),
]);

const amphibiousVehicle = new Vehicle([
  new DriveMovement(),
  new SwimMovement()
]);
```

Composition over Inheritance

Výhody kompozice:

- **Flexibilita** - snadno měníme chování
- **Znovupoužitelnost** - komponenty lze kombinovat
- **Loose coupling** - volnější vazby
- **Testovatelnost** - snadnější testování jednotlivých částí

Composition over Inheritance

Praktický příklad - UI komponenty:

```
interface Plugin {
  apply(): void;
}
```

Composition over Inheritance

```
class Editor {
  private plugins: Plugin[] = [];

  addPlugin(plugin: Plugin) {
    this.plugins.push(plugin);
  }

  initialize() {
    this.plugins.forEach(p => p.apply());
  }
}
```

Composition over Inheritance

```
class SpellCheckPlugin implements Plugin {
  apply() { console.log("Spell check enabled"); }
}

class AutoSavePlugin implements Plugin {
  apply() { console.log("Auto-save enabled"); }
}
```

Composition over Inheritance

```
const editor = new Editor();
editor.addPlugin(new SpellCheckPlugin());
editor.addPlugin(new AutoSavePlugin());
editor.initialize();
```

Můžeme snadno přidávat/odebírat funcionality!

Composition over Inheritance

Kdy použít dědičnost vs. kompozici?

Použijte dědičnost když:

- Máte jasný vztah „je“ (Dog **je** Animal)
- Sdílíte společný kód mezi příbuznými třídami
- Potřebujete polymorfismus

Použijte kompozici když:

- Máte vztah „má“ nebo „umí“
- Chcete flexibilně kombinovat funkce nebo často měnit funkcionalitu

Závěr

Shrnutí

Co jsme se naučili:

TypeScript základy

- Typový systém, základní typy
- Union, Intersection typy
- Type Guards a Discriminated Unions
- Type aliases a assertions

Shrnutí

Objektově orientované programování

- Třídy, konstruktory, vlastnosti
- Statické členy
- Zapouzdření (public, private, protected)
- Dědičnost (extends, super)
- Polymorfismus

Shrnutí

Abstrakce a rozhraní

- Abstraktní třídy
- Rozhraní (interfaces)
- Index Signatures
- Rozdíly a kdy co použít

Shrnutí

SOLID principy

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Shrnutí

Pokročilé koncepty

- Generika pro znovupoužitelný kód
- Utility typy (Partial, Required, Pick, …)
- Dekorátory pro metadata
- Mixiny pro kompozici
- Composition over Inheritance

Shrnutí

Klíčové poznatky

- TypeScript přidává **typovou bezpečnost** do JavaScriptu
- OOP principy pomáhají **organizovat** a **škálovat** kód
- **SOLID principy** vedou k lepšímu návrhu tříd a rozhraní
- Generika umožňují psát **flexibilní** a **znovupoužitelný** kód
- **Composition over inheritance** pro větší flexibilitu
- Dekorátory a mixiny jsou **pokročilé vzory** pro rozšíření funkcionality

Užitečné zdroje

Oficiální dokumentace

- TypeScript Handbook:
  https://www.typescriptlang.org/docs/
- TypeScript Playground (online editor):
  https://www.typescriptlang.org/play
- TypeScript GitHub:
  https://github.com/microsoft/TypeScript

Užitečné zdroje

Doporučená četba

- Effective TypeScript (Dan Vanderkam)
- Programming TypeScript (Boris Cherny)
- TypeScript Deep Dive
  https://basarat.gitbook.io/typescript/

Užitečné zdroje

Nástroje a ekosystém

- TSLint / ESLint - lintování kódu
- Prettier - formátování kódu
- ts-node - spouštění TypeScript bez kompilace
- DefinitelyTyped - typové definice pro JavaScript knihovny

Užitečné zdroje

Best practices

- Používejte **strict mode** v tsconfig.json
- Preferujte **interface** nad type alias pro objekty
- Používejte **readonly** kde je to možné
- Vyhněte se `any`, používejte `unknown`
- Dokumentujte kód pomocí JSDoc komentářů