



# TypeScript

## Operátory typov a genericita

kurz Vývoj progresívnych  
webových aplikácií

Eduard Kuric

# Typ prienik

- Kombinácia jedného a viac typov na vytvorenie nového typu, ktorý bude obsahovať všetky atribúty existujúcich typov
- Poriadie typov nie je dôležité
- Používame operátor &

```
type typeAB = typeA & typeB;
```

# Typ prienik /2

```
interface Identity {  
    id: number;  
    fullName: string;  
}  
interface Contact {  
    email: string;  
    phone: string;  
}  
type Employee = Identity & Contact;  
let e: Employee = {  
    id: 100,  
    fullName: 'John Doe',  
    email: 'john.doe@example.com',  
    phone: '(408)-897-5684'  
};
```

# Typ prienik /2

```
interface Identity {  
    id: number;  
    fullName: string;  
}  
interface Contact {  
    email: string;  
    phone: string;  
}  
type Employee = Identity & Contact  
let e: Employee = {  
    id: 100,  
    fullName: 'John Doe',  
    email: 'john.doe@example.com',  
    phone: '(408)-897-5684'  
};
```

*// Poznamenajme typ union*  
let varName = typeA | typeB;

varName môže obsahovať hodnotu  
typu A **alebo** B

# Operátor typeof

```
type alphanumeric = string | number;
```

```
function add(a: alphanumeric, b: alphanumeric) {  
    if (typeof a === 'number' && typeof b === 'number') {  
        return a + b;  
    }  
  
    if (typeof a === 'string' && typeof b === 'string') {  
        return a.concat(b);  
    }  
  
    throw new Error('Invalid arguments. Both arguments  
                        must be either numbers or strings.');
```

```
}
```

# Operátor instanceof

```
class Customer {}  
class Supplier {}  
type BusinessPartner = Customer | Supplier;  
  
function signContract(partner: BusinessPartner) : string {  
    if (partner instanceof Customer) {}  
    if (partner instanceof Supplier) {}  
    ...  
}
```

# Operátor **in**

- kontrola na existenciu atribútu/metódy v objekte

```
class Customer {  
    isCreditAllowed(): boolean {}  
}  
  
let partner = new Customer()  
  
if ('isCreditAllowed' in partner) {  
    // true  
}
```

# Konverzia typov (type casting)

- Používame operátor `as` alebo `<>`

```
let a: typeA;
```

```
let b = a as typeB;
```

```
let a: typeA;
```

```
let b = <typeB>a;
```

```
HTMLElement extends HTMLElement element Element
```

```
let el: HTMLElement;
```

```
el = new HTMLElement(); // subclass
```



# Uplatnenie typu (type assertion)

- Inštruujú transpilátor, aby zaobchádzal s hodnotou ak so uvedeným typom (**nejde o konverziu typov**)
- Používame operátory `as` a `<>`

```
function getNetPrice(price: number, discount: number,  
                    format: boolean): number | string {  
    let netPrice = price * (1 - discount);  
    return format ? `${netPrice}` : netPrice;  
}
```

```
let netPrice = getNetPrice(100, 0.05, true) as string;  
let netPrice = <number>getNetPrice(100, 0.05, false);
```

# Genericita

- Vytváranie znovupoužiteľných, zovšeobecnených a typovobezpečných funkcií, tried a rozhraní
- Uvažujme funkcie:

```
function getRandomNumberElement(items: number[]): number {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}  
  
function getRandomStringElement(items: string[]): string {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}
```

# Genericita

- Vytváranie znovupoužiteľných, zovšeobecnených a typovobezpečných funkcií, tried a rozhraní
  - Typová kontrola v čase transpilácie
  - Eliminácia konverzie typov (type casting)
  - Umožňujú implementovať generické algoritmy
- Uvažujme funkcie

```
function getRandomElement(items: number[]): number {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}  
  
function getRandomStringElement(items: string[]): string {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}
```

Logika je rovnaká/duplicitná

# Genericita – premenná T

- Môžeme použiť typ any

```
function getRandomAnyElement(items: any[]): any {}
```

- Riešenie funguj, ale nie je typovo bezpečné
  - neumožňuje nám vynútiť typ vráteného elementu
- Použijeme radšej generický typ – **typová premenná T**

```
function getRandomElement<T>(items: T[]): T {  
    let randomIndex = Math.floor(Math.random() * items.length);  
    return items[randomIndex];  
}  
  
let numbers = [1, 5, 7, 4, 2, 9];  
let randomEle = getRandomElement<number>(numbers);
```

# Genericita – premenná T

- Môžeme použiť typ any

```
function getRandomAnyElement(items: any[]): any {}
```

- Riešenie

- 

V praxi používame odvodenie typu pre argument, neuvádzame explicitne typ

- Použitie

```
function
```

```
let
```

```
return
```

```
}
```

```
let numbers = [1, 5, 7, 4, 2, 9];
```

```
let randomEle = getRandomElement<number>(numbers);
```

# Genericita – viacero typov

```
// spoji objekty
function merge<U, V>(obj1: U, obj2: V) {
    return {
        ...obj1,
        ...obj2
    };
}

let result = merge(
    { name: 'John' },
    { jobTitle: 'Frontend Developer' }
);

// vystup: { name: 'John', jobTitle: 'Frontend Developer' }
```

# Genericita – obmedzenie typu

- Nič nám nebráni, aby sme zavolali funkciu merge (z predošlého slajdu) takto:

```
let result = merge(  
    { name: 'John' },  
    2  
);
```

- Výstup: { name: 'John' } // TS nevyhodi chybu

# Genericita – extends

- Použijeme extends na určenie typov

```
function merge<U extends object,  
                V extends object>(obj1: U, obj2: V) {  
    return {  
        ...obj1,  
        ...obj2  
    };  
}
```

- Funkcia už bude **fungovať iba s objektami** (typ object), transpilátor by už vyhodil chybu



# Typ atribútu v objekte `extends keyof`

```
// error Type 'K' cannot be used to index type 'T'.
function prop<T, K>(obj: T, key: K) {
    return obj[key];
}

// riesenie, zaistime, aby K bolo klucom T
function prop<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}

// transpiler OK
let str = prop({ name: 'John' }, 'name');

// transpiler error
// Argument of type '"age"' is not assignable to parameter of type '"name"'.
let str = prop({ name: 'John' }, 'age');
```

# Generická trieda

```
class Stack<T> {  
    private elements: T[] = [];  
    constructor(private size: number) {}  
    isEmpty(): boolean {}  
    isFull(): boolean {}  
    push(element: T): void {}  
    pop(): T {}  
}  
let numbers = new Stack<number>(5);
```

# Generické rozhranie

- **Atribúty/vlastnosti objektu**

```
interface Pair<K, V> {  
    key: K;  
    value: V;  
}  
  
let month: Pair<string, number> = {  
    key: 'January',  
    value: 1  
};
```

# Generické rozhranie /2

- **Metódy**

```
interface Collection<T> {  
    add(o: T): void;  
    remove(o: T): void;  
}  
class List<T> implements Collection<T>{  
    private items: T[] = [];  
    add(o: T): void {}  
    remove(o: T): void {}  
}  
let list = new List<number>();
```

# Generické rozhranie /3

- Typy indexov

```
interface Options<T> {  
    [name: string]: T  
}  
  
let inputOptions: Options<boolean> = {  
    'disabled': false,  
    'visible': true  
};
```