Generics

Interfaces, Generic Functions and Classes

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

http://softuni.bg

Have a Question?

Software University

sli.do

#TypeScript

Table of Content

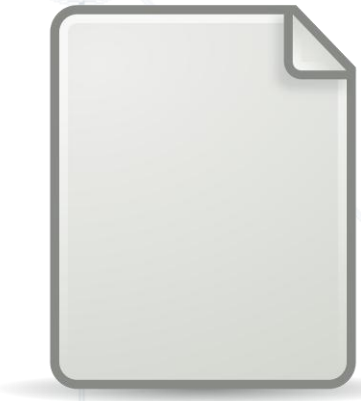
Software University

1. Generics
   - **Generic functions**
   - **Generic interfaces**
   - **Generic classes**
   - **Generic type constraints**
   - **Mapped Types using Generics**
   - **Advanced Mapped Types**

Generics

Definition

- Used to build **reusable** software components
- The components will work with a **multitude** of types instead of a single type
- Defined by type variable - **<LETTER>**
- Follows the **DRY** (**D**on't **R**epeat **Y**ourself) principle
- Allows us to **abstract** the type
- Generics can be applied to **functions**, **classes**, **interfaces** and **mapped types**



Example: Generic vs Non-Generic

- Generic

```
function echo<T>(arg: T): T {
    console.log(typeof arg);
    // It will print number and
string when the function is
invoked
    return arg;
}
echo(11111);
echo('Hello');
```

- Non-generic

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

```
function echo(arg: string): string {
    return arg;
}
```

Generic Functions

Software University

- Generic functions allow us to work with user input with **unknown** data type
- It is a way of telling the function that whatever **type** is **passed** to it the **same** type shall be **returned**
- Put some **constraints** to user input
- We can put **more than one** type variable in the generic function

Example: Generic Functions

Software University

```
const takeLast = <T>(array: T[]) => {
    return array.pop();
}
const sample = takeLast(['Hello', 'World', 'TypeScript']);
const secondSample = takeLast([1, 2, 3, 4]);
console.log(sample, secondSample); //TypeScript, 4
```

```
const makeTuple = <T, V>(a: T, b: V) => {
    return [a, b];
}
const firstTuple = makeTuple(1, 2);
const secondTuple = makeTuple('a', 'b');
console.log(firstTuple, secondTuple); //[1, 2], [a, b]
```

Generic Interfaces

- Using **generic interfaces** we can define **generic functions** too

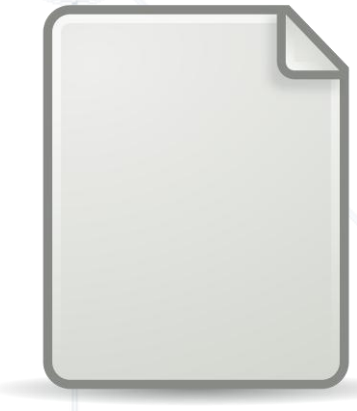
```
interface GenericConstructor<T, V> {
    (arg: T, param: V): [T, V];
}

const generatedFn: GenericConstructor<string, string> = <T, V>(arg: T, param: V)
 => {
       return [arg, param];
}

const sample = generatedFn('Hello', 'World');
console.log(sample); // [Hello, World]
```

Generic Classes

- Generics can be used on:
  - The **properties** of the class
  - The **methods** of the class
- To define a generic class we put **<LETTER> after the name of the class**
- We can use **multiple** type variables
- Generic classes can implement **generic interfaces**



Example: Generic Class Using Single Parameter

Software University

```
class Collection<T> {
    public data: T[];
    constructor(...elements: T[]) { this.data = elements; }

    addElement(el: T) { this.data.push(el); }

    removeElement(el: T) {
        let index = this.data.indexOf(el);
        if (index > -1) {
            this.data.splice(index, 1);
        }
    }

    reverseElements() { return this.data.reverse(); }

    showElements() { return this.data; }
}
```

Example: Generic Class Using Multiple Parameters


```
class UserInput<F, S> {
    public first: F;
    public second: S;
    constructor (f: F, s: S) {
        this.first = f;
        this.second = s;
    }

    showBoth() {
        return  `First: ${this.first}, second: ${this.second}`;
    }
}

let sample = new UserInput('Ten', 10);
let test = new UserInput(1, true);
console.log(sample.showBoth()); // First: Ten, second: 10
console.log(test.showBoth()); // First: 1, second: true
```

Example: Generic Class Implements Interface

Software University

```
interface ShowEnum<T> { returnPair(): [string, T | number]; }

class EnumOption<T> implements ShowEnum<T> {
    public key: string;
    public value: T | number;
    static counter = 0;
    constructor(k: string, v: T) {
        this.key = k;
        this.value = v ?? EnumOption.counter++;
    }
    returnPair(): [string, T | number] { return [this.key, this.value]; }
}

let test: ShowEnum<string> = new EnumOption('January', 'jan');
console.log(test.returnPair()); // ['January', 'jan']
let test2: ShowEnum<number | undefined> = new EnumOption('January', undefined);
console.log(test2.returnPair()); // ['January', 0]
let test3: ShowEnum<number | undefined> = new EnumOption('February', 2);
console.log(test3.returnPair()); // ['February', 2]
```

Generic Type Constraints

Software University

- In TypeScript we can make sure that a variable **has** at least **some specific information** contained in it
- Constraints are enforced by **extends** keyword and can be any type including advanced types

```
function fullName<T extends {fName: string} & {lName: string}>(obj: T) {
      return `The full name is ${obj.fName} ${obj.lName}.`;
}

let output = fullName({fName: 'Svetoslv', lName: 'Dimitrov'});

console.log(output); // The full name is Svetoslav Dimitrov
```

Mapped Types using Generics

Software University

- Creates new types by **transforming each property of an existing type**
  - Uses **Generics** to be reusable

```
type Point = { x: number; y: number; };
type Colors = { red: string; blue: string; };

type Optional<T> = { [K in keyof T]?: T[K] };

type PartialPoint = Optional<Point>;
// { x?: number; y?: number; }
type PartialColors = Optional<Colors>;
// { red?: string; blue?: string; }
```

Indexed Access Types

- Allow us to **look up specific properties** in a type
- The index used is also a type, so we can use **literals**, **unions**, **keyof** and **other types**

```
type Person = { name: string, age: number, isLocal: boolean};

type Age = Person['age'];
type All = Person[keyof Person];
type local = Person['name' | 'isLocal'];
type temp = 'age' | 'isLocal';
type census = Person[temp];

type test = Person['test']        //Error: 'test' not in Person
```

number

string | number | boolean

string | boolean

number | boolean

Conditional Types

- Allow us to choose types based on conditional logic
- Takes the form **A extends B ? TrueType : FalseType**
  - Also works as **type narrowing**

```
type Age = { age: number};
type Person = { name: string, age: number};

type strOrNum = Person extends Age ? string : number;

type NameType<T> = T extends {name: unknown} ? T['name'] : T;
```

type strOrNum = string

OK, since true condition must have 'name'

Advanced Mapped Types

Software University

- Combining **Conditional Types** with **Generics** and **Mapped Types** can allow us to create some very useful functionality

```
type Employee = { name:string, age: number, salary: number};

type NumberPropertyNames<T> = {
  [K in keyof T]: T[K] extends number ? K : never;
}[keyof T];

type numberKeys = NumberPropertyNames<Employee>
```

Type for values that never occur

Automatically dropped from unions like keyof results

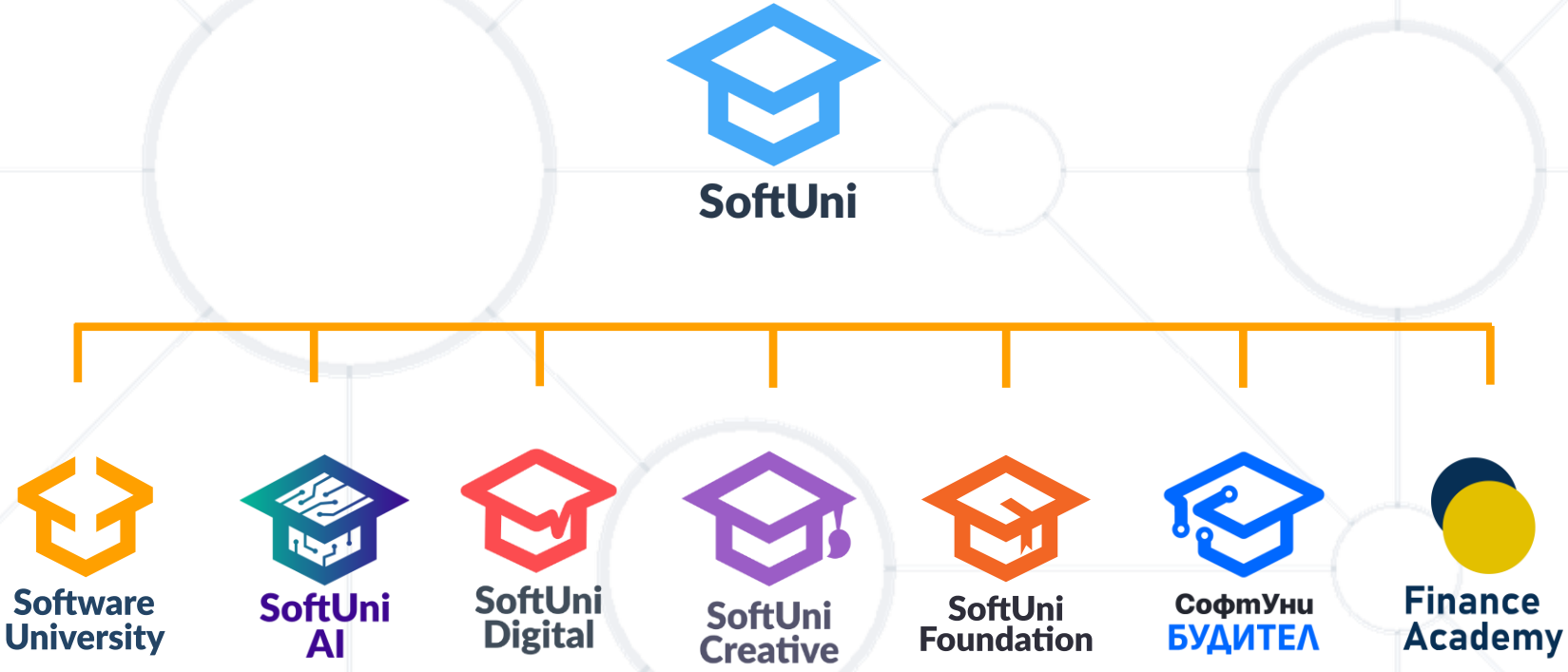
Type indexing

'age' | 'salary'

Summary

- Generics are used to:
  - Abstract data types
  - Build reusable components
  - Allow flexible constraints on parameters
- We can use them in:
  - Functions
  - Classes - their properties and methods
  - Interfaces
  - Mapped Types

Questions?



SoftUni Diamond Partners

Software University

Coca-Cola HBC Bulgaria

Postbank

top EMPLOYER България Bulgaria 2025

FOR A BETTER WORLD OF WORK

SUPER HOSTING .BG

createX

DRAFT KINGS

THE CROWN IS YOURS

INDEAVR

Serving the high achievers

encorp.io

VIVACOM

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

Software University

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg