

TypeScript Generics

Interfaces, Generic Functions and Classes



SoftUni Team
Technical Trainers



**Software
University**



**SoftUni
Foundation**



Software University

<http://softuni.bg>

1. Interfaces

2. Generics

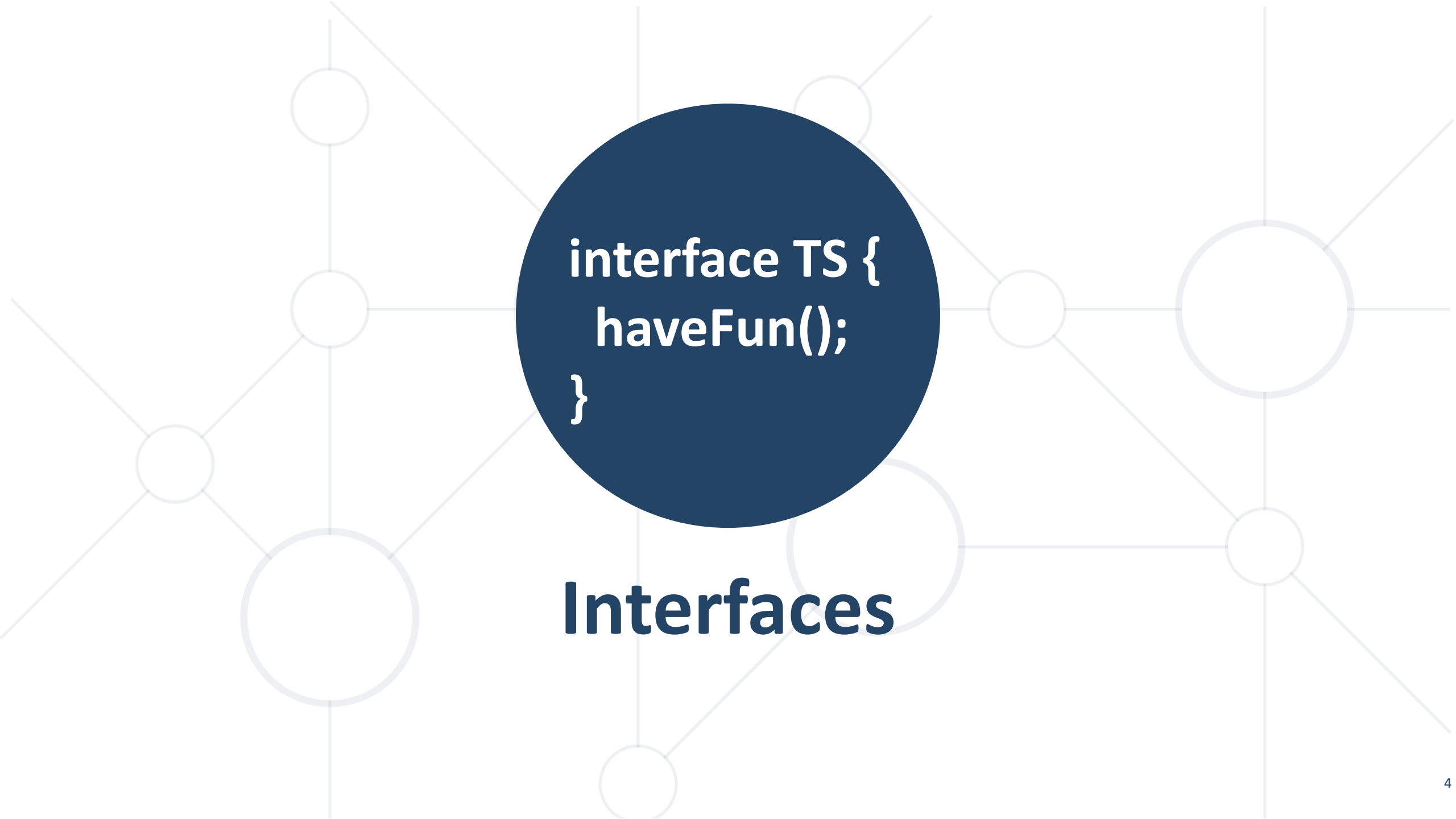
- Generic functions
- Generic interfaces
- Generic classes
- Generic type constraints



Have a Question?

sli.do

#tbd

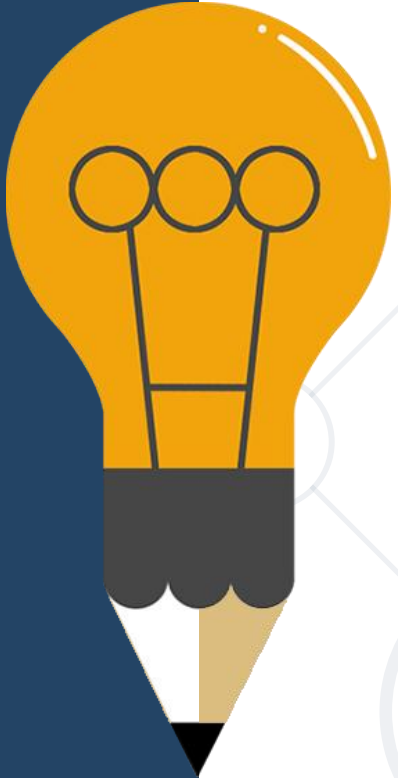


```
interface TS {  
  haveFun();  
}
```

Interfaces

Definition

- Defined by using keyword **interface**
- Often called **duck typing** or **structural typing**
- We can define **properties**, **methods** and **events** also called **members** of the interface
- The interface **contains** only **the declaration** of its members
- Helps to **standardize the structure** of the deriving classes



Example: Basic interface

```
interface Person {  
  fullName: string,  
  email: string,  
}
```

Interface declaration

```
let thomas: Person = {  
  fullName: 'Thomas Doe',  
  email: 'thomas@test.test',  
}
```

Declare a **variable** with the **interface as type** in order to follow the **structure**

```
console.log(thomas.fullName) //Thomas Doe
```

- Interfaces in TypeScript can also describe function types
 - They are constructed in the following way:

```
interface Name {  
  (paramOne: type, paramTwo: type, ...paramN: type): type;  
}
```

- Where in the parantheses we put the **parameters** we want to pass to the function with their **types**, splitted by comma.
- On the right side is the **return type** of the function

Example: Describe function types

```
interface Calculator {  
    (numOne: number, numTwo: number, operation: string): number;  
}  
  
let calc: Calculator = function (a: number, b: number, operation:  
    string): number {  
    let result: number = 0;  
    const addition = () => result = a + b; ;  
    const parser = {  
        'addition': addition,  
    }  
    parser[operation]();  
    return result;  
}
```


- Interfaces can be implemented by classes using the keyword **implement**
- A class that implements an interface **must have** all the properties defined in the interface
 - Describes the **public** side of the class

```
interface Person { ... }
```

```
class Teacher implements Person { ... }
```

Example: Implemented by class

```
interface ClockLayout {  
    hour: number;  
    minute: number;  
    showTime(h: number, m: number): string;  
}  
  
class Clock implements ClockLayout {  
    public hour;  
    public minute;  
    constructor(h: number, m: number) {  
        this.hour = h;  
        this.minute = m;  
    }  
    showTime() {  
        return `Current time: ${this.hour}:${this.minute}`;  
    }  
}
```

- Interfaces can extend **classes** and other **interfaces**
 - Extending **classes**
 - The extended interface **inherits** all of the members of the class including **private** and **protected** members
 - The interface **does not inherit** the **implementations** of the members (e.g. method implementations)
 - Extending other **interfaces**
 - Creates a **combination** of all interfaces

Example: Extending interfaces

```
class Computer {  
    public RAM;  
    constructor(r: number) { this.RAM = r; }  
    showParams(): string { return `${this.RAM}`; }  
}  
interface Parts extends Computer {  
    CPU: string;  
    showParts(): string;  
}  
class PC extends Computer implements Parts {  
    public keyboard;  
    public CPU;  
    constructor(RAM: number, CPU:string) { super(RAM); this.CPU = CPU; }  
    showParts() {  
        return `${this.RAM} ${this.CPU}`;  
    }  
}
```



Generics

Definition

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



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 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

```
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]
```

- 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]
```

- Generics can be used on:
 - The **properties** of the class
 - The **methods** of the class
- To define 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

```
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

```
interface ShowComponents<T, V> {  
    print(key: T, value: V): string;  
}  
  
class Components<T, V> implements ShowComponents<T, V> {  
    public key: T;  
    public value: V;  
    constructor(k: T, v: V) {  
        this.key = k;  
        this.value = v;  
    }  
    print(){  
        return `Key: ${this.key} and value: ${this.value}`;  
    }  
}  
  
let test: ShowComponents<string, string> = new Components('New', 'Test');  
console.log(test.print('Test', 'Hello')); // Key: New and value: Test
```

- In TypeScript we can make sure that sudden type variable **has** at least **some information** containing in it
- Constraints are enforced by **extends** keyword

```
function fullName<T extends { fName: string, lName: string }>(obj: T) {  
    return `The full name is ${obj.fName} ${obj.lName}.`;  
}  
  
let output = fullName({fName: 'Svetoslav', lName: 'Dimitrov'});  
console.log(output);// The full name is Svetoslav Dimitrov
```



Live Exercises

- Generics are use to:
 - **Abstract** data types
 - Build **reusable** components
- We can use them in:
 - **Functions**
 - **Classes** - their **properties** and **methods**
 - **Interfaces**



SoftUni Diamond Partners



XSsoftware



SBTech
we know sports



telenor



SoftwareGroup
doing it right

NETPEAK



SmartIT



Postbank

Решения за твоето утре



INDEAVR

Serving the high achievers



INFRAGISTICS®



STEMO®
Computer Systems & Software

SUPERHOSTING.BG

SoftUni Organizational Partners



One
SOFTV

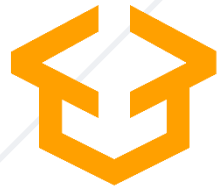


WORLD
OF
MYTHS

Questions?



SoftUni



**Software
University**



**SoftUni
Svetlina**



**SoftUni
Creative**



**SoftUni
Digital**



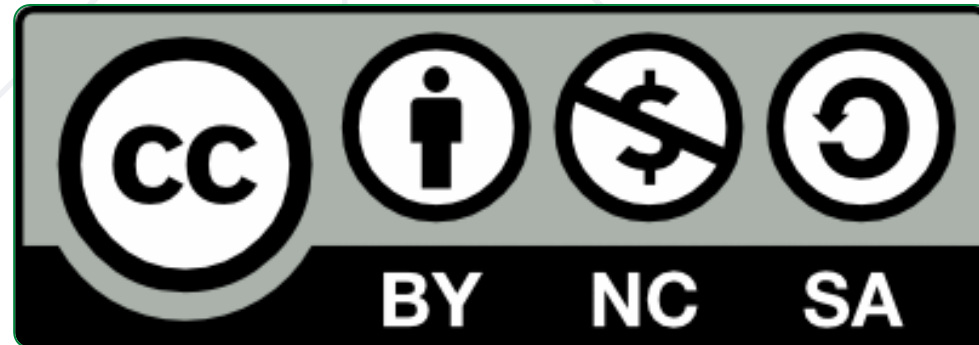
**SoftUni
Foundation**



**SoftUni
Kids**



- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license



Trainings @ Software University (SoftUni)

- Software University - High-Quality Education and Employment Opportunities
 - softuni.bg
- Software University Foundation
 - <http://softuni.foundation/>
- Software University @ Facebook
 - facebook.com/SoftwareUniversity
- Software University Forums
 - forum.softuni.bg

