Advanced Data Types

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#TypeScript

Table of Contents

Advanced Data Types

Advanced Data Types

Software University

- Union type – represent an **either-or** combination of **multiple types**
- Has **only the shared** properties of the combined types without **type narrowing** to a single type

```
function greet(message: string | string[]) {
    console.log(message.length); // ok, since both have length
    if (typeof message === "string") {    // narrowed to string
        return message;
    }
    return message.join(' ');    // narrowed to string[]
}
let greeting = 'Hello world';
let greetingArray = ['Dear', 'Sir/Madam'];
console.log(greet(greetingArray)); // Dear Sir/Madam
```

Advanced Data Types

Software University

- Intersection types – combine **multiple types** in one
- Has **all** properties of the combined types

```
function showContact(contactPerson:
{name: string} & {email: string}) {
    return contactPerson;
}
let contactPerson: {name: string} & {email: string} =
{
    name: 'Svetoslav Dimitrov',
    email: 'test@test.com'
}
console.log(showContact(contactPerson));
```

Literal Types

- String Literal Type

```
let status: "success" | "error";
status = "success"; // valid
```

- Number Literal Type

```
let errorCode: 500 | 400 | 404;
errorCode = 500; // valid
```

Type Aliases

Type Aliases

Software University

- A **custom name** for a type, declared with the **type** keyword, can target:
  - Primitives
  - Objects
  - Advanced types

```
type Age = number;    // primitive
type User = { age: Age };  // uses the type alias Age
type Person = { name: string };  // object
type Combined = User & Person;  // advanced type

const user: Combined = {name: 'John', age: 20 };
```

"keyof" Typescript Operator

Software University

- TypeScript Operator that does not exist in JS
- Retrieves the keys of an object type as a **union of string or numeric literals**

```
type Point = { x: number; y: number; };
type PointKeys = keyof Point; // 'x' | 'y'

type Colors = { red: string; blue: string; };
type ColorKeys = keyof Colors; // 'red' | 'blue'
```

"in" Usage

Software University

- JS operator that checks if a given key exists in an object
- TypeScript can use the JS "**in**" operator:
  - As a **TypeGuard**
  - To iterate over the results of the "**keyof**" operator

```
type A = { name: string};
type B = { age: number};
let val: A | B = …;
//used as a type guard to narrow 'val' to type B
if('age' in val) console.log(val.age); //valid
```

"typeof" Usage

Software University

- TypeScript can also leverage the JS **typeof** operator to extract TS type information from variables
  - This **does not change** the runtime functionality

```
type Point = { x: number; y: number; };
let point1: Point = {x: 12, y:4};
type Point2 = typeof point1;   // { x: number; y: number; }

console.log(typeof point1)  // object
type Color = {red: number};
let color1 = { red: 20};

// true since both are objects
console.log(typeof point1 === typeof color1)    // true
```

Mapped Types

Software University

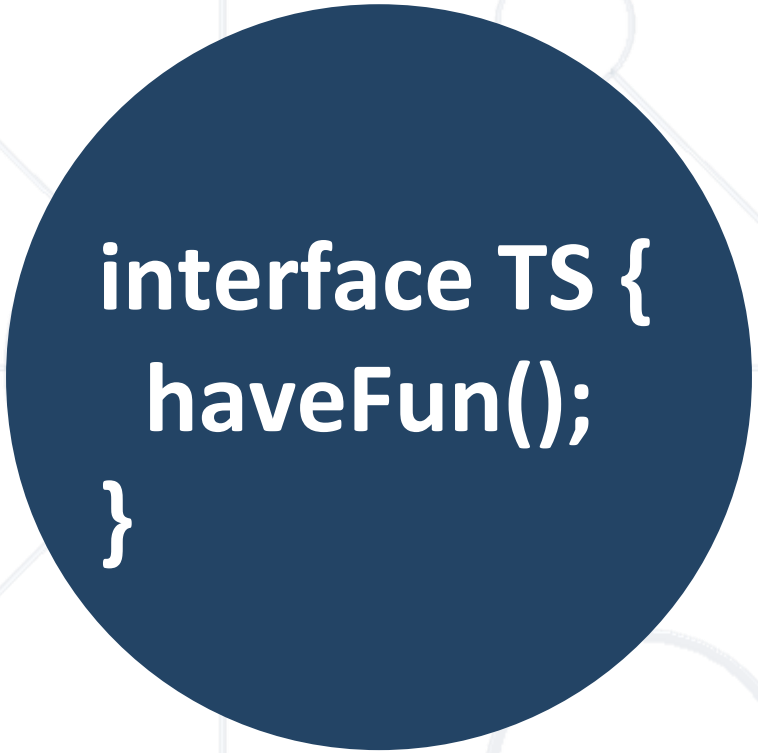
- Creates new types by **transforming each property of an existing type**

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

type PartialPoint = {[K in keyof Point]?: Point[K]};
// { x?: number; y?: number; }

type ReadonlyColors =
{ readonly [K in keyof Colors]: Colors[K] };
// { readonly red: string; readonly blue: string; }
```

Recursive Types

- **Recursive types** are vital for **representing complex, self-referential** data structures

```
type TreeNode = {
 value: number;
 left?: TreeNode;
 right?: TreeNode;
}

let root: TreeNode = {
 value: 20,
 left: {value: 5} // also of type TreeNode
};
```



Interfaces

Definition

Software University

- Defined by using keyword **interface**
- Often called **duck typing** or **structural typing**
- We can define **properties** and **methods** on an interface 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,
}

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

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

Interface declaration

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

Describe Function Types

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

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

    - In the parentheses we put the **parameters** we want to pass to the function with their **types**, split by a comma
    - On the right side is the **return type** of the function

Example: Describe Function Types

Software University

```
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;
}
```

Implemented by Classes

- 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}`;
    }
}
```

Extending Interfaces

- 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}`;
    }
}
```

Interfaces vs Types

Interfaces vs Types

- In many cases, they can be used interchangeably depending on personal preference
  - Interfaces: Defines a contract that the **object must adhere to**
  - Types:
    - create new name for **primitive data types**
    - define **union**, **tuple** and **more complex types** and many more

Interfaces vs Types

- Interface

```
interface Person {
  firstName: string;

  lastName: string;

  greeting: () =>  string;
}
```

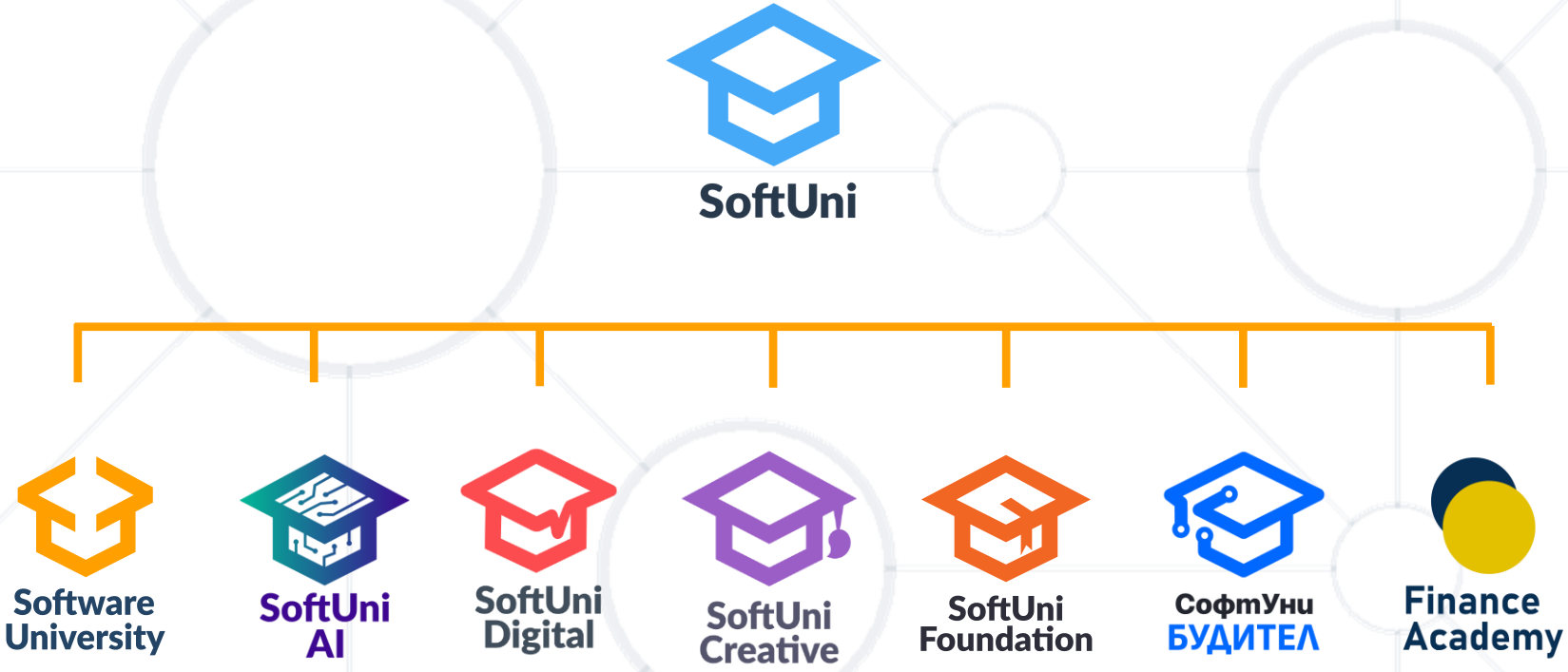
- Type

```
type Person = {
  firstName: string;

  lastName: string;

  greeting: () =>  string;
}
```

Summary

- TypeScript provides a lot more advanced data types and advanced typing for complex use cases:
  - union, intersection types and variety of literals
  - type aliases, recursive types , "keyof" and many more
- There are types and interfaces that can help us extend our typing even to the next level

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