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 - combine **multiple types** in one type

```
function greet(message: string | string[]) {
    if (typeof message === "string") {
        return message;
    }
    return message.join(' ');
}
let greeting = 'Hello world';
let greetingArray = ['Dear', 'Sir/Madam'];

console.log(greet(greetingArray)); //Dear Sir/Madam
```

Advanced Data Types

- Intersection types - combine **multiple types** in one type

```
interface Person { fullName: string | string[]; }
interface Contact { email: string; }
function showContact(contactPerson: Person & Contact) {
    return contactPerson;
}
let contactPerson: Person & Contact = {
    fullName: '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

Software University

- Simple Type Alias

```
type Age = number;
const myAge: Age = 25;
```

- Object Type Alias

```
type User = { id: number; name: string; };
const user: User = { id: 1, name: 'John Doe' };
```

"keyof" Usage

- 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'
```

Mapped Types

Software University

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

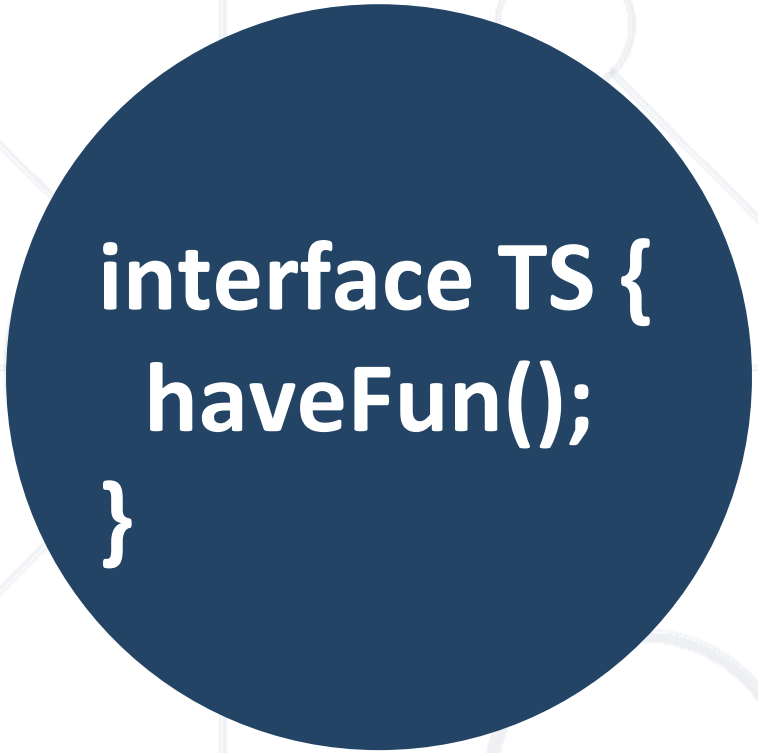
```
type Optional<T> = { [K in keyof T]?: T[K] };
type PartialPoint = Optional<Point>;
// { x?: number; y?: number; }

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

Recursive Types and Interfaces

Software University

- **Recursive types** are vital for **representing complex, self-referential** data structures
- **Type inference** allows TypeScript to **automatically deduce types, improving code readability** and **development speed**

```
interface TreeNode {
 value: number;
 left?: TreeNode;
 right?: TreeNode;
}
```



Interfaces

Definition

Software University

- 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,
}

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;
}
```

    - 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: str
ing): 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

Software University

```
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;
}
```

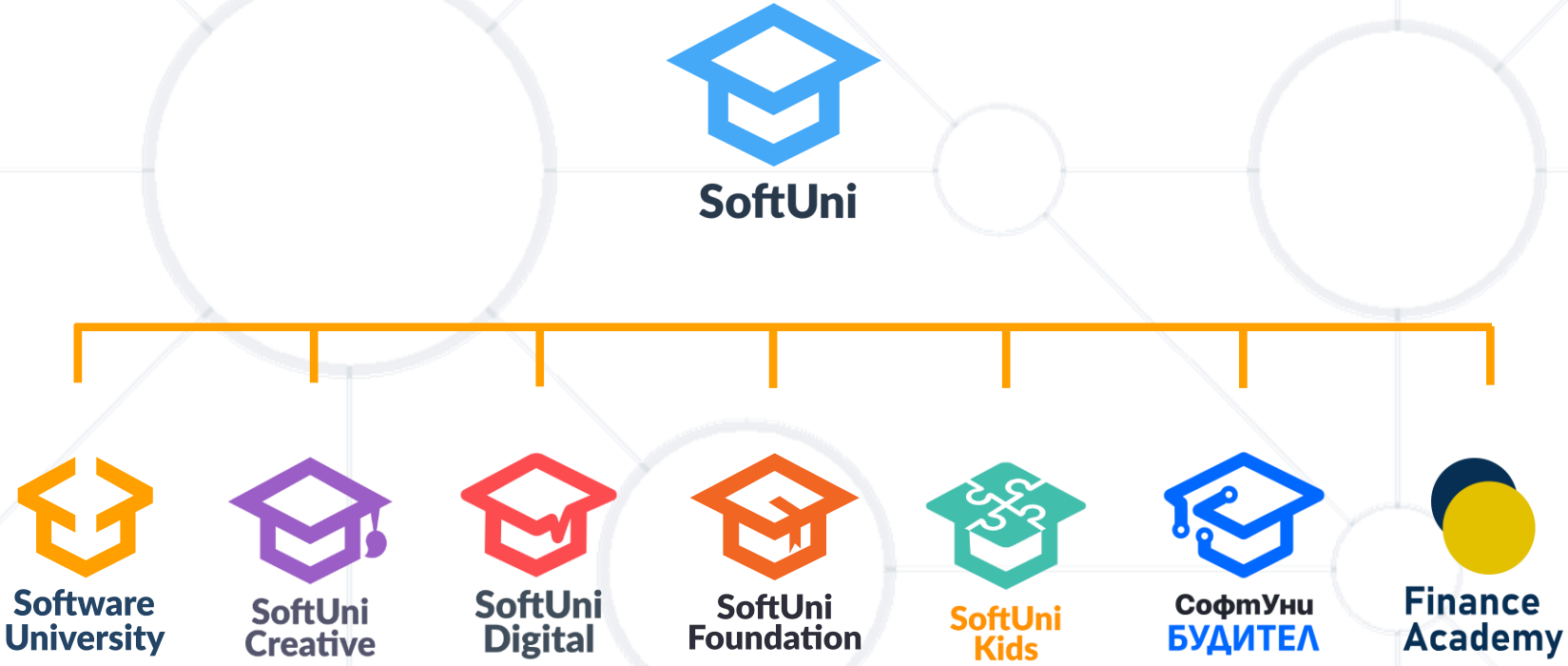
- Type

```
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, insertion 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

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

THE CROWN IS YOURS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

PHAR VISION

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