TYPESCRIPT

References: **W3School**

Nguyen Quang Phu - PDZ

Table of Contents

1. Introduction

> TypeScript is JavaScript with **added** syntax for types. TypeScript is a **syntactic superset** (means that it shares the same base syntax, but adds something) of JavaScript which adds **static** typing.

Compare

- *JavaScript* can be difficult to understand what **types** of data being passed. JavaScript **functions** and **variables** do not have any information.

- *TypeScript* allows specifying types of data being passed, can report error when types do not match

2. Simple Types

There are 3 main primitives in JavaScript and TypeScript.

- `boolean` - `true` or `false` values
- `number` - whole numbers and floating point values
- `string` - text values

There are 2 main ways TypeScript assigns a type:

- Explicit
- Implicit

Example:

```
// Explicit
let firstName: string = "QuangPhu";
// Implicit - guess the type
let firstName = "QuangPhu";
```

Error in Type Assignment

Example:

```
let firstName: string = "Dylan"; // type string
firstName = 33; // attempts to re-assign the value to a different type
```

3. Special Types

There are some special types that may not refer to any specific type of data.

Types: any

`any` is a type that disables type checking and effectively allows all types to be used

```
let u = true;
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.

let v: any = true;
v = "string"; // no error as it can be "any" type
```

Types: unknown

`unknown` is a similar but safer alternative to `any`

```
let w: unknown = 1;
w = "string"; // no error
```

Types: never

`never` effectively throws an error whenever it is defined

Type: undefined & null

These are JavaScript primitives.

```
let y: undefined = undefined;
let z: null = null;
```

4. Arrays

TypeScript has a specific syntax for typing arrays

Example:

```
const names: string[] = [];
names.push("Dylan"); // no error
// names.push(3); // Error: Argument of type 'number' is not assignable to parameter of type 'string'.
```

Readonly

`readonly` is a keyword that prevents arrays from being changed

Example:

```
const names: readonly string[] = ["Dylan"];
names.push("Jack"); // Error: Property 'push' does not exist on type 'readonly string[]'.
```

> 📝**Note**: TypeScript can infer the type of an array if it has values.

Example:

```
const numbers = [1, 2, 3]; // inferred to type number[]
numbers.push(4); // no error
// comment line below out to see the successful assignment
numbers.push("2"); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.
let head: number = numbers[0]; // no error
```

5. Tuples

- **tuple** is a typed **array** with a *pre-defined length* and *types* for each index
- **tuple** can allow each element in the array to be a *known type* of value

Example:

```
// define our tuple
let ourTuple: [number, boolean, string];

// initialize correctly
ourTuple = [5, false, 'Coding God was here'];
```

Readonly Tuple

This will not throw an error

Example:

```
// define our tuple
let ourTuple: [number, boolean, string];
// initialize correctly
ourTuple = [5, false, 'Coding God was here'];
// We have no type safety in our tuple for indexes 3+
ourTuple.push('Something new and wrong');
console.log(ourTuple);
```

BUT when using readonly, it will throw an error

Example:

```
// define our readonly tuple
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];
// throws error as it is readonly.
ourReadonlyTuple.push('Coding God took a day off');
```

Destrucuring Tuples

```
const graph: [number, number] = [55.2, 41.3];
const [x, y] = graph;
```

6. TypeScript Object Types

Example:

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};

car.type = "Ford"; // no error
car.type = 2; // Error: Type 'number' is not assignable to type 'string'.
```

6.1. Optional Properties

Example without an optional property

```
const car: { type: string, mileage: number } = { // Error: Property 'mileage' is missing in type '{ type: string; }' but required in type '{ type: string; mileage: number; }'.
  type: "Toyota",
};
car.mileage = 2000;
```

Example with an optional property

```
const car: { type: string, mileage?: number } = { // no error
  type: "Toyota"
};
car.mileage = 2000;
```

6.2. Index Signature

```
const nameAgeMap: { [index: string]: number } = {};
nameAgeMap.Jack = 25; // no error
nameAgeMap.Mark = "Fifty"; // Error: Type 'string' is not assignable to type 'number'.
```

7. Enums

`enum` is a special "**class**" that represents a group of **constants** (unchangable)

7.1. Numeric Enums - Default

```
enum CardinalDirections {
  North, // 0
  East,
  South,
  West
}
let currentDirection = CardinalDirections.North;
console.log(currentDirection); // 0
```

7.2. Numeric Enums - Intialize

```
enum CardinalDirections {
  North = 18,
  East,
  South,
  West
}
let currentDirection = CardinalDirections.West;
console.log(currentDirection); // 21
```

7.3. String Enums

```
enum CardinalDirections {
  North = 'North',
  East = "East",
  South = "South",
  West = "West"
};
// logs "North"
console.log(CardinalDirections.North);
```

> **Note**: Recommended not to match string and numeric enum values

8. Type Aliases and Interfaces

> TypeScript allows types to be defined separately from the variables that use them

8.1. Type Aliases

> Allow defining types with a custom name (an Alias) Can be used for primitives like `string` or more complex types such as `objects` and `arrays`

```
type CarYear = number
type CarType = string
type CarModel = string
type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
}

const carYear: CarYear = 2001
const carType: CarType = "Toyota"
const carModel: CarModel = "Corolla"
const car: Car = {
  year: carYear,
  type: carType,
  model: carModel
};
```

8.2. Interfaces

> Similar to type aliases, except they **only** apply to `object` types

```
interface Rectangle {
  height: number,
  width: number
}

const rectangle: Rectangle = {
  height: 20,
  width: 10
};
```

8.3. Extending Interfaces

Interfaces can extend other's definitions

Extending an interface means you are *creating a new interface* with the **same properties** as the original, plus something **new**.

```
// Maybe like Inheritance
interface Rectangle {
  height: number,
  width: number
}

interface ColoredRectangle extends Rectangle {
  color: string
}

const coloredRectangle: ColoredRectangle = {
  height: 20,
  width: 10,
  color: "red"
};
```

9. Union Types

> **Union types** are used when a value can be more than a single type.

Union (| - OR)

```
function printStatusCode(code: string | number) {
  console.log(`My status code is ${code}.`)
}
printStatusCode(404);
printStatusCode('404');
```

10. Functions

10.1. Return Type

Value returned by the function can eb explicitly defined

```
function getTime(): number {
  return new Date().getTime();
}
```

10.2. Void Return Type

```
function printHello(): void {
  console.log('Hello!');
}
```

10.3. Parameters Type

```
function multiply(a: number, b: number) {
  return a * b;
}
```

> 📝**Note**: If no parameter type is defined, TypeScript will default to using any

10.4. Optional Parameters

By default, TypeScript will assume all parameters are required, BUT they can be explicitly optional

```
// the `?` operator here marks parameter `c` as optional
function add(a: number, b: number, c?: number) {
  return a + b + (c || 0);
}
```

10.5. Default Parameters

```
function pow(value: number, exponent: number = 10) {
  return value ** exponent;
}
```

10.6. Rest Parameters

Can be typed like normal parameters, but rest parameters are always arrays

```
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0);
}
```

11. Casting

> Sometimes it's necessary to **override** the type of a variable

11.1. Casting with `as`

Directly change the type of the given variable

```
let x: unknown = 'hello';
console.log((x as string).length);
```

> 📝**Note**:
>
> - Casting doesn't actually change the type of the data
> - TypeScript will still attempt to typecheck casts to prevent casts that don't seem correct

11.2. Casting with `<>`

```
let x: unknown = 'hello';
console.log((<string>x).length);
```

> 📝**Note**: This type of casting now work with TSX, such as working on React files

11.3. Force casting

> To override type errors that TypeScript may throw when casting, first cast to **unknown**, then to the **target type**.

```
let x = 'hello';
console.log(((x as unknown) as number).length); // x is not actually a number so this will return undefined
```

12. Classes

> TypeScript adds **types** and **visibility modifiers** to JavaScript classes.

12.1. Members: Types

```
class Person {
    name: string;
}

const person = new Person();
person.name = "Jane";
```

12.2. Members: Visibility

There are 3 main visibility modifiers:

- `public` - (default) allows access to the class member from anywhere
- `private` - **only** allows access to the class member from within the class
- `protected` - allows access to the class member from itself and any classes that inherit it

```
class Person {
    private name: string;

    public constructor(name: string) {
        // this refers to the instance of the class
        this.name = name;
    }

    public getName(): string {
        return this.name;
    }
}

const person = new Person("Jane");
console.log(person.getName()); // person.name isn't accessible from outside the
class since it's private
```

12.3. Parameter Properties

Can add a visibility modifier to the parameters

```
class Person {
    // name is a private member variable
    public constructor(private name: string) {}

    public getName(): string {
        return this.name;
    }
}

const person = new Person("Jane");
console.log(person.getName());
```

12.4. Readonly

`readonly` **prevent** class members from being **changed**

```
class Person {
    private readonly name: string;

    public constructor(name: string) {
        // name cannot be changed after this initial definition, which has to be
either at it's declaration or in the constructor.
        this.name = name;
    }

    public getName(): string {
        return this.name;
    }

    // public setName(name: string) {
    //     this.name = name;
    // }
    // cannot do this as name
    // is read-only (unchangeable)
}

const person = new Person("Jane");
console.log(person.getName());
```

12.5. Inheritance (`implements`)

Can implement multiple interfaces by: `class A implements interface1, interface2 {};`

```
interface Shape {
    getArea: () => number;
}

class Rectangle implements Shape {
    public constructor(protected readonly width: number, protected readonly height: number) {}

    public getArea(): number {
        return this.width * this.height;
    }
}
```

12.6. Inheritance: Extends

> **Note**:
>
> - A class can only extends one other class
> - `super` keyword below is used to call methods or access properties of a parent class from within the subclass. Below, in `class Square`, it calls the constructor of the `class Rectangle`

```
interface Shape {
    getArea: () => number;
}

class Rectangle implements Shape {
    public constructor(protected readonly width: number, protected readonly height: number) {}

    public getArea(): number {
        return this.width * this.height;
    }
}
```

```
class Square extends Rectangle {
    public constructor(width: number) {
        super(width, width);
    }

    // getArea gets inherited from Rectangle
}
```

12.7. Override

When a class **extends** parent class, it can **replace** the members of the parent class with the **same name**

```
interface Shape {
    getArea: () => number;
}

class Rectangle implements Shape {
    // using protected for these members allows access from classes that extend from this class, such as Square
    public constructor(protected readonly width: number, protected readonly height: number) {}

    public getArea(): number {
        return this.width * this.height;
    }

    public toString(): string {
        return `Rectangle[width=${this.width}, height=${this.height}]`;
    }
}

class Square extends Rectangle {
    public constructor(width: number) {
        super(width, width);
    }

    // this toString replaces the toString from Rectangle
    public override toString(): string {
        return `Square[width=${this.width}]`;
    }
}
```

12.8. Abstract Classes

Classes can be written in a way that allows them to be used as a **base class** for other classes **without having to implement** all the members.

> 📝**Note**: Abstract classes cannot be directly instantiated

```
abstract class Polygon {
    public abstract getArea(): number;

    public toString(): string {
        return `Polygon[area=${this.getArea()}]`;
    }
}

// super called still required
// it used to indicate that the subclass is invoking
// the constructor of its superclass
class Rectangle extends Polygon {
    public constructor(protected readonly width: number, protected readonly height: number) {
        super();
    }

    public getArea(): number {
        return this.width * this.height;
    }
}

const myRect = new Rectangle(10,20);

console.log(myRect.getArea());
```

13. Basic Generics

Generics allow creating '**type variables**' which can be used to *create classes, functions & type aliases* that *don't need to explicitly define* the **types** that they use.

> This may look the same as *Template in CPP*

13.1. Functions

```
// S, T here like a hidden type that
// we will decide when using
function createPair<S, T>(v1: S, v2: T): [S, T] {
  return [v1, v2];
}
console.log(createPair<string, number>('hello', 42)); // ['hello', 42]
```

13.2. Classes - Default Value

```
class NamedValue<T = string> {
    private _value: T | undefined;

    constructor(private name: string) {}
    public setValue(value: T) {
        this._value = value;
    }

    public getValue(): T | undefined {
        return this._value;
    }

    public toString(): string {
        // this is string literal in JavaScript
        return `${this.name}: ${this._value}`;
    }
}

let value = new NamedValue<number>('myNumber');
value.setValue(10);
console.log(value.toString()); // myNumber: 10
```

13.3. Type Aliases

Allow creating types that are more **reusable**

```
type Wrapped<T> = { value: T };

const wrappedValue: Wrapped<number> = { value: 10 };
```

13.4. Using with Extends

Constraints can be added to generics to **limit** what's allowed, they make it possible to rely on a **more specific type** when using the generic type.

```
function createLoggedPair<S extends string | number, T extends string | number>
(v1: S, v2: T): [S, T] {
    console.log(`creating pair: v1='${v1}', v2='${v2}'`);
    return [v1, v2];
}
```

14. Utility Types

> TypeScript comes with a large number of types that can help with **some common type manipulation**, usually referred to as **utility types**.

14.1. Partial

It changes all the properties in an object to be **optional**.

```
interface Point {
    x: number;
    y: number;
}

let pointPart: Partial<Point> = {}; // `Partial` allows x and y to be optional
pointPart.x = 10;
```

14.2. Required

It changed all the properties in an object to be **required**.

```
interface Car {
    make: string;
    model: string;
    mileage?: number;
}

let myCar: Required<Car> = {
    make: 'Ford',
    model: 'Focus',
    mileage: 12000 // `Required` forces mileage to be defined
};
```

14.3. Record

`Record` is a **shortcut** to defining an **object type** with a specific **key type** and **value type**.

```
const nameAgeMap: Record<string, number> = {
  'Alice': 21,
  'Bob': 25
};
```

> 📝**Note**: `Record<string, number>` is equivalent to `{ [key: string]: number }`

14.4. Omit

`Omit` **removes keys** from an object type.

```
interface Person {
    name: string;
    age: number;
    location?: string; // optional
}

const bob: Omit<Person, 'age' | 'location'> = {
    name: 'Bob'
    // `Omit` has removed age and location from the type and they can't be defined
here
};
```

14.5. Pick

`Pick` **removes all but the specified keys** from an object type.

> 📝**Note**: It only remove keys of an object, not remove key of the "parent" interface

```
interface Person {
    name: string;
    age: number;
    location?: string;
}

const bob: Pick<Person, 'name'> = {
    name: 'Bob'
    // `Pick` has only kept name, so age and location were removed from the type
and they can't be defined here
};
```

14.6. Exclude

`Exclude` removes **types** from a **union**.

```
type Primitive = string | number | boolean
const value: Exclude<Primitive, string> = true; // a string cannot be used here
since Exclude removed it from the type.
```

14.7. ReturnType

`ReturnType` extracts the return type of a function type

```
// here the return type of a function is an object
type PointGenerator = () => { x: number; y: number; };
const point: ReturnType<PointGenerator> = {
  x: 10,
  y: 20
};
```

14.8. Parameters

`Parameters` extracts the **parameter types** of a function type as an array.

```
type PointPrinter = (p: { x: number; y: number; }) => void;
const point: Parameters<PointPrinter>[0] = {
  x: 10,
  y: 20
};
```

15. Keyof

15.1. `keyof` with explicit keys

`keyof` creates a **union** type with those keys

```
interface Person {
    name: string;
    age: number;
}

// `keyof Person` here creates a union type of "name" and "age", other strings will not be allowed
function printPersonProperty(person: Person, property: keyof Person) {
    console.log(`Printing person property ${property}: "${person[property]}"`);
}

let person = {
    name: "Max",
    age: 27
};

// Printing person property name: "Max"
printPersonProperty(person, "name");
```

15.2. `keyof` with index signatures

`keyof` can be used with index sugnatures to extract the index type.

```
type StringMap = { [key: string]: unknown };
// `keyof StringMap` resolves to `string` here
function createStringPair(property: keyof StringMap, value: string): StringMap {
    return { [property]: value };
}
```

16. Null & Undefined

> **Note**:

> - By default, `null` and `undefined` **handling** is **disabled**, and can be **enabled** by setting `strictNullChecks` to **true**
> - `null` and `undefined` are *primitive* types and can be used like other types.

16.1. Optional Chaining

- Work well with TypeScript's **null handling**
- `?.` operator when accessing properties on an object that may or may not exists, with a compact syntax

```
interface House {
    sqft: number;
    yard?: {
        sqft: number;
    };
}

function printYardSize(house: House) {
    const yardSize = house.yard?.sqft;
    if (yardSize === undefined) {
        console.log('No yard');
    } else {
        console.log(`Yard is ${yardSize} sqft`);
    }
}

let home: House = {
    sqft: 500
};

printYardSize(home);
```

16.2. Nullish Coalescence

- Use `??` operator
- Work well with TypeScript's **null handling**
- Allows writing expressions that have a fallback specifically when dealing with `null` otr `undefined`

```
function printMileage(mileage: number | null | undefined) {
  console.log(`Mileage: ${mileage ?? 'Not Available'}`);
}

printMileage(null); // Prints 'Mileage: Not Available'
printMileage(0); // Prints 'Mileage: 0'
```