Contents

TS

Strongly typed JavaScript Superset -> more robust code

Features

- type definition + type inference
- interfaces <> Java interfaces: more about structure, also attributes!
- classes

- decorators: add or change functionality of a class, function, property
- modules: export/import with ECMAScript modules, require/exports with CommonJS modules

Types

In TypeScript, a type is a way to define the shape, structure, or behavior of data, allowing you to specify and enforce what kind of values a variable, parameter, or return value can hold.

- can represent not only objects but also unions, intersections, primitives, and other types

Primitive Types

Three main primitives: `number` (int, float), `string`, `boolean` (true/false)

```
let name = 'Rajesh'; // implicit type string
let anotherName: string = 'Anna-Maria'; // explicit type
```

Special Types

- `any`: avoid type checking (bad practice, can be reassigned to different data type)
- `null`
- `undefined`: placeholder for uninitialized value

With `"strictNullChecks==On"`, you have to check for **null** explicitly:

```
function doSomething(x: string | null) {
    if (x !== null) {
        console.log("Hello, " + x.toUpperCase());
    }
}
```

Arrays

Arbitrary types in **JS** In **TS** type has to be *homogeneous*

```
const stringArray = ['string 1', 'string 2'];
const stringArray2 = new Array<string>(); // string array using generics
const stringArray3: string[] = []; // string array using type annotation
```

Object

```
// type for an obj
let employee: {
    firstName: string;
    lastName: string;
```

```
    age: number;
    jobTitle: string;
};

employee = {
    firstName: "test",
    lastName: "test",
    age: 24,
    jobTitle: "test"
}

// explicit obj, saves typing
const peter: { name: string; toString: () => string } = {
    name: 'Peter',
    toString() {
        return this.name; // or 'Anna'
    }
}
```

Union

```
function add(a: any, b: any) {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }
    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    } throw new Error('Parameters must be numbers or strings');
}
console.log(add(1, 2));     // output: 3
console.log(add('1', '2')); // output: "12"
console.log(add(true, false)); // Error: Parameters must be numbers or strings
// type narrowing: TS' inference mechanism narrows down "any" to either number of string
// due to the way the parameters are used in the control flow

// use union types to avoid "any"
function add(a: number | string, b: number | string): number | string {
    if (typeof a === 'number' && typeof b === 'number') {
        return a + b;
    }
    if (typeof a === 'string' && typeof b === 'string') {
        return a.concat(b);
    }
    throw new Error('Parameters must be numbers or strings');
}
```

Intersection

```
type Personal = {
    name: string;
    age: number;
};
type Contact = {
    email: string;
    phone: string;
};
type Candidate = Personal & Contact;
let candidate: Candidate = {
    name: "Joe",
    age: 25,
    email: "joe@example.com",
    phone: "(408)-123-4567"
};
```

Type Aliases

```
// define new type for existing & more complex types
type Name = string;
let firstName: Name = 'Anna';

type Person = {
    name: string;
    age: number;
};
let anna: Person = {
    name: 'Anna',
    age: 34
};

type alphanumeric = string | number;
let alnum1: alphanumeric = 'a string', alnum2: alphanumeric = 123.56;
```

Interfaces

Primarily used to define the shape of an object, specifying properties and methods. - Classes typically implement one or more interfaces. - An interface may also extend other interfaces which means it inherits the properties of each extended interface.

```
interface Person {
    name: string;
    age: number;
    speak(): void;
```

```
}

// capabilities
interface JSONResponse extends Response, HTTPable {
    version: number;
    payloadSize: number;
    outOfStock?: boolean; // optional property

    // 2 ways to define functions
    update: (retryTimes: number) => void;
    update(retryTimes: number): void;

    (): JSONResponse; // call this obj. via ()
    // functions in JS are Obj. that can be called

    new(s: string): JSONResponse; // new operator

    [key: string]: number; // any property not described is assumed to exist
    // -> must be a number

    readonly body: string; // property cannot be changed
}
```

Generics

- Type that has one or more *parameters* which are types
- Generic functions, classes & interfaces possible

```
// generic function
function getRandomElement<T>(items: T[]): T {
    let randomIndex = Math.floor(Math.random() * items.length);
    return items[randomIndex];
}
let numbers = [1, 5, 7, 4, 2, 9];
getRandomElement<number>(numbers);
let colors = ['red', 'green', 'blue'];
getRandomElement(colors);

// generic interface
interface Pair<T, U> {
    first: T;
    second: U;
}

let numberPair: Pair<number, number> = { first: 1, second: 2 };
let stringNumberPair: Pair<string, number> = { first: "Age", second: 30 };
```

Type Assertion

```
let inputElement = document.getElementById("user-input");
(inputElement as HTMLInputElement).value = "Hello";
```

String Literal Types

Define type that accepts only one specified string literal

```
String Literal Types
let click: 'click'; // define variable click of type 'click'
click = 'click';    // ok
click = 'abc';      // compiler error

// usecase
type MyMouseEvent = 'click' | 'dblclick' | 'mouseup' | 'mousedown';
let mouseEvent: MyMouseEvent;
mouseEvent = 'click';     // valid
mouseEvent = 'dblclick';  // valid
mouseEvent = 'mouseup';   // valid
mouseEvent = 'mousedown'; // valid
mouseEvent = 'mouseover'; // compiler error
```

Enums

```
enum ApprovalStatus {
    draft = 1,
    submitted,
    approved,
    rejected
};

// usecase
const request =  {
    id: 1,
    status: ApprovalStatus.approved,
    description: 'Please approve this request'
};

if(request.status === ApprovalStatus.approved) {
    console.log('Send email to the Applicant...');
}
```

Tuples

Array with fixed length & fixed element types

```ts
type RGBValue = [number, number, number];
let color: RGBValue = [255, 0, 0];
```

Functions

Like js functions, but with parameter & return types, default values possible

```ts
function applyDiscount(price: number, discount: number = 0.05): number {
    return price * (1 - discount);
}
applyDiscount(100);
```

Optional Parameters

```ts
function applyDiscount(price: number, discount?: number): number {
    if (!discount) {
        discount = 0.05;
    }
    return price * (1 - discount);
}
applyDiscount(100);
```

Rest Parameters

Variable number of arguments from 0 to n Note: only **one** rest parameter of type array is allowed which must be the **last** parameter

```ts
function createMap<T>(...elements: T[]): Map<number, T> {
    const m = new Map<number, T>();
    elements.forEach((value: T, index: number) =>  m.set(index, value));
    return m;
}
createMap(1, 2, 3);
createMap("apple", "banana", "cherry");
```

Function Types

```ts
function calculate(value1: number, value2: number,
        operator: (v1: number, v2: number) => number): number {
    return operator(value1, value2);
}
calculate(2, 10, (x, y) => x + y);
```

Function Overloading

Define **multiple signatures** for a single function and provide one implementation

```ts
function toArray(item: string): string[];
function toArray(item: number): number[];
```

```ts
function toArray(item: any): string[] | number[] {
    return Array.isArray(item) ? item : [ item ];
}
```

Classes

```ts
class User
        extends Account
        implements Updatable, Serializable{

    id: string;
    displayName?: boolean;
    name!: string; // always defined
    #attributes: Map<string, any> // private field

    constructor(id: string, public age: number, email: number) {
        super(email);
        this.id = id;
        this.#attributes = new Map<string, any>();
    }

    get id() { return this.id; }
    set id(value: string) { this.id= value; }

    setName(name: string) {
        this.name = name;
    }
    private getAge() { // not usable outside or subclass
        return this.age;
    }
    protected setAge(value: number) { // my be used in subclass
        this.age = age;
    }

    static #userCount = 0; // static private field
    static registerUser(user: User) { } // static method
    static { this.#userCount = 1; } // initialization of static fields
}
```

Abstract Classes

- Cannot be instantiated, only subclassed
- Can have abstract methods that must be implemented by subclasses
- Can have implemented methods that can be used by subclasses
- Classes can only inherit from one abstract class, but can implement multiple

interfaces

```
abstract class Employee {
    constructor(private firstName: string, private lastName: string) {}

    abstract getSalary(): number; // abstract

    get fullName(): string { // implemented
        return `${this.firstName} ${this.lastName}`;
    }
    compensationStatement(): string {
        return `${this.fullName} makes ${this.getSalary()} a month.`;
    }
}
```

Control Flow Analysis

TS introduces a feature called **call flow analysis** that enhances the type system & enables stricter type checking -> **type narrowing**

```
function printId(id: number | string) {
    if (typeof id === "string") {
        console.log(id.toUpperCase());
    }
    else {
        console.log(id);
    }
}
```

Useful Operators

Spread Operator

```
const source = [1, 2, 3];
console.log([...source]); // merge objects
const obj1 = { value: 12.45, unit: 'km/h' };
const obj2 = { date: new Date(), unit: 'm/s' };
console.log({ ...obj1, ...obj2 }); // obj2.unit overwrites obj1.unit!

// pass array to function
function addNumbers(a: number, b: number, c: number): number {
    return a + b + c;

}const numbers = [1, 2, 3];
console.log(addNumbers(...numbers)); // Output: 6

// convert string to array
```

```
const greeting = "Hello";
const characters = [...greeting];
console.log(characters); // Output: ['H', 'e', 'l', 'l', 'o']
```

Optional Chaining Operator

```
type Address = { street?: string; city?: string; };
type User2 = { name: string; address?: Address; };

const user: User2 = {
    name: "Alice"
};
console.log(user?.address?.city); // eliminates the need for null checks
```

Destructuring

```
const numbers = [1, 2, 3, 4];
const [first, second, third] = numbers;
const [first2, , third2] = numbers;
const [first3, ...rest] = numbers;
```

Assertion Operator

- Tell compiler that a value is certainly not null or undefined
- User operator with caution

```
let element = document.getElementById("myElement");
element!.style.backgroundColor = "red";
```

Decorators

- Add annotations & meta programming syntax
- Can be attached to *classes*, *methods*, *properties*, *accessors*, *parameters*
- Use cases: logging, profiling, validation etc.

```
// simple decorator
function Greeter(target: Function) { //
    target.prototype.greet = function() {
        console.log("Hello, " + this.name);
    }
}

@Greeter
class Person {
    name: string;
    constructor(name: string) {
        this.name = name;
    }
```

```
}

const person = new Person("Alice");

// does not have greet() function per class definition, therefore cast to any
(person as any).greet();

// method decorator
// applied to the Property Descriptor for the method
function logExecutionTime(originalMethod: any, _context: any) {
    function replacementMethod(this: any, ...args: any[]) {
        console.time(originalMethod.name); // Start timing
        const result = originalMethod.call(this, ...args);
        console.timeEnd(originalMethod.name); // End timing
        return result;
    }
    return replacementMethod;
}

class MathOperations {
    @logExecutionTime
    calculateSquare(n: number): number {
        // Simulate a time-consuming task
        for (let i = 0; i < 1_000_000_000; i++) {}
        return n * n;
    }
}

const math = new MathOperations();
console.log(`Result: ${math.calculateSquare(3)}`);
```

Module System

ES Modules (ECMAScript Modules)

- import & export

```
// mathUtils.ts
export function add(a: number, b: number): number {
    return a + b;
}
export function subtract(a: number, b: number): number {
    return a - b;
}
// or: export { add, subtract };

import { add, subtract as sub } from './mathUtils';
```

```ts
add(5, 3);


// when exporting single value or obj. use default export
export default function greet(name: string): void {
    console.log(`Hello, ${name}!`);
}

import greet from './greeter'; greet("Alice");

// type declaration
// some third party libraries do not provide TS type declarations, provide your own
declare module 'mathUtils' {
    export function add(a: number, b: number): number;
    export function subtract(a: number, b: number): number;
}
```

CommonJS Modules

- `require` & `module.exports`
- Used in Node.js apps
- Convention: name `.cjs` instead of `.js`