Basics in TypeScript

Topics

- Basics
- Objects
- Enums
- Arrays
- Functions
- Classes and Interfaces
- Generics
- Basic Types
- Type Operators
- Working with Types
- Debugging

Basics

- Install with `npm i -g typescript`
- Run with `tsc file-name`
- tsconfig.json: -outDir,

Basics: JS vs TS

- larger than js
- static typing
- code completion

Basics: Type safety

- Type safety: Using types to prevent programs from doing invalid things
- Statically Typed: I made a mistake when I compile the program
- Dynamically Typed: I made a mistake when I RUN the program
- TypeScript is Statically Typed differently from JavaScript that is Dynamically Typed

Basics: Compiler and Compiling Process

- TypeScript Compiler (TSC)
- How compilation works: text (code) -> compiler -> abstract syntax tree (AST) -> bytecode -> feed it into runtime and get results
- but before: TS code -> TS AST -> Typechecker -> JS Source
- Typechecker: A special program that verifies that your code is typesafe
- Important fact: when TSC compiles your code from TypeScript to JavaScript, it won't look at your types

Basics: TypeChecker

- Type System: A set of rules that a typechecker uses to assign types to your program
- General Rule: Type Explicitly Declared and Type Automatically Inferred
- Typescript does both: It can infer from example and you can declare it
- Good Programming style: Write where necessary, infer if it's possible

Basics: tsconfig.json

- Every TypeScript project should include a file called tsconfig.json in its root directory
- It's a configuration file where you can set different properties of the compiling process
- You can set: which file should be compiled, which directory compile them to, which version of JavaScript to emit
- you can configure the tsconfig file also by command line

Basics: tslint.json

- tslint.json for configuration and management of the code formatting style

Types

- string, boolean, number
- any
- undefined
- null

```
let n: number = 10;
```

Types: Basic Types

- A set of values and the things you can do with them
- Example: boolean, string, number

Types: TypeScript's type hierarchy

- Every type extends `unknown`
- `any` extends `unknown`
- number, bigint, boolean, string, symbol, Object types extend `any`
- bounds of types: a variable upper bound (in type) is number. It cannot be a string or more than a number it's not assignable

Types: Fundamentals

- any:
  - avoid, if you can
  - use it when you and the typechecker are not be able to infer the type
  - you can do everything and it can be everything
  - working with any is like working in JavaScript, without TypeChecker
- unknown

Types: Fundamentals (2)

- boolean: as always, for the moment
- number: as always, for the moment
  - use_separators: 1_000_000
- bigint: as always, defined by n
- string

Types: Fundamentals (Object)

```
let a = {
    b: 'x'
}
console.log(a.b);
let b = {
    c: {
        d: 'f'
    }
}
let a: {b: number} = {
    b: 12
}
let c: {
    firstName: string
    lastName: string
} = {firstName: 'john', lastName: 'barrowman' }
```

Type: Classes

```
class Person {
    constructor (
        public firstName: string, // public is shorthand for
        public lastName: string // this.firstName = firstName
    ) {}
}
c = new Person('matt', 'smith');
```

Types: Type Aliases

- Use it for DRYing up repeated complex types

```
type Age = number
type Person = {
    name: string
    age: Age
}
// you can also add function to a type aliases. In this case you are stating
// the name of the function,
// the input expected by the function and
// the type of the value returned by the function.
type Persona = {
    name: string;
    greet: (messaggio: string) => void;
}
```

Types: Union and Intersection Types

```
type Cat = {name: string, purrs: boolean}
type Dog = {name: string, barks: boolean, wags: boolean}
type CatOrDogOrBoth = Cat | Dog
type CatAndDog = Cat & Dog

let b: CatAndDog {
    name: 'Domino',
    barks: true,
    purrs: true,
    wags: true
}
```

Types: Fundamentals (3)

- symbol
  - alternative to string keys in object and in map
- Objects
- Arrays
- Tuples
- null, undefined, void and never
- Enums

Types: Arrays

```
let a = [1,2,3] // number[]
var b = ['a','b'] // string[]
let c: string[] = ['a'] // string[]
let d = [1,'a'] // (string | number)[]
const e = [2,'b'] // (string | number)[]
let f = ['red']
f.push('blue')
f.push(true) // Error TS2345: Argument of type 'true' is not assignable to parameter of type 'string'
let g = [] // any[]
g.push(1) // number[]
g.push('red') // (string | number)[]
let h: number[] = [] // number[]
h.push(1) // number[]
h.push('red') // Error TS2345: Argument of type '"red"' is not assignable to parameter of type  'number'
```

Types: Tuples

```
let a: [number] = [1]
// A tuple of [first name, last name, birth year]
let b: [string, string, number] = ['malcom', 'gladwell', 1963]

b = ['queen', 'elizabeth', 'ii', 1926] // Error TS2322: Type 'string' is not assignable to type 'number'

// With optional element
let trainFares: [number, number?][] = [
    [3.75],
    [9.25, 7.70],
    [10.50]
]
// Equivalently
let moreTrainFares: ([number] | [number, number])[] = [
    //...
]
```

Types: readonly Tuples and Arrays

```
let as: readonly number[] = [1,2,3]
let bs: readonly number[] = as.concat(4)
as[4] = 5 // Error readonly
as.push(6) // Error readonly
```

Types: null, undefined, void and never

- undefined: used in case of something hasn't been defined yet
- null means an absecnce of value
- void: used in function that not return a value
- never: used in function that never returns a value, e.g throws an exception or cycle undefinitely

Types: Enums

```
enum Language {
    English, Spanish, Russian
}
enum Language {
    English = 0, Spanish = 1, Russian = 2
}
let myFirstLanguage = Language.Russian
let mySecondLanguage = Language['English']
// You can split the definition in two parts
enum Language {
    English = 0,
    Spanish = 1
}
enum Language {
    Russian = 2
}
```

Types: Enums (2)

```
enum Language {
    English = 100,
    Spanish = 200 + 300,
    Russion // inferred 501
}
enum Color {
    Red = '#c10000',
    Blue = #007ac1',
    Pink = 0xc10050,
    White = 255
}
let d = Color[6] // doesn't give you an error
const enum Language {
    English,
    Spanish,
    Russian
}
let a = Language.English
let b = Language.Tagalog // error!!!
let d = Language[6] // error!!!
```

Functions

- Return type
- Optional parameters
- Narrowing
- Optional call
- Optional operator

Functions: Declaring and Invoking Functions

```
function add(a: number, b: number) {
    return a + b
}
function add(a: number, b: number): number {
    return a + b
}
```

Functions: Optional and Default Parameters

```
function log(message: string, userId?: string) {
    let time = new Date().toLocaleTimeString()
    console.log(time,message,userId  || 'Not signed in')
}
function log(message: string, userId = 'Not signed in') {
    let time = new Date().toISOString()
    console.log(time, message, userId)
}
```

Functions: Variable number params

```
function sum(numbers: number[]): number {
    return numbers.reduce( (total, n) => total + n, 0 )
}
```

Function: Variable number params (2)

- Arity in functions is defined by the number of parameters that are expected to be passed
- A function is called Variadic, if the number of parameter is not fixed
- A function is called fixed-arity function, if the number of parameter is given and fixed

Function: Variable number params (3)

- A safe way to realize a variadic function in TypeScript is that in which you use the rest operator, declaring the type of the array

```
function sumVariadicSafe(...numbers: number[]): number {
    return numbers.reduce((total, n) => total + n, 0)
}
```

Function: call, apply and bind

```
function add(a: number, b: number): number {
    return a + b
}
add(10,20)                    // evaluates to 30
add.apply(null, [10,20])      // by spreading, evaluates to 30
add.call(null, 10, 20)        // by not spreading, evaluates to 30
add.bind(null, 10, 20)()      // by not spreading and not directly invoking, evaluates to 30
```

Function: Typing this

- The value of the `this` keyword depends on the context where it is used and
- It is managed by JavaScript
- In general, we can have the following usages:
  - in classes: it referes to the class where it is contained
  - in object: it referes to the object where it is contained
  - with call, apply and bind: it referes to the object that is bounded

Function: Type Narrowing

```
function print(val: string | number) {
    if (typeof val === "string") {
        console.log(val.toUpperCase());
    } else {
        console.log(val.toFixed(2));
    }
}
// another way of doing Type Narrowing
if (error instanceof Error) {
}
```

Function: in operator narrowing

```
type Bird = { fly: () => void };
type Fish = { swim: () => void };

function move(animal: Bird | Fish) {
    if ("fly" in animal) {
        animal.fly();
    } else {
        animal.swim();
    }
}
```

Function: typeof and instanceof

- typeof is more used about primitive types
  - typeof returns the type of the argument, as 'string'
- instanceof is created for class
  - instanceof can only verify if a given arg is of that type or not, returning a boolean

Function: type predicate

```
function isFish(pet: Fish | Bird): pet is Fish {
    return (pet as Fish).swim !== undefined;
}
```

Function: Discriminated Union

Function: Generator Functions

- Generators are a way to produce a stream of values

```
function* createFibonacciGenerator() { // the asterisk before function's name makes that function a generator.
    let a = 0
    let b = 1
    while(true) {
        yield a;
        [a,b] = [b, a+b]
    }
}

let fibonacciGenerator = createFibonacciGenerator()
fibonacciGenerator.next()
fibonacciGenerator.next()
fibonacciGenerator.next()
fibonacciGenerator.next()
fibonacciGenerator.next()
fibonacciGenerator.next()
```

Function: Iterators

- Iterator are a way to consume values

```
let numbersIterator = {
    *[Symbol.iterator]() {
        for (let n = 1; n <= 10; n++){
            yield n;
        }
    }
}
for (let a of numbersIterator) {
    console.log(a);
}
```

- spreading an iterator: `let allNumber = [...numbers]`
- resting an iterator: `let [one, two, ...rest] = numbers`

Classes and Interfaces

Classes and Interfaces: General Structure

```
class Person {
    name: string;
    age: number;

    constructor (name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet(): void {
        console.log(`Hi, my name is ${this.name} and I'm ${this.age} years old`);
    }
}

const henry = new Person("Henry", 30);
henry.greet(); // Hi, my name is Henry and I'm 30 years old
```

Classes and Interfaces: Access Modifier

- `public` (default): accessible from anywhere
- `protected`: accessible from class and subclasses
- `private`: accessible only from the class

Classes and Interfaces: Inheritance

```
class Vehicle {
    switchOn(): void {
        console.log("Vehicle switched on!");
    }
}

class Car extends Vehicle {
    drive(): void {
        console.log("Driving the car!");
    }
}

const tesla = new Car();
tesla.switchOn();
tesla.drive();
```

Classes and Interfaces: readonly property

```
class Book {
    readonly title: string;

    constructor(title: string) {
        this.title = title;
    }
}

const book = new Book("1984");
book.title = "Animal Farm";
```

Classes and Interfaces: Getters and Setters

```
class Product {
    private _price: number;

    constructor(price: number) {
        this._price = price;
    }

    get price(): number {
        return this._price;
    }

    set price(val: number) {
        if (val > 0)
            this._price = val;
    }
}
```

- a class can extend only one other class and implements multiple interface

Classes and Interfaces: Getters and Setters (2)

- if `get` exists but no `set`, the property is automatically `readonly`
- if the type of the setter parameter is not specified, it is inferrend from the returned type of the getter
- getters and setters are accessors, it means they should use in this way:

```
class Person {
    _name: string;
    _age: string;

    constructor(name:string, age:string) {
        this._name = name;
        this._age = age;
    }

    get name():string {
        return this._name;
    }

    set name(name:string) {
        this._name = name;
    }
}
const myPerson = new Person("ric",30);
myPerson.name = "par";
```

Classes and Interfaces: Static Properties and Methods

```
class Math {
    static PI = 3.14;

    static squared (x: number): number {
        return x * x;
    }
}
console.log(Math.PI);
console.log(Math.squared(5));
```

Classes and Interfaces: General Information on Interfaces

- Using the keyword `implements` you can say that a particular class satisfies a particular interface
- The class implementing the interface must implement all of the methods declared in it
- Interfaces can state readonly properties, but it cannot declare access modifiers (they are public by default)
- A class can implements a number of interface

Classes and Interfaces: Interfaces and Implementations

- An implementation as example

```
interface Animal {
    name: string;
    makeSound(): void;
}
class Dog implements Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
    makeSound(): void {console.log("Bau!");}}
```

Classes and Interfaces: Abstract Class

- Another way of model a concept is by using Abstract Classes. These can have constructors, default implementations and set access modifiers for properties and methods.

```
abstract class Animal {
    constructor(public name: string) {}

    greet(): void {
        console.log("Hi, my name is ${this.name}");
    }

    abstract makeNoise(): void;
}
```

- Abstract classes cannot be instantiated, the implemented methods are inherited as they are and abstract methods must be realized before instantiation

Classes and Interfaces: Use Interfaces or Abstract Class?

- An interface give you a shape to be satisfied
- An abstract class give you a structure that all of the class implementing that should satisfied
- In general: use an interface to give functionalities, meanwhile use an abstract class to structure a class which behavior cannot be realized at the beginning

Classes and Interfaces: TypeScript is Structurally Typed

- Differently from Java, C#, Scala and other languages that are Nominally Typed, TypeScript doesn't make difference between based on the name of the class but based on the functionalities it expose
- This feature is related to the Duck Typing Philosophy
- A small exception is related to private and protected fields of the class: if a class have same private or protected field and the shape is not an instance of that class or a subclass, the shape is not assignable to the class

Classes and Interfaces: Details On Inheritance

- If the super class have some private field, the subclass inherits the private field, but it cannot access to it

Classes and Interfaces: super call

- super call: if the child class overrides a method defined on its parent class, the child instance can make a super call to call its parent's version of the method

Classes and Interfaces: this as return type

```
class Set {
    has(value: number): boolean {
        // ...
    }
    add(value: number): this {
        // ...
    }
}
```

Classes and Interfaces: Type Aliases and Interfaces (1)

- Comparison between type aliases and interfaces

```
type Sushi = {
    calories: number
    salty: boolean
    tasty: boolean
}
interface Sushi = {
    calories: number
    salty: boolean
    tasty: boolean
}
```

Classes and Interfaces: Type Aliases and Interfaces (2)

- Comparison between type aliases and interfaces

```
type Food = {
    calories: number
    tasty: boolean
}
type Sushi = Food & {
    salty: boolean
}
type Cake = Food & {
    sweet: boolean
}
```

```
interface Food {
    calories: number
    tasty: boolean
}
interface Sushi extends Food {
```

Classes and Interfaces: Type Aliases and Interfaces (3)

- Differences between Type Aliases and Interfaces:
  - Inheritance: Interfaces can be extended
  - Interface merging: type can be extended only with `&` (intersection), and it cannot be redefined
  - Union and Intersection: only type aliases can create advanced type with union and intersection
  - Type is better to use with primitive types and tuples

Classes and Interfaces: Constructor Overloading

```
class User {
    name: string;
    age?: number;

    constructor(name: string);
    constructor(name: string, age: number);
    constructor(name: string, age?: number) {
        this.name = name;
        this.age = age;
    }
}
```

- You cannot specify the behavior for every example, instead based on type, you specify the logic for every case inside the only one body of the costructor

Classes and Interfaces: super or this?

- The inheritance mechanism is managed by javascript using the prototype chaining
- The property are setted in the actual object, meanwhile
- The methods and the accessors are referred by the prototype chaining, so they live in the super class and not in the actual class
- In summary:
  - use `this` to access to the props of the subclasses and all of the superclasses
  - use `super` to access to methods and accessors of the superclasses

Classes and Interfaces: Declaration Merging

```
class A {
    do(): string {
        return `do`;
    }
}
class A {
    anotherDo(): string {
        return `another do`;
    }
}
```

- this code is correct in TypeScript and the resulting class A have two different method: "do" and "anotherDo"

Classes and Interfaces: Declaration Merging (2)

- This holds also for interfaces

```
interface A {
    do(): string;
}
interface A {
    anotherDo(): string;
}
// the resulting interface have two differents methods: do and anotherDo
```

- This holds for Classes, Interfaces and Enums but not for Type Aliases

Classes and Interfaces: Declaration Merging (3)

- There is some exception:

```
interface Person {
    phone_number: string
}
interface Person {
    phone_number: number // Error the type should be the same
}
```

Classes and Interfaces: Declaration merging (4)

- Same happen for generic interfaces

```
interface Person<Phone extends number> {
    phone: Phone
}
interface Person<Phone extends string> {
    phone: Phone
}
```

- this will rise an error where all declaration must have the same type parameters

Classes and Interfaces: Anonymous Inner Classes

- In TypeScript is possible to define an Anonymous Inner Class

```
function createSpecialObject() {
    return new class {
        greet() {
            console.log("Hi from anonymous inner class!");
        }
    }
}
const o = createSpecialObject();
o.greet();
```

Classes and Interfaces: Anonymous Inner Classes (2)

- You can also use the Anonymous Inner Class to implement an Interface

```
interface Greetings {
    greet(): void;
}
function createImplementation(msg: string): Greetings {
    return new class implements Greetings {
        greet(): void {
            console.log(msg);
        }
    }
}
const myGreet = createImplementation("Hi from Richard!");
myGreet.greet();
```

Classes and Interfaces: Mixin (Implementation)

```
type ClassConstructor = new(...args: any[]) => {}

function withEXDebug<C extends ClassConstructor>(Class: C) {
    return class extends Class {
        debug() {
            let Name = Class.constructor.name
            let value = this.getDebugValue()
            return Name + '(' + JSON.stringify(value) + ')'
        }
    }
}
```

- With Type Aliases and Anonymous Inner Classes in mind, we have realized a mixin in TypeScript

Classes and Interfaces: Mixin

- Mixin are a way to share functionalities betwen class or interfaces without directly extends or inherits that
- Natively not supported by TypeScript, but they can be implemented building a function that get as input a class and returns a new anonymoud class that extends the one given as input and introducing new functionalities (as method or fields) in the first one

Generics

Generics: Base Syntax

```
function id<T>(arg: T): T {
    return arg;
}

let result = id<string>("ciao");
let result = id("ciao");
```

Generics: Class and Interfaces

```
class Container<T> {
    private value: T;
    constructor(value: T) {
        this.value = value;
    }
    getValue(): T {
        return this.value;
    }
}
interface Pair<T, U> {
    key: T;
    value: U;
}
```

- Note that a static method do not have access to the generics type of the class

Generics: Constrain on Generics

```
interface HasDistance {
    distance: number;
}
function logDistance<T extends HasDistance>(arg: T): void {
    console.log(arg.distance);
}
```

In this way, `logDistance` will accept only types that have a distance property.

Generics: Primitive Type in Parametric Type

- You can use primitive type in parametric type also

Generics: Generics in Type Aliases

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

Generics: Bounding Generics By Type Aliases

Generics: Bounding Generics By Structurally Shape

Handling Errors

Handling Errors: Base syntax

```
try {
    let result = riskyFunction();
    console.log(result);
} catch(error) {
    console.error("An error occurred!", error);
} finally {
    console.log("This runs no matter what.");
}
```

Handling Errors: Typed error handling

```
try {
    throw new Error("Something went wrong");
} catch (err: unknown) {
    if (err instanceof Error) {
        console.error("Error:", err.message);
    } else {
        console.error("Unknown error");
    }
}
```

Handling Errors: Custom error classes

```
class InvalidValueerro extends Error {
    constructor(value: string) {
        super(`Invalid value: ${value}`);
        this.name = "InvalidValueError";
    }
}

function check(value: string) {
    if (value !== "ok") {
        throw new InvalidValueError(value);
    }
}
```