Practical Typescript

Azat Satklichov
azat.satklichov@broadcom.com

Agenda

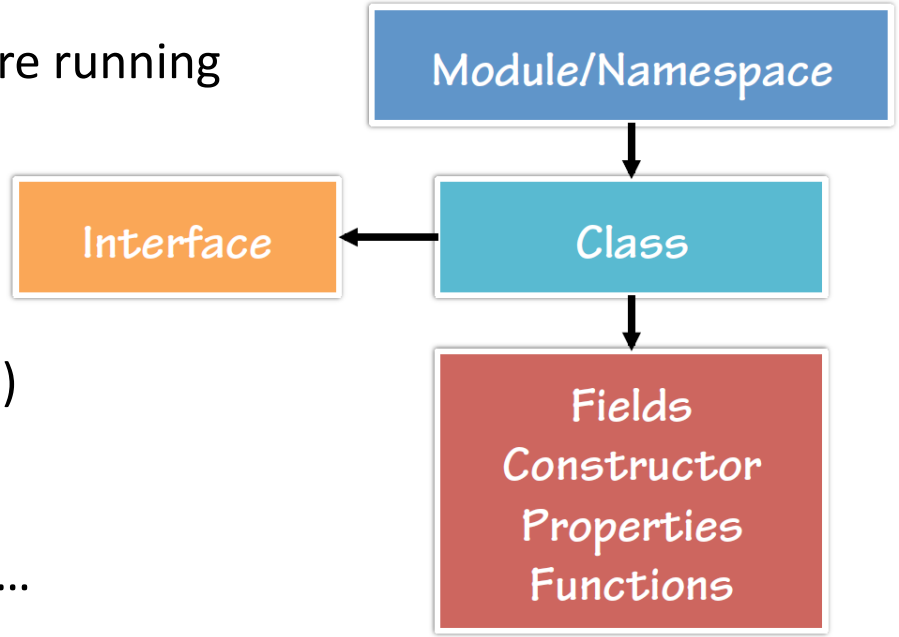
- ❑ Why Typescript
- ❑ Type Annotations/Inferences, Data Types
- ❑ Access modifiers, Properties
- ❑ Type Definition Files, Ambient Declaration. Typings
- ❑ Functions (Arrow), Params options, Overloading, …

- ❑ Powerful Features (**Minimizing TS**)
  - ✓ Class Expression
  - ✓ Destruction
  - ✓ Spread operator
  - ✓ Combined Types (Union, Intersection)
  - ✓ String literal types, Type aliases
  - ✓ Declaration Merging
  - ✓ Type Guard

https://www.typescriptlang.org/play

Why VSCode

- Javascript don't dos (null, undefined, ===, this, ..) still may exist, so follow Do's and Don't in TS
- To get rid off Function Spaghetti Code → Ravioli Code (JavaScript Patterns) (Each JS module has specific concern)
- TS is Super Set of Typescript. Transpiled to JS code (ES3, ES5, Ecma2015[ES6], ..) , IIFE, ..
- Static Typing - Compile Time Type Checking eliminates errors before running
- Types declarations (*d.ts) - once using DOM, jQuery, lodash, ..
- Features
  - EcmasScript 2015 – let/const (block scope, not hoisted, unique def.)
  - Classes, Abstract classes, Enums, Tuples, Interfaces assistances
  - Type Inference, Types Erasure, Destructions, Spread ..
  - Async Programming (callbacks, Promise, async/awat), Decorators, …
- TS alternatives – Dart, CoffeeScript, Applying JavaScript Patterns, ..
- Cross Platform, Open Source



Data Types (primitive, Object)

Built-in: boolean, number, string, any (try to avoid no typing benefit), and Ecmascript 2015 new primitive type 'symbol'
Custom: enum, classes, interfaces, array, ..

Type annotation and inference

```
var num1 = 1; //Type inference (number)
var num3:number; //safe via Type Annotation
var num2: number = 23; //Type Annotation and the Value
//safe via type inference
num1 = "d";
```

What about intellisense support with **ANY?** **Casting**

```
let msg; //implicit ANY
msg = 'abc';
//msg.   NO INTELLISENCE
//1-way cast
(<string>msg).startsWith('a');
//2-way
(msg as string).startsWith('a');

(<number> msg).toPrecision;
```

Data Types (Enums, Arrays, Tuples ..)

```
//Cobol Punch Card has 5 fields on positions 0-6, 7, 8-11, 12-72, 72-80
enum PunchCard { Sequence = 0, Indicator = 7, AreaA = 8, AreaB = 12, IdentificationArea = 73 };

let startPosition: PunchCard = PunchCard.AreaA;
console.log(startPosition); //8
console.log(PunchCard.IdentificationArea); //73
let fieldName: string = PunchCard[startPosition];
console.log(fieldName); //AreaA
console.log(PunchCard[12]); //AreaB

//Also enums can be used with string values, mix, …
enum PrintMedia { Newspaper = "NEWSPAPER", Magazine = "MAGAZINE"}
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE

//Arrays
let arr1: string[] = ['a', 'b', 'c'];
let arr2: Array<string> = ['a', 'b', 'c'];
let arr3: any[] = ['a', true, 23];

//Tuple
let tuple: [number, string] = [123, "Broadcom"];
```

Access modifiers, Properties

Access modifiers (on fields methods), Encapsulation.
Public, [Also public if not defined], Private, Protected

```
class Point {
    x: number;
    y: number;
    constructor(x?: number, y?: number) {
    this.x = x;
    this.y = y;
}
```

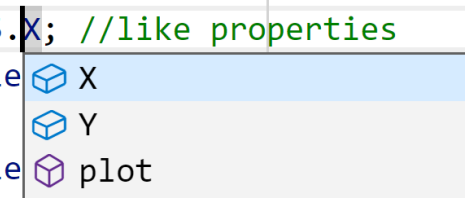
Constructor pattern (shorthand way). No multiple **constructor** (use optional params, or static factory methods)
TS compiler will generate fields implicitly with same name & initialize.

```
class Point {
    constructor(private x?: number, private y?: number) { //can be also public, then mutable,..
    }
}
```

How to access private fields?

1) Getters & Setters.
2) Concept of Properties: camelCase fields (get X **clashes** with x, so use _x)

```
//concept of properties
let p5 = new Point5(23, 40);
//p5.x;
x = p5.X; //like properties
console
console
```



Other Usage of '_' : E.g. app.get('/forms', (_req, res) => {..} to defer warning "req" is declared but its never read.

Type Definition Files, Ambient Declaration

- Once you work with Javascript & DOM (Table, Input .. elements)
- lib.d.ts is referenced by default for the **DOM** and JavaScript in TS.
- Ambient Declarations (**declare**) do not appear anywhere in the JavaScript
- *.d.ts files not to run but to give context for code-hints, err-detection
- Primarily used as a TS wrapper for JS libraries

TS

```
declare var document;

document.title = "Hello";
```

Javascript

```
document.title = "Hello";
```

E.g.
```
var table: HTMLTableElement  = document.createElement('table');
```

- Once working with third-party libs (jQuery, lodash etc.) you need *.d.ts file. E.g.

```
///</// <reference path="lodash.d.ts" />
import * as _ from "lodash";
console.log(_.snakeCase('UEFA Champions League')); //uefa_champions_league
```

Type Definition Files. Typings

- Get *.d.ts sources from GitHub (even you can contribute via PR ;))

https://github.com/borisyankov/DefinitelyTyped, http://definitelytyped.org/

- Tools to manage: direct download from GitHub, Nuget, tsd, typings
- 1-way: npm i --save lodash --save-dev @types/lodash

- 2-way: using tsd.json [deprecated] – Find&download *.d.ts files, keeps all /// in single file
  npm install tsd –g, then tsd install lodash –save, or tsd install jquery –save // typings folder

- 3-way: typings.json is new and like tsd but gets files from multiple sources GitHub, SVN, … add typings folder. Configure tsconfig.json files:["typings/main.d.ts“] to remove ///<…

➢ npm install typings –global

➢ typings install jquery --save

> typings -v

2.1.1

Arrow Functions. Void return type

- Compact form of expressions.
- Omit function keyword
- Have scope this

```
var myFun = function (x: number, y: number) {
            return x * y;
}

var myArrFun = (x: number, y: number) => x * y;
```

TS function with return type **void**. Still returns undefined, no compile error

```
greetMe('Hello!');
let x = greetMe('Hello!');
console.log(x);
```

TS

```
v();
System.out.println(v());
}
```

Cannot resolve method 'println(void)'

JAVA

Declaring Parameters.

In JS by default all parameters are OPTIONAL. But in Typescript all parameters are required by default.

```
//optional, default params.
var myFun2 = function (x?: number, y: string = 'Lidl') {
        //TBD
}


//No warning for un-used variables
public var(_variable: string) {
     console.log('Just see, no warning even variable not used' );
}


//rest parameters
var myFun3 = function (x: number, ... ids: number[])  {
   //tbd
}

myFun3(2);
myFun3(2, 3);
myFun3(2, 55, 453);
```

Overloaded Functions

In TS once types are removed during transpilation to JS, this adds ambiguity .. .

```
//define overloaded functions
function getIds(user: string): string[];
function getIds(active: boolean): string[];

//implementation function
function getIds(factor: any): string[] {
    if (typeof factor == 'string') {
          //tbd
     } else if (typeof factor == 'boolean') {
        //tbd
     }
     return []; //TBD result
};
```

Class Expressions

In JS function expressions were used a lot.

```
abstract class Animal {
    abstract swim(txt: string): void;
}
//giving a class name is optional
let Dolphin = class extends Animal {
   swim(txt: string): void {
      console.log('swim like ' + txt);
   }
}
let myDolphin = new Dolphin();
myDolphin.swim('Dolphin');


//how to use class expression in extension
class Sharq extends class {name : string} { //not confuse with object, this is decl.
   elasmobranchii : string;
}
let mySharq = new Sharq();
mySharq.elasmobranchii = 'elasmobranchii Sharq';
mySharq.name = 'Alpha';
```

Destructing assignments

The process of assigning the elements of an array or the properties of an object to individual variables.

```
let apples: string[] = ['Granny Smith', '
Opal', 'Goldspur', 'Ligol', 'Melba'];

let apple1 = apples[0];
let apple2 = apples[1];
let apple3 = apples[3];
console.log(apple1);//Granny Smith

//destructing arrays
let [jablko1, jablko2, jablko3] = apples;
console.log(jablko1);//Granny Smith
```

```
let car  = {
    model: 'Skoda Fabia Kombi',
    karoserie: 'Kombi',
    assembly: 'Mlada Boleslav'
};

// let model = car.model;
// let karoserie = car.karoserie
// let assembly = car.assembly;

//property names exact match.
let {model, karoserie, assembly} = car;
console.log(model);//new variable
//if object props not match or different naming
let {model: znacka, karoserie:style, assembly:factory} = car;
console.log(znacka);//new variable
```

Spread operator

```
let greenApples: string[] = ['Granny Smith', 'Lodi', 'Smeralda'];

//using spread operator
let allApples: string[] = ['Opal', 'Goldspur', 'Ligol', 'Melba', ...greenApples];
```

Combining Types (union, intersection)

```
//union types used for parameters and return types
function getReport1(id: number | string) {
        //TBD
}
```

```
//intersection types - all properties must match
function getReport2(id: Mobile & Tablet) {
  //TBD
}
```

String Literal Types

```
//string literals - acts like distinct types
let mr : 'Mister'; //type of variable
let mr1: 'Mister' = 'Mister';
let mr2: 'Mister' = 'Madam'; // err – type is not assignable
//gives enum like behavior, e.g. to finite values
let mr3: 'Mister' | 'Madam' = 'Madam';
let mr4: 'Mister' | 'Madam' = 'Miss';//err
```

Type aliases

```
//type aliases
let mr5: 'Mister' | 'Madam' | 'Miss' = 'Miss';
type mrCategory= 'Mister' | 'Madam' | 'Miss';
let mr6: mrCategory= 'Madam';
let mr7: mrCategory= 'Madam';
let mr8: mrCategory= 'Mrs'; // err - is not assignable to type
```

Declaration Merging

Declaration merging – compiler merges two separate declarations declared with same name into one

```
interface Mashyn {
  name: string;
  go(): () => void;
}

//somewhere in app another Mashyn interface
interface Mashyn {
    color: string;
    stop(): () => void;
}

//TS compliler merges it and sees as single interface. Code completion support, ..
class Volga implements Mashyn {
    name: string;
     go(): () => void {
                    throw new Error("Method not implemented.");
      }
      color: string;
      stop(): () => void {
          throw new Error("Method not implemented.");
      }
}
```

Can be merged: Interfaces, Enums, Namespaces, Namespaces with classes | functions | enums

Can not be merged: Classes with classes. Workaround is Mixin concept

Type Guards, User defined Types

Compiler can do check more errors on narrowed block based on type guards [typeof, instanceof] .
Typesof is also used in Overloaded method implementation, …

```
let x: string | number = 144;
 if(typeof x === 'number') {
   //TYPE is NARROWED to NUMBER
   //(not exist in Java until Java 15)
} else {
   //narrowed to STRING. Compiler does this
 }
```

typeof drawback - only used for specific types
(string, number, booleand and symbol)

Instanceof - works on other types, which has a constructor, ..

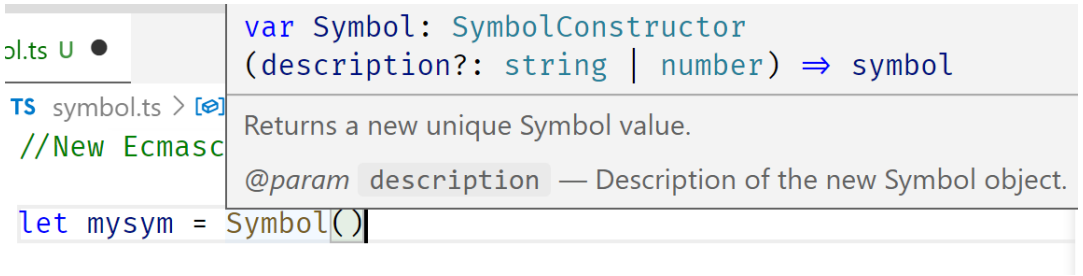
```
class Football {}
class Hockey {}
let sport: Football | Hockey = new Football();
if(sport instanceof Football){
  //narrowed to Football, so safe to use
}
```

User defined Type Guards

```
interface Drink { taste: string}
//Java has sealed Classes concept, can be used for similar usecase
function isDrink(d : any) : d is Drink {
    return (<Drink> d).taste !==undefined;
}
let f = new Football();
if(isDrink(f)){
 console.log('Yes it is a drink type');
} else {
   console.log('It is not a drink');
}
```

Symbols

New Ecmascript 2015 feature, in tsconfig.json change compiler option: "target": "ES2015“ (no ES5).
No **new** used to create symbol. No constructor function. Symbols are new primitive data type. Its type is SYMBOL. It is unique




```
let mySym1 = Symbol(23);
let mySym2 = Symbol(23);
console.log(mySym1 === mySym2);//false
console.log(typeof mySym2);//symbol
```

Functional programming, Currying in TS

```
const test = (a: string, b: string) => b + " " + a;
test("I am arg1", " I am arg2"); // I am arg1 I am arg2

//currying  -  nesting returning functions and be able to partially consume a function
const curr = (a: string) => (b: string) => b + " " + a;
curr("I am arg1")(" I am arg2"); // I am arg1 I am arg2

//const compute = (a: number, f: (x:number) => number) : number => f(a);
const compute : (a: number) => (f: (x : number) => number) => number = a => f => f(a)
```

Thank You

References

https://www.typescriptlang.org/docs/
https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html
https://github.com/danielstern/compiling-typescript
https://medium.com/@cb.yannick/functional-programming-with-typescript-part-1-3f7167a2c0ad

BROADCOM®