Typescript 1.8

**Types**

* **Simple** Types: **boolean/number/string/T[]|Array<T>/enum T {V1, V2, …}/any/void/[T1,T2,…]**. // [T1,T2,…] is tuple.
  + An **Array of different Types**: **any[]**. e. g. let anyArray: any[] = [42, true, ‘life’];
  + **Tuple**: An **Array of hybrid data types**.
    - **Init**: let myTuple: **[number, boolean, string]** = **[42, true, ‘life’]**;
    - **Access**: myTuple**[2];**  // returns ‘life’
  + **Cast or Type Assertion**: (**<T>**myValue) / myValue **as** T. e.g. <string> // in **JSX** only **‘as’** is supported
* **Custom** Types: **Function Type / Class Type** (see below)
* **String:** 
  + Regular: **" … "**, **' … '**
  + Multi-line Template String: **` … `** , Embedded Expression: **` … ${ variable } `**
* **Array**: let list: **string[]** = **[** “aaa”, “bbb”, “ccc” **]**;
* **Tuple**:
  + let x: [string, number] = [“aaa”, 42]; // heterogeneous type Array
  + Access: x[0] // OK, x[3] = “bbb”; // OK using union type. need more details!
* **Enum**: enum Color {Red, Green, Blue};
  + **name to number**: Color**.Green** // returns number 1
  + **number to string**(name): Color**[1]** // returns **string “Green”**
* **Any**:
  + Good use case on array of multiple types: let list: **any[]** = [1, true, "free"];

**Variable Capturing by Function – A different perspective:** the **Scope** where the variable belongs to **IS CAPTURED!**

* function foo() can capture variable (i.e. not via params) exists outside function foo(). And those captured variable can still exist even AFTER the code within its scope has finished excecuting. How? Bcos the **Scope** didn’t go away and instead **IS CAPTURED**.

function s() { // **begin of scope**

var **a** = 10;

return function **foo()** { var b = **a** + 1; return b; } // **function foo() capture variable ‘a’**

} // **end of scope**

var f = s(); // s returns function foo()

// After s() is called, all code within s() has finished executing, however the scoped is still available since it is captured by foo()

alert(f()); // variable ‘a=10’ is still available when executing foo(). alert: 11

**const, let, var:**

* Declaring Variable: **let/const**/var. **Always** use **let and const** bcos they are **block-scoped**! Avoid use var.
* **var** is **function-scoped:**  **function f()** { … } // e.g. if use ‘var’ in a **for loop, ONLY 1 scope** will be created and captured
  + **Function capturing ‘Scoped Variable’**: **ONLY 1 copy** of the variable **from 1 scope** is captured

for (**var** i = 0; i < 10 ; i++) { // ‘**var i**’ will look for an **outer function scope** to put itself into

setTimeout(function() {console.log(i); }, 100 \* i); // output ‘10’ everytime

}

* **let** is **block-scoped:**  **{** … **}** // e.g. if use ‘let’ in a **for loop** with **n iterations**, **n scopes** will be created and captured
  + **Function capturing ‘Scoped Variable’**: **n copies** of the variable **from n scope** are captured

for (**let** i = 0; i < 10 ; i++) { // ‘**let i**’ will **honor the block-scope** here

setTimeout(function() {console.log(i); }, 100 \* i); // output 0 to 9

}

* e.g. const x: number = 42; let y: number = 42; ~~var z: number = 42;~~

**Destructing Assignment – extract data from array/object into variables**

* **Variable Array Destructing:**
  + **let [x, y, z]** = [42, 43, 44];// create **variables x, y & z** whose values are **42, 43 & 44** respectively
  + **let [x]**  = [42, 43, 44];// create **1 variable x** whose value is **42**
  + **let [, x, ]**  = [42, 43, 44];// create **1 variable x** whose value is **43**
  + **let [a, …rest]** = [42, 43, 44];// create **variables a & rest** whose values are **42 & [43, 44]** respectively
* **Variable Object Destructing:**
  + **let {x, y, z} = { x: 42, y: “aaa”, z: true };** // create **variables x, y & z** whose values are **42, “aaa” & true** respectively
    - p.s. the **name of variables** **MUST MATCH** the **name of the object member**
  + **Default value**: let { x **= 18**, y, z } = otherObject;
  + **Pre-declared variable**: let a, b; **(** {a, b} = {a: "baz", b: 101} **)**; // **()** is required because otherwise **{ }** will be treated as block
  + **Rename Object Property**: let { a**: newNameA**, b**: newNameB** } = {a: "baz", b: 101}; // **rename** ‘a’ to newNameA

**Function - Standalone**

* **Define (i.e. with Impl.):** can capture variable OUTSIDE the function body!
  + **Named**: **function MyFunc(**name: string, id: number): string { return name + id; }
  + **Anonymous**: **let** myFunc **= function(**name: string, id: number): string { return name + id; }
  + **Lambda 1 line**: let myFunc **=** (name: string, id: number): string **=>** name + id; // no need to use the ‘return’ keyword
  + **Lambda 2+ lines**: let myFunc **=** (name: string, id: number): string **=> {** **return** name + id; **}**
    - ‘**this**' captured by **lambda** will be **bound at function creation** rather than invocation like javascript does.
* **Function** **Type** **‘ : ’** : i.e. **without body / impl.**
  + e.g. **let** myFunctionType**: (p1: T1, p2: T2) => TResult;** // TResult can be any data type including **void**
  + use **=>** when declare it as a **standalone** Function Type
  + use **:** when declare it as a **Function Type Interface**
* **Parameter**:
  + **Optional: p?**: T // when optional param is missing, the value is ‘undefined’. **if(x === “undefined”)** …
  + **Default:** p: **T =** someDefaultValue
  + **Rest: …params: T[]** // Collects a group of **parameters into a single array**
    - **Access individual param: params[0]**
* **Overload**

function foo(x: **{**suit: string; card: number; **}[]**): **number**;

function foo(x: **number**): **{**suit: string; card: number; **}**;

function foo(x): **any** {

if (**typeof** x **==** **"object"**) { // **testing** if it is an array

return someNumber;

}

else if (**typeof** x **==** **"number"**) {

return **{** … **}**;

}

}

**Interface**: [more](https://www.typescriptlang.org/docs/handbook/interfaces.html)

* Duck Typing. i.e. **Anything(incl. *object literal*)** in the **same shape** of the interface can be used on an interface param.
* **Interfaces** are capable of describing a **Contract** for
  + **Object Literal Interface**: Javascript **Object Literal with 1+ Properties**:- **p1: T1; / Optional** property - **p1?:T1;**
    - **Match both name & type of the properties defined in the interface**
    - **Object literal** provided can have **more properties than** the **interface** defined.
    - interface IFoo {

name: string; // Simple Type

age: number; // Simple Type

addRating: (rating: number) => void; // Standalone Function Type

}

* + - Usage: let obj:IFoo; obj = { name: “Vincent”, age: 42 };
  + **Function Type Interface**: Javascript **Function Type Object:-** **(p1: T1, p2: T2): TResult;**
    - **Match** the **signature**. **Param names no need to match**
    - interface IFoo {

(name: string): string; **// Cannot have more than 1 Function Type in a single Interface**

}

* + - Usage: // **supports both anonymous & lambda** function
      * let ift = **<IFoo>** **function** (name: string): string { … return someString; };
      * let ift: IFoo; ift = foo; // where let foo = <IFoo> function(name: string): string { … return someString; };
  + **Index Type Interface**: Javascript **Indexable Type Object like Array/Dictionary:-**  **[index: number/string]: TE;**
    - **Match both Index type & Element type**. **Support 2 kinds** of signature:
      * [index: **number**]: TE; // e.g. a = [“Bob”, “Carl”, “Den”]. a[0]
      * [index: **string**]: TE; // e.g. myObj[“propertyName”]
      * Advanced: support BOTH at the same time.
    - interface IFoo {

[index: number]: string;

[index: string]: string;

}

* + **Class Interface**: Class to implement – **class** MyClass **implements** MyInterface
    - Properties:- p1: T1; p2: T2;
    - Methods:- MyMethod(p1: T1, p2: T2): TR;
    - For **public** contract **only**. No private!
* Dynamic Interface (a **Hybrid** of Contracts above) – useful for dealing with the dynamic nature of ‘native Javascript’

interface Counter {

(start: number): string; **// Declare Function Object Type**

interval: number; **// Declare Class Property**

reset(): void; **// Declare Class Method**

}

function createCounter(): Counter { // Create a Hybrid Object

let counter = **<Counter>** **function** (start: number) { }; **// Define Function Object. Looks like ONLY Anonymous Function works!**

counter.interval = 123; **// Define Class Property**

counter.reset = function () { }; **// Define Class Method**

return counter;

}

let c = createCounter(); // Instantiate Object

c(10); **// Use: Function Object Type**

c.reset(); **// Use: Class Method**

c.interval = 5.0; **// Use Class Property**

* Interface **Extend** Composition
  + interface MyInterface3 **extends** MyInterface2, MyInterface1 { … }
* Inteface **extends** Classes (Advanced) :- interface MyInterface **extends** MyClass { … }

Class

class MyClass {

name: string; // property

**constructor**(authorName: string) { name = authorName; }

**get** AuthorName(): string { return name; } // no need to use **this** keyword

**set** AuthorName(arthorName: string) { name = arthorName; }

**static** Desc: string = ‘hello world’; // accessing by prepending: **MyClass.Desc**

**MyMethod(**myParam: string**)**: void { name = myParam; } // p.s. **no *function* keyword**

**private/protected** foo():void { … } // **default public**

}

* **Constructor**
  + **ONLY ONE** ctor allowed. e.g. class MyClass { **constructor**(p1: T1) { … } … }
  + **Overload**: use optional params to give us a similar effect of overload. e.g. **constructor**(p1: T1, **p2?**: T2, **p3?**: T3) { … }
* **Parameter ’Properties’**: let you create and initialize a class **‘Property’** in 1 place.
  + e.g. class MyClass { constructor(**private** name: string) { }
    - will **auto generate** a **Property**:- private name: string; and then init with whatever passed to the constructor!
* **‘new’ up a class**: let myInstance = **new** MyClass(42);
* **accessors**: the ***signature is a function***
  + **get** foo**(): T** { **return** this.MyProp; }
  + **set** foo**(value: T)** { this.MyProp = **value**; }
* **Access Modifier**: **public (default) / private / protected**
* **Constant**: cannot use **const** inside a class!!! Alternative: **static PI: number = 3.14;**
* ‘**this**’ always refer bound to the **current object instance**.
* **Inheritance**
  + class MyDerived **extends** MyBased { … } // inherit **base class**
  + class MyDerived **implements** IBased { … } // inherit **base interface**
  + class MyDerived **extends** MyBased **implements** IBased { … } // inherit BOTH base class & interface
  + calling **base class *ctor***: **constructor**() { … **super(…);** }
  + calling **base class *properties/methods***: **super.myProp**/**super.myMethod**(…)
* **Abstract Class / Method**
  + **abstract** **class** MyClass { **abstract** **MyMethod**(p1: T1): TResult; … }
* Class Expression ???
  + let MyClassExp = class extends MyBased { // impl. Base class methods … }
  + let myInstance = new MyClassExp( … );
* Instance vs Static side of a class ???
  + Instance
    - let c: MyClass;
    - c\_Instance = **new** MyClass(…); // using ‘instance’ side of MyClass
  + Static
    - let c\_Static: **typeof** MyClass= MyClass; // using ‘static’ side of MyClass … the constructor
    - let o: MyClass = new c\_Static();

Iteration

let list = [4, 5, 6];

**for** (let i **in** list) { // for in returns a list of **keys**

console.log(i); // "0", "1", "2",

}

**for** (let i **of** list) { // for of returns a list of **values**

console.log(i); // "4", "5", "6"

Namespace – internal use ONLY

* Create mynamespace.ts

**namespace** MyNamespace {

**export** class MyClass { … } // **expose** class/interface/function/nested namespace/

}

* Use

/// <reference path=”mynamespace.ts” />

**MyNamespace.**MyClass/MyFunction/MyNestedNamespace

* **Alias** for long namespace or nested namespace
  + **import** *shortname* = **veryverylongnamespace**;
  + **import** *shortname2* = mynamespace.**nestednamespace**

Modules

* **DON’T USE namespace inside** a module!
* Each module is a file
* Module Loader: Angular 2 using System.js
* Export
  + **Member level**: e.g. **export** interface MyInteface … / **export** class MyClass …
  + **Export Statement**: **export {** MyInterface**,** MyClass, MySomethingElse **}** // Centralize on 1 line
  + **Default Export:**
    - **export default class** { … }
* Import
  + **Selective Import: import {** MyInterface**,** MyClass **} from** ‘./**themodule**’ // from **file themodule.ts without** the file **extension**
    - **Opt-in ONLY** the interested (MyInterface, MyClass) exported members of a module.
  + **ALL**: import **\* as TheModuleAlias** from ‘./**themodule**’
  + Import **Default Export**: import **TheModuleAlias** from ‘./themodule’

Generic

* Generic Function: function Foo**<T>**(p1**: T**)**: T** { … return p1;)
* Generic Interface: interface MyInterface**<T>** { getLatestItem: () => **T**; addItem: (item: T) => void; getAllItem: () => Array<T>; }
* Generic Class:
  + class MyClass<T> implements MyInterface<T> {
    - private \_items: Array<T> = new Array<T>();
    - getLatest(): T { return this.\_items[0]; }
    - addItem(item: T): void { this.\_items.push(item); }
    - getAllItem(): Array<T> { return \_items; }
  + }

Type Definitions

* **Ambient Module**:

**declare module** “*underscore*" {

export = \_;

}

* Get Type Definitions: [Typescript Definition Manager (TSD)](http://definitelytyped.org/tsd/), [Nuget](http://www.nuget.org/packages?q=Definitelytyped), [GitHub](https://github.com/DefinitelyTyped/DefinitelyTyped)
* Usage:
  + **/// <reference path="***underscore***.d.ts" />**
  + **Import \* as \_ from “***underscore***”**