TypeScript

Notes by rajeeshvengalapurath@gmail.com

Multiple lines & embedded expressions

```
let sentence: string = `Hello, my name is ${ fullName }.
                        I'll be ${ age + 1 } years old next month.`;
```

Interfaces

```
interface IInterface{
    label: string;
}
```

```
✓ let o1 : IInt = {label: "test"} ;
✕ let o2 : IInt = {label: "test", xyz: 2} ;
```

Optional

```
interface IInt{
    label: string;
    name?: string //optional
}
```

```
✓ let o1 : IInt = {label: "test"} ;
✓ let o2 : IInt = {label: "test", name: "ee"};
✕ let o3 : IInt = {label: "test", xyz: "ee"}; //xyz is an unknown property
```

The advantage of optional properties is that you can describe these possibly available properties while still also **preventing use of properties that are not part of the interface**

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = {color: "white", area: 100};
    if (config.clor) {
        // Error: Property 'clor' does not exist on type 'SquareConfig'
        newSquare.color = config.clor;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}
```

Excess Property

```
interface IInt{
    label: string;
    name?: string
    [propname: string] : any;
}

✓ let o1 : IInt = {label: "test", name: "ee", abc:3, xyz:"test"};
```

Function Types

```
interface ISearchFunc {
    (source: string, subString: string): boolean;
}

let mySearch: ISearchFunc = function(source: string, subString: string) {
    ...
    return result > -1;
};
```

Indexable Types

```
interface IStringArray {
    [index: number]: string;
}

let myArray: IStringArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

Sample 2

```
class Animal {
    name: string;
}
class Dog extends Animal {
    breed: string;
}

interface NotOkay {
    [x: number]: Animal;
    [x: number]: Dog;
}
```

Class Types

Interfaces describe the public side of the class
Same as c#

```
interface ClockInterface {
    currentTime: Date;
}

class Clock implements ClockInterface {
    currentTime: Date = new Date();
    xyz: number; //not defined by the interface. Same as c#
    constructor(h: number, m: number) { }
}


let o : ClockInterface = new Clock(2,3);
console.log(o.currentTime);
```

Methods

//Same as c#

```
interface ClockInterface {
    myF(a: number): number;
}

class Clock implements ClockInterface {
    myF(a: number): number {
        return a;
    }
}

let o : ClockInterface = new Clock();
console.log(o.myF(9));
```

Extending Interfaces

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = {} as Square;
square.color = "blue";
square.sideLength = 10;
```

Extending Multiple Interfaces

```
interface Shape {
    color: string;
}

interface PenStroke {
    penWidth: number;
```

```
}

interface Square extends Shape, PenStroke {
    sideLength: number;
}

let square = {} as Square;
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```

Hybrid Types

When interacting with 3rd-party JavaScript, you may need to use patterns like this to fully describe the shape of the type

```
interface Counter {
    (start: number): string;
    interval: number;
    reset(): void;
}

function getCounter(): Counter {
    let counter = (function (start: number) { }) as Counter;
    counter.interval = 123;
    counter.reset = function () { };
    return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;
```

Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class.

```
class Control {
    private state: any;
}

interface SelectableControl extends Control {
    select(): void;
}
```

Classes

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```

when we refer to one of the members of the class we prepend **`this`**.. This denotes that it's a member access.

Inheritance

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Public, private, and protected modifiers

Public by default

In TypeScript, each member is public by default.

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

ECMAScript Private Fields

With TypeScript 3.8, TypeScript supports the new JavaScript syntax for private fields

```
class Animal {
    #name: string;
    constructor(theName: string) { this.#name = theName; }
}

new Animal("Cat").#name; // Property '#name' is not accessible outside class
'Animal' because it has a private identifier.
```

TypeScript's private

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;

//or

class Animal {
   constructor(private name: string) { }
}
```

protected

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

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

Protected Constructor

```
class Person {
```

```
    protected name: string;
    protected constructor(theName: string) { this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The 'Person' constructor is protected
```

Readonly modifier

Readonly properties **must be initialized** at their **declaration** or in the **constructor**.

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Accessors (getters/setters)

```
const fullNameMaxLength = 10;

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (newName && newName.length > fullNameMaxLength) {
            throw new Error("fullName has a max length of " + fullNameMaxLength);
        }

        this._fullName = newName;
    }
```

```
}

let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

Static Properties

```
class Grid {
    static origin = {x: 0, y: 0};
}
```

Abstract Classes

```
abstract class Department {
    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {
    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }
}
```

Using a class as an interface

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

interface Point3d extends Point {
    z: number;
}

let point3d: Point3d = {x: 1, y: 2, z: 3};
```

Functions

Function Types

```
let myAdd: (x: number, y: number) => number; //function type declaration
myAdd = function(x: number, y: number): number { return x + y; }; //assignment
myAdd(3,4); //call
```

Inferring the types (contextual typing)

This helps cut down on the amount of effort to keep your program typed.

```
//(x: number, y: number) => number //this part is absent
// myAdd has the full function type
let myAdd = function(x: number, y: number): number { return  x + y; };

// The parameters 'x' and 'y' have the type number
let myAdd: (baseValue: number, increment: number) => number =
    function(x, y) { return x + y; };
```

Optional and Default Parameters

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

✕ let result1 = buildName("Bob");                  // error, too few parameters
✕ let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
✓ let result3 = buildName("Bob", "Adams");         // ah, just right

function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

✓ let result1 = buildName("Bob");                  // works correctly now
✕ let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
✓ let result3 = buildName("Bob", "Adams");         // ah, just right
```

this and arrow functions

In JavaScript, this is a variable that's set when a function is called.

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        // NOTE: the line below is now an arrow function, allowing us to capture
        //'this' right here
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}
```

Overloads

```
let suits = ["hearts", "spades", "clubs", "diamonds"];
```

```
function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
    if (typeof x == "object") {
        let pickedCard = Math.floor(Math.random() * x.length);
        return pickedCard;
    }
    else if (typeof x == "number") {
        let pickedSuit = Math.floor(x / 13);
        return { suit: suits[pickedSuit], card: x % 13 };
    }
}
```

Generics

Generic Types

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

let myIdentity: <T>(arg: T) => T = identity;
```

We could also have used a **different name for the generic type** parameter in the type, so long as the number of type variables and how the type variables are used line up.

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

let myIdentity: <U>(arg: U) => U = identity;
```

generic type as a call signature of an **object literal type**

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

let myIdentity: {<T>(arg: T): T} = identity;
```

Generic Interface

```
interface GenericIdentityFn {
    <T>(arg: T): T;
}

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

let myIdentity: GenericIdentityFn = identity;
```

```
//

interface GenericIdentityFn<T> {
    (arg: T): T;
}

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

let myIdentity: GenericIdentityFn<number> = identity;
```

Generic Classes

```
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };
```

Generic Constraints

```
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);  // Now we know it has a .length property, so no more
                              //error
    return arg;
}
```

Type Parameters in Generic Constraints

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {
    return obj[key];
}
```

Using Class Types in Generics

When **creating factories** in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<T>(c: {new(): T; }): T {
    return new c();
}
```

A more advanced example uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types.

```
class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag;  // typechecks!
createInstance(Bee).keeper.hasMask;   // typechecks!
```

Enums

```
enum E {
    A = getSomeValue(),
    B, // Error! Enum member must have initializer.
}
```

String enums

```
enum Direction {
    Up = "UP",
    Down = "DOWN",
    Left = "LEFT",
    Right = "RIGHT",
}
```

Heterogeneous enums

Technically enums can be mixed with string and numeric members, but it’s not clear why you would ever want to do so :D

```
enum BooleanLikeHeterogeneousEnum {
    No = 0,
    Yes = "YES",
}
```

Unless you're really trying to take advantage of JavaScript's runtime behavior in a clever way, it's advised that you don't do this.

Computed Enum

```
enum FileAccess {
    // constant members
    None,
    Read    = 1 << 1,
    Write   = 1 << 2,
    ReadWrite  = Read | Write,
    // computed member
    G = "123".length
}
```

Type Compatibility

Type compatibility in TypeScript is based on **structural subtyping**. Structural typing is a way of relating types based **solely on their members**. This is in contrast with nominal typing. Consider the following code:

```
interface Named {
    name: string;
}

class Person {
    name: string;
}

let p: Named;
// OK, because of structural typing
p = new Person();
```

In nominally-typed languages like **C#** or **Java**, the **equivalent code** would be an **error** because the Person **class does not explicitly** describe itself as being an **implementer** of the **Named interface**.

TypeScript's structural type system was designed based on how JavaScript code is typically written. **Because JavaScript widely uses anonymous objects** like function expressions and object literals, it's much more natural to represent the kinds of relationships found in JavaScript libraries with a structural type system instead of a nominal one.

Nullable

```
let sn: string | null = "bar";
```

Tree Nodes

```
type Tree<T> = {
    value: T;
    left: Tree<T>;
    right: Tree<T>;
}
```

String Literal Types

enum-like behavior

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";
class UIElement {
    animate(dx: number, dy: number, easing: Easing) {
        if (easing === "ease-in") {
            // ...
        }
        else if (easing === "ease-out") {
        }
        else if (easing === "ease-in-out") {
        }
        else {
            // error! should not pass null or undefined.
        }
    }
}

let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // error: "uneasy" is not allowed here
```

Numeric Literal Types

```
function rollDice(): 1 | 2 | 3 | 4 | 5 | 6 {
    // ...
}
```

These are seldom written explicitly, but they can be useful when narrowing issues and can catch bugs:

```
function foo(x: number) {
    if (x !== 1 || x !== 2) {
        //         ~~~~~~~
        // Operator '!==' cannot be applied to types '1' and '2'.
```

```
    }
}
```

Discriminated Unions

```
interface Square {
    kind: "square";
    size: number;
}
interface Rectangle {
    kind: "rectangle";
    width: number;
    height: number;
}
interface Circle {
    kind: "circle";
    radius: number;
}

type Shape = Square | Rectangle | Circle;
Now let’s use the discriminated union:

function area(s: Shape) {
    switch (s.kind) {
        case "square": return s.size * s.size;
        case "rectangle": return s.height * s.width;
        case "circle": return Math.PI * s.radius ** 2;
    }
}
```

in operator

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

Type Aliases

Type aliases create a new name for a type. Type aliases are sometimes similar to interfaces, but can name primitives, unions, tuples, and any other types that you’d otherwise have to write by hand.

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === "string") {
```

```
        return n;
    }
    else {
        return n();
    }
}

//My test
let s: NameResolver;
s = ()=>{return "d";}
//output=d
```

Interface vs Type Alias

- Interfaces have many similarities to types however interfaces provide us with more capabilities and therefore they are preferred.
- An interface, for example, can be used in conjunction with the extends and implements keywords,
    - whereas a type alias for an object cannot.
- An interface can also have multiple merged declarations, TypeScript merges them together:
    - and a type alias for an object type literal cannot.

```
type User = {
  name: string;
  age: number;
}

const myUser: User = {
  name: "John",
  age: 33
};

//

interface User {
  name: string;
  age: number
}
const myUser: User = {
  name: "John",
  age: 33
};

// merged interfaces

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

```
interface User {
  email: string;
}

const myUser: User = {
  name: "John",
  age: 33,
  email: "me@test.com"
};
```

Symbols (GUID?)

Symbol values are created by calling the Symbol constructor.

```
let sym1 = Symbol();
let sym2 = Symbol("key"); // optional string key
```

Symbols are immutable, and unique.

```
let sym2 = Symbol("key");
let sym3 = Symbol("key");

sym2 === sym3; // false, symbols are unique
```

Just like strings, symbols can be used as keys for object properties.

```
const sym = Symbol();

let obj = {
    [sym]: "value"
};

console.log(obj[sym]); // "value"
```

Well-known Symbols

```
Symbol.iterator
Symbol.match
etc..
```

Iterators and Generators

An object is deemed iterable if it has an implementation for the Symbol.iterator property

for..of statements

```
let someArray = [1, "string", false];

for (let entry of someArray) {
    console.log(entry); // 1, "string", false
```

```
}
```

for..of vs. for..in statements

```
let list = [4, 5, 6];

for (let i in list) {
    console.log(i); // "0", "1", "2",
}

for (let i of list) {
    console.log(i); // "4", "5", "6"
}
```

Modules

- Modules are **executed within their own scope**, not in the global scope;
- this means that variables, functions, classes, etc. declared in a module are **not visible outside the module unless they are explicitly exported** using one of the export forms.
- Conversely, to consume a variable, function, class, interface, etc. exported from a different module, **it has to be imported** using one of the import forms.
- Any file containing a top-level import or export is considered a module. Conversely, a file without any top-level import or export declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

```
export interface StringValidator {
    isAcceptable(s: string): boolean;
}

2.

import { StringValidator } from "./StringValidator";

export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

Export statements

```
class ZipCodeValidator implements StringValidator {
    ..
}
export { ZipCodeValidator };
```

```
export { ZipCodeValidator as mainValidator };
```

Re-exports

File: ParseIntBasedZipCodeValidator.ts

```
export class ParseIntBasedZipCodeValidator {
    isAcceptable(s: string) {
        return s.length === 5 && parseInt(s).toString() === s;
    }
}

// Export original validator but rename it
export {ZipCodeValidator as RegExpBasedZipCodeValidator} from "./ZipCodeValidator";
```

AllValidators.ts

```
export * from "./StringValidator";
export * from "./ZipCodeValidator";
```

Import

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

imports can also be renamed

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

Import the entire module into a single variable, and use it to access the module exports

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

Namespacing

```
namespace Validation {
    export interface StringValidator {
        ..
    }

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
        ..
    }

    export class ZipCodeValidator implements StringValidator {
        ;;
    }
}
```

```
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
validators["Letters only"] = new Validation.LettersOnlyValidator();
```

Multi-file namespaces

Validation.ts

```
namespace Validation {
    export interface StringValidator {
        ..
    }
}
```

LettersOnlyValidator.ts

```
namespace Validation {
    export class LettersOnlyValidator implements StringValidator {
        ..
    }
}
```

Promise

```
export class AppComponent {

  promiseOutput
  posts = []
  err = true

  sendPost(post){
    return new Promise((resolve, reject)=>{
      setTimeout(() => {
        this.posts.push(post)

        if(!this.err){
          resolve('success')
          this.err=true
        } else {
          reject('some error')
          this.err=false
        }

      }, 2000);
    })
  }

  createPost(){
    this.sendPost(1).then((resp)=>{
      this.promiseOutput=resp
    }).catch((err)=>{
      this.promiseOutput=err
    })
  }
```

```
}
```

Handling promises with async await (An elegant way)

```
async function init(){
   await createPost({ title: 'test', body: 'body test'});
   getPosts();
}
```

Callback function with type

```
fn1(){
    this.fn2(this.myCallback)
}
fn2(cb: (n: number)=>{}){
    let str = cb(3) //str is "text 3"
}
myCallback(n: number){
    return "text " + n
}
```

Custom asynchronous function

```
testfn(){
       console.log("a");

       setTimeout(() => {
              let i: number=0;
              for (let index = 0; index < 1000000000; index++) {
                     I++;
              }
              console.log(i);
       }, 0);

       console.log("b");
}

//output
//a
//b
//1000000000
```

Code inside setTimeout function will take its own time to execute. Set second argument as 0

Observable vs Promise

```
const myPromise = new Promise(resolve => {
   setTimeout(() => {
```

```
      resolve('dog') ✓ Only first function is called
      resolve('cat') ✕
      resolve('dog') ✕
   }, 100)
}

myPromise.then(result => {
   console.log('promise: ', result)
})

/*
promise: dog
*/

const myObservable = new Rx.Observable(observer => {
   setTimeout(() => {
      resolve('dog')
      resolve('cat')
      resolve('dog')
   }, 100)
})

myObservable.subscribe(result => {
   console.log('observable: ', result)
})

/*
observable: dog
observable: cat
observable: bird
*/

myObservable
.filter(result => result === 'bird')
.subscribe(result => {
   console.log('observable: ', result)
})

/*
observable: bird
*/
```