Intro to TypeScript Types and Interfaces

CS569 – Web Application Development II

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

Maharishi University of Management - Fairfield, Iowa

TypeScript

TypeScript is an open-source object-oriented language developed and maintained by Microsoft. It is a typed superset of JavaScript that compiles to plain JavaScript.

TypeScript was first released in October 2012.

It's the official language adopted by the Google Angular Team to write Angular projects.

Official website: https://www.typescriptlang.org

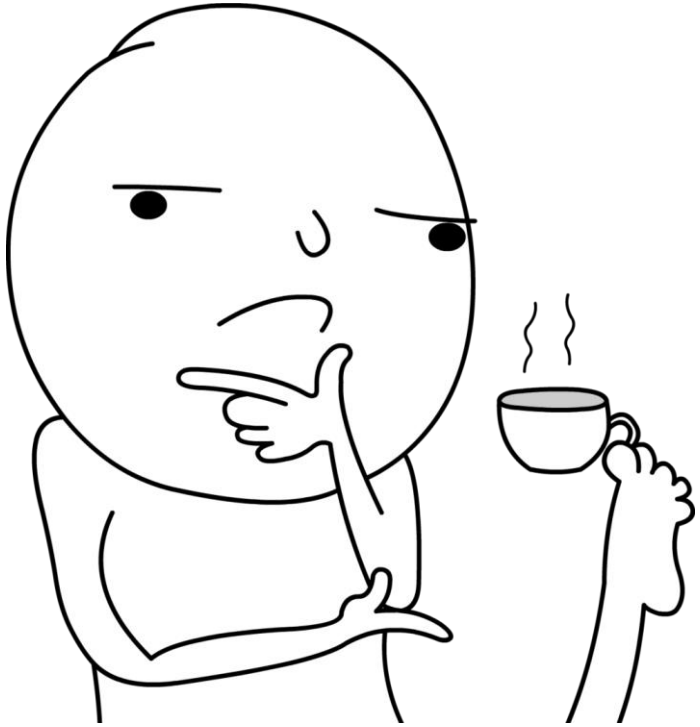
Source code: https://github.com/Microsoft/TypeScript

Quiz!

Is it ever possible that

`(a==1 && a==2 && a==3)`

could evaluate to **`true`** in JavaScript?



Why TypeScript

JavaScript is a dynamic programming language with no type system.

A no type system means that a variable in JavaScript can have any type of value such as string, number, boolean etc.

The type system increases the code quality, readability and makes it an easy to maintain and refactor code base.

Errors can be caught at compile time (development) rather than at run time.

Enterprise Web Applications

Without the type system, it is difficult to scale JavaScript to build complex applications with large teams working on the same code.

The reason to use TypeScript is that it allows JavaScript to be used at scale.

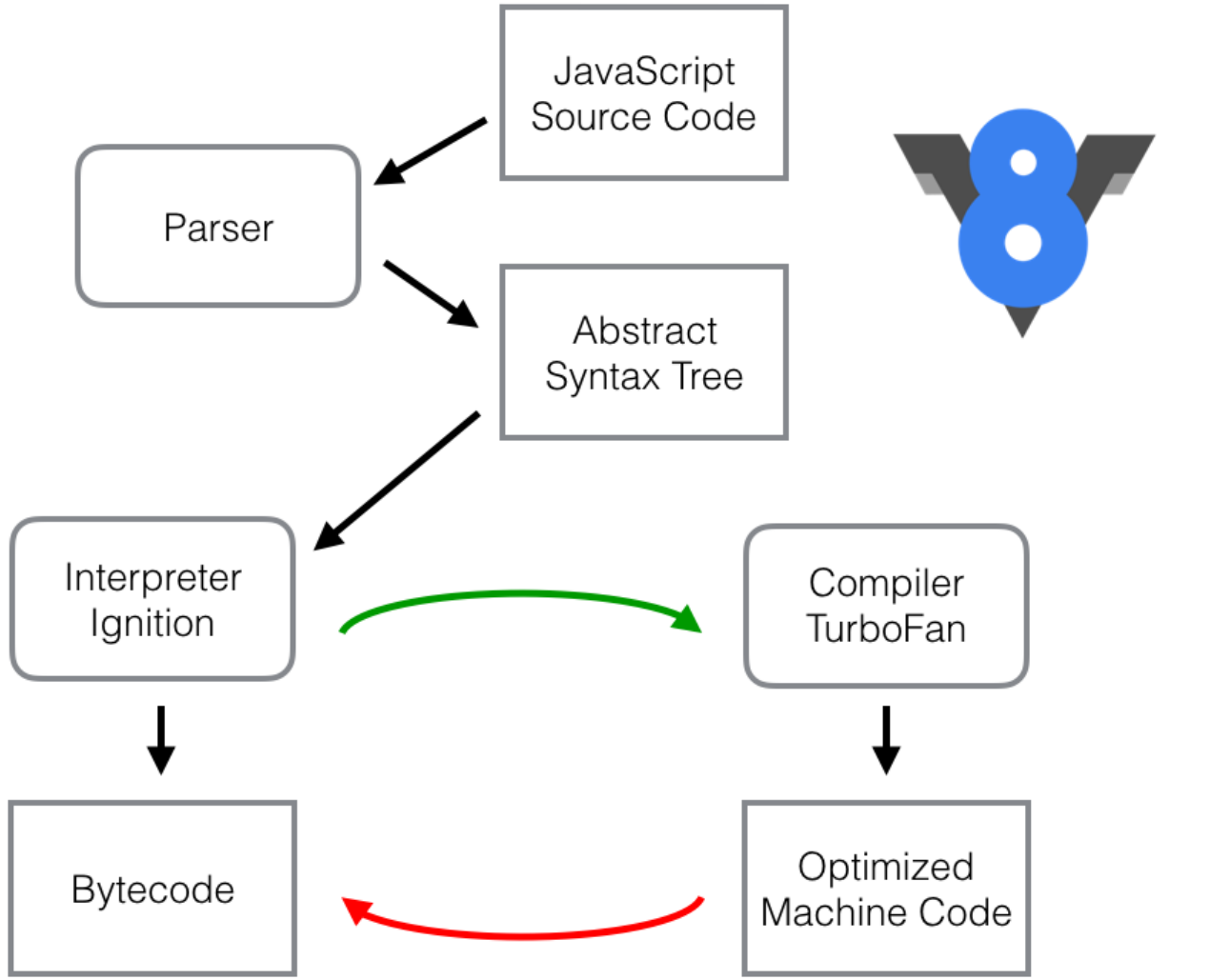
Why Types?

One of the great things about type checking is that:

1. It helps writing safe code because it can prevent bugs at compile time.
2. Compilers can improve and run the code faster.

It’s worth noting that types are **optional** in TypeScript.

V8



Google

@fhinkel

TypeScript Compiler

TypeScript compiles into simple JavaScript.

A TypeScript code is written in a file with **`.ts`** extension and then compiled into JavaScript using the TypeScript compiler.

A TypeScript compiler needs to be installed on your platform. Once installed, the command **`tsc filename.ts`** compiles the TypeScript code into a plain JavaScript file.

TypeScript Features

- Cross-Platform
- Object Oriented Language
- Static type-checking
- Optional Static Typing
- DOM Manipulation
- ES6/next Features

Setup

```
npm install -g typescript
tsc -v
tsc filename.ts
```

Microsoft provides Visual Studio Code with TypeScript support built in.

https://www.typescriptlang.org/play

tsconfig.json

To create a TypeScript configuration file:

```
tsc -init //it creates a tsconfig.json
```

*Since all configurations live in this file we could simply type **tsc** and it's going to automatically find all ***.ts** and compile them to JavaScript.*

```
{
   "compilerOptions": {
       "module": "commonjs",
       "target": "es5",
       "noImplicitAny": false,
       "noEmitOnError": true,
       "sourceMap": true,
       "outDir": "./js",
   },
   "exclude": [ "node_modules" ]
}
```

tsconfig.json

Click here for full list of options

Deno runtime

Deno is a simple, modern and secure runtime for JavaScript and TypeScript that uses V8. It is created by **Ryan Dahl** the founder of NodeJS. It supports TypeScript out of the box.

```
choco install deno
deno run filename.ts
```

https://deno.land/

First Example

```
function add(a: number, b: number) {
    return a + b;
}
const sum1: number = add(10,15);

const sum2: number = add('WAD2', 5);
```

Transpile the code to JS.

Type Annotations

We can specify the type using **`:type`** after the name of the variable, parameter or property.

TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
const grade: number = 90; // number variable
const name: string = "Asaad"; // string variable
const isFun: boolean = true; // Boolean variable
```

Type Annotation of Parameters

```
function hello(id:number, name:string) {
    console.log(`Id = ${id}, Name = ${name}`);
}
```

Type annotations are used to enforce type checking. It is not mandatory in TypeScript to use type annotations. Type annotations help the compiler in checking types and helps avoid errors dealing with data types.

Type Annotation in Object

```
let employee : {
    id: number;
    name: string;
};

employee = {
  id: 100,
  name : "John"
}
```

If you try to assign a string value to `id` then the TypeScript compiler will give the following error:

```
error TS2322: Type '{ id: string; name: string; }' is not assignable to type '{ id:number; name: string; }'.
Types of property 'id' are incompatible. Type 'string' is not assignable to type 'number'.
```

number, string, and boolean

```
let grade:number = 90; // number
let n = new Number(90);
console.log(n) // Output: a number object with value 90
console.log(typeof n) // Output: object

let n2 = n.valueOf(); // returns the primitive value of the number
console.log(n2) //Output: 123
console.log(typeof n2) //Output: number

let employeeName:string = 'John Smith';
let isPresent:boolean = true;
```

Note that, **Boolean** with an uppercase B is different from **boolean** with a lowercase b. Upper case **Boolean** is an object type whereas lower case **boolean** is a primitive type.

Array

There are two ways to declare an array:

1. Using **square brackets**

```
let values: number[] = [12, 24, 48];
```

2. Using a **generic array type**, `Array<elementType>`

```
let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
```

You can always initialize an array with many data types elements, but you will not get the advantage of TypeScript's type system.

Multi Type Array

An array in TypeScript can contain elements of different data types.

```
let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

Tuple

Tuple is a new data type where a variable can include multiple data types in the specified array position.

```
let user: [number, string, boolean, number, string];
user = [1, "Asaad", true, 20, "Faculty"];

let family: [number, string][];
family = [[1, "Asaad"], [2, "Mike"], [3, "Mada"]];
```

Enum

Enums allow us to declare a set of named constants, a collection of related values that can be numeric or string values.

Enum values start from zero and increment by 1 for each member.

Enum in TypeScript supports reverse mapping.

```
enum Technologies {
  Angular,
  React,
  ReactNative
}

// Technologies.React; returns 1
// Technologies["React"]; returns 1
// Technologies[0]; returns Angular
```

```
console.log(Technologies);

{
    '0': 'Angular',
    '1': 'React',
    '2', 'ReactNative',
    Angular: 0,
    React: 1,
    ReactNative: 2
}
```

Union Type

Union type allows us to use more than one data type for a variable or a function parameter.

(type1 | type2 | type3 | .. | typeN)

```
let course: (string | number);
let data: string | number;
function process(code: (string | number)){}
```

Any

When you do not have prior knowledge about the type of some variables and deal with dynamic content, we can **any** type.

```
let something: any = 'Asaad';
something = 569;
something = true;

let data: any[] = ["Asaad", 569, true];
```

Type Inference

It is not mandatory to annotate types in TypeScript, as it infers types of variables when there is no explicit information available in the form of type annotations.

```
let text = "some text";
text = 123; // Type '123' is not assignable to type 'string'
```

Type inference in complex objects

TypeScript looks for the most common type to infer the type of the object.

```
let data = [1, 2, "Asaad"];

data.push(3);
data.push(true); // Type 'true' is not assignable to type 'string | number'
```

Type Assertion

Type assertion allows you to set the type of a value and tells the compiler **not to infer** it. *(similar to type casting)*

```
let code: any = 123;
let courseCode = <number> code;

console.log(typeof(courseCode)); // number
```

Type Assertion with Object

```
// the compiler assumes that the type of employee is {} with no properties.
let employee = {};
employee.name = 'Asaad'; // Compiler Error: Property 'name' does not exist on type '{}'
```

Interfaces are used to define the structure of variables. TS compiler will autocomplete employee properties

```
interface Employee {
    name: string;
}

let employee = <Employee>{};
employee.name = 'Asaad'; // OK
```

There are two ways to do Type Assertion

1. Using the angular bracket **<>** syntax

```
let code: any = 123;
let courseCode = <number> code;
```

2. Using **as** keyword

```
let code: any = 123;
let courseCode = code as number;
```

Function Parameters

In TypeScript, the compiler expects a function to receive the exact number and type of arguments as defined in the function signature.

The parameters that may or may not receive a value can be appended with a '**?**' to mark them as optional.

```
function Sum(x: number, y: number) : number {
    return x + y;
}

function Greet(greeting: string, name?: string  = "my dear" ) : string {
    return greeting + ' ' + name + '!';
}
```

Function Overloading

You can have multiple functions with the same name but **different parameter types and return type**. However, the number of **parameters** should be the same.

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {
    return a + b;
}

add("Hello ", "Asaad"); // returns "Hello Asaad"
add(10, 20); // returns 30
```

Interface

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

An interface is defined with the keyword interface and it can include properties and method declarations using a function or an arrow function.

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number;
    getManagerName(number): string;
}
```

Interface as Type

Interface in TypeScript can be used to **define a type** and also to **implement** it in the class. We can have optional properties, marked with a "?". We can mark a property as read only.

```
interface KeyPair {
    readonly key: number;
    value?: string;
}

let kv1: KeyPair = { key:1, value:"Asaad" };
let kv2: KeyPair = { key:2 };
kv2.key = 3; // Compiler error
```

Interface as Function Type

```
interface KeyValueFunction{
    (key: number, value: string): void;
};

let kvf: KeyValueFunction = function (key:number, value:string):void {
    console.log(key = ' + key + ', value = ' + value)
};

kvf(1, 'Asaad'); // 'key = 1, value = Asaad'
```

A function that does not return a value, actually returns **undefined.** only **null** or **undefined** is assignable to **void.**

Interface for Array Type

```
interface NumList {
    [index:number]:number
}

let numArr: NumList = [1, 2, 3];
numArr[0];
numArr[1];
```

Extending Interfaces

Interfaces can extend one or more interfaces. The object from the extended interface **must include all the properties and methods from both interfaces**, otherwise, the compiler will show an error.

```
interface ICity {
    name: string;
}

interface IZipcode extends ICity {
    zipcode: number;
}

let northStreet: IZipcode = {
    zipcode:52557,
    name:"Fairfield",
}
```

Implementing an Interface

Interfaces can be implemented with a Class. The Class implementing the interface needs to **strictly conform to the structure of the interface**.

The implementing class can define extra properties and methods, but at least it must define all the members of an interface.

Implements Example

```
interface ICourse {
    code: number;
    name: string;
    getGrade:(number)=>number;
}

class Course implements ICourse {
    code: number;
    name: string;
    constructor(code: number, name: string) {
                this.code = code;
                this.name = name;
    }
    getGrade(code:number):number {
        return 90;
    }
}

let course = new Course(569, "Web Application Development II");
```

Class

Classes are the fundamental entities used to create reusable objects. Functionalities are passed down to other classes and objects can be created from classes.

The class in TypeScript is compiled to plain JavaScript function constructor by the TS compiler to work across platforms and browsers.

A class can include the following:

- Constructor
- Properties
- Methods

Constructor

The constructor is a special method which is called when creating an object. An object of the class can be created using the **new** keyword.

It is not necessary for a class to have a constructor.

Inheritance

TypeScript classes can be extended to create new classes with inheritance, using the **`extends`** keyword.

```
class B extends A {}
```

This means that the B class now includes all the members of the A class.

The constructor of the B class initializes its own members as well as the parent class's properties using the **`super`** keyword.

Inheritance Example

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

class MSD extends Course {
    code: number;
    constructor(code: number, name:string) {
        super(name);
        this.code = code;
    }
    displayName():void {
        console.log("Name = " + this.name +  ", Course Code = CS" + this.code);
    }
}

let course = new MSD(569, "Web Application Development");
course.displayName(); // Name = Web Application Development, Course Code = CS569
```

We must call **super()** method first before assigning values to properties in the constructor of the derived class.

A class can implement multiple interfaces

```
class MSD implements ICourse, ICode {
    code: number;
    name: string;

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

    display(): void {
        console.log("this.name +  ", Course Code = CS" + this.code);
    }
}

let wad:ICourse = new MSD(569, "Web Application Development II");
wad.display(); // Web Application Development II, Course Code = CS569

let c:ICode = new MSD(569, "Web Application Development II");
c.display(); //Compiler Error: Property 'display' does not exist on type 'ICode'
```

The **MSD** class implements two interfaces - **ICourse** and **ICode**. So, an instance of the **MSD** class can be assigned to a variable of **ICourse** or **ICode** type. However, an object of type **ICode** cannot call the **display()** method because **ICode** does not include it.

```
interface ICourse {
    name: string;
    display():void;
}

interface ICode {
    code: number;
}
```

Method Overriding

```
class Meditator {
    name: string;
    constructor(name: string) {this.name = name }
    meditate(duration:number = 20) {
        console.log(this.name + " is meditating for " + duration + " mins!");
    }
}

class Sidha extends Meditator {
    constructor(name: string) {super(name)}
    meditate(duration = 40) {
        console.log('Meditation started')
        super.meditate(duration);
    }
}

let asaad = new Sidha("Asaad");
asaad.meditate(60);  // Meditation started Asaad is meditating for 60 mins!
```

When a child class defines its own implementation of a method from the parent class, it is called method overriding.

Abstract Class

Abstract classes are mainly for inheritance where other classes may derive from them. **We cannot create an instance of an abstract class**.

An abstract class includes one or more **`abstract` methods or properties**.

The class which extends the abstract class **must** define all the abstract methods.

Abstract Class Example

```
abstract class Course {
    faculty: string;
    abstract name: string;
    constructor(faculty: string) {this.faculty = faculty}
    abstract findByFaculty(string): Course;
}

class MSD extends Course {
        name: string;
        code: number;
    constructor(faculty: string, name: string, code: number) {
        super(faculty); // must call super()
        this.name = name;
        this.code = code;
    }
    findByFaculty(faculty: string): Course {
        // execute AJAX request to find a course from db
        return new MSD(..);
    }
}
```

The class which implements an abstract class must call super() in the constructor.

Access Modifiers

There are three types of access modifiers: **`public`**, **`private`** and **`protected`**. Encapsulation is used to control class members' visibility.

public

By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

```
class Course {
    public code: string;
    name: string;
}

let course = new Course();
course.code = 569;
course.name = "WAD2";
```

code and **name** are accessible outside of the class using an object of the class.

private

The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class Course {
    private code: string;
    name: string;
}

let course = new Course();
course.code = 569; // Compiler Error
course.name = "WAD2"; // OK
```

`protected`

The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

Protected Example

Property **code** is protected and only accessible within class **Course** and its subclasses.

```
class Course {
    public name: string;
    protected code: number;
    constructor(name: string, code: number){
        this.name = name;
        this.code = code;
    }
}

class MSD extends Course{
    private department: string;
    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.department = department;
    }
}

let wad = new MSD("Web Application Development", 569, "Computer Science");
wad.code; // Compiler Error
```

ReadOnly

Read-only members can be accessed outside the class, but their value cannot be changed. Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor.

```
class Course {
    readonly code: number;
    name: string;
    constructor(code: number, name: string){
        this.code = code;
        this.name = name;
    }
}
let course = new Course(569, "WAD");
course.code = 571; // Compiler Error
course.name = 'Web Application Development'; // Ok
```

Static

ES6 includes static members and so does TypeScript. The static members of a class are accessed using the class name and dot notation, without creating an object.

```
class Circle {
    static pi: number = 3.14;

    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```