TypeScript

Rujuan Xing

Maharishi University of Management - Fairfield, Iowa

Context

- Writing large applications in JavaScript is difficult, not originally designed for large complex applications (mostly a scripting language, with functional programming constructs)
- Lacks structuring mechanisms like Class, Module, Interface. (Some are before ES2015)
- TypeScript is a language for **application scale JavaScript development**.
- TypeScript is **typed superset of JavaScript** that **complies to plain JavaScript**.

Fix/Improve JavaScript – different approaches

- Through Libraries or Frameworks
  - jQuery, AngularJS, Knockout, Ext JS…
- New language that extend/improve language features of JavaScript. Superset of JavaScript, compiles to plain JavaScript.
  - CoffeeScript, TypeScript
- Entirely new language with many new features that compile to JavaScript
  - GWT(Google Web Toolkit), Dart

What is 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

What and Why Types?

- When declare variables, we can specify the type of the variables, function parameters and object properties.

```
var age: number = 32; // number variable
```

- One of the great things about type checking is that:
  - It helps writing safe code because it can prevent bugs at compile time.
  - Compilers can improve and run the code faster.

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

Type Annotations

- We can specify the type using `:Type` after the name of the variable, parameter or property. There can be a space after the colon.

- Type Annotation in TypeScript

```
var age: number = 32; // number variable
var name: string = "John";// string variable
var isUpdated: boolean = true;// Boolean variable
```

- Type Annotation of Parameters

```
function display(id: number, name: string) {
    console.log("Id = " + id + ", Name = " + name);
}
```

- Type Annotation in Object

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

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

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.
- TSC is a command-line application written in TypeScript.

TypeScript Setup

https://www.typescriptlang.org/play

- Download and Install Node https://nodejs.org/en/download/
- Install typescript globally
  - `npm install -g **typescript**`
- Check typescript version
  - **`tsc -v`**
- Compile your typescript code with TSC
  - **`tsc`** `filename.ts`
  - **`tsc`** `filename.ts –w //This is in watch mode`
- `Run your code with NodeJS`
  - **`node`** `filename.js`

tsconfig.json

- Best practices, tsconfig.json is located in project root directory.
- It used for
  - Which files should be complied
  - Which directory to compile them to
  - Which version of JavaScript to emit
  - …

tsconfig.json

```
{
    "compilerOptions": {
        "lib": [
            "ES2015",
            "DOM"
        ],
        "module": "commonjs",
        "outDir": "dist",
        "strict": true,
        "target": "ES2015"
    },
    "include": [
        "src"
    ]
}
```

- `include`: which folders should TSC look in to find your TypeScript files?
- `lib`: Which APIs should TSC assume exist in the environment you'll be running your code in? This includes things like ES5's `Function.prototype.bind`, DOM's `console.log`
- `module`: which module system should TSC compile your code to (CommonJS, ES2015, etc.)
- `outDir`: Which folder should TSC put your generated JS code in?
- `strict`: Be as strict as possible when checking for invalid code. This options enforces that all of your code is properly typed.
- `target`: Which JavaScript version should TSC compile your code to (ES3, ES5, ES3015, ES2016, etc.)?

- When input files are specified on the command line, tsconfig.json files are ignored.
  - For example: `tsc filename.ts`
  - use `tsc` instead

TypeScript Features

- Data Types Supported
- Optional Static Type Annotation
- Classes
- Interface
- Modules
- Arrow Expressions
- Type Assertions

Data Types

Data Types: Any

- Any is used when it’s impossible to determine the type

```
let notSure: any = 4;

notSure = 'Maybe a string instead';

notSure = false;
```

Data Types: Primitive

- All numbers are floating point values and type is number
- boolean – true/fasle value
- string – both single/double quote can be used
- void – used in a function returning nothing
- null, undefined – same as JS

```
const isDone: boolean = true;
const lines: number = 42;
const greeting: string = "Hello World";

function bigHorribleAlert(): void {
    alert("I'm a little annoying box!");
}
```

Data Types: Enum

- Enum allows 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. You can change this by manually setting the value of one its members.

```
enum Color { Red, Green, Blue };

let c: Color = Color.Red;


enum Color2 { Red = 0, Green, Blue };

enum Color3 { Red = 3, Green, Blue };
```

Data Types: 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, "John", true, 20, "Faculty"];


let family: [number, string][];

family = [[1, "John"], [2, "Mike"], [3, "Mada"]];
```

Data Types: Union Type

- Union type allows us to use more than one data type for a variable or a function parameter.
- Syntax: **(type1 | type2 | type3 | .. | typeN)**

```
let course: (string | number);

let data: string | number;

function process(code: (string | number)) { }
```

Data Types: 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’];
```

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

```
let fruits2: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
let fruits3: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana’];
```

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.
- Four ways of variable declaration
  - Type and Value in one statement
  - Type but no Value, then Value will be `undefined`
  - Value but no Type, it will be of Any type but may be inferred based on its value
  - Neither Value nor Type, then Type will be Any, Value will be `undefined`.

```
let message1: string = 'Hello World';

let message2: string;

let message3 = 'Hello World';

let message4;
```

Interface & Classes

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 IKeyPair {
    readonly key: number;
    value?: string;
}

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

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.

```
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(445, "Modern Asynchronous Programming");
```

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(445, "Modern Asynchronous Programming");
course.displayName(); // Name = Modern Asynchronous Programming, Course Code = CS445
```

A class can implement multiple interfaces

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

interface ICode {
    code: number;
}

class MAP 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: MAP = new MAP(445, "Modern Asynchronous Programming");
wad.display(); // Modern Asynchronous Programming, Course Code = CS445
```

- The MAP class implements two interfaces - **ICourse** and **ICode**. So, an instance of the MAP 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.

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: number = 40) {
        console.log('Meditation started')
        super.meditate(duration);
    }
}

let john = new Sidha("John");
john.meditate();  // Meditation started John is meditating for 40 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.
- Mostly used when child classes want to share the some but not all behavior, it should be used primarily for objects that are closely related.

Abstract Class Example

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

class MAP extends Course {
    name: string;
    code: number;
    constructor(faculty: string, name: string, code: number)
{
        super(faculty); // must call super()
        this.name = name;
        this.code = code;
    }
    checkPrerequisite(faculty: string): boolean {
        return true;
    }
}
```

- 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 = "CS445";
course.name = "MAP";
```

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

Class Example - Shortcut

- Adding **access modifiers** to the constructor arguments lets the class know that they're properties of a class. If the arguments don't have access modifiers, they'll be treated as an argument for the constructor function and not properties of the class.

```
interface Book {
    bookName: string;
    isbn: number;
}

class Course {

    // public is shorthand for this.name = name, this.code = code
    constructor(public name: string, public code: number) { }

    useBook(book: Book) {
        console.log(`Course ${this.name} is using the textbook:
                ${book.bookName} who's ISBN = ${book.isbn}`);
    }
}
```

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 = "CS445"; // Compiler Error
course.name = "MAP"; // OK
```

protected

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

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

class MAPCourse extends Course {
    private details: string;
    constructor(name: string, code: number, department: string) {
        super(name, code);
        this.details = `${department} - ${this.code}`;
    }
}

let map = new MAPCourse("Modern Asynchronous Programming", 445, "Computer Science");
map.code; // Compiler Error
```

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

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 = 445; // Compiler Error
course.name = 'Modern Asynchronous Programming'; // 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
```

Type Assertion

Type Assertion

- Type assertion allows you to set the type of a value and tells the compiler **not to infer** it.
- This is when you might have a better understanding of the type of a variable than what TypeScript can infer on its own.

```
let code: any = 123;
//we know the code is of type number, even it has been declared as 'any'
//While assigning code to employeeCode, we have asserted that code is of type numbe
r. We're certain about it.
let employeeCode = <number>code;
//so the type of employeeCode is number
console.log(typeof (employeeCode)); //Output: number
```

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;`

Type Assertion with Object

```
let student = {};
student.name = "John"; //Compiler Error: Property 'name' does not exist on type '{}'
student.code = 123; //Compiler Error: Property 'code' does not exist on type '{}'
```

- In the above example, the compiler assumes that the type of employee is {} with no properties.

- Avoid the situation by using Type Assertion: Interfaces are used to define the structure of variables.

```
interface Student {
    name: string;
    code: number;
}

let student = <Student>{};
student.name = "John"; // OK
student.code = 123; // OK
```

Webpack

ES6 Modules

- Within any JS module, everything is considered private, until we export it, we can have two kinds of exports:
- **`export default`** (can be used once) *default export*
- **`export`** (can be used multiple times) *named export*

- To import what is explicitly exported we use:
- **`import`** varForDefault, **`{`**desctructuredExports**`}`** **`from`** './module.js’

ES6 Modules in the Browser

- To use a JS file in the browser that imports JS modules, we must add type attribute to indicate that this is JS file is using modules:

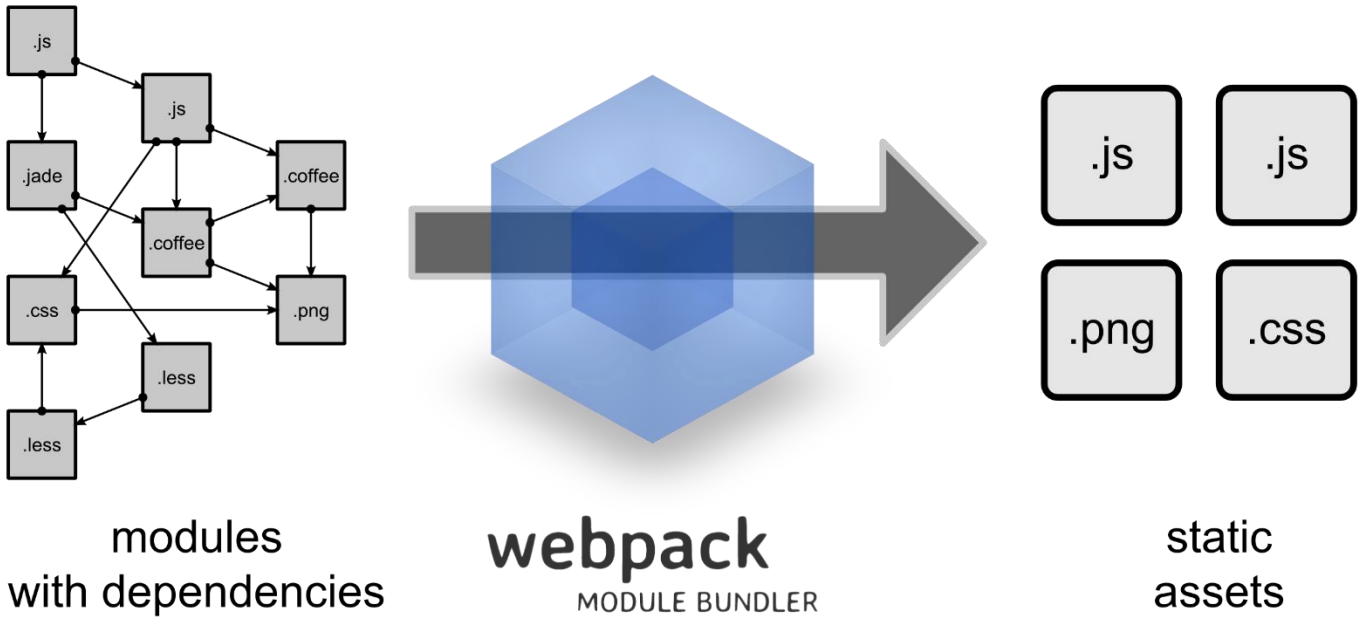
```
<script src="app.js" type="module"></script>
```

- Only modern browsers support modules. As a fallback, we can still bundle the code and provide one JS file for the application.

1. Chrome: cors problem – must deploy your html in some serve. Here we install “live server” plugin in VSC.
2. Click “Go Live” to view your page

Webpack – Module Bundler

- webpack is an open-source JavaScript module bundler.
- It allows you to split your JavaScript into separate modules in development while letting you compile those modules into a single bundle in production.



Using Webpack Bundler – Only Demo what is used for

1. Create a new folder: `learn-webpack`
2. Init package, will generate `package.json`: `npm init -y`
3. Install webpack webpack-cli on dev environment: `npm install webpack webpack-cli --save-dev`
4. Create a file named: `index.js` under `learn-webpack/src` folder
5. Add scripts

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "dev": "webpack --mode development",
  "build": "webpack --mode production"
},
```

6. Start webpack in development mode: `npm run dev`
7. Webpack will generate a `main.js` file under `learn-webpack/dist` folder
8. Add a `index.html` under `learn-webpack/dist` folder. In the index.html, link to `main.js`
9. Now you can open `index.html` in browser.

Using Webpack Bundler - continued

10. Add a `component.js` under `learn-webpack/src` folder

```
export default (text = "Hello, Webpack!") => {
    const element = document.createElement("p");
    element.innerHTML = text;
    return element;
};
```

11. Modify `index.js` to use `component.js`

```
import component from './component';

window.onload = function() {
    document.getElementById("main-content").appendChild(component());
}
```

12. Start webpack in development mode: `npm run dev`. It'll regenerate `main.js` which bundle all js files into a single js file.

13. Reopen `index.html` in browser to see the change.