TypeScript Fundamentals

- By **Srihari Sridharan**

As of TypeScript V 4.5.4 - 2022 02 14 - I ❤️ TypeScript!

To try code samples, copy a snippet and paste it in TypeScript Playground and hit 'Run'! Observe the 'JS' and 'Logs' tabs on the right side!

Need for TypeScript!

JavaScript vs. Maintainability!

This has been a challenge with full-stack JavaScript. There are design patterns to solve this problem, that said, super set languages like TypeScript makes this easier and simpler.

Maintainable Code!

Importance of source code maintainability!

Encapsulation in JavaScript

- Function Spaghetti Code to Ravioli Code (JavaScript Patterns)

JavaScript Dynamic Types

- JS provides dynamic type system
- The good:
    - Variables can hold any object (number, string, array, object, function, etc.)
    - Types determined on the fly
    - Implicit type coercion (ex: string to number)
- The bad:
    - Enterprise applications can have many 1000s of lines of code to maintain
    - Difficult to ensure proper types are passed without tests - more code required for validating and ensuring type safety
    - Not all developers use === for comparison

Alternatives for TS

- DART - https://www.dartlang.org
- CoffeeScript - http://coffeescript.org/
- Pure JS and apply JS Patterns

TypeScript Features

What is TypeScript?

- The TypeScript site defines it as - "TypeScript is JavaScript with syntax for types. TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale."

- Flexible Options
    - Any Browser
    - Any Host
    - Any OS
    - Open Source
    - Tool Support

- Key TypeScript Features
    - Supports standard JavaScript code
    - Provides static typing
    - Encapsulation through classes and modules
    - Support for constructors, properties and modules
    - Define interfaces
    - Lambdas for functions (Arrow syntax `() => {}`)
    - Intellisense and syntax checking

TypeScript to JavaScript

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

```
let greeter = new Greeter("world");
alert(greeter.greet());
// Generated JavaScript
"use strict";
class Greeter {
    constructor(message) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter("world");
alert(greeter.greet());
```

The Fundamentals

Syntax

- Being superset of JavaScript - follows the same syntax rules:
    - `;` ends an expression, but it is optional and follows the same rules as JavaScript
    - {} brackets define the code blocks
- Most of JavaScript key words hold good.
- Remember: TS is super set of JS - meaning every valid JS code is a valid TS code.

Keywords and Operators

- `class` - Container for members such as properties and functions
- `constructor` - Provides initialization functionality in a class
- `exports` - Export a member from a module
- `extends` - Extend a class or interface
- `implements` - Implement an interface
- `imports` - Import a module
- `interface` - Defines code contract that can be implemented by types
- `module/namespace` - Container for classes and other code
- `public/private` - Member visibility modifiers
- `...` - Rest parameter syntax
- => - Arrow syntax used with definitions and functions
- `<type-name>` - < > characters use to cast/convert between types
- `:` - Separator between variable/parameter names and types

Code Hierarchy

- Module/Namespaces
    - Classes (implement interfaces)
        - Fields
        - Constructor
        - Properties
        - Functions

Framework Options and Tools

- Node.js
- TypeScript Playground
- Sublime
- Emacs
- Vi
- Visual Studio
- Visual Studio Code

Basic Types

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a `boolean` value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 64;
let hex: number = 0xf01d;
let binary: number = 0b1110;
let octal: number = 0o734;
```

String

As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

```
let color: string = "blue";
color = 'red';

// Template Strings
let fullName: string = `Srihari Sridharan`;
let age: number = 30;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next year.`;
console.log(sentence);
```

Arrays

2 ways of declaring arrays!

```
// 1. Using []  syntax
let numbers1: number[] = [1, 2, 3, 4 , 5];

// 2. Using Array<Type> syntax
let numbers2: Array<number> = [1, 2, 3, 4 , 5];
```

Tuple

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error, Type 'string' is not assignable to type
'number'.
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, Property 'substr' does not exist on
type 'number'.
```

When accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = "world"; // Error, Tuple type '[string, number]' of length '2' has
no element at index '3'.
console.log(x[5].toString()); // OK, 'string' and 'number' both have
'toString'
x[6] = true; // Error, Tuple type '[string, number]' of length '2' has no
element at index '6'.
```

Enum

A helpful addition to the standard set of data types from JavaScript is the enum. As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Days {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}
```

The equivalent JavaScript looks like:

```
"use strict";
var Days;
(function (Days) {
    Days[Days["Monday"] = 0] = "Monday";
    Days[Days["Tuesday"] = 1] = "Tuesday";
    Days[Days["Wednesday"] = 2] = "Wednesday";
    Days[Days["Thursday"] = 3] = "Thursday";
    Days[Days["Friday"] = 4] = "Friday";
    Days[Days["Saturday"] = 5] = "Saturday";
    Days[Days["Sunday"] = 6] = "Sunday";
})(Days || (Days = {}));
```

Another example below:

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
console.log(c);

// Try this yourself
console.log(Color); // Interesting isn't it?
```

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
console.log(c);
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
console.log(c);
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren't sure what that mapped to in the Color enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];
console.log(colorName); // Displays 'Green' as it's value is 2 above.
// Recollect how it is stored as an object in JS? If not try
console.log(Color); now.
```

```
console.log(Color);
console.log(Color.Blue);
```

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
console.log(notSure);

notSure = false; // okay, definitely a boolean
console.log(notSure);
```

The `any` type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect Object to play a similar role, as it does in other languages. But variables of type Object only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't
check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type
'Object'.
```

The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];
console.log(list[0]);
console.log(list[1]);
list[1] = 100;
console.log(list[1]);
console.log(list[2]);
```

Void

`void` is a little like the opposite of `any`: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    alert("This is my warning message!");
}

warnUser();
```

Declaring variables of type void is not useful because you can only assign `undefined` or `null` to them:

```
let unusable: void = undefined;
console.log(unusable); // :D
unusable = 10; // Error: Type 'number' is not assignable to type 'void'.
```

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their own types named `undefined` and `null` respectively. Much like `void`, they're not extremely useful on their own:

```
// Can't assign to these variables!
let u: undefined = undefined;
let n: null = null;
console.log(u);
console.log(n);
```

By default `null` and `undefined` are subtypes of all other types. That means you can assign `null` and `undefined` to something like number.

However, when using the `--strictNullChecks` flag, `null` and `undefined` are only assignable to void and their respective types. This helps avoid many common errors. In cases where you want to pass in either a `string` or `null` or `undefined`, you can use the union type `string | null | undefined`.

As a note: we encourage the use of `--strictNullChecks` when possible, but for the purposes of this handbook, we will assume it is turned off.

Never

The `never` type represents the type of values that never occur. For instance, `never` is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type `never` when narrowed by any type guards that can never be true.

The `never` type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, `never` (except never itself). Even any isn't assignable to `never`.

Some examples of functions returning `never`:

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

```
let someValue: any = "this is a string";
let str: string = someValue as string;
let strLength: number = (str).length;
console.log(strLength);
```

A note about `let`

You may have noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. Many common problems in JavaScript are alleviated by using `let`, as it is block scoped, so you should use it instead of `var` whenever possible. `var` is function scoped!

Object Types

- Examples - functions, classes, modules, interfaces, and literal types.
- May encompass -
    - Properties
        - public or private
        - required or optional
    - Call signatures
    - Construct signatures
    - Index Signatures

```
// Object literals
let rectangle = { h: 10, w: 20 };
let points: Object = { x: 10, y: 20 };

// Functions
let multiply = function(x: number, y: number) {
    return x * y;
};

// Declared as type 'Function'!
let square : Function;
square = function(x: number) {
    return x * x;
```

```
};

console.log(multiply(2,3));
console.log(square(2));

console.log(rectangle);
console.log(points);
```

Functions

- Parameter types (required and optional)
- Arrow function expressions
    - Compact form of function expressions
    - Omit the function keyword
    - Have scope of “this”
- Void
    - Used as the return type for functions that return no value

```
// TypeScript Code
let areaOfRectangle1 = function(h: number, w: number) {
    return h * w;
};

// The above function can be simplified using arrow function as () => {};
let areaOfRectangle2 = (h: number, w: number) => h * w;

console.log(areaOfRectangle1(2,3));
console.log(areaOfRectangle2(2,3));
"use strict";
let areaOfRectangle1 = function (h, w) {
    return h * w;
};
// The above function can be simplified using arrow function as () => {};
let areaOfRectangle2 = (h, w) => h * w;
console.log(areaOfRectangle1(2, 3));
console.log(areaOfRectangle2(2, 3));
// void used as return type for functions that return no value
let greet: (msg: string) => void;
greet = (message) => console.log(message);
greet("Hello!");
```

Note: the declaration for variable `greet`. This can be seen as a delegate definition as in `C#`. `greet` can point to any function that satisfies this signature. `(msg: string) => void;`

```
// Emits the JavaScript Code
"use strict";
var greet;
greet = (message) => console.log(message);
greet("Hello!");
```

Classes

Defining a Class

- Classes act as containers for different members
- Typical members of a class include:
    - Fields
    - Constructors
    - Properties
    - Functions

```
class Car{
    engine: string;

    constructor(engine: string) {
        this.engine = engine;
    }
}
const c:Car = new Car("Ford");
console.log(c);
// Can be simplified as
class Car{
    // Shorthand way to declare a field
    constructor (public engine:string) { }

    // Public Methods
    start() {
        return "Started " + this.engine;
    }

    stop() {
        return "Stopped " + this.engine;
    }
}

const c = new Car("MPFI");
console.log(c);
console.log(c.start());
console.log(c.stop());
```

- Properties can be defined as shown below. Note: you might need a backing variable.

```
class Car {
    private _engine: string;

    constructor(engine: string) {
        this._engine = engine;
    }

    get engine(): string {
        return this._engine;
    }

    set engine(value: string) {
        if (value == '') throw 'Supply an Engine!';
```

```
        this._engine = value;
    }
}

const c = new Car("MPFI");
console.log(c);
console.log(c.engine);
// c.engine = ''; // Uncomment and check
c.engine = 'SPFI';
console.log(c.engine);
```

Complex Types

- Complex / user defined types can be used for declaring a variable.

```
class Engine {
    constructor(
        public horsePower: number,
        public engineType: string
    ) { }
}

class Car {
    private _engine: Engine;

    constructor(engine: Engine) {
        this._engine = engine;
    }
}
```

Instantiating a Type

- Types are instantiated using the `new` keyword.

```
let engine = new Engine(300, 'V8');
let car = new Car(engine);
console.log(car);
```

Extending Types - Inheritance

- Types can be extended using the TypeScript `extends` keyword
- Child class constructor must call base class (`super`) constructor

```
class ChildClass extends ParentClass {
    constructor() {
        super();
    }
}
class Engine {
    constructor(
        public horsePower: number,
        public engineType: string
    ) { }
```

```
}

class Auto {
    engine: Engine;
    constructor(engine: Engine) {
        this.engine = engine;
    }
}

class Truck extends Auto {
    fourByFour: boolean;

    constructor(engine: Engine, fourByFour: boolean) {
        super(engine);
        this.fourByFour = fourByFour;
    }
}

const engine = new Engine(700, "MFPI");
const truck = new Truck(engine,true);
console.log(truck.engine);
console.log(truck);
```

Interfaces

- What is an interface?
    - An interface is a description of the actions that an object can do.
    - What rather than how?
    - eg: Plug point
- E.g. A factory requires that all engines being built have a standard "interface".

```
interface IEngine {
    start(callback: (startStatus: boolean, engineType: string) => void) :
void;
    stop(callback: (stopStatus: boolean, engineType: string) => void) :
void;
}
```

- Breaking this definition down, `start()` and `stop()` take 1 parameter and return void.
    - The parameter is `callback` - a function.
    - `callback` in turn is a pointer to the function which takes 2 arguments and returns void.
        - `startStatus/stopStatus` of type `boolean`
        - `engineType` of type `string`

Optional Members

```
interface IAutoOptions {
```

```
    engine: IEngine;
    basePrice: number;
    state: string;
    // make, model, and year are optional. Indicated a by a '?' next to
the member name.
    make?: string;
    model?: string;
    year?: number;
}
```

Implementing an Interface

- Interfaces provide a way to enforce a `contract`

```
class Engine implements IEngine {
  constructor(public horsePower: number, public engineType: string) { }

  start(callback: (startStatus: boolean, engineType: string) => void) {
    window.setTimeout(() => {
      callback(true, this.engineType);
    }, 10000);
  }

  stop(callback: (stopStatus: boolean, engineType: string) => void) {
    window.setTimeout(() => {
      callback(true, this.engineType);
    }, 10000);
  }
}

const engine = new Engine(500, "MFPI");
engine.start((status: boolean, engineType: string) => {
  console.log(`Started ${engineType} ${status}`);
});

engine.stop((status: boolean, engineType: string) => {
    console.log(`Started ${engineType} ${status}`);
});
```

Using interface as a Type

- Interfaces help ensure that proper data is passed.

```
class Auto {
    engine: IEngine;
    basePrice: number;
    //More fields…

    constructor(data: IAutoOptions) {
        this.engine = data.engine;
        this.basePrice = data.basePrice;
    }
}
```

Extending an Interface

Defining an Extended Interface

- Extend the definition of an existing interface.

```
interface IAutoOptions {
    engine: IEngine;
    basePrice: number;
    state: string;
    make?: string;
    model?: string;
    year?: number;
}

// Defines IAutoOptions members plus custom members
interface ITruckOptions extends IAutoOptions {
    bedLength?: string;
    fourByFour: boolean;
}
```

Using as Extended Interface

- The class Truck takes `ITruckOptions` as input.

```
class Truck extends Auto {
    bedLength: string;
    fourByFour: boolean;

    constructor(data: ITruckOptions) {
        super(data); // This is something interesting!
        this.bedLength = data.bedLength;
        this.fourByFour = data.fourByFour;
    }
}
```

TypeScript Modules and Namespaces

Why Modules?

- Separation of concerns
- Testable
- Maintainable
- Reusable

What is a Module?

A note about terminology: It's important to note that in TypeScript 1.5, the nomenclature has changed. "Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that `module X {` is equivalent to the now-preferred `namespace X {`).
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.
Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the **CommonJS module loader for Node.js** and **require.js for Web applications**.

In TypeScript, just as in ECMAScript 2015, any file containing a top-level import or export is considered a module.

Export

- Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the export keyword.

```
// Code in StringValidator.ts
export interface StringValidator {
  isAcceptable(s: string): boolean;
}

// Code in ZipCodeValidator.ts
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

- Export statements are handy when exports need to be renamed for consumers.

```
import { StringValidator } from "./StringValidator";
export const numberRegexp = /^[0-9]+$/;
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
```

```
    }
}

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

Re-exports

- A re-export does not import it locally, or introduce a local variable.

```
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";
```

- A module can wrap one or more modules and combine all their exports using export * from "module" syntax.

```
// Code in AllValidators.ts
export * from "./StringValidator"; // exports interface 'StringValidator'
export * from "./LettersOnlyValidator"; // exports class
'LettersOnlyValidator'
export * from "./ZipCodeValidator";  // exports class 'ZipCodeValidator'
```

Import

- Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the `import` forms below:

Import a single export from a module

```
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();
```

Namespaces

- Namespaces (previously internal modules) are used to organize code in TypeScript.
- Additionally, anywhere the `module` keyword was used when declaring an internal module, the `namespace` keyword can and should be used instead.
- What happens behind the scenes?

```
namespace App{
    export class Person {
        constructor(public name: string, public age: number) { }
    }
}
```

JavaScript code looks like:

```
"use strict";
var App;
(function (App) {
    class Person {
        constructor(name, age) {
            this.name = name;
            this.age = age;
        }
    }
    App.Person = Person;
})(App || (App = {}));
```

Let us explore further:

```
namespace App.Shapes {
    export class Rectangle {
        constructor(public height: number, public width: number) { }
    }
}
"use strict";
var App;
(function (App) {
    var Shapes;
    (function (Shapes) {
        class Rectangle {
            constructor(height, width) {
                this.height = height;
                this.width = width;
            }
        }
        Shapes.Rectangle = Rectangle;
    })(Shapes = App.Shapes || (App.Shapes = {}));
})(App || (App = {}));
```

- Code in a single file.

```
// Code in validation.ts
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
```

```
    const lettersRegexp = /^[A-Za-z]+$/;
    const numberRegexp = /^[0-9]+$/;

    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }

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

- Hard to maintain in a single file. Can be split across files. The `/// <reference path="..." />` directive is the most common of this group. It serves as a declaration of dependency between files. Triple-slash references instruct the compiler to include additional files in the compilation process.

```
// Code in Validation.ts
namespace Validation {
    export interface StringValidator {
        isAcceptable(s: string): boolean;
    }
}
// Code in LettersOnlyValidator.ts
/// <reference path="Validation.ts" />
namespace Validation {
    const lettersRegexp = /^[A-Za-z]+$/;
    export class LettersOnlyValidator implements StringValidator {
        isAcceptable(s: string) {
            return lettersRegexp.test(s);
        }
    }
}
// Code in ZipCodeValidator.ts
/// <reference path="Validation.ts" />
namespace Validation {
    const numberRegexp = /^[0-9]+$/;
    export class ZipCodeValidator implements StringValidator {
        isAcceptable(s: string) {
            return s.length === 5 && numberRegexp.test(s);
        }
    }
}
// Code in Test.ts
/// <reference path="Validation.ts" />
/// <reference path="LettersOnlyValidator.ts" />
/// <reference path="ZipCodeValidator.ts" />

// Some samples to try
let strings = ["Hello", "98052", "101"];

// Validators to use
let validators: { [s: string]: Validation.StringValidator; } = {};
validators["ZIP code"] = new Validation.ZipCodeValidator();
```

```
validators["Letters only"] = new Validation.LettersOnlyValidator();

// Show whether each string passed each validator
for (let s of strings) {
    for (let name in validators) {
        console.log(
            """ + s + "" "
            + (validators[name].isAcceptable(s) ? " matches " : " does not
match ")
            + name
        );
    }
}
```

- Once there are multiple files involved, we'll need to make sure all of the compiled code gets loaded. There are two ways of doing this.
    - Compilation into a single file
        - Concatenated output using the `--outFile` flag to compile all of the input files into a single JavaScript output file: `tsc --outFile sample.js Test.ts`. The compiler will automatically order the output file based on the reference tags present in the files.
        - The files can also be specific individually `tsc --outFile sample.js Validation.ts LettersOnlyValidator.ts ZipCodeValidator.ts Test.ts`
    - Per-file compilation (the default) along with `<script>` tag.
    - `<script src="Validation.js" type="text/javascript" />`
    - `<script src="LettersOnlyValidator.js" type="text/javascript" />`
    - `<script src="ZipCodeValidator.js" type="text/javascript" />`
      `<script src="Test.js" type="text/javascript" />`

Alias Names

- Using alias names to shorten lengthy namespace parts:

```
namespace Shapes {
    export namespace Polygons {
        export class Triangle { }
        export class Square { }
    }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square(); // Same as 'new Shapes.Polygons.Square()'
```

Further Reading

Please consider perusing the **'TypeScript Handbook'** which I used to refer and create this example in addition to adding items based on my experience.

Hope you fell in with ❤️ TypeScript!