HOA SEN UNIVERSITY

www.hoasen.edu.vn

TYPESCRIPT

Phạm Thị Kim Ngôn

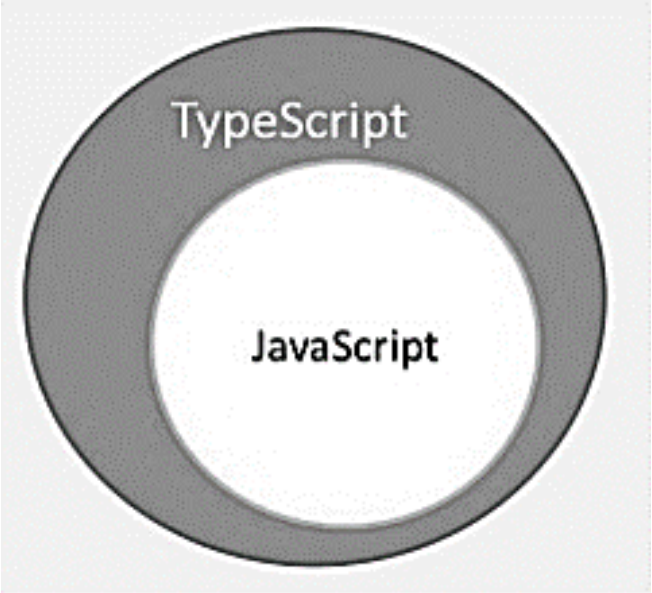
ngon.phamthikim@hoasen.edu.vn

Content

- What is TypeScript?
- Why Use TypeScript?
- Basic Syntax
- Types
- Object Orientation
- Operators
- Functions
- Modules

What is TypeScript?

- TypeScript is a strongly typed, object oriented, compiled language.
- It was designed by Anders Hejlsberg (designer of C#) at Microsoft.
- TypeScript is both a language and a set of tools.
- TypeScript is a typed superset of JavaScript compiled to JavaScript. In other words, TypeScript is JavaScript plus some additional features.



Features of TypeScript

- TypeScript is just JavaScript.
- Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. **All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.**
- TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.
- JavaScript is TypeScript. **This means that any valid .js file can be renamed to .ts and compiled with other TypeScript files.**

TypeScript and ECMAScript

- TypeScript adopts its basic language features from the ECMAScript5 specification, i.e., the official specification for JavaScript.
- TypeScript language features like Modules and class-based orientation are in line with the EcmaScript 6 specification. Additionally, TypeScript also embraces features like generics and type annotations that aren't a part of the EcmaScript6 specification.

Why Use TypeScript?

- **Compilation** - The TypeScript transpiler provides the error-checking feature.
- **Strong Static Typing -** JavaScript is not strongly typed. The type of a variable, declared with no type, may be inferred by the TLS (TypeScript Language Service) based on its value.
- TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

Components of TypeScript

- At its heart, TypeScript has the following three components
  - **Language** – It comprises of the syntax, keywords, and type annotations.
  - **The TypeScript Compiler** – The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
  - **The TypeScript Language Service** – The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.

The TypeScript Compiler

- The TypeScript compiler is itself a .ts file compiled down to JavaScript (.js) file. The TSC (TypeScript Compiler) is a source-to-source compiler.

TypeScript - Basic Syntax

- Modules
- Functions
- Variables
- Statements and Expressions
- Comments

Your First TypeScript Code

- Multiple files can be compiled at once.

```
tsc file1.ts, file2.ts, file3.ts
```

```
var message:string = "Hello World“
console.log(message)
```

```
//Generated by typescript 1.8.10
var message = "Hello World";
console.log(message);
```

Identifiers in TypeScript

- Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are:
  - Identifiers can **include both, characters and digits**. However, the identifier **cannot begin with a digit.**
  - Identifiers **cannot include special symbols** except for underscore **(_)** or a dollar sign **($)**.
  - Identifiers **cannot be keywords**.
  - They must **be unique**.
  - Identifiers are case-sensitive.
  - Identifiers **cannot contain spaces**.

Identifiers in TypeScript

Valid identifiers	Invalid identifiers
firstName	Var
first_name	first name
num1	first-name
$result	1number

TypeScript – Keywords

break	as	any	switch
case	if	throw	else
var	number	string	get
module	type	instanceof	typeof
public	private	enum	export
finally	for	while	void
null	super	this	new
in	return	true	false
any	extends	static	let
package	implements	interface	function
new	try	yield	const
continue	do	catch	

Whitespace and Line Breaks

- TypeScript ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.
- Each line of instruction is called a **statement**.
- A single line can contain multiple statements. However, these statements must **be separated by a semicolon**.

Comments in TypeScript

- Single-line comments (//) – Any text between a // and the end of a line is treated as a comment

- Multi-line comments (/* */) – These comments may span multiple lines.

TypeScript and Object Orientation (1)

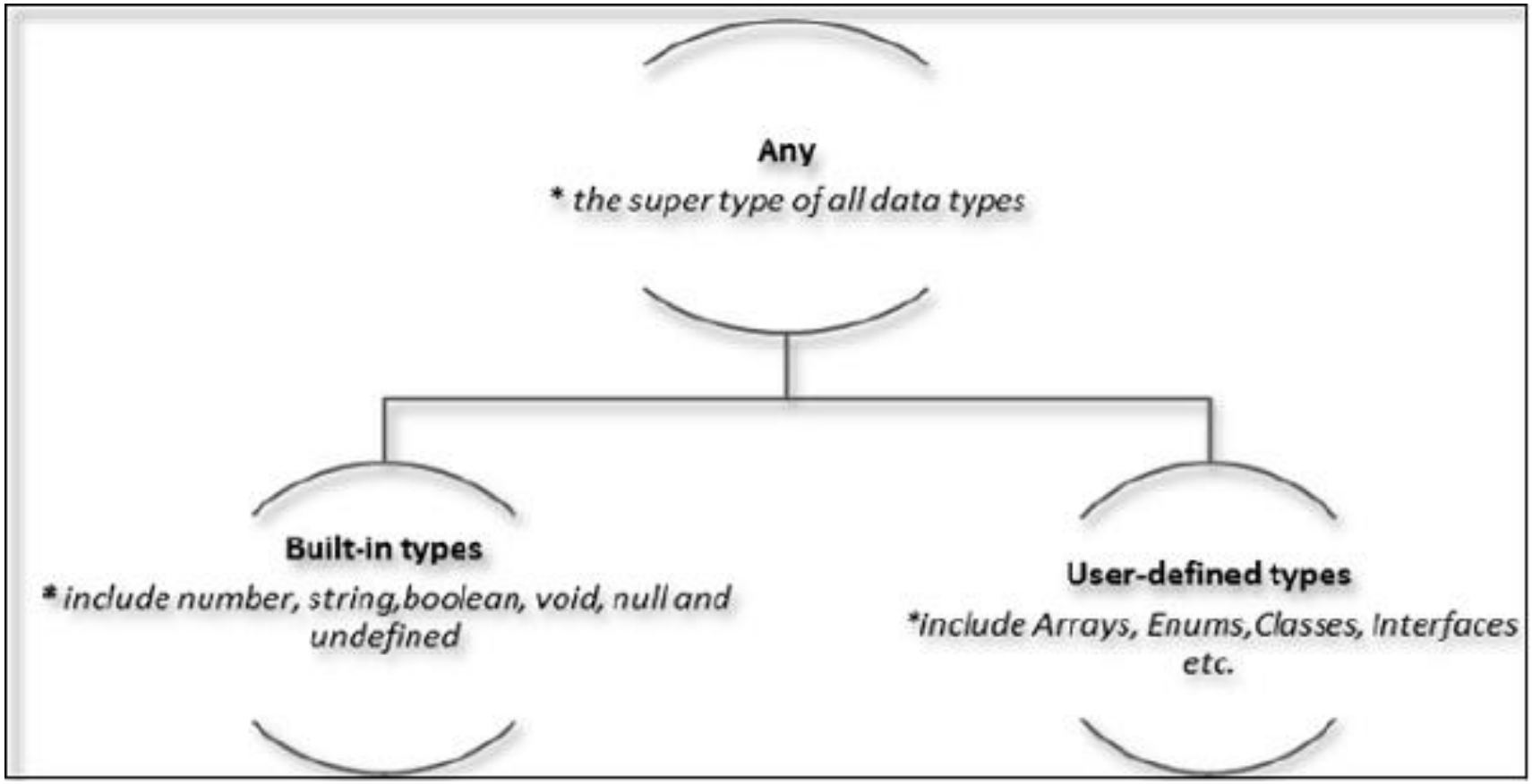
- Object – An object is a real time representation of any entity. Every object must have three features :
  - **State** – described by the attributes of an object
  - **Behavior** – describes how the object will act
  - **Identity** – a unique value that distinguishes an object from a set of similar such objects.
- **Class** – A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object
- **Method** – Methods facilitate communication between objects.

TypeScript and Object Orientation(2)

```
class Greeting {
        greet():void {
                console.log("Hello World!!!")
        }
}
var obj = new Greeting();
obj.greet();
```

```
//Generated by typescript 1.8.10
var Greeting = (function () {
        function Greeting() { }
        Greeting.prototype.greet = function () {
                console.log("Hello World!!!");
        };
        return Greeting;
}());
var obj = new Greeting();
obj.greet()
```

TypeScript – Types (1)



TypeScript – Types (2)

- **Built-in types**

Data type	Keyword	Description
Number	number	Double precision 64-bit floating point values. It can be used to represent both, integers and fractions.
String	string	Represents a sequence of Unicode characters
Boolean	boolean	Represents logical values, true and false
Void	void	Used on function return types to represent non-returning functions
Null	null	Represents an intentional absence of an object value.
Undefined	undefined	Denotes value given to all uninitialized variables

TypeScript – Types (3)

- **User-defined Types**

  User-defined types include **Enumerations (enums), classes**, **interfaces**, **arrays**, and **tuple**.

TypeScript – Types (4)

- **Number**

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

- **String**

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

TypeScript – Types (5)

- **String**
  - You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote **(`)** character, and embedded expressions are of the form **${ expr }**.

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.
I'll be ${ age + 1 } years old next month.`;
```

```
let sentence: string = "Hello, my name is " + fullName +
".\n\n" + "I'll be " + (age + 1) + " years old next month.";
```

TypeScript – Types (6)

- **Array**

  TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

```
let list: Array<number> = [1, 2, 3];
```

TypeScript – Types (6)

- **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.

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

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

```
x[3] = "world";
// OK, 'string' can be assigned to 'string | number'
console.log(x[5].toString());
// OK, 'string' and 'number' both have 'toString'
x[6] = true; // Error, 'boolean' isn't 'string | number'
```

TypeScript – Types (7)

- Enum

A helpful addition to the standard set of datatypes from JavaScript is the enum.

- By default, enums begin numbering their members starting at **0**.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

- 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;
```

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

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];
console.log(colorName); // Displays 'Green' as its value is 2 above
```

TypeScript – Types (8)

- **Any**

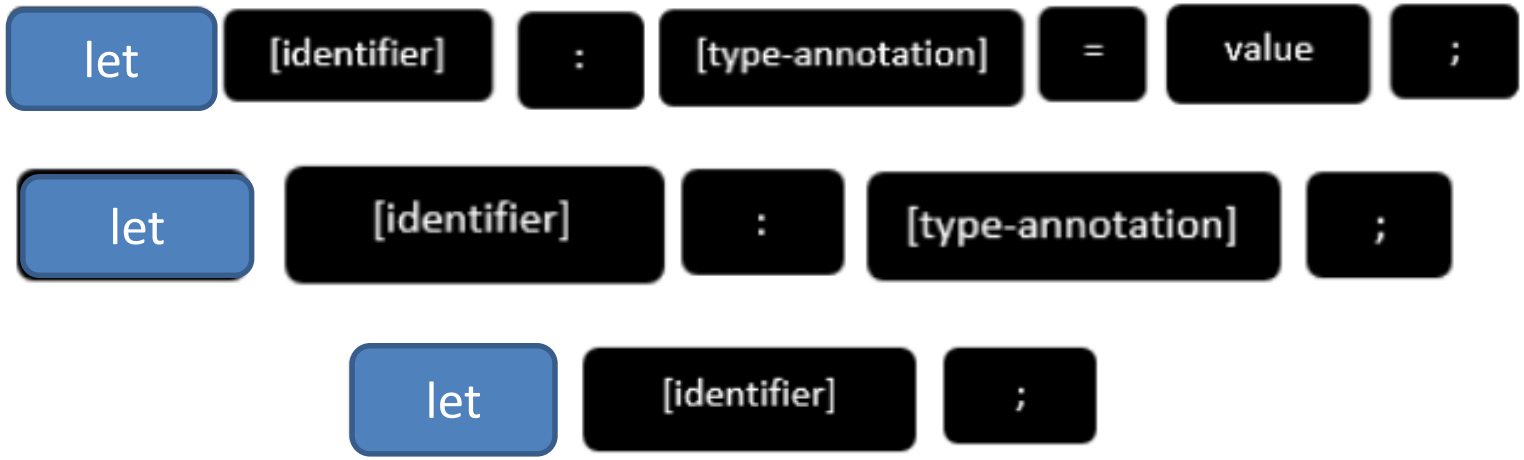
```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Variables

- Variable names can contain alphabets and numeric digits.
- They cannot contain spaces and special characters, except the underscore (_) and the dollar ($) sign.
- Variable names cannot begin with a digit.
- A variable must be declared before it is used. Use the **let** or **var** keyword to declare variables.



Example - Variables

```
var myname:string = "John"
var score1:number = 50;
var score2:number = 42.50
var sum = score1 + score2
console.log("name"+myname)
console.log("first score: "+score1)
console.log("second score: "+score2)
console.log("sum of the scores: "+sum)
```

```
var num = 2; // data type inferred as number
console.log("value of num "+num);
num = "12";
console.log(num);
```

error TS2011: Cannot convert 'string' to 'number'.

Variable Scope (1)

- **Global Scope** – Global variables are declared outside the programming constructs. These variables can be accessed from anywhere within your code.
- **Class Scope** – These variables are also called **fields**. Fields or class variables are declared within the class but outside the methods. These variables can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.
- **Local Scope** – Local variables, as the name suggests, are declared within the constructs like methods, loops etc. Local variables are accessible only within the construct where they are declared.

Variable Scope (2)

```
var global_num = 12 //global variable
class Numbers {
    num_val = 13; //class variable
    static sval = 10; //static field
    storeNum():void {
        var local_num = 14; //local variable
    }
}
console.log("Global num: "+global_num)
console.log(Numbers.sval) //static variable
var obj = new Numbers();
console.log("Global num: "+obj.num_val)
```

Output:

Global num: 12
10
Global num: 13

Variable Scope (3)

- **let** declarations

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        var b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

Operators (1)

- The same some popular languages: C#, C, Java,…
- Conditional Operator (?)
  - **Test** – refers to the conditional expression
  - **expr1** – value returned if the condition is true
  - **expr2** – value returned if the condition is false

Test ? expr1 : expr2

```
let num:number = -2
let result = num > 0 ?"positive":"non-positive"
console.log(result)
```

Operators (2)

- **typeof** operator : It is a unary operator. This operator returns the **data type of the operand**

```
let num = 12
console.log(typeof num); //output: number
```

- **Instanceof** operator: This operator can be used to test if an **object is of a specified type or not.**

If, loop

- if
- for loop
- while, do...while

The same C language

Function

function function_name (param1[:type], param2[:type], param3[:type])

```
function
disp_details(id:number,name:string,mail_id?:string) {
    console.log("ID:", id);
    console.log("Name",name);
    if(mail_id!=undefined)
        console.log("Email Id",mail_id);
}
disp_details(123,"John");
disp_details(111,"mary","mary@xyz.com");
```

Function

- Rest Parameters

```
function addNumbers(...nums:number[]) {
    let i;
    let sum:number = 0;
    for(i = 0;i<nums.length;i++) {
        sum = sum + nums[i];
    }
    console.log("sum of the numbers",sum)
}
addNumbers(1,2,3)
addNumbers(10,10,10,10,10)
```

Default Parameters

function function_name(param1[:type],param2[:type] = default_value) { }

```
function calculate_discount(price:number,rate:number = 0.50) {
        var discount = price * rate;
        console.log("Discount Amount: ",discount);
}
calculate_discount(1000)
calculate_discount(1000,0.3)
```

Output

```
Discount amount : 500
Discount amount : 300
```

Anonymous Function (1)

- Functions that are not bound to an identifier (function name) are called as **anonymous functions**.
- These functions are dynamically declared at runtime.
- Anonymous functions can accept inputs and return outputs, just as standard functions do.
- An anonymous function is usually not accessible after its initial creation.
- Variables can be assigned an anonymous function. Such an expression is called a function expression.

```
let res = function( [arguments] ) { ... }
```

```
let res = function(a:number,b:number) {
        return a*b;
};
console.log(res(12,2))
```

Anonymous Function (2)

- **The Function Constructor**

```
let res = new Function( [arguments] ) { ... }
```

  - The new Function() is a call to the constructor which in turn creates and returns a function reference.

```
let myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3);
console.log(x);
```

Anonymous Recursive Function(3)

```
(function () {
   var x = "Hello!!";
   console.log(x)
})() // the function invokes itself using a pair of
       //parentheses ()
```

Lambda Functions

- Lambda refers to anonymous functions in programming.
- Lambda functions are a concise mechanism to represent anonymous functions.
- These functions are also called as **Arrow functions**.
- **Lambda Function - Anatomy**
  - There are **3 parts** to a Lambda function:
    - **Parameters** – A function may optionally have parameters
    - **The fat arrow notation/lambda notation (=>)** – It is also called as the goes to operator
    - **Statements** – represent the function's instruction set

Lambda Expression

- It is an anonymous function expression that points to a single line of code.

```
( [param1, parma2,…param n] )=>statement;
```

```
var foo = (x:number)=>10 + x
console.log(foo(100)) //outputs 110
```

Lambda Statement

- Lambda statement is an anonymous function declaration that points to a block of code. This syntax is used when the function body spans multiple lines.

```
var foo = (x:number)=> {
    x = 10 + x
    console.log(x)
}
foo(100)
```

Syntactic Variations

```
let func = (x)=> {
    if(typeof x=="number") {
        console.log(x+" is numeric")
    } else if(typeof x=="string") {
        console.log(x+" is a string")
        }
}
func(12)
func("Tom")
```

```
//Generated by typescript 1.8.10
var func = function (x) {
        if (typeof x == "number") {
                console.log(x + " is numeric");
        } else if (typeof x == "string") {
        console.log(x + " is a string");
        }
};
func(12);
func("Tom");
```

Syntactic Variations

- Optional parentheses for a single parameter

```
var display = x=> {
        console.log("The function got "+x)
}
display(12)
```

- Optional braces for a single statement, Empty parentheses for no parameter

```
var disp =()=> {
        console.log("Function invoked");
}
disp();
```

MODULES

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.**
- 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

Export (1)

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

Validation.ts

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

ZipCodeValidator.ts

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

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

Export (2)

- **Export statements**

  Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

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

Export (3)

- **Export statements**

  Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

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

Export (4)

- Optionally, a module can wrap one or more modules and combine all their exports using **export * from "module"** syntax

AllValidators.ts

```
// exports interface 'StringValidator'
export * from "./StringValidator";
// exports class 'LettersOnlyValidator'
export * from "./LettersOnlyValidator";
// exports class 'ZipCodeValidator'
export * from "./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();
```

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

Import(2)

- 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();
```

References

- https://www.tutorialspoint.com/typescript/typescript_overview.htm
- https://www.typescriptlang.org/docs/handbook/modules.html