Table of Contents

Introducing TypeScript

Introduction

TypeScript is a super set of JavaScript that compiles into JavaScript. TypeScript supports type-safety, data types, classes, interfaces, inheritance, modules and much more. It also supports latest standard and evolving features of JavaScript including **ES5 and ES6**.

Typescript 1.0 was released on 1st Oct, 2012 by **Microsoft** as a free and open source programming language for JavaScript based enterprise application development. At the time of writing this book, the latest version of TypeScript was 2.0.

If you or your team come from a strongly typed language like C#, Java and Php then you don't need to learn JavaScript syntax and style to develop your JavaScript application; TypeScript is a good alternative. Since it will give you the feel of object-oriented programming language.

Advantages of TypeScript

- Simplify JavaScript code which is easier to read and debug.
- Provides Type Safety at compile time.
- Unlike JavaScript, it supports classes, Interfaces etc.
- Supports latest standard and evolving features of JavaScript including ES5 and ES6.
- Supports modularity.
- Open source.

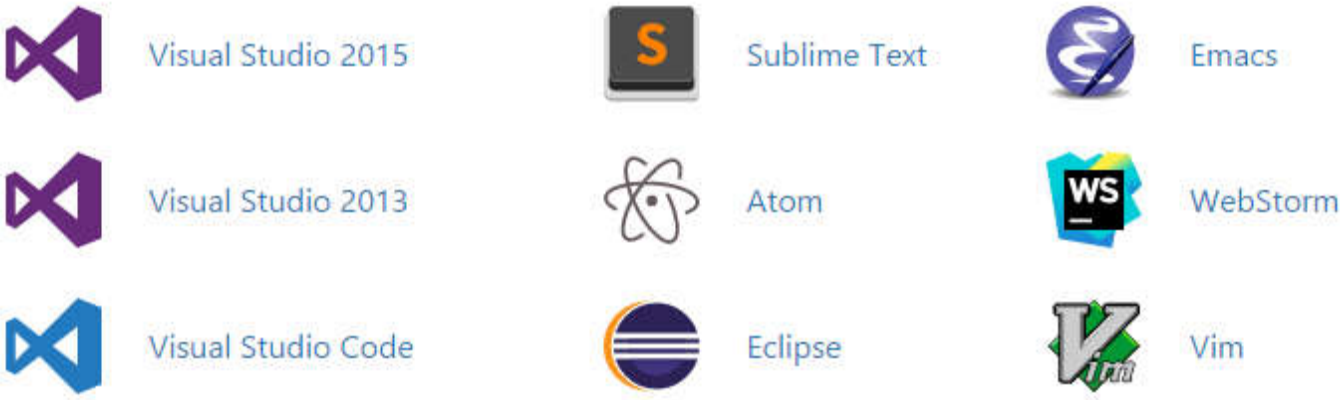
TypeScript based JavaScript Frameworks

TypeScript is used by the many popular JavaScript Framework for developing desktop applications, web applications as well as hybrid and native mobile apps. Some of the frameworks are shown in the fig.

Source: www.typescriptlang.org

Tools And IDE

TypeScript provides highly-productive development tools for JavaScript IDE like WebStrom, Visual Studio Code, Visual Studio, Atom, Brackets, Sublime, Eclipse, Vim etc. These tools support static type checking and code refactoring while developing your JavaScript applications using TypeScript.



Source: www.typescriptlang.org

Getting Started with TypeScript using Visual Studio Code

To get started with TypeScript make sure you have TypeScript Supported IDE. In this book, we will use free, open source and cross platform IDE - Visual Studio Code. Make sure you have installed following software to get started with TypeScript development.

- Node.js
- Visual Studio Code

Install TypeScript using NPM

To get started with TypeScript, just install Typescript using NPM by running following command.

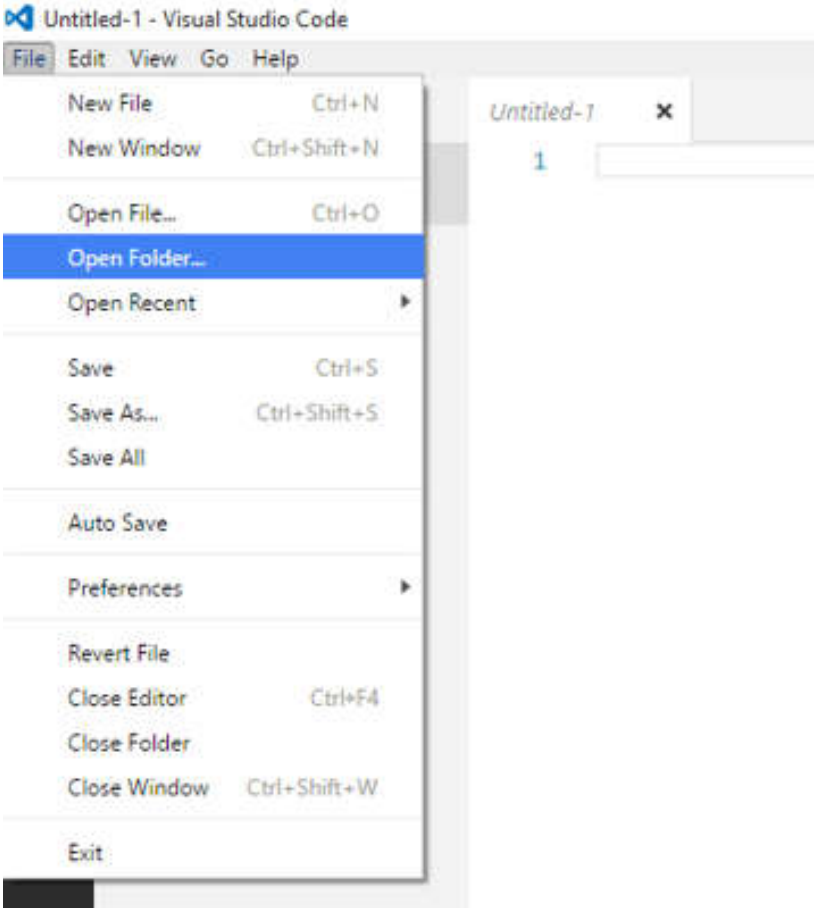
```
Command Prompt
```

```
npm install -g TypeScript
```

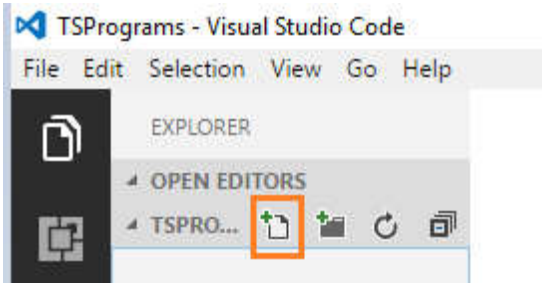
This command will install typescript as global package.

Open Visual Studio Code

So far, we have installed all the prerequisites to build first TypeScript program. Now open VS code and make an empty folder in your system named as TSPrograms. Open this folder with the help of VS code.



Create a new file named as *hello.ts* by clicking *New File* option in the MyProject row of the explore sidebar as given below:



Add following line of code as given below:

Hello.ts

```
class Program {
    constructor(private msg: string) {
    }
    showDetails() {
        console.log("Your message is : " + this.msg);
    }
}

let obj = new Program("Hello TypeScript");
obj.showDetails();
```

Setting TypeScript compilation option in Visual Studio Code

It's time to setup typescript compilation target as ES3 or ES5 or ES6 as per your target browsers. For this one you have to add *tsconfig.json* file and add the following lines of code.

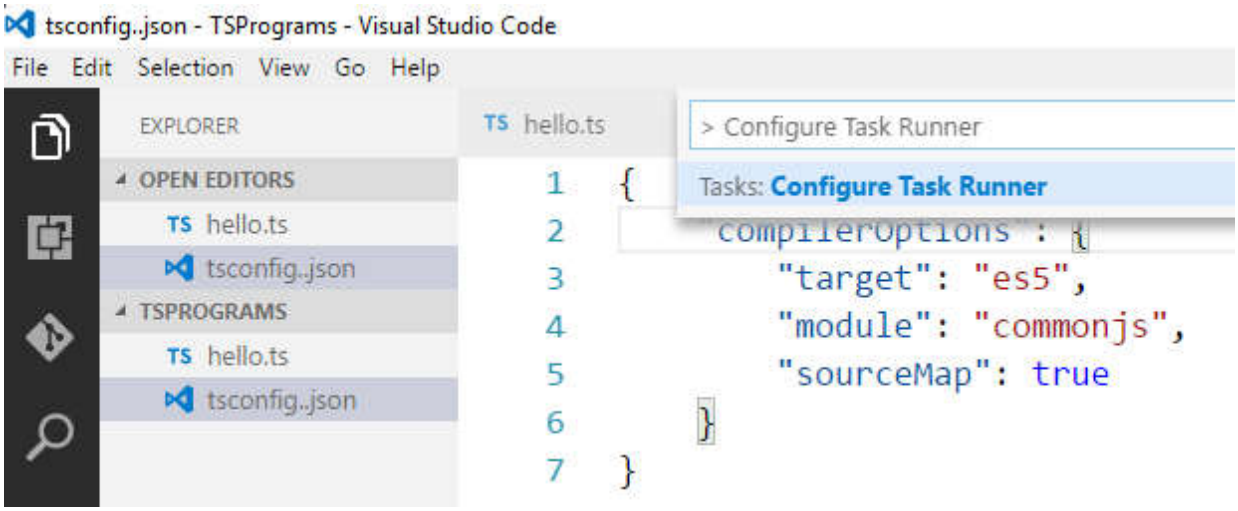
tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "sourceMap": true
    }
}
```

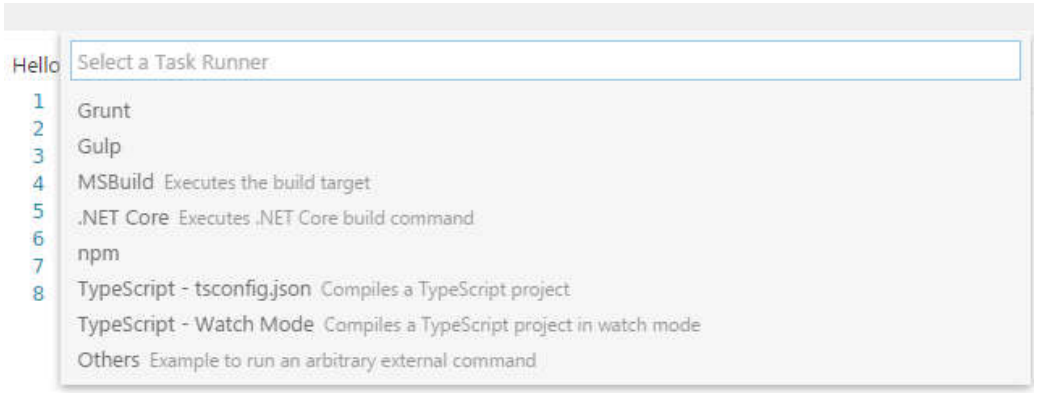
Creating Task to Transpile TypeScript into JavaScript

VS Code integrates with tsc (typescript compile) through integrated task runner. You can use this to transpile .ts files into .js files.

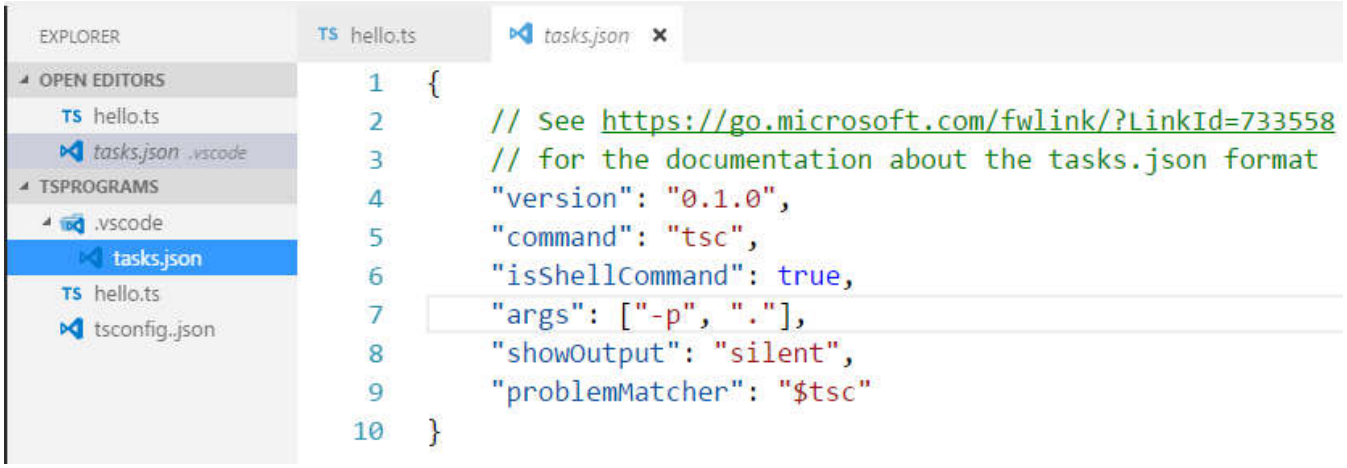
To create tasks.json, open the Command Palette with *Ctrl+Shift+P* and type *Configure Task Runner*, press Enter to select it.



This shows a selection box with templates you can choose from. Choose *TypeScript – tsconfig.json* option and press Enter.



It will create a *tasks.json* file in *.vscode* folder as given below:



It's time to execute the task. It can be executed by simply pressing Ctrl+Shift+B (Run Build Task). At this point you will see an additional file show up in the file list with name *hello.js*.

Running Code using Node

Now, you can run your code by opening up a terminal and running the following command.

```
Command Prompt
```

```
Node hello
```

For more help refer the given link https://goo.gl/PbJNsh

Important Information

- TypeScript never executed by browser, so you have to add reference of compiled JS file (Generated by TypeScript compiler) to your webpage.
- TypeScript code can be compiled as per ES3 standard to support older browser.
- TypeScript code can be compiled as per ES5 and ES6 standard to support latest browser.
- A piece of JavaScript code is a valid TypeScript code.

TypeScript Compilation

TypeScript is compiled into JavaScript and JavaScript is an implementation of the ECMAScript standard. Version 4 of the ECMAScript specification was abandoned, so technically it does not exist.

You can compile your TypeScript code into three versions of ECMAScript – ES3, ES5 and ES6. When you deal with browsers, you should know the browsers supports for all these ECMAScript standards.

ECMAScript 3 (ES3)

ECMAScript 3 standard was published in 1999 and it is widely supported by all the browsers available in market. If you are targeting ES3 compilation for your TypeScript file, limited number of TypeScript features are supported. But it offers the widest support for your code in all the browsers.

ECMAScript 5 (ES5)

ECMAScript 5 standard was published in Dec, 2009 and it is widely supported by all the latest browsers available in market. If you are targeting ES5 compilation for your TypeScript file, maximum number of TypeScript features are supported. ES5 is supported in the following browsers:

- Chrome 7 or higher
- FF 4 or higher
- IE 9 or higher
- Opera 12 or higher
- Safari 5.1 or higher

ECMAScript 6 (ES6)

ECMAScript 6 standard was published in June, 2015 and its features are only supported by the latest browsers. If you are targeting ES6 compilation for your TypeScript file, all the TypeScript features are supported but your code might not work in older browsers like IE7, IE8, IE9 and IE10. ES6 is supported in the following browsers:

- Chrome 30 or higher
- FF 20 or higher
- IE 11, Edge 12 or higher
- Opera 15 or higher
- Safari 7 or higher

For detailed analysis of ES5 and ES6 features support by the browsers, please refer the given link: https://goo.gl/YmLnMl

Types

Introduction

TypeScript is referred to as optional static type language which means you can ask the compiler to ignore the type of a variable if you want to take the advantage of dynamic type. This mixing of static and dynamic typing is also available in C# where you can define the static typed using primary datatype and secondary datatype and also you can define the dynamic type using the dynamic keyword.

Here, we will discuss about the TypeScript Types which you can use to write your TypeScript code.

Any Type

The Any type is a supertype of all types which represents any JavaScript value. You can assign any type of value to it. The **any** keyword is used to define Any type in TypeScript.

The Any type is useful, when we do not know the type of value (which might come from an API or a 3rd party library) and we want to skip the type-checking on compile time.

any.ts

```
function ProcessData(x: any, y: any) {
    return x + y;
}

let result: any;
result = ProcessData("Hello ", "Any!"); //Hello Any!
result = ProcessData(2, 3); //5
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

any.js

```
function ProcessData(x, y) {
    return x + y;
}
var result;
result = ProcessData("Hello ", "Any!"); //Hello Any!
result = ProcessData(2, 3);
```

Primitive Types

The primitive types in TypeScript are the Number, Boolean, String, Void, Null, Undefined types and Enum type.

Number

The Number primitive type is equivalent to JavaScript primitive type *number*, which represents double precision 64-bit floating-point values. The **number** keyword is used to define Number type in TypeScript. TypeScript supports decimal, hexadecimal, binary and octal literals.

number.ts

```
let nondecimal: number = 2;
let decimal: number = 2.4;
let hexadecimal: number = 0xf;
let binary: number = 0b100;
let octal: number = 0o7;
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

number.js

```
var nondecimal = 2;
var decimal = 2.4;
var hexadecimal = 0xf;
var binary = 4;
var octal = 7;
```

Boolean

The Boolen primitive type is equivalent to JavaScript primitive type *boolen* which accepts a value that is either *true or false*. The **boolean** keyword is used to define Boolean type in TypeScript.

boolen.ts

```
let yes: boolean = true;
let no: boolean = false;
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

boolen.js

```
var yes = true;
var no = false;
```

String

The String primitive type is equivalent to JavaScript primitive type *string* and represents the sequence of characters stored as Unicode UTF-16 code units. The **string** keyword is used to define String type in TypeScript.

Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

string.ts

```
let name: string = "Shailendra Chauhan";
let site: string = 'www.gurukulsight.com';
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

string.js

```
var name = "Shailendra Chauhan";
var site = 'www.gurukulsight.com';
```

Void

The Void primitive type represents the absence of a value. The **void** keyword is used to define Void type in TypeScript but it is not useful because you can only assign *null* or undefined values to it.

The void type is mostly used as a **function return type** that does not return a value or as a **type argument** for a generic class or function.

void.ts

```
function displayMeassge(): void {
    console.log("Welcome to Gurukulsight!");
}
let unusable: void = undefined;
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

void.js

```
function displayMeassge() {
    console.log("Welcome to Gurukulsight!");
}
var unusable = undefined;
```

Null

The Null primitive type is equivalent to JavaScript primitive type *null* which accepts the one and only value *null*. The **null** keyword is used to define Null type in TypeScript but it is not useful because you can only assign *null* value to it.

The Null type is a **subtype** of all types **except** the undefined type. Hence, you can assign *null* as a value to all primitive types, object types, union types, and type parameters.

null.ts

```
let num: number = null;
let bool: boolean = null;
let str: string = null;
let n: null = null; //not useful, since you can assign only null value
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

null.js

```
var num = null;
var bool = null;
var str = null;
var n = null;
```

Undefined

The Undefined primitive type is equivalent to JavaScript primitive type *undefined* and all **uninitialized variables** in TypeScript and JavaScript have one and only value that is *undefined*. The **undefined** keyword is used to define Undefined type in TypeScript but it is not useful because you can only assign undefined value to it.

The Undefined type is also a **subtype** of all types. Hence, you can assign *undefined* as a value to all primitive types, object types, union types, and type parameters.

undefined.ts

```
let num: number = undefined;
let bool: boolean = undefined;
let str: string = undefined;
let un: undefined = undefined; //not useful, since you can assign only undefined
value
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

undefined.js

```
var num = undefined;
var bool = undefined;
var str = undefined;
var n = undefined;
```

Symbols

A symbol is a new, primitive data type introduced in ES6 just like number and string. A symbol value is created by calling the Symbol constructor.

symbols1.ts

```
let s1 = Symbol();
let s2 = Symbol("mySymbol");
```

Symbols are immutable, and unique. When you create two symbols with the same description then they will be unique and immutable. Just like strings, symbols can be used as keys for object properties.

symbols2.js

```
let s1 = Symbol("mySymbol Description");
let s2 = Symbol("mySymbol Description");
s1 === s2; // false, symbols are unique
```

Here is the list of some built-in symbols:

1. **Symbol.hasInstance** - Used for determining whether an object is one of the constructor's instance.
2. **Symbol.match** - Used for matching the regular expression against a string.
3. **Symbol.iterator** - Returns the default iterator for an object and called by the for-of loop.
4. **Symbol.search** - Returns the index number within a string that matches the regular expression.
5. **Symbol.split** - Splits a string at the indices that match the regular expression.

Enum

An enum is a way to give friendly names to a set of numeric values. The **enum** keyword is used to define Enum type in TypeScript. By default, the value of enum's members start from 0. But you can change this by setting the value of one of its members.

enum.ts

```
enum Courses {TypeScript, Ionic, Angular2, NodeJS};
let tscourse: Courses = Courses.TypeScript;
console.log(tscourse); //0
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

enum.js

```
var Courses;
(function (Courses) {
    Courses[Courses["TypeScript"] = 0] = "TypeScript";
    Courses[Courses["Ionic"] = 1] = "Ionic";
    Courses[Courses["Angular2"] = 2] = "Angular2";
    Courses[Courses["NodeJS"] = 3] = "NodeJS";
})(Courses || (Courses = {}));
;
var tscourse = Courses.TypeScript;
console.log(tscourse); //0
```

In the previous example, we can start the enum's members value from 2 instead of 0.

enum.ts

```
enum Courses {TypeScript=2, Ionic, Angular2, NodeJS};
let tscourse: Courses = Courses.TypeScript;
console.log(tscourse); //2
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

enum.js

```
var Courses;
(function (Courses) {
    Courses[Courses["TypeScript"] = 2] = "TypeScript";
    Courses[Courses["Ionic"] = 3] = "Ionic";
    Courses[Courses["Angular2"] = 4] = "Angular2";
    Courses[Courses["NodeJS"] = 5] = "NodeJS";
})(Courses || (Courses = {}));
;
var tscourse = Courses.TypeScript;
console.log(tscourse); //2
```

An enum type can be assigned to the Number primitive type, and vice versa.

enum.ts

```
enum Courses { TypeScript = 2, Ionic, Angular2, NodeJS };
let tscourse: Courses = Courses.Ionic;
let c: number = tscourse; //3
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

enum.js

```
var Courses;
(function (Courses) {
    Courses[Courses["TypeScript"] = 2] = "TypeScript";
    Courses[Courses["Ionic"] = 3] = "Ionic";
    Courses[Courses["Angular2"] = 4] = "Angular2";
    Courses[Courses["NodeJS"] = 5] = "NodeJS";
})(Courses || (Courses = {}));
;
var tscourse = Courses.Ionic;
var c = tscourse;
```

Object Types

The primitive types in TypeScript are the Array, Tuple, Function, Class and Interface.

Array

The Array type is used to store same type of multiple values in a single variable and further your array elements you can access using the index number.

An Array type can be defined using square brackets '[ ]' followed by *elemType* or generic array type 'Array<elemType>' as given below:

array.ts

```
let numberList: number[] = [1, 2, 3];
//OR
let numList: Array<number> = [1, 2, 3];
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

array.js

```
var numberList = [1, 2, 3];
var numList = [1, 2, 3];
```

Tuple

The Tuple type represents a JavaScript array where you can define the datatype of each element in array.

tuple.ts

```
let tuple: [string, number];
tuple = ["Gurukulsight", 1];
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

tuple.js

```
var tuple;
tuple = ["Gurukulsight", 1];
```

Function

Like JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. In TypeScript, you can add types to each of the parameter and also add a return type to function.

function.ts

```
//named function with number as return type
function add(x: number, y: number): number {
    return x + y;
}
//anonymous function with number as return type
let sum = function (x: number, y: number): number {
    return x + y;
};
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

function.js

```
function add(x, y) {
    return x + y;
}
var sum = function (x, y) {
    return x + y;
};
```

Class

ECMAScript 6 or ES6 provides class type to build JavaScript application by using object-oriented class-based approach. In TypeScript, you can compile class type down to JavaScript standard ES5 that will work across all major browsers and platforms.

class.ts

```
class Student {
    rollNo: number;
    name: string;

    constructor(rollNo: number, name: string) {
        this.rollNo = rollNo;
        this.name = name;
    }
    showDetails() {
        return this.rollNo + " : " + this.name;
    }
}
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

class.js

```
var Student = (function () {
    function Student(rollNo, name) {
        this.rollNo = rollNo;
        this.name = name;
    }
    Student.prototype.showDetails = function () {
        return this.rollNo + " : " + this.name;
    };
    return Student;
}());
```

Interface

Interface acts as a contract between itself and any class which implements it. Interface cannot be instantiated but it can be referenced by the class object which implements it.

Hence, it can be used to represent any non-primitive JavaScript object.

interface.ts

```
interface IHuman {
    firstName: string;
    lastName: string;
}
class Employee implements IHuman {
    constructor(public firstName: string, public lastName: string) {
    }
}
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

interface.js

```
var Employee = (function () {
    function Employee(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    return Employee;
}());
```

Template Strings

JavaScript uses double quotes (") or single quotes (') to surround string data. String data within double or single quotes can only be in one line and there is no way to insert data into these strings. This results into a lot of string concatenation.

strings

```
let firstname: string = "Shailendra";
let site: string = 'Gurukulsight';

//string concatenation
let str = 'Hello, my name is ' + firstname + ' and I am the founder of ' + site;
console.log(str);
```

To solve these issues ES6 introduces a new type of string literal that is marked with back ticks (`). By using back ticks (`) you can manipulate strings easily.

template_strings

```
let firstname: string = "Shailendra";
let site: string = 'Gurukulsight';

//template string
let str = `Hello, my name is ${firstname}
           and I am the founder of ${site}`;
console.log(str);
```

Type Annotation

Type Annotations means defining the type of a variable explicitly. The type used to specify an annotation can be a primitive type, an object type or any complex structure to represent data.
The type of a variable is defined by using colon (:) followed by type.

Syntax

```
<variable_declaration> <identifier> : <type_annotation> = <value>;
<variable_declaration> <identifier> : <type_annotation>;
```

type_annotation.ts

```
let num: number = 2; //type of num is number
let str: string = "Gurukulsight"; //type of str is string
let numberList: number[] = [1, 2, 3]; //type of numberList is number array
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

type_annotation.js

```
var num = 2;
var str = "Gurukulsight";
var numberList = [1, 2, 3];
```

Type Inference

Type inference is a mechanism to determine the type of data at compile time in the absence of explicit type annotations. TypeScript will infer the type from the initial value.

type_inference.ts

```
let myString = 'Hello Gurukulsight'; // string
let myBool = true; // boolean
let myNumber = 1.23; // number
let myArray = ['Hello', 'Gurukulsight']; // string[]
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

type_inference.js

```
var myString = 'Hello Gurukulsight'; // string
var myBool = true; // boolean
var myNumber = 1.23; // number
var myArray = ['Hello', 'Gurukulsight']; // string[]
```

Type Assertion

A type assertion is just like a type casting, but it doesn't perform special type checking or restructuring of data just like other languages like C# and Java. This process entirely works on compile time. Hence it has no impact on runtime.

Type assertions can be done using two ways:

1. **Angle-bracket syntax**

anglebracket.ts

```
let anyValue: any = "Welcome to www.gurukulsight.com!";
let strLength: number = (<string>anyValue).length;
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

anglebracket.js

```
var anyValue = "Welcome to www.gurukulsight.com!";
var strLength = anyValue.length;
```

2. **As syntax**

anglebracket.ts

```
let anyValue: any = "Welcome to www.gurukulsight.com!";
let strLength: number = (<string>anyValue).length;
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

anglebracket.js

```
var anyValue = "Welcome to www.gurukulsight.com!";
var strLength = anyValue.length;
```

Statements

Introduction

TypeScript is a superset of JavaScript which provides static type checking for JavaScript. TypeScript uses JavaScript syntax to define statements.

Here, we will discuss about the TypeScript statements which you can use to write your TypeScript code.

Variable Statement

In TypeScript, a variable statement is extended to include optional type annotations.

Syntax

```
<variable_declaration> <identifier> : <type_annotation> = <value>;
<variable_declaration> <identifier> : <type_annotation>;
```

Variable Declaration using var, let and const

Before ES6, JavaScript has only *var* keyword to define a variable. Now, ES6 provides let and const keywords to declare a variable in JavaScript.

- **var** - It supports only function scope. It doesn't support block scope.
- **let** - It is similar to *var* but it supports block scope. Unlike var you cannot redeclare it twice in the same scope.
- **const** - It is similar to *let* but its value cannot be changed. Also, at the time of declaration you have to assign value to it.

variable.js

```
var x = 4;
let number = 50;
const key = 'xyz123';

//OR
var x: number = 4;
let number: number = 50;
const key: string = 'xyz123';
```

Block Scoping

let supports block scope. When you define, a variable using let within a block like if statement, for loop, switch statement etc., block-scoped variable is not visible outside of its containing block.

block.js

```
function foo(flag: boolean) {
    let a = 100;

    //if block
    if (flag) {
        // 'a' exists here
        let b = a + 1;
        return b;
    }
    // Error: 'b' doesn't exist here
    return b;
}
```

Hoisting

When you use *var* to declare a variable, you have to deal with hoisting. Hoisting is JavaScript's default behavior of **moving** variable declarations to the **top** of the **current scope** (i.e. top of the current js file or the current function). It means, in JavaScript a **variable** can be **used before** it has been **declared**.

hoisting.js

```
var x = "Gurukulsight";
function foo() {
    console.log(x); //Gurukulsight
}

function bar() {
    console.log(x); //undefined
    var x; //x declaration is hoisted, that's why x is undefined
}
foo();
bar();
```

In above bar function declaration of x variable will move to the top of bar function variable declaration and within bar function, we have not specified any value to x, hence it's default value is *undefined*.

Important Information

- JavaScript only hoists variable declarations, not variable initializations.
- To avoid issues because of hoisting, always declare all variables at the beginning of every scope.
- Use ES6 - let and const to avoid hoisting.

hoistingfixed.js

```
var x = "Gurukulsight";
function bar() {
    console.log(x); //Error: x is not defined
    let x; //x declaration is not hoisted
}
bar();
```

Destructuring

Destructuring means breaking up the structure. Destructuring is a way to quickly extract the data from a single object or array and assign it to multiple variables within a single statement. ES6 also supports object destructuring.

TypeScript supports the following forms of Destructuring:

1. Object Destructuring
2. Array Destructuring

destructuring.ts

```
let list = ['one', 'two', 'three'];
let [x, y, z] = list; //destructuring the array values into three variables x,y,z

console.log(x); // 'one'
console.log(y); // 'two'
console.log(z); // 'three'

let obj = {x: 'one', y: 'two', z: 'three'};
let {x, y, z} = obj; // destructuring the object properties x, y, z

console.log(x); // 'one'
console.log(y); // 'two'
console.log(z); // 'three'
```

Destructuring can also be used for passing objects into a function which allow you to extract specific properties from an object.

destructuring_auguments.ts

```
function greet({firstName, lastName}): void {
    console.log(`Hello, ${firstName} ${lastName}!`);
}

let p1 = { firstName: 'Shailendra', lastName: 'Chauhan' };
let p2 = { firstName: 'Kanishk', lastName: 'Puri' };

greet(p1) // -> Hello, Shailendra Chauhan!
greet(p2) // -> Hello, Kanishk Puri!
```

If, do..while, while and for Statements

In TypeScript, if, do..while, while and for statements you can use in same way as you use in JavaScript.

statements.ts

```
if (true) {
    console.log("if statement");
}

let i: number = 1;
do {
    console.log("do..while loop");
    i++;
} while (i <= 1);

let j: number = 1;
while (j <= 1) {
    console.log("while loop");
    j++;
}

for (let k: number = 1; k < 4; k++) {
    console.log(k);
}
```

For..In Statement

The for...in loop iterates over the properties of an iterable object i.e. list type. The for...in loop returns a list of keys on the object being iterated and using the keys on that object you can access values.

forin.ts

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

for (let i in list) {
    console.log(i); //keys: "0", "1", "2"
    console.log(list[i]); //values: "4", "5", "6"
}
```

For..Of Statement

The for...of loop iterates over the properties of an iterable object i.e. list type. The for...of loop returns a list of values on the object being iterated.

forin.ts

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

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

Operators

TypeScript supports all of the standard JavaScript operators like arithmetic operators, comparison operators, assignment operators, conditional operators, bitwise operators, logical operators and Type Operators.

Functions

Introduction

TypeScript extends JavaScript functions with typed parameters, return type annotations, overloads, default parameter values, and rest parameters. Like JavaScript, TypeScript functions can be created both as a named function or as an anonymous function.

function.ts

```
//named function with number as parameters type and return type
function add(x: number, y: number): number {
    return x + y;
}
//anonymous function with number as parameters type and return type
let sum = function (x: number, y: number): number {
    return x + y;
};
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

function.js

```
function add(x, y) {
    return x + y;
}
var sum = function (x, y) {
    return x + y;
};
```

Optional Parameters

In JavaScript, you can call a function without passing any arguments, even the function specifies parameters. Hence every parameter in JavaScript function is optional and when you do this, each parameter value is undefined.

In TypeScript, functions parameters are not optional. The compiler checks each call and warns you if you are not passing the values as per the function receiving parameters type.

TypeScript supports **optional parameter** by suffixing question mark '? ' to the parameter. Also, optional parameter you can add after any required parameters in the parameter list.

optionalparameter.ts

```
//here, z is optional parameter added after required parameters x and y
function add(x: number, y: number, z?: number): number {
    if (z !== undefined)
        return x + y + z;
    else
        return x + y;
}

let resul1 = add(2, 3); //5
let resul2 = add(2, 3, 5); //10
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

optionalparameter.js

```
function add(x, y, z) {
    if (z !== undefined)
        return x + y + z;
    else
        return x + y;
}
var resul1 = add(2, 3); //5
var resul2 = add(2, 3, 5); //10
```


Default Parameters

In TypeScript, you can set a default value to a function parameter and when user does not pass the value for that parameter, default value will be used for that parameter. Also, default parameter you can add after any required parameters in the parameter list.

defaultparameter.ts

```
//here, z is default parameter added after required parameters x and y
function add(x: number, y: number, z: number = 0): number {
    return x + y + z;
}

let resul1 = add(2, 3); //5
let resul2 = add(2, 3, 5); //10
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

defaultparameter.js

```
function add(x, y, z) {
    if (z === void 0) { z = 0; }
    return x + y + z;
}
var resul1 = add(2, 3); //5
var resul2 = add(2, 3, 5); //10
```

Rest Parameters

The Rest parameter allow you to pass zero or more values to a function. A Rest parameter is prefixed by three consecutive dot characters '...' and allow the functions to have a variable number of arguments without using the arguments object. The rest parameter is an instance of Array, so all array methods work.

A rest parameter must follow following three rules:

1. Only one rest parameter is allowed.
2. The rest parameter type must be an array type.
3. The rest parameter must be the last parameter in the parameter list.

restparameter.ts

```
function add(x: number, ...y: number[]): number {
    let result = x;
    for (var i = 0; i < y.length; i++) {
        result += y[i];
    }
    return result;
}

let result1 = add(2, 5); //7
let result2 = add(2, 5, 7, 2); //16
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

restparameter.js

```
function add(x) {
    var y = [];
    for (var _i = 1; _i < arguments.length; _i++) {
        y[_i - 1] = arguments[_i];
    }
    var result = x;
    for (var i = 0; i < y.length; i++) {
        result += y[i];
```

```
    }
    return result;
}
var result1 = add(2, 5); //7
var result2 = add(2, 5, 7, 2); //16
```

Spread Operator

The spread operator is introduced with ES6. It allows you to expand an array into multiple formal parameters. In ES6, the spread operator example is given below:

spreadoperator_es6.js

```
function add(x, y, z) {
    return x + y + z;
}

let nums = [2, 5, 5];
let result = add(...nums); //12
```

In TypeScript, the spread operator acts like a reverse of rest parameter. The spread operator in TypeScript you can use with rest parameter as given below:

spreadoperator.ts

```
function add(...x: number[]): number {
    let result = 0;
    for (var i = 0; i < x.length; i++) {
        result += x[i];
    }
    return result;
}

let nums: number[] = [2, 5, 5];
let result = add(...nums); //12
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

spreadoperator.js

```
function add() {
    var x = [];
    for (var _i = 0; _i < arguments.length; _i++) {
        x[_i - 0] = arguments[_i];
    }
```

```
    var result = 0;
    for (var i = 0; i < x.length; i++) {
        result += x[i];
    }
    return result;
}
var nums = [2, 5, 5];
var result = add.apply(void 0, nums); //12
```

Function Overloads

In TypeScript, Function overloads is purely a compile-time process. It has no impact on the compiled JavaScript code.

The parameter list of a function overload **cannot** have **default parameters**. But you can define **optional parameter** using question mark '? ' in function overloads.

functionoverloads.ts

```
function add(x: string, y: string, z: string): string;
function add(x: number, y: number, z: number): number;

// implementation signature
function add(x: any, y: any, z: any): any {
    let result: any;
    if (typeof x == "number" && typeof y == "number" && typeof z == "number") {
        result = x + y + z;
    }
    else {
        result = x + y + " " + z;
    }
    return result;
}

let result1 = add(4, 3, 8); // 15
let result2 = add("Gurukul", "sight", "website"); //Gurukulsight website
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

functionoverloads.js

```
function add(x, y, z) {
    var result;
    if (typeof x == "number" && typeof y == "number" && typeof z == "number") {
        result = x + y + z;
    }
```

```
    else {
        result = x + y + " " + z;
    }
    return result;
}
var result1 = add(4, 3, 8); // 15
var result2 = add("Gurukul", "sight", "website"); //Gurukulsight website
```

Arrow Function

ES6 provides shorthand syntax for defining anonymous function. Arrow function omit the function keyword and have lexical scoping of *this* keyword. TypeScript extends arrow function with types parameters.

arrowfunction.ts

```
//arrow function with typed parameters
let add = (x: number, y: number)=> {
    return x + y;
};

let result = add(2, 3); //5
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

arrowfunction.js

```
var add = function (x, y) {
    return x + y;
};
var result = add(2, 3);
```

The arrow function is typically useful for writing callbacks, which often have an undefined or unexpected *this*. The use of an arrow function causes the callback to have the same *this* as the surrounding method have. Let's understand the concept with the following example.

arrowcallback.ts

```
class Messenger {
    message: string = "Hello Gurukulsight!";
    greetArrow(): void {
       //arrow function as a callback
       setTimeout(() => console.log(this.message), 2000); //contains parent scope
    }
    greetAnonymous(): void {
```

```
        //anonymous function as a callback
        setTimeout(function () {
             console.log(this.message); // doesn't contain parent scope
         }, 3000);
    }
};

let m1 = new Messenger();
m1.greetArrow(); //Hello Gurukulsight!
m1.greetAnonymous(); //undefined
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

arrowcallback.js

```
var Messenger = (function () {
    function Messenger() {
        this.message = "Hello Gurukusight!";
    }
    Messenger.prototype.greetArrow = function () {
        var _this = this;
        setTimeout(function () { return console.log(_this.message); }, 2000);
    };
    Messenger.prototype.greetAnonymous = function () {
        setTimeout(function () {
            console.log(this.message);
        }, 3000);
    };
    return Messenger;
}());

var m1 = new Messenger();
m1.greetArrow();
m1.greetAnonymous();
```

Classes

Introduction

ECMAScript 6 or ES6 provides class type to build JavaScript application by using object-oriented class-based approach. TypeScript extends ES6 class type with typed members and access modifiers like classes in C# or Java programming languages.

In TypeScript, you can compile class type down to JavaScript standard ES5 that will work across all major browsers and platforms.

class.ts

```
class Student {
    private rollNo: number;
    private name: string;

    constructor(_rollNo: number, _name: string) {
        this.rollNo = _rollNo;
        this.name = _name;
    }
    showDetails() { //public : by default
        console.log(this.rollNo + " : " + this.name);
    }
}

let s1 = new Student(1, "Shailendra Chauhan");
s1.showDetails(); //1 : Shailendra Chauhan

let s2 = new Student(2, "kanishk Puri");
s2.showDetails(); //2 : kanishk Puri
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

class.js

```
var Student = (function () {
    function Student(_rollNo, _name) {
        this.rollNo = _rollNo;
        this.name = _name;
    }
    Student.prototype.showDetails = function () {
```

```
        console.log(this.rollNo + " : " + this.name);
    };
    return Student;
}());
var s1 = new Student(1, "Shailendra Chauhan");
s1.showDetails();

var s2 = new Student(2, "kanishk Puri");
s2.showDetails();
```

Constructors

Just like object oriented programming languages C# or Java, TypeScript class type supports two types of constructors - default constructor and parameterized constructor.

Unlike C# or Java, in typescript constructor, you can make public, private or protected instance members of a class.

constructor.ts

```
class Customer {
 //instance members with access modifiers
 constructor(private id:number, public name:string, protected address:string) { }
    showDetails() {
        console.log(this.id + " : " + this.name + " : " + this.address);
    }
}

let c1 = new Customer(1, "Shailendra Chauhan", "Noida");
c1.showDetails(); //1 : Shailendra Chauhan : Noida
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

constructor.js

```
var Customer = (function () {
    //instance members with access modifiers
    function Customer(id, name, address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
    Customer.prototype.showDetails = function () {
        console.log(this.id + " : " + this.name + " : " + this.address);
    };
    return Customer;
```

```
}());
var c1 = new Customer(1, "Shailendra Chauhan", "Noida");
c1.showDetails();
```

Instance and Static Members

TypeScript class members are either instance members or static members.

Instance members are members of the class type and its instance. Within constructors, member functions and accessors, *this* represents to the instance of the class.

Static members are declared using the static keyword and are the members of constructor function type. Within static functions and static accessors, *this* represents to the constructor function type.

Static members are accessible by using the class name only, not by using the class instance. Inside static method, you can access only static members.

members.ts

```
class Booklist {
    //instance members
    private books: string[] = []; //instance property or instance member
    constructor(public name: string) {
    }
    addBook(book: string) { //member function or instance member function
        if (this.books.length >= Booklist.maxBookCount) {
            throw new Error('Booklist is full');
        }
        else {
            this.books.push(book);
            Booklist.totalBooksCount++;
        }
    }
    //static members
    static totalBooksCount: number = 0;
    static maxBookCount: number = 20;
    static totalBooks() { //static methods
        return Booklist.totalBooksCount;
    }
}

let booklist = new Booklist('My Book List');

// acessing instance members using class instance
let listName = booklist.name;
booklist.addBook("Gurukulsight - TypeScript Training Book");
booklist.addBook("Gurukulsight - Angular2 Training Book");
```

```
// accessing static members using class name
let maxBooks = Booklist.maxBookCount; //20
let totalBooks = Booklist.totalBooks(); //2

console.log(maxBooks);
console.log(totalBooks);
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

members.js

```
var Booklist = (function () {
    function Booklist(name) {
        this.name = name;
        this.books = [];
    }
    Booklist.prototype.addBook = function (book) {
        if (this.books.length >= Booklist.maxBookCount) {
            throw new Error('Booklist is full');
        }
        else {
            this.books.push(book);
            Booklist.totalBooksCount++;
        }
    };
    Booklist.totalBooks = function () {
        return Booklist.totalBooksCount;
    };
    return Booklist;
}());

Booklist.totalBooksCount = 0;
Booklist.maxBookCount = 20;
var booklist = new Booklist('My Book List');

var listName = booklist.name;
booklist.addBook("Gurukulsight - TypeScript Training Book");
booklist.addBook("Gurukulsight - Angular2 Training Book");

var maxBooks = Booklist.maxBookCount;
var totalBooks = Booklist.totalBooks();
console.log(maxBooks);
console.log(totalBooks);
```

Access Modifiers

TypeScript supports three access modifiers - public, private, and protected.

- **Public** - By default, members (properties and methods) of TypeScript class are public - so you don't need to prefix members with the public keyword. Public members are accessible everywhere without restrictions
- **Private** - A private member cannot be accessed outside of its containing class. Private members can be accessed only within the class.
- **Protected** - A protected member cannot be accessed outside of its containing class. Protected members can be accessed only within the class and by the instance of its sub/child class.

In compiled JavaScript code, there will be no such types of restriction on the members.

accessmodifiers.ts

```
class Foo {
    private x: number;
    protected y: number;
    public z: number;
    saveData(foo: Foo): void {
        this.x = 1; // ok
        this.y = 1; // ok
        this.z = 1; // ok

        foo.x = 1; // ok
        foo.y = 1; // ok
        foo.z = 1; // ok
    }
}

class Bar extends Foo {
    getData(foo: Foo, bar: Bar) {
        this.y = 1; // ok
        this.z = 1; // ok

        bar.y = 1; // ok
        bar.z = 1; // ok
        foo.z = 1; // ok

        foo.x = 1; // Error, x only accessible within A
        bar.x = 1; // Error, x only accessible within A
        bar.y = 1; // Error, y only accessible through instance of B
    }
}
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

accessmodifiers.js

```
var Foo = (function () {
    function Foo() {
    }
    Foo.prototype.saveData = function (foo) {
        this.x = 1;
        this.y = 1;
        this.z = 1;
        foo.x = 1;
        foo.y = 1;
        foo.z = 1;
    };
    return Foo;
}());
var Bar = (function (_super) {
    __extends(Bar, _super);
    function Bar() {
        return _super.apply(this, arguments) || this;
    }
    Bar.prototype.getData = function (foo, bar) {
        this.y = 1;
        this.z = 1;
        bar.y = 1;
        bar.z = 1;
        foo.z = 1;
        foo.x = 1;
        bar.x = 1;
        bar.y = 1;
    };
    return Bar;
}(Foo));
```

Readonly Modifiers

TypeScript supports readonly modifiers on property level by using the readonly keyword. The Readonly properties must be initialized at their declaration or in the constructor.

readonly.ts

```
class Company {
    readonly country: string = "India";
    readonly name: string;

    constructor(_name: string) {
```

```
        this.name = _name;
    }
    showDetails() {
        console.log(this.name + " : " + this.country);
    }
}

let c1 = new Company("Dot Net Tricks Innovation");
c1.showDetails(); // Dot Net Tricks Innovation : India

c1.name = "TCS"; //Error, name can be initialized only within constructor
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

readonly.js

```
var Company = (function () {
    function Company(_name) {
        this.country = "India";
        this.name = _name;
    }
    Company.prototype.showDetails = function () {
        console.log(this.name + " : " + this.country);
    };
    return Company;
}());
var c1 = new Company("Dot Net Tricks Innovation");
c1.showDetails();
c1.name = "TCS";
```

Accessors

Just like C# properties accessors, TypeScript supports *get/set* accessors to **access** and **to set** the value to a member of an object. This way give us control over how a member of an object is accessed and set.

accessors.ts

```
class Employee {
    private passcode: string;
    private _fullName: string;

    constructor(_passcode?: string) {
        this.passcode = _passcode;
    }
    get fullName(): string { //accessing value
        return this._fullName;
```

```
    }
    set fullName(newName: string) { //setting value
        if (this.passcode == "secret_passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized update of employee!");
        }
    }
}

let e1 = new Employee("secret_passcode");
e1.fullName = "Shailendra Chauhan";
if (e1.fullName) {
    console.log(e1.fullName);
}

let e2 = new Employee();
e2.fullName = "Kanishk Puri"; //Error: Unauthorized update of employee
if (e2.fullName) {
    console.log(e1.fullName);
}
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

accessors.js

```
var Employee = (function () {
    function Employee(_passcode) {
        this.passcode = _passcode;
    }
    Object.defineProperty(Employee.prototype, "fullName", {
        get: function () {
            return this._fullName;
        },
        set: function (newName) {
            if (this.passcode == "secret_passcode") {
                this._fullName = newName;
            }
            else {
                console.log("Error: Unauthorized update of employee!");
            }
        },
        enumerable: true,
        configurable: true
    });
    return Employee;
}());
```

```
var e1 = new Employee("secret_passcode");
e1.fullName = "Shailendra Chauhan";
if (e1.fullName) {
    console.log(e1.fullName);
}
var e2 = new Employee();
e2.fullName = "Kanishk Puri"; //Error: Unauthorized update of employee
if (e2.fullName) {
    console.log(e1.fullName);
}
```

Inheritance

Inheritance is an important aspect of object oriented programming languages like C# and Java. Inheritance is a mechanism of acquiring the features and behaviors of a class by another class. The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived/child class**. Further, in child class you can override or modify the behaviors of its parent class and also can add new one.

Traditional JavaScript uses functions and prototype-based inheritance but TypeScript **supports** object-oriented **class-based inheritance**. Hence, in TypeScript you can extend the features and behaviors of an existing class into new one through the *extends* keyword.

Like C# and Java languages, TypeScript supports only **single inheritance** and **multi-level inheritance**. It **doesn't support multiple** and **hybrid** inheritance.

inheritance.ts

```
class Person {
    private firstName: string;
    private lastName: string;
    constructor(_firstName: string, _lastName: string) {
        this.firstName = _firstName;
        this.lastName = _lastName;
    }
    fullName(): string {
        return this.firstName + " " + this.lastName;
    }
}

class Employee extends Person {
    id: number;
    constructor(_id: number, _firstName: string, _lastName: string) {
        //calling parent class constructor
        super(_firstName, _lastName);
```

```
        this.id = _id;
    }
    showDetails(): void {
        //calling parent class method
        console.log(this.id + " : " + this.fullName());
    }
}

let e1 = new Employee(1, "Shailendra", "Chauhan");
e1.showDetails(); //1 : Shailendra Chauhan
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

inheritance.js

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype,
new __());
};
var Person = (function () {
    function Person(_firstName, _lastName) {
        this.firstName = _firstName;
        this.lastName = _lastName;
    }
    Person.prototype.fullName = function () {
        return this.firstName + " " + this.lastName;
    };
    return Person;
}());
var Employee = (function (_super) {
    __extends(Employee, _super);
    function Employee(_id, _firstName, _lastName) {
        var _this =
        _super.call(this, _firstName, _lastName) || this;
        _this.id = _id;
        return _this;
    }
    Employee.prototype.showDetails = function () {
        console.log(this.id + " : " + this.fullName());
    };
    return Employee;
}(Person));
var e1 = new Employee(1, "Shailendra", "Chauhan");
e1.showDetails();
```

Abstract Class

An Abstract class is a special type of class which cannot be instantiated and acts as a base class for other classes. Abstract class members marked as abstract must be implemented by derived classes.

The purpose of an abstract class is to provide basic or default functionality as well as common functionality that multiple derived classes can share and modify.

Just like C# or Java, in TypeScript *abstract* keyword is used to define *abstract classes* as well as *abstract methods* within an *abstract class*. An abstract class cannot be *instantiated* directly. Unlike an interface, an abstract class may contain *non-abstract* members.

abstract.ts

```
abstract class Shapes {
    // must be implemented in derived classes
    abstract Area(): number;
}
class Square extends Shapes {
    side: number = 0;
    constructor(n: number) {
        super(); //mandatory to call parent class constructor
        this.side = n;
    }
    // implemented Area method
    Area(): number {
        return this.side * this.side;
    }
}
class Rectangle extends Shapes {
    length: number = 0;
    width: number = 0;
    constructor(length: number, width: number) {
        super(); //mandatory to call parent class constructor
        this.length = length;
        this.width = width;
    }
    // implemented Area method
    Area(): number {
        return this.length * this.width;
    }
}
let s = new Square(4);
s.Area(); //16

let r = new Rectangle(4, 3);
s.Area(); //12
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

abstract.js

```
var __extends = (this && this.__extends) || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype,
new __());
};
var Shapes = (function () {
    function Shapes() {
    }
    return Shapes;
}());
var Square = (function (_super) {
    __extends(Square, _super);
    function Square(n) {
        var _this = _super.call(this) || this;
        _this.side = 0;
        _this.side = n;
        return _this;
    }
    Square.prototype.Area = function () {
        return this.side * this.side;
    };
    return Square;
}(Shapes));
var Rectangle = (function (_super) {
    __extends(Rectangle, _super);
    function Rectangle(length, width) {
        var _this = _super.call(this) || this;
        _this.length = 0;
        _this.width = 0;
        _this.length = length;
        _this.width = width;
        return _this;
    }
    Rectangle.prototype.Area = function () {
        return this.length * this.width;
    };
    return Rectangle;
}(Shapes));
var s = new Square(4);
s.Area();
var r = new Rectangle(4, 3);
s.Area();
```

Interfaces

Introduction

Interface acts as a contract between itself and any class which implements it. It means a class that implement an interface is bound to implement all its members. Interface cannot be instantiated but it can be referenced by the class object which implements it. Interfaces can be used to represent any non-primitive JavaScript object.

interface.ts

```
interface IHuman {
    firstName: string;
    lastName: string;
}
class Employee implements IHuman {
    constructor(public firstName: string, public lastName: string) {

    }
}
```

The compiled JavaScript (ES5) code for the above TypeScript code is give below:

interface.js

```
var Employee = (function () {
    function Employee(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    return Employee;
}());
```

Use of Interfaces

Interfaces are particularly **useful** for **validating** the required structure of properties, objects passed as parameters, and objects returned from functions.

Also, Interfaces are only TypeScript compile-time construct and compiled JavaScript code have no such representation.

Interface Inheritance

An interface can be inherited from zero or more base types. The **base type** can be a **class** or **interface**.

Let's understand the interface inheritance with the following examples:

Interface1.ts

```
interface IStore {
    Read(): void;
    Write(): void;
}
interface ICompress {
    Compress(): void;
    Decompress(): void;
}

interface IDocument extends IStore, ICompress {
    Print(): void;
}
```

Class Implementing Interfaces

Just like C# and Java, a TypeScript class can implement multiple interfaces.

Interface2.ts

```
interface IStore {
    Read(): void;
    Write(): void;
}
interface ICompress {
    Compress(): void;
    Decompress(): void;
}

class DocStore implements IStore, ICompress {
    Read(): void {
        console.log("Read Method for IStore");
    }
    Write(): void {
        console.log("Write Method for IStore");
    }
    Compress(): void {
        console.log("Compress Method for ICompress");
    }
```

```
    Decompress(): void {
        console.log("Decompress Method for ICompress");
    }
}

let I1: IStore = new DocStore();
//can access only IStore members
I1.Read();
I1.Write();

let I2: ICompress = new DocStore();
//can access only ICompress members
I2.Compress();
I2.Decompress();
```

Interface Extending Class

Unlike C# or Java, TypeScript interfaces can inherit (extend) classes. When an interface extends a class, type it inherits the members of the class but not their implementations i.e. the members' declaration is available in interface. Also, anything added to the class will also be added to the interface.

Interface.ts

```
class Store {
    Read(): void {
        console.log("Read Method for Store");
    }
    Write(): void {
        console.log("Write Method for Store");
    }
}

interface IDocStore extends Store {
   //inherited Read and Write methods with declaration only
}
class Doc implements IDocStore {
    //mandatory to implement Read and Write methods of IDocStore
    Read(): void {
        console.log("Read Method for Doc");
    }
    Write(): void {
        console.log("Write Method for Doc");
    }
}
let I1: IDocStore = new Doc();
I1.Read();
I1.Write();
```

Generics

Introduction

Generics enforce type safety without compromising performance, or productivity. In generics, a type parameter is supplied between the open (<) and close (>) brackets and which makes it strongly typed collections i.e. generics collections contain only similar types of objects.

In TypeScript, you can create generic functions, generic methods, generic interfaces and generic classes.

generic.ts

```
function doReverse<T>(list: T[]): T[] {
    let revList: T[] = [];
    for (let i = (list.length - 1); i >= 0; i--) {
        revList.push(list[i]);
    }
    return revList;
}
let letters = ['a', 'b', 'c', 'd', 'e'];
let reversedLetters = doReverse<string>(letters); // e, d, c, b, a

let numbers = [1, 2, 3, 4, 5];
let reversedNumbers = doReverse<number>(numbers); // 5, 4, 3, 2, 1
```

Generic Functions

When a function implementation includes type parameters in its signature then it is called a generic function. A generic function has a type parameter enclosed in angle brackets (< >) immediately after the function name.

The following is an example of generic function:

genericfunction.ts

```
function doReverse<T>(list: T[]): T[] {
    let revList: T[] = [];
    for (let i = (list.length - 1); i >= 0; i--) {
        revList.push(list[i]);
    }
```

```
    return revList;
}
let letters = ['a', 'b', 'c', 'd', 'e'];
let reversedLetters = doReverse<string>(letters); // e, d, c, b, a

let numbers = [1, 2, 3, 4, 5];
let reversedNumbers = doReverse<number>(numbers); // 5, 4, 3, 2, 1
```

Generics are purely compile-time representation; compiled JavaScript code does not have such representation. The compiled JavaScript (ES5) code for the above TypeScript code is give below:

genericfunction.js

```
function doReverse(list) {
    var revList = [];
    for (var i = (list.length - 1); i >= 0; i--) {
        revList.push(list[i]);
    }
    return revList;
}
var letters = ['a', 'b', 'c', 'd', 'e'];
var reversedLetters = doReverse(letters);

var numbers = [1, 2, 3, 4, 5];
var reversedNumbers = doReverse(numbers);
```

Generic Classes

A generic class has a type parameter enclosed in angle brackets (< >) immediately after the class name. In generic class, same type parameter can be used to annotate method parameters, properties, return types, and local variables.

The following is an example of generic class:

genericclass.ts

```
class ItemList<T>
{
    private itemArray: Array<T>;
    constructor() {
        this.itemArray = [];
    }

    Add(item: T) : void {
        this.itemArray.push(item);
    }
```

```
    GetAll(): Array<T> {
        return this.itemArray;
    }
}
let fruits = new ItemList<string>();
fruits.Add("Apple");
fruits.Add("Mango");
fruits.Add("Orange");

let listOfFruits = fruits.GetAll();
for (let i = 0; i < listOfFruits.length; i++) {
    console.log(listOfFruits[i]);
}
/* -- Output ---
Apple
Mango
Orange
*/
```

Generics are purely compile-time representation; compiled JavaScript code does not have such representation. The compiled JavaScript (ES5) code for the above TypeScript code is give below:

genericclass.js

```
var ItemList = (function () {
    function ItemList() {
        this.itemArray = [];
    }
    ItemList.prototype.Add = function (item) {
        this.itemArray.push(item);
    };
    ItemList.prototype.GetAll = function () {
        return this.itemArray;
    };
    return ItemList;
}());
var fruits = new ItemList();
fruits.Add("Apple");
fruits.Add("Mango");
fruits.Add("Orange");
var listOfFruits = fruits.GetAll();
for (var i = 0; i < listOfFruits.length; i++) {
    console.log(listOfFruits[i]);
}
```

Generic Interfaces

A generic interface has a type parameter enclosed in angle brackets (< >) immediately after the interface name. In generic interface, same type parameter can be used to annotate method parameters, properties, return types, and local variables.

The following is an example of generic interface:

genericinterface.ts

```
interface IItemList<T> {
    itemArray: Array<T>;
    Add(item: T): void;
    GetAll(): Array<T>;
}

class ItemList<T> implements IItemList<T>
{
    itemArray: Array<T>;
    constructor() {
        this.itemArray = [];
    }

    Add(item: T): void {
        this.itemArray.push(item);
    }
    GetAll(): Array<T> {
        return this.itemArray;
    }
}

let fruits = new ItemList<string>();
fruits.Add("Apple");
fruits.Add("Mango");
fruits.Add("Orange");

let listOfFruits = fruits.GetAll();
for (let i = 0; i < listOfFruits.length; i++) {
    console.log(listOfFruits[i]);
}
/* -- Output ---
Apple
Mango
Orange
*/
```

Generics are purely compile-time representation; compiled JavaScript code does not have such representation. The compiled JavaScript (ES5) code for the above TypeScript code is give below:

genericinterface.js

```
var ItemList = (function () {
    function ItemList() {
        this.itemArray = [];
    }
    ItemList.prototype.Add = function (item) {
        this.itemArray.push(item);
    };
    ItemList.prototype.GetAll = function () {
        return this.itemArray;
    };
    return ItemList;
}());
var fruits = new ItemList();
fruits.Add("Apple");
fruits.Add("Mango");
fruits.Add("Orange");
var listOfFruits = fruits.GetAll();
for (var i = 0; i < listOfFruits.length; i++) {
    console.log(listOfFruits[i]);
}
```

Modules and Namespaces

Introduction

A module is a container to a group of related variables, functions, classes, and interfaces etc. Variables, functions, classes, and interfaces etc. declared in a module are not accessible outside the module unless they are explicitly exported using *export* keyword. Also, to consume the members of a module, you have to import it using *import* keyword.

Modules are declarative and the relationship between modules is specified in terms of imports and exports at the file level. Modules are also available in ES6/ECMAScript 2015.

Important Information

> From TypeScript 1.5, "Internal modules" are "namespaces" and "External modules" are simply "modules" to match ES6 module terminology

Export

In TypeScript, you can export any declaration such as a variable, function, class, type alias, or interface by using export keyword.

myModule.ts

```
//exporting Employee type
export class Employee {
    constructor(private firstName: string, private lastName: string) { }
    showDetails() {
        return this.firstName + ", " + this.lastName;
    }
}

//exporting Student type
export class Student {
    constructor(private rollNo: number, private name: string) { }
    showDetails() {
        return this.rollNo + ", " + this.name;
    }
}
```

Import

In TypeScript, you can import an exported declaration by using import keyword. Let's import the myModule file types within app file as given below:

app.ts

```
//importing the exporting types Student and Employee from myModule file
import { Student, Employee } from "./myModule";

let st = new Student(1, "Mohan");
let result1 = st.showDetails();
console.log("Student Details :" + result1);

let emp = new Employee("Shailendra", "Chauhan");
let result2 = emp.showDetails();
console.log("Employee Details :" + result2);
```

Re-Export

In TypeScript, sometimes modules extend other modules, and partially expose some of their features. In this case, you can re-export some of their features either using their original name or introducing a new name. Let's re-export the myModule file only Student type as show below:

re_export.ts

```
//re-exporting types Student as st from myModule file
export { Student as st } from "./myModule";

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

Now, you can import the types form re_export file as given below:

app.ts

```
//importing the exporting types from reexport file
import { st as Student, numberRegexp } from "./reExport"; importing st as Student

let st = new Student(1, "Mohan");
let result1 = st.showDetails();
console.log("Student Details :" + result1);
```

Namespaces

Namespaces are simply named JavaScript objects in the global scope. Namespaces are used for grouping of variables, functions, objects, classes, and interfaces, so that you can avoid the naming collisions. Namespaces are declared using the *namespace* keyword.

namespace.ts

```
namespace Gurukulsight {
    //exporting outside the namespace body
    export class Student {
        constructor(private rollNo: number, private name: string) { }
        showDetails() {
            return this.rollNo + ", " + this.name;
        }
    }
    // Only available inside the namespace body
    let maxCount: number = 100;
    class Employee {
        constructor(private firstName: string, private lastName: string) { }
        showDetails() {
            return this.firstName + ", " + this.lastName;
        }
    }
}
namespace DotNetTricks {
    //accessing Gurukulsight namespace student class
    import Student = Gurukulsight.Student;

    export class Person {
        constructor(private firstName: string, private lastName: string) { }
        fullName(): string {
            return this.firstName + " " + this.lastName;
        }
    }
    //creating object of student class
    let st = new Student(1, "Mohan");
    st.showDetails();
}
```

Important Information

- A namespace can be described in multiple files and allow to keep each file to a maintainable size.
- The members of a module body, you can access only within that module body.
- To make a member available outside the namespace body, you need to prefix the member with the export keyword.

Export and Import Namespaces

In TypeScript, you can export a namespace by prefixing export keyword and to use its members use *import* keyword. Also, to make a member available outside the namespace body, you need to prefix that member with the export keyword.

Let's understand the namespace import and export with the help of following example.

gurukulsight.ts

```
export namespace Gurukulsight {
    //exporting outside the namespace body
    export class Student {
        constructor(private rollNo: number, private name: string) { }
        showDetails() {
            return this.rollNo + ", " + this.name;
        }
    }
    // Only available inside the namespace body
    let maxCount: number = 100;
    class Employee {
        constructor(private firstName: string, private lastName: string) { }
        showDetails() {
            return this.firstName + ", " + this.lastName;
        }
    }
}
```

Now, let's import the Gurukulsight namespace as given below:

main.ts

```
//importing the namespace
import { Gurukulsight } from "./namespace";

//accessing Gurukulsight namespace student class
import Student = Gurukulsight.Student;

//creating object of student class
let st = new Student(1, "Mohan");
st.showDetails();
```

Decorators

Introduction

Decorators are simply functions that are prefixed @ symbol, and can be attach to a class declaration, method, accessor, property, or parameter. The decorators supply information about the class, method, property or parameters.

Decorators are proposed for future version of JavaScript and they are available as an experimental feature of TypeScript.

There are following types of decorators, you can use with TypeScript:

1. Class Decorators
2. Method Decorators
3. Accessor Decorators
4. Property Decorators
5. Parameter Decorators

Class Decorator

A Class Decorator is attached to a class declaration and tell about the class behaviors. The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition.

classdecorator.ts

```
@sealed
class Employee {
    constructor(private firstName: string, private lastName: string) { }
    showDetails() {
        return this.firstName + ", " + this.lastName;
    }
}
```

In above example, @sealed decorator will seal both the constructor and its prototype so that you cannot inherit the class.

Method Decorator

A method Decorator is attached to the methods of a class.

methoddecorator.ts

```
class ItemList {
    itemArray: Array<string>;
    constructor() {
        this.itemArray = [];
    }
    @log
    Add(item: string): void {
        this.itemArray.push(item);
    }
    GetAll(): Array<string> {
        return this.itemArray;
    }
}
```

In above example, @log decorator will log the new item entry.

Property Decorator

A property Decorator is attached to the properties of a class.

propertydecorator.ts

```
class Company {
    @ReadOnly
    name: string = "Dot Net Tricks";
}

let company = new Company();
t.name = 'TCS'; // you can't change it's name
console.log(t.name); // 'Dot Net Tricks'
```

In above example, @ReadOnly decorator will make the name property as readonly, so you can't change its value.

Important Information

This book has been written by referring to official website of TypeScript at www.typescriptlang.org.