Contents

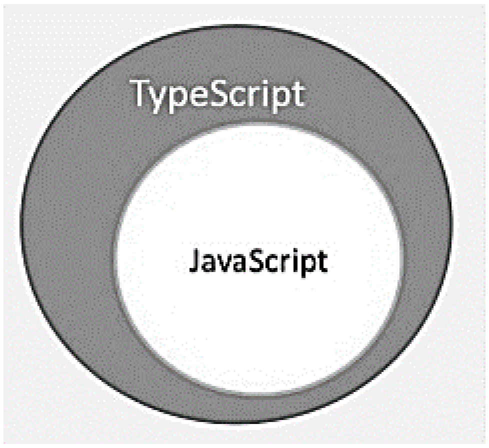
TypeScript – Overview

JavaScript was introduced as a language for the client side. The development of Node.js has marked JavaScript as an emerging server-side technology too. However, as JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse the code. Moreover, its failure to embrace the features of Object Orientation, strong type checking and compile-time error checks prevents JavaScript from succeeding at the enterprise level as a full-fledged server-side technology. TypeScript was presented to bridge this gap.

What is TypeScript?

By definition, "TypeScript is JavaScript for application-scale development."

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 starts with JavaScript and ends with 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.

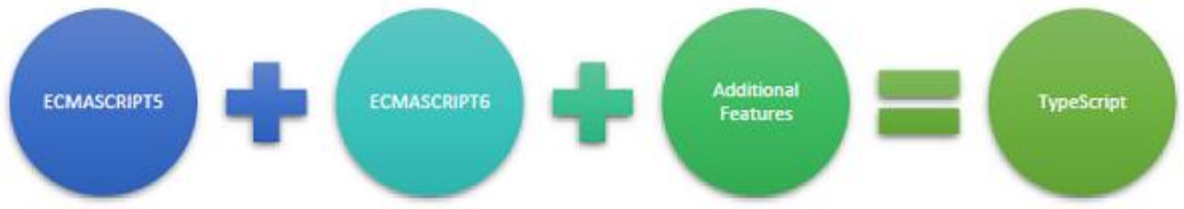
TypeScript supports other JS libraries. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

JavaScript is TypeScript. This means that any valid .js file can be renamed to .ts and compiled with other TypeScript files.

TypeScript is portable. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn't need a dedicated VM or a specific runtime environment to execute.

TypeScript and ECMAScript

The ECMAScript specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published. Version 6 of the standard is codenamed "Harmony". TypeScript is aligned with the ECMAScript6 specification.



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.

TypeScript - Environment Setup

Typescript is an Open Source technology. It can run on any browser, any host, and any OS. You will need the following tools to write and test a Typescript program −

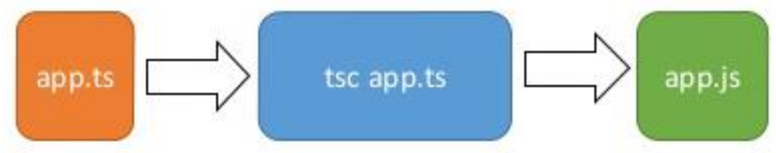
A Text Editor

The text editor helps you to write your source code. Examples of a few editors include Windows/Mac are Visual Code (My Favorite), Notepad, Notepad++, Emacs, vim or vi, etc. Editors used may vary with Operating Systems and user preference.

The source files are typically named with the extension .ts

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 (transcompiler / transpiler).



The TSC generates a JavaScript version of the .ts file passed to it. In other words, the TSC produces an equivalent JavaScript source code from the Typescript file given as an input to it. This process is termed as transpilation.

However, the compiler rejects any raw JavaScript file passed to it. The compiler deals with only .ts or .d.ts files.

Installing Node.js

Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code. You may download Node.js source code or a pre-built installer for your platform. Node is available here – https://nodejs.org/en/

Get TypeScript

The command-line TypeScript compiler can be installed as a Node.js package.

INSTALL

```
npm install -g typescript
```

TypeScript - Basic Syntax

Syntax defines a set of rules for writing programs. Every language specification defines its own syntax. A TypeScript program is composed of –

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

Your First TypeScript Code

Let us start with the traditional "Hello World" example –

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

On compiling, it will generate following JavaScript code.

```
var message = "Hello World";
console.log(message);
```

- Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.
- Line 2 prints the variable's value to the prompt. Here, console refers to the terminal window. The function log () is used to display text on the screen.

Compile and Execute a TypeScript Program

Let us see how to compile and execute a TypeScript program using Visual Studio Code. Follow the steps given below –

To compile the file use the following command on the terminal window.

```
tsc Test.ts
```

The file is compiled to Test.js. To run the program written, type the following in the terminal.

```
node Test.js
```

Compiler Flags

Compiler flags enable you to change the behavior of the compiler during compilation. Each compiler flag exposes a setting that allows you to change how the compiler behaves.

The following table lists some common flags associated with the TSC compiler. A typical command-line usage uses some or all switches.

S.No.	Compiler flag & Description
1.	--help Displays the help manual
2.	--module Load external modules
3.	--target Set the target ECMA version

4.	--declaration Generates an additional .d.ts file
5.	--removeComments Removes all comments from the output file
6.	--out Compile multiple files into a single output file
7.	--sourcemap Generate a sourcemap (.map) files
8.	--module noImplicitAny Disallows the compiler from inferring the any type
9.	--watch Watch for file changes and recompile them on the fly

Note − Multiple files can be compiled at once.

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

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.

The following tables lists a few examples of valid and invalid identifiers –

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

TypeScript – Keywords

Keywords have a special meaning in the context of a language. The following table lists some keywords in TypeScript.

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.

TypeScript is Case-sensitive

TypeScript is case-sensitive. This means that TypeScript differentiates between uppercase and lowercase characters.

Semicolons are optional

Each line of instruction is called a **statement**. Semicolons are optional in TypeScript.

Example

```
console.log("hello world")
console.log("We are learning TypeScript")
```

A single line can contain multiple statements. However, these statements must be separated by a semicolon.

Comments in TypeScript

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like author of the code, hints about a function/ construct etc. Comments are ignored by the compiler.

TypeScript supports the following types of comments −

- **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.

Example

```
//this is single line comment

/* This is a
   Multi-line comment
*/
```

TypeScript and Object Orientation

TypeScript is Object-Oriented JavaScript. Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation considers a program as a collection of objects that communicate with each other via mechanism called methods. TypeScript supports these object oriented components too.

- **Object** − An object is a real time representation of any entity. According to Grady Brooch, 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.

Example: TypeScript and Object Orientation

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

The above example defines a class Greeting. The class has a method greet (). The method prints the string "Hello World" on the terminal. The new keyword creates an object of the class (obj). The object invokes the method greet ().

On compiling, it will generate following JavaScript code.

```
//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()
```

The output of the above program is given below −

```
Hello World!!!
```

TypeScript - Variables

A variable, by definition, is "a named space in the memory" that stores values. In other words, it acts as a container for values in a program. TypeScript variables must follow the JavaScript naming rules −

- 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.

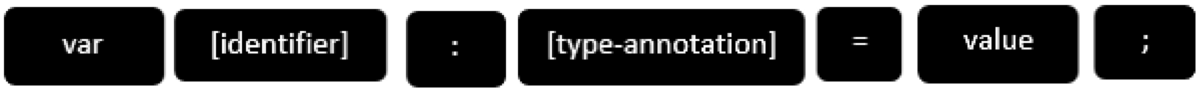
A variable must be declared before it is used. Use the **var** keyword to declare variables.

Variable Declaration in TypeScript

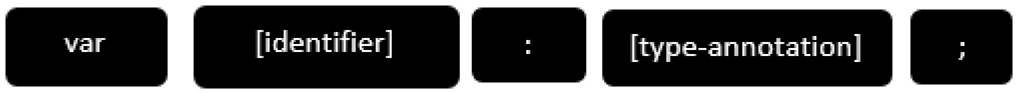
The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the **var** keyword to declare a variable.

When you declare a variable, you have four options −

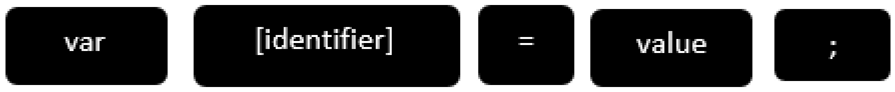
- Declare its type and value in one statement.



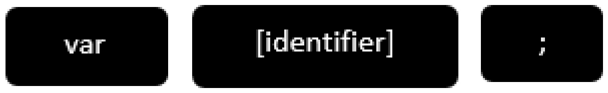
- Declare its type but no value. In this case, the variable will be set to undefined.



- Declare its value but no type. The variable type will be set to the data type of the assigned value.



- Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.



The following table illustrates the valid syntax for variable declaration as discussed above −

S.No.	Variable Declaration Syntax & Description
1.	**var name:string = ”mary”** The variable stores a value of type string
2.	**var name:string;** The variable is a string variable. The variable’s value is set to undefined by default
3.	**var name = ”mary”** The variable’s type is inferred from the data type of the value. Here, the variable is of the type string

4.	**var name;** The variable's data type is any. Its value is set to undefined by default.

Example: Variables in TypeScript

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

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var name = "John";
var score1 = 50;
var score2 = 42.50;
var sum = score1 + score2;
console.log("name" + name);
console.log("first score: " + score1);
console.log("second score : " + score2);
console.log("sum of the scores: " + sum);
```

The output of the above program is given below −

```
name:John
first score:50
second score:42.50
sum of the scores:92.50
```

The TypeScript compiler will generate errors, if we attempt to assign a value to a variable that is not of the same type. Hence, TypeScript follows Strong Typing. The Strong typing syntax ensures that the types specified on either side of the assignment operator (=) are the same. This is why the following code will result in a compilation error −

```
var num:number = "hello"     // will result in a compilation error
```

Type Assertion in TypeScript

TypeScript allows changing a variable from one type to another. TypeScript refers to this process as *Type Assertion*. The syntax is to put the target type between < > symbols and place it in front of the variable or expression. The following example explains this concept −

Example

```
var str = '1'
var str2:number = <number> <any> str   //str is now of type number
console.log(typeof(str2))
```

If you hover the mouse pointer over the type assertion statement in Visual Studio Code, it displays the change in the variable's data type. Basically it allows the assertion from type S to T succeed if either S is a subtype of T or T is a subtype of S.

The reason why it's not called "type casting" is that casting generally implies some sort of runtime support while, "type assertions" are purely a compile time construct and a way for you to provide hints to the compiler on how you want your code to be analyzed.

On compiling, it will generate following JavaScript code.

```
"use strict";
var str = '1';
var str2 = str;
console.log(typeof (str2)); //output string
```

Inferred Typing in TypeScript

Given the fact that, Typescript is strongly typed, this feature is optional. TypeScript also encourages dynamic typing of variables. This means that, TypeScript encourages declaring a variable without a type. In such cases, the compiler will determine the type of the variable on the basis of the value assigned to it. TypeScript will find the first usage of the variable within the code, determine the type to which it has been initially set and then assume the same type for this variable in the rest of your code block.

The same is explained in the following code snippet −

Example: Inferred Typing

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

In the above code snippet −

- The code declares a variable and sets its value to 2. Note that the variable declaration doesn't specify the data type. Hence, the program uses inferred typing to determine the data type of the variable, i.e., it assigns the type of the first value that the variable is set to. In this case, **num** is set to the type number.
- When the code tries to set the variable's value to string. The compiler throws an error as the variable's type is already set to number.

It will produce the following output −

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

TypeScript Variable Scope

The scope of a variable specifies where the variable is defined. The availability of a variable within a program is determined by its scope. TypeScript variables can be of the following scopes −

- **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.

The following example illustrates variable scopes in TypeScript.

Example: Variable Scope

```
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)
```

On transpiling, the following JavaScript code is generated −

```
var global_num = 12;              //global variable
var Numbers = (function () {
   function Numbers() {
      this.num_val = 13;          //class variable
   }
   Numbers.prototype.storeNum = function () {
      var local_num = 14;        //local variable
   };
   Numbers.sval = 10;            //static field
   return Numbers;
}());

console.log("Global num: " + global_num);
console.log(Numbers.sval);       //static variable
var obj = new Numbers();
```

```
console.log("Global num: " + obj.num_val);
```

It will produce the following output −

```
Global num: 12
10
Global num: 13
```

If you try accessing the local variable outside the method, it results in a compilation error.

```
error TS2095: Could not find symbol 'local_num'.
```

ECMAScript (ES6, ES7) //Most commonly used

- `let and const`
- `Template literals`
- `Desctructuring`
- `Spread operator`
- `Arrow Functions using =>`
- `Classes`

Declaring Variables with let instead of var

var framework = 'React';
var framework = 'Angular'
console.log(framework); //Output is Angular as the last variable declared was assigned this value

This is very dangerous and might drive the code to an incorrect output. Other programming language like C, Java, and C# do not allow this behavior. With ES6, a new keyword is introduced, called **let.**
Example**:**

let framework = 'React';
let framework = 'Angular'; // throws error
console.log(framework); //Output is Angular as the last variable declared was assigned this value

Constants

ES6 introduced the keyword ***const***. Its behavior is the same thing as the keyword ***let***, the only difference is that the variable defined by ***const*** has a read-only value, meaning a constant value. Example:

const PI = 3.141593;
PI = 3.0; //throws error
Console.log(PI);

Template Literals

Template literals are really very nice because we can create strings without the need to concatenate the values.

Example:

```
        var book = {
            name: 'Learning JavaScript DataStructures and Algorithms'
        };
        Console.log(‘You are reading ‘ + book.name + ‘.,\n and this is a new line \n and so is this’);
```

We can improve the above syntax of the console.log output with the following code

```
        console.log(`You are reading ${book.name}. ,       //variable interpolation and no need to use /n
anymore
                    and this is a new line
                    and so is this.`);
```

Arrow functions

Arrow functions are great way of simplifying the syntax of function in ES6. Consider the following example:

```
        var circleArea = function circleArea(r) {
                var PI = 3.14;
                var area = PI * r * r;
                return area;
        };
        Console.log(circleArea(2));
```

Simplified version using arrow function is

```
        var circleArea = (r) => {
                var PI = 3.14;
                var area = PI * r * r;
                return area;
        };
        let circleArea2 = (r) => 3.14 * r * r; //Further simplified
```

Default Parameter values for functions

```
        function sum (x = 1, y = 2, z = 3) {
                return x + y + z
        };
        console.log(sum(4,2)); //outputs 9
```

Spread operator

```
var a = [1, 23, 12, 9, 30, 2, 50];

var b = [3,3,...a] ; //b= [3, 3, 1, 23, 12, 9, 30, 2, 50]
```

Generics

Introduction

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, which is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

Hello World of Generics

```
function identity(arg: number): number {
    return arg;
}
```

Or, we could describe the identity function using the **any** type:

```
function identity(arg: any): any {
    return arg;
}
```

While using any is certainly generic in that it will cause the function to accept **any** and all types for the type of **arg**, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a type variable, a special kind of variable that works on types rather than values.

```
function identity<T>(arg: T): T {
    return arg;
}
```

We've now added a type variable **T** to the identity function.

Boostrap React and SPFx in Demo Project

References

- Learning JavaScript Data Structures and Algorithms, Second Edition ISBN 978-1-78528-549-3
- https://www.typescriptlang.org/(TypeScript – JavaScript that scales)
- https://reactjs.org/ (React - A JavaScript library for building user interfaces)
- https://docs.microsoft.com/en-us/sharepoint/dev/spfx/sharepoint-framework-overview (Overview of the SharePoint Framework)