
# TypeScript

## Beginner to Advanced

**Author**

Rupesh Kahane

C#Corner

## About The Author

**Rupesh Kahane** works as a Senior Software Engineer in MNC company located at Pune (India). He did a master's in computers science and currently having more than 6+ years' experience in Web Application development. He is a four-time C# Corner MVP. He has working knowledge of technologies like ASP.NET MVC 5.0, C#, Angular, TypeScript, Azure, SQL Server, Entity Framework, LINQ, Kendo UI, Telerik Controls, TDD etc. He had conducted more than 120 hands on Seminars, Workshops for College, Schools and Training Institutes. His main passion is learning new technologies, sharing knowledge and traveling.

## Contents

# Chapter 1: Introduction to TypeScript

TypeScript is a modern Web development programming language created by Anders Hejlsberg at Microsoft. The first version of TypeScript, v0.8, was launched in October 2012. TypeScript 0.9 was released in 2013. The current version of TypeScript is 3.8. You can download the latest version from https://www.typescriptlang.org.

This chapter is an introduction to the TypeScript programming language. In this chapter, we will cover the following:

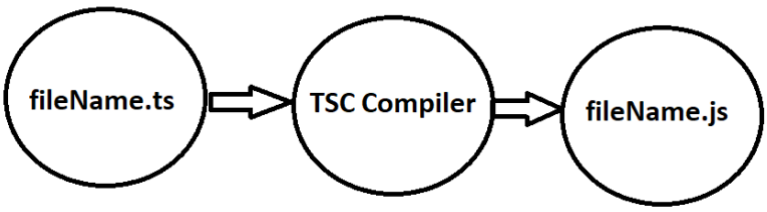
1. What is TypeScript
2. Why TypeScript
3. TypeScript Installation
4. TypeScript Basics
5. TypeScript and Object-Oriented

## 1. What is TypeScript?

TypeScript is a superset of JavaScript; i.e., it includes all the elements of JavaScript & some other features as well. TypeScript is an open-source programming language that is developed and maintained by Microsoft. When the user trans compiles the TypeScript code, it is converted into JavaScript. The trans compiler, also known as a source-to-source compiler, is a special type of compiler that converts the source code of a program into another language.

TypeScript is designed to build large scale Web applications. The extension of the TypeScript file is fileName.ts. It supports object-oriented features, static typing, type inference, cross platforms & Intellisense. The latest TypeScript version, 3.8, is available now.

The below image shows the TypeScript compilation process.



## 2. Why TypeScript?

There are several reasons why one should use TypeScript.

- It is an open-source programming language.
- It supports cross-platform development.
- Optional static typing means it detects the errors while writing the script.
- Typing mistakes/errors show immediately so we can check the types easily without debugging.
- Features of object-oriented programming language are supported in TypeScript.
- Intellisense is the best advantage that gives us ideas while coding.
- The code is reusable & easy to understand.
- Advanced features are supported like Classes, Generics, Optional parameters to use while creating function & modules support is easily available.

**Example:**

Consider the below code written in JavaScript.

JavaScript is not strict about type checking while writing our code, which means we can assign any type of value to our variable.

In the code below for our variable X, we are able to add any values to a variable.

```
<script>
  var X = "Welcome !!!";
  console.log(X);
  X = 200;
  console.log(X);
  X = [10,20,30];
  console.log(X);
  X = false
  console.log(X);
</script>
```

Now, we will run the above code in JavaScript & print the data on the console.

Welcome !!!

200

(3) [10, 20, 30]

false

Now if we write the code in TypeScript then we will see how strict type checking works.

Consider the below example written in TypeScript.

```
let X: string;
X = 'Welcome !!!'
console.log(X);
X = 200;
```

We will get an error message when we are trying to assign a number to a string type as follows, this is strict type checking.

```
let X: string
Type '200' is not assignable to type 'string'. (2322)
```

## 3. TypeScript Installation

Different ways to install TypeScript are as follows

1. By using npm command - Node.js package manager
2. By using Visual Studio plugin

**1. By using npm command:**

**What is Node.js?**

It was developed in 2009 and it is the server-side run environment used in server-side applications. It is open-source, uses JavaScript and is built on Chrome's V8 JavaScript engine. In single-page applications, this is widely used by Node.js. By using Node.js developers can do CRUD operations easily.

To download & install the latest version of Node.js use link https://nodejs.org/en/download/

After installation developers can check the version information using the command

node -v

```
G:\Web Development Fundamentals\TypeScript>node -v
v10.15.1

G:\Web Development Fundamentals\TypeScript>
```

Now install TypeScript package using below command

npm install -g typescript

```
G:\Web Development Fundamentals\TypeScript>npm install -g typescript
```

After successful installation, if you would like to know the version information of TypeScript or would like to check if the installation is done properly or not then use the below command

tsc -v

**2. By using Visual Studio plugin**

When you are using Visual Studio 2019, and then download the latest TypeScript packages using NuGet package manager, it should be associated with the version itself.

Update is also available in the NuGet package manager.

When you are using Visual Studio 2017 then download the latest TypeScript SDK from the below site for Visual Studio 2017 https://www.microsoft.com/en-us/download/details.aspx?id=55258

**By using Visual Studio Code**

Download the Visual Studio Code set up from https://code.visualstudio.com and install it. Now open the appropriate folder in Visual Studio code & add a new file by giving the and extension as .ts

**By using Mac OS X & Linux**

Download the Visual Studio Code set up from https://code.visualstudio.com/Docs/editor/setup

&https://code.visualstudio.com/Docs/editor/setup

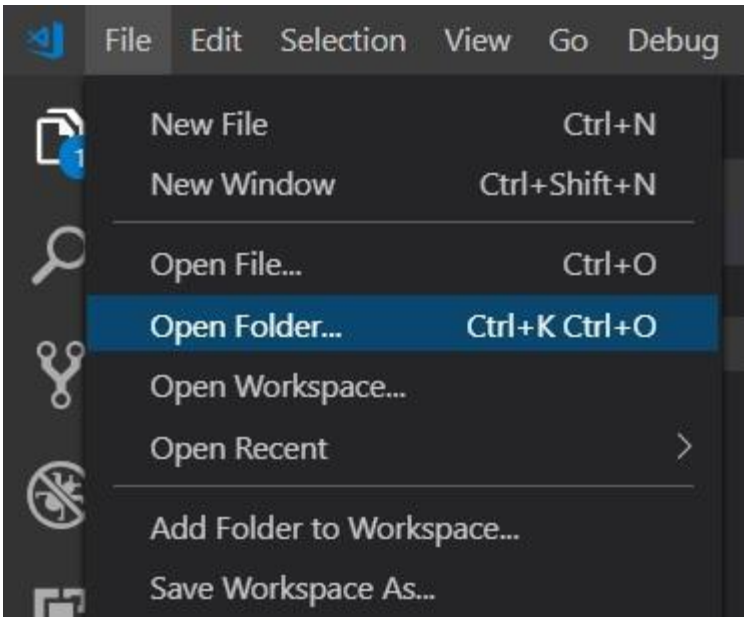
## 4. Basic Syntaxes

The below topics will cover

1. Variable Declaration
2. Function calls using Object-Oriented Concepts
3. Comment
4. Compile options

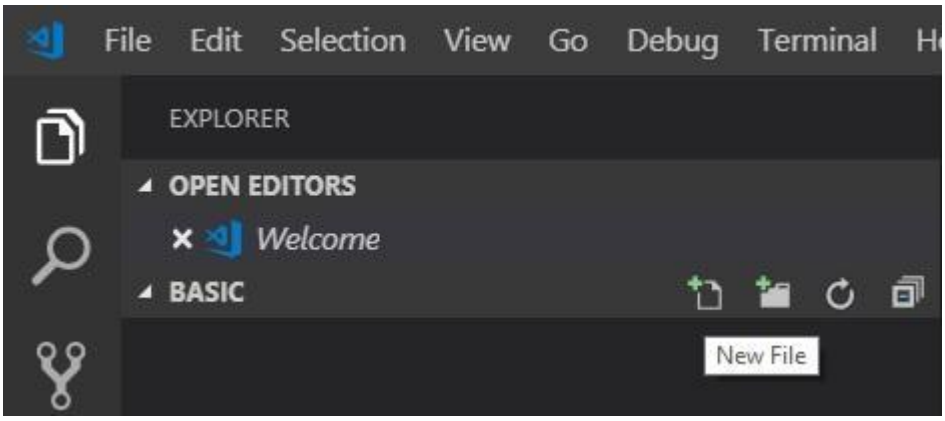
**Step 1**

I am using Visual Studio Code. Open folder from the file



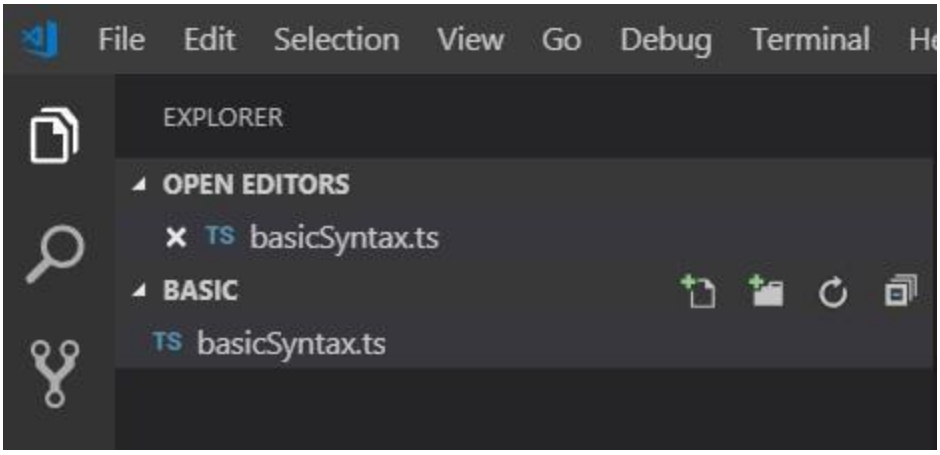

**Step 2**

Create a new file & give the name to the file with extension as .ts (TypeScript)



**Step 3**

Here I am creating a TypeScript file with the name basicSyntax.ts




**Variable declaration:**

[keyword] [variable_Name] : [Type] = [Optional_Value];

Here

keyword: We can use let/var keyword depending upon the scope of our variable.

Variable_Name:  Unique variable name.

Type: Data type of variable like a number, string, Boolean, etc.

Optional_value: Value to be assigned  is optional, we can assign the value later as well.

**Example:**

Now write the below code to declare a variable. We know the syntax as

```
let firstNumber: number= 501;

console.log(firstNumber);
```

Now to run the above code, open terminal & use the below commands

tsc basicSyntax.ts

```
PS G:\Web Development Fundamentals\TypeScript\basics> tsc basicSyntax.ts
```

After execution of the above command TypeScript compiler will convert this file into a .js file. Here typescript compiler trans-compiles the file.

To execute the code, use the below command

node basicSyntax.js

We will see the output in the console.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL                    1: powershell
PS G:\Web Development Fundamentals\TypeScript\basics> tsc basicSyntax.ts
PS G:\Web Development Fundamentals\TypeScript\basics> node basicSyntax.js
501
PS G:\Web Development Fundamentals\TypeScript\basics>
```

We can **explicitly** declare type syntax of variable as

```
let firstNumber: number = 501;
```

**Warnings:**

```
let firstNumber: number= 501;

firstNumber = 'Welcome';
```

Now if you are trying to assign the string value to our variable then it will give us the error / warning as => Type '"Welcome"' is not assignable to type 'number'.

```
let firstNumber: number
Type '"Welcome"' is not assignable to type 'number'. ts(2322)
```

## 5. TypeScript and Object-Oriented

TypeScript is an object-oriented programming language that revolves around classes and objects.

A class is a blueprint of newly-created things. Blueprint means that before making things we can write down the basic things on paper. The class contains characteristics & behavior. Characteristics mean data

members which are written inside a class & behavior means a function that is applied to those characteristics.

I am going to create a class having one function which will take two parameters as a string and that function will return both the strings as concatenated output/return format.

**Real-Time Example**: Consider a light bulb

Image URL

**Characteristics**: coil, cover, etc.

**Behavior**: Turn on / off.

**What is Object?**

The object is an instance of a class through which we access the member inside of that class, which means it can access the characteristics & behavior of that class. The object has a lifespan.

Consider that the below class, concatString, contains a function, displayConcatString, which returns a concatenated string.

**Example**

```
class concatString{

displayConcatString(str1: string, str2:string){

return (str1 + ' '+ str2);

   }

}

let objectStr = new concatString();

console.log(objectStr.displayConcatString('Hello', 'World'));
```

In the above example, I have created an object of class through which we are calling a function, displayConcatString.

Now to run the application we will use the same commands as above & our output will be like this:

```
PS G:\Web Development Fundamentals\TypeScript\basics> tsc basicSyntax.ts
PS G:\Web Development Fundamentals\TypeScript\basics> node basicSyntax.js
Hello World
```

**Comments**:

Typescript also supports two types of comments

**1. Single line comment**- By using the // sign developers can write a comment whenever required

**Example**: It is used to maintain the version number of specific file type we can use

```
// TypeScript version 3.8
```

**2. Multi-line comment**- By using the /* */ sign developers can write comment whenever required

**Example**: It is used to maintain information about developer name, date time, etc in the application.

```
/*
  Developer Name: www.CoderFunda.com
  Date: 28-02-2020
*/
```

**Compiler Options**

**For a complete list of all compiler options, use this URL**

Will explain how these options are useful in coding.

**Note –**These options are not useful in the command line, these are useful in the tsconfig.json

1) **--out**

This is used to compile multiple files into a single output file.

2) **--removeComments**

To remove all comments from the file.

3) **--module**

Load the module.

4) **--w / --watch**

Watch the file changes done by the developer.

I have written some code as below, myModule is the module, and webURL & completeDetails are two variables that I am combining into a single string to print as output.

```
module myModule{

/* Welcome info about www.CoderFunda.com*/

let webURL: string = 'www.CoderFunda.com';

let completeDetails : string = ' \n Welcome to ' + webURL +

'- articles & blogs by Rupesh Kahane;

console.log(completeDetails);

}
```

I am just executing the above code & will see the output

```
Welcome to www.CoderFunda.com -articles & blogs by Rupesh Kahane
```

Here I am using **--out** to copy the output code into another file, also I will use **--removeComments** to remove comments in the file. I will use the below command to create new output file as

**Command:** tsc --removeComments --out newOutputFile.js basicSyntax.ts

```
tsc --removeComments --out newOutputFile.js basicSyntax.ts
```

We will see the code in new file newOutputFile.js as

```
var myModule;
(function (myModule) {
var webURL = 'www.CoderFunda.com';
var completeDetails = ' \n Welcome to ' + webURL +
' - articles & blogs by Rupesh Kahane';
console.log(completeDetails);
})(myModule || (myModule = {}));
```

As there are no comments in the new files because we used **--removeComments.**

## Summary

This chapter was an introduction to Microsoft's new open-source programming language, TypeScript. In this chapter, we covered what TypeScript is, why we use TypeScript, how to install TypeScript, and how to build our first TypeScript program. We also learned the fundamentals of TypeScript.

In the next chapter, I will cover variable declaration & operators in TypeScript.


C#Corner

# Chapter 2. Variable Declaration & Operators in TypeScript

## Introduction:

TypeScript is a superset of JavaScript, so it supports variable declaration using let and const. Also, it supports operators to perform the operations while creating the programs. In this chapter, we will learn Variable & Operators in TypeScript programming language. In this chapter, we will cover the following

1. What are variables?
2. Why use variables?
3. Variable Declaration
4. Re-initialize / re-declaring the variables
5. Block Scoped variables - let & const declaration
6. What is the Operator?
7. Types of Operators

## 1. What are Variables?

Variables are the names used in the programming language, and they have some values associated with their data types. Values can be changed during the execution of the program. The variable name itself explains that value can be vary/changed any time. A variable name should be meaningful so anyone can understand the code easily.

**Keywords are not allowed** as a variable name. Keywords are the reserve words in the TypeScript language like **break, as, any, switch, if, string, else,** etc.

**Variable declaration:**

[keyword] [variable_Name] : [Type] = [Optional_Value];

Here

keyword: We can use the let/var keyword depending upon the scope of our variable.

Variable_Name: Unique variable name.

Type: Data type of variable like the number, string, Boolean, etc.

Optional_value: Value to be assigned which is optional, we can assign the value later as well.

**Important points related to the naming convention:**

Spaces are not allowed while declaring the variable name in TypeScript. It must contain letters, numbers & may contain underscore (_). It is case sensitive.

var first: number;

var FIRST: number;

In the above code variable first & FIRST are treated as different in TypeScript.

**Example:** Consider the world-famous game Cricket

Image URL

We will consider the below real-life example which will give us the information about the highest score & player name by year.

1) Player Name - Rahul

Highest Score - 501

Year - 1989

2) Player Name - Sachin

Highest Score - 895

Year - 1992

Here in real life variable names are the same, but the values are different every year. There are some conditions to use the variable name. We can use the same variable name in different scope as in the above example year is different.

**Important Points**

A variable name should be unique; it is used to store value by the user/developer. The scope depends upon the usage. Every variable has a type which is used to store specific/mixed type of data depending upon its data type.

## 2. Why use variables?

As we know by using programming languages, we can create an application, or we create a structure that is being used by day to day activities. The application can be used to store some data, perform some actions on this data, or give us a report.

**Example:** Birth of a child.

Birth of a child consists of some information that will be used to calculate the population of a city, state, or country.

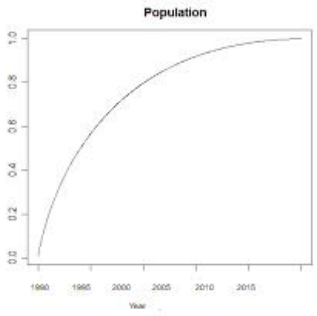



Image URLGraph Image URL

To maintain such a record application, we need a person's **name, gender, date of birth, birth time, birth location, city, state, country,** etc. to maintain the record.

The number of children born in every city has different data each day, so variable names have different data, but the structure remains the same. We can keep these records in such a manner so we can do the calculation based on this data & generate the result/report/graphs etc.

So we will use some unique variable names such as **personName, gender, dateOfBirth, birthTime, birthLoction, city, state, country, etc.**

## 3. Variable Declaration:

Variables can store single values & this can be changed any time during execution of the program. Variable declaration by using **var** keyword is as it is in JavaScript.

**Syntax:** var [variable_name] = [optional-value];

Or in typescript we can define like

var [variable_name] :[data-type]= [optional-value];

**Example:**

```
var person_name = 'Sam';
```

Or

```
var person_name: string = 'Sam';
```

Consider the below code snippet, initially car speed is 0, if a car starts, value is true, then car speed value can be changed to 60. So the value of the variable can be changed any time during execution of the program.
Here **carSpeed**variable is a **global variable** when it is declared outside the function. Value can be preserved outside of condition or function as well.

**speedLimit** variable is the local variable, we can access it within its construct.

```
var carSpeed = 0;
var carStarts = true;
if(carStarts)
{
     function calculateCarSpeed()

      {

var speedLimit = 120;
        carSpeed = 60;
    }
}
console.log(carSpeed);
```

## 4. Re-initializing / redeclaring the variables:

We can use the same variable name in a different function/block scope.

We will write code in which two different functions are having the same variable name as "**message**"& that is accessed within that function or only in the scope block of the function. It is allowed in TypeScript / JavaScript.

```
function displayMessage()
{
  var message = 'Welcome from displayMessage function';
  console.log(message);
}
displayMessage();

function displayMessage_1(){
  var message = 'Welcome from displayMessage_1 function';
  console.log(message);
}
displayMessage_1();
```

**Code explanation:**

We have written two functions having different function names; i.e., **displayMessage()** & **displayMessage_1()**, also we have written the same variable name; i.e., **message** being used in each function. We are just printing our **message** variable value on the console.

## 5. Block Scoped variables:

Consider the below code snippet. By default, variable isActive is false, having type boolean. In if condition we check the value of this Boolean variable and if it's true then it will go inside the condition & return a value from a function.

We will write one function in which we will declare a num2 variable in the  if condition. We are accessing this variable outside if condition / blocked scope is also possible.

```
function getValue(num1: number, isActive: boolean = false)
```

```
{
  if(isActive)
  {
    var num2;
    num2 = num1 + 1000;
  }
  return num2;
}
console.log(printNumber(1));
console.log(printNumber(3, true));
```

We will execute the above functions by passing parameters to function. The function is returning a value of num2.

We will pass one parameter which is a type of number & print the actual num2 value in the console. The second parameter in function is by default false. So, while executing this condition if the condition is false so the function will return **undefined**.

Now in the second scenario, we will pass two parameters as first is number & second is Boolean value **true.** So, the function will return a value of num2, which is accessible outside if condition/block scoped.

To execute the above script using terminal use the below syntax

```
PS G:\Web Development Fundamentals\TypeScript\variable> tsc variable.ts
PS G:\Web Development Fundamentals\TypeScript\variable> node variable.js
undefined
1003
```

### A) let declaration & const declaration:

**Why let & const in TypeScript?**

let & const are the new variable declarations used in TypeScript. These come into the picture to resolve problems in the var declaration. let is used to declare a block-scoped variable which means anything declared within the {} is block-scoped. **static type** checking is possible by using let variable declaration, so while writing the script the developer can get suggestions/ warnings, etc. **const** is similar to let, except we cannot **update** or **re-declare it.**

Consider the below code snippet for **block-scoped:**

```
let webURL = 'welcome to www.CoderFunda.com';
let isArticlesReadble = true;
if(isArticlesReadble)
{
    let message = 'Thanks for reading articles on  www.CoderFunda.com';
    console.log(message);
}
console.log(message);
```

**Code explanation:**

Here while printing the **message** variable value on a console outside the scope i.e. {}, will get an error **Cannot find name message.** This means let variables have been **block-scoped**.

Consider the below code snippet **cannot Re-declare a variable in the same scope:**

```
if(true)
{
let a = 100;
let a = 200;
}
```

**Code Explanation:**

TypeScript will give us a warning as we cannot redeclare the block-scoped variable **'a'**.

Consider the below code snippet for **Re-declaring a variable in a different scope:**

```
let message = 'Welcome to www.CoderFunda.com';
let isArticlesReadble = true;
if(isArticlesReadble)
{
    let message = 'Thanks for reading articles on  www.CoderFunda.com';
    console.log(message);
}
console.log(message);
```

**Code Explanation:**

This code will execute successfully because we have defined the same variable i.e. **message** in different scope. After running this code will get output like this:

*Thanks for reading articles on www.CoderFunda.com*

*Welcome to www.CoderFunda.com*

Consider the below code snippet for **Static Type Checking:**

```
let age: number = 100;
age = 'updated age';
```

Here **age** variable is of type number. In the second line if we try to assign a string type value it will give us a warning as **Type 'updated age' is not assignable to type 'number'.** So here **static type** checking is also done in TypeScript.

**B) const Declaration**

Consider the below code for const declaration:

**Syntax:** **const** [variable_name] = [value];

Or

**const** [variable_name] : [data_type] = [value];

**Example:**const totalMonths = 12;

Or

const totalMonths : number = 12;

So, every **const** declaration must be initializing while declaring.

Consider below code snippet for **cannot re-initialize**:

```
const empName: string = 'sam';
empName = 'raj';
```

In the above code, we cannot re-initialize const variable empName.

```
const day: string;
```

```
const hours;
```

In the above code, we just have to initialize the const variables, without initializing it will give us warning for variable hours.

**Important Points**

Illegal to declare a variable as in below code snippet

1) before variable declaration make an increment of a variable

```
num1++;
let num1: number = 100;
```

2) before assigning a value to make an addition

```
let num2 = (num2 + 200);
```

3) If the type of variable is decided, then assign the value of that variable to a different type of variable

```
let webURL = 'www.CoderFunda.com';
var i: number = webURL;
```

4) Keywords are not used as variable names

```
var else: string = 'Welcome';
var switch: number = 150;
```

## 6. What is Operator?

An operator is a symbolic sign / single character which is used to operate. Some mathematical operators are used to perform some operations like addition (+), subtraction (-), multiplication (*) & division (/). Some other operators are used to perform some operations on numbers, string, etc. like Boolean. In terms of a programming language, when we are writing or solving any problem, we need to manage the flow of our data on which we can make a decision. These are called **Conditional operators.**

**Example**: Consider a DJ Operator who operates music by using some keys to make volume up /down, bass increase/decrease, mix the sounds of two songs, etc. For this operator, every key is used as **Operators.**

Image URL

## 7. Types of Operators

### 1) Arithmetic Operators

These operators are used to perform the mathematical operations like addition (+), subtraction (-), multiplication (*) & division (/).

**Example:** Consider the following two variables having values A having 50, B having 30 & we will perform addition, subtraction, multiplication & division of these two variables as below. We will see the output into a console.

```
let A: number = 50;
let B: number = 30;
console.log('Addition of two variables using + operator is:', (A+B));
console.log('Subtraction of two variables using - operator is:', (A-B));
console.log('Multiplication of two variables using * operator is:', (A*B));
console.log('Division of two variables using / operator is:', (A/B));
```

Run the above code using the terminal & see the output as

Addition of two variables using + operator is: 80

Subtraction of two variables using - operator is: 20

Multiplication of two variables using * operator is: 1500

Division of two variables using / operator is: 1.6666

### 2) Assignment Operators

We can assign values to the variables by using these operators. Also, we can declare **shorthand notations/compound assignment operators.**

In the previous example, we assign **100** value to variable **A** by using **= operator**

We can add, subtract the value from **A** variable by using **-operator** as

```
let A: number = 100;
let B: number = 50;
console.log('Addition using Shorthand notation ',(A+=5))
console.log('Subtraction using Shorthand notation ',(B-=5))
```

**Code Explanation:** In the above example **(A += 5)** means **(A = A + 5) ,** so **A** variable has value while initialization is 100. Now our code will execute as **A = (Value of A variable) + 5;** i.e. **A = 100 + 5;**

This is the same for subtraction as well **B = 50 – 5;**

Run the above code using terminal & see the output as

```
PS G:\Web Development Fundamentals\TypeScript\Operators> tsc operators.ts
PS G:\Web Development Fundamentals\TypeScript\Operators> node operators.js
Addition using Shorthand notation  105
Subtraction using Shorthand notation  45
```

**Consider the below code snippet**

```
let A : number = 100;
let B : number = 50;
let C = 11;
let D = 15;
let E = 20;
let F = 30;
console.log('Multiplication : ',(A*=5));
console.log('Division : ',(B/=5));
console.log('Mod value : ',(C%=5));
console.log('Increment operator : ',(++E));
console.log('Decrement operator : ',(--F));
```

We can use multiplication, division, the modular operator (to calculate reminder), increment, decrement, etc. It works the same as the above addition, subtraction.

The output of the above code is:

```
PS G:\Web Development Fundamentals\TypeScript\Operators> tsc operators.ts
PS G:\Web Development Fundamentals\TypeScript\Operators> node operators.js
Multiplication :  500
Division :  10
Mod value :  1
Increment operator :  21
Decrement operator :  29
```

### 3) Relational Operators:

These operators are used to compare the variables.

**== (Equal To) Operator:** To check the values of two variables, if it is true then it will go into a condition.

**!= (Not Equal To) Operator:** To check whether the value of two variables is equal or not.

**> (Greater Than) Operator:** To check if the value of the left side variable is greater than the right side variable, if it is true then it goes into a condition.

**< (Less Than) Operator:** To check if the value of the left side variable is less than the right side variable, if it is true then it goes into a condition.

**>= (Greater Than Equal To) Operator:** To check if the value of the left side variable is greater than or equal to the right side variable, if it is true then it goes into a condition.

**=< (Less Than Equal To) Operator:** To check if the value of the left side variable is less than or equal to the right side variable, if it is true then it goes into a condition.

**Example:**

```
let A = 150, B = 250;
if(A==B)
{
    console.log('A & B Both are equal');
}
if(A!=B)
{
```

```
    console.log(''A & B Both are not equal');
}
if(A>B)
{
    console.log('A is greater than B');
}
if(A<B)
{
    console.log('A is less than B');
}
if(A>=B)
{
    console.log('A is greater than or equal to B');
}
if(A<=B)
{
    console.log('A is less than or equal to B');
}
```

Here A has a value of  150 & B has a  value 250, so it will match some conditions and print the message on the console if the condition is true.

If we run the above code snippet then the output will like:

A & B Both are not equal

A is less than B

A is less than or equal to B

**4) Logical Operator:**

It is used in the programs when we are writing some logical conditions.

**&& (Logical AND) Operator:** This is a logical AND operator, if both the conditions are true i.e. left & right side then condition will execute.

**|| (Logical OR) Operator:** This is a logical OR operator, if one of the conditions is true from the left or right side then the condition will execute.

**! (Logical NOT) Operator:** This is a  logical Not operator, it checks the opposite of the condition.

**Example:**

```
let A : boolean = true, B : boolean = false;
if(A&&B)
{
    console.log('You are in && condition');
}
if(A||B)
{
    console.log('You are in || condition');
}
if(!B)
{
    console.log('You are in ! condition');
}
```

It will check the conditions according to the operators executing the condition & printing a message on the console.

If we run the above code snippet then the output will be:

```
PS G:\Web Development Fundamentals\TypeScript\Operators> tsc operators.ts
PS G:\Web Development Fundamentals\TypeScript\Operators> node operators.js
You are in || condition
You are in ! condition
```

**Summary:** In this chapter, we covered what variable is, naming conventions, why we use variables, variable declaration, Re-initializing / re-declaring of variables, blocked scope, use of let & const, Operator & its types in TypeScript. In the next chapter, I will cover Data Types in TypeScript.

## Chapter 3. Data Types in TypeScript

## Introduction:

As we have seen how we can declare a variable in the previous chapter, now in this chapter we are going to learn what are different data types in TypeScript & how can we use them all. Data types are the most important part of every programming language. In this chapter, we will cover the following different data types

1. number
2. string
3. boolean
4. tuples
5. union
6. any
7. void
8. null & undefined
9. never

### 1. number data type in TypeScript

**What is the number?**

Numbers are the digits used in the programming language; a Number is an object which holds the numeric values & which are meaningful. TypeScript supports decimal, hexadecimal, octal, binary as well whenever required. In general, suppose we consider a landline or mobile phone & everyone has a unique number which is a combination of some digits.

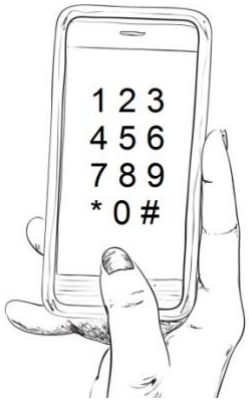



Image URL

Consider the sample below to initialize the numbers.

```
let A = new Number(500);
let B: Number = 600;
let C: number = 700;
console.log("Value of A: ", A);
console.log("Value of B: ", B);
console.log("Value of C: ", C);
```

In the above example, we can declare the number like A, B, C & assign the value. Will Execute the above code & we will get this output:.

```
PS G:\Web Development Fundamentals\TypeScript\numbers> tsc numbers.ts
PS G:\Web Development Fundamentals\TypeScript\numbers> node numbers.js
Value of A:  [Number: 500]
Value of B:  600
Value of C:  700
```

- **Min Max Value:** In a programming language, Integer numbers have maximum & minimum values. We can access it as shown in the below code snippet.

```
Number.MAX_VALUE

Number.MIN_VALUE
```

Here in above example value of Max is 1.7976931348623157e+308 & value of Min is 5e-324

- **Safe Integer:** Whenever we are doing any addition of +1 / subtraction of -1 in below largest integer number it is unsafe:

```
Number.MAX_SAFE_INTEGER

Number.MIN_SAFE_INTEGER
```

Here in above example MAX_SAFE_INTEGER having value 9007199254740991 & MIN_SAFE_INTEGER having value -9007199254740991

- **NaN:** Whenever the calculation result is not a valid number then it returns as a NaN

```
var D = 100;
D = Number.NaN;
console.log("Value of D: ",D);
console.log("Square root: ",Math.sqrt(-1));
```

Here in the above example the value of D variable & Square root of -1 will get NaN as

```
Value of D:  NaN
Square root:  NaN
```

As we have seen with the operators in the previous chapter, we can compare the NaN value and it will get the result as true or false depending on the condition. Consider the below example

```
console.log(NaN === NaN);
```

Execute the above code & you will get the output as false.

- **Positive & Negative Infinity:** For the number we can check the Infinity values manually or using static class members of Number

```
console.log(Number.POSITIVE_INFINITY);
console.log(Number.NEGATIVE_INFINITY);
console.log(Number.length);
console.log(Number.POSITIVE_INFINITY === Infinity);
console.log(Number.NEGATIVE_INFINITY === -Infinity);
```

Execute the above code & we will get this output:

```
Infinity
-Infinity
1
true
true
```

- **Decimal:** In Typescript we can define or calculate the number in a decimal format, decimal has a base 10

```
let N = 10.50;
console.log("Value of N: ",N);
console.log(.4 + .2);
```

Execute the above code & you will get the output as

```
Value of N:  10.5
0.6000000000000001
```

- **Hexadecimal:** Typescript supports defining the Hexadecimal number. Hex having base 16, so while calculating multiply to the 16th power

  Refer to the below chart for Hex to Decimal Conversion

| Hexadecimal | Decimal |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| A | 10 |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |

Suppose we create a variable having value 0x1D9 then we can get actual value a:

0x1D9

$0x1D9 = (1 * 16^2) + (13 * 16^1) + (9 * 16^0)$

$0x1D9 = (1 * 256) + (13 * 16) + (9 * 1)$

$0x1D9 = 256 + 208 + 9$

0x1D9= 473

```
let M = 0x1D9;
console.log("Value of M: ",M);
```

The  output after executing the above code:

```
Value of M:  473
```

- **Octal & Binary:** Typescript allows us to define a number as octal or binary. Octal has a base 8.

  Suppose the below example has K & L values respectively.

```
let K  =0o477 ;
console.log("Value of K: ",K);
let L = 0b111000;
console.log("Value of L: ",L);
```

We will execute the above code & get the output as

```
Value of K:  319
Value of L:  56
```

**Number Methods:** TypeScript supports some inbuilt methods in the number of data types as below.

A. **toExponential():** This method will return the exponential notation in string format.
B. **toFixed():** This method will return the fixed-point notation in string format.
C. **toLocaleString():** This method will convert the number into a local specific representation of the number.
D. **toPrecision():** This method will return the string representation in exponential or fixed-point to the specified precision.
E. **toString()**: This method will return the string representation of the number in the specified base.
F. **valueOf():** This method will return the primitive value of the number.

**Example**

Consider the below example in which you will use number methods as explained above.

```
let first = 100;
console.log("toExponential: ",first.toExponential(2));
```

```
let second = 1001.2589;
console.log("toFixed: ",second.toFixed(2));
console.log("toLocaleString: ",second.toLocaleString());
console.log("toPrecision: ",second.toPrecision(2));
console.log("toString: ",second.toString());
let third = second.valueOf();
console.log("valueOf: ",third);
console.log("typeof: ",typeof third);
```

We will execute the above code & get the output as:

```
toExponential:  1.00e+2
toFixed:  1001.26
toLocaleString:  1,001.259
toPrecision:  1.0e+3
toString:  1001.2589
valueOf:  1001.2589
typeof:  number
```

## 2. string data type in TypeScript

**What is a string?**

The string is the sequence of Unicode characters or a continuous array of characters. The string can be stored in the format of single/double quotation marks. The string has a meaningful valid name used in the programming language. It can contain an alphabet, numbers, special characters, etc. In general suppose a person object has some properties like the first name, last name, email id, address, etc.

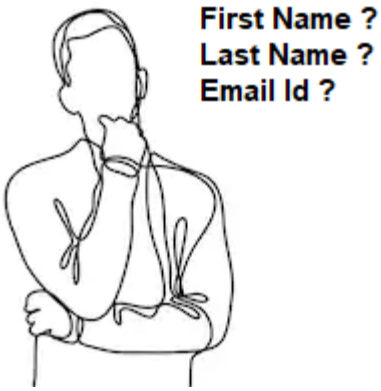


Image URL

**Syntax** to declare string type variable

var [variable_name] = [optional-value];

Or

var [variable_name] :[data-type]= [optional-value];

Here [data-type] we will use as a string.

We can define string data type variables as below

```
let firtName: string = 'John';
let lastName: string;
let userName = 'Sam';
lastName = "Roy";
console.log('First Name is: ',firtName);
console.log('Last Name is: ',lastName);
console.log(typeof(firtName));
console.log(typeof(lastName));
console.log(typeof(userName));
```

In the above code firstName, lastName & userName are of type string. We can initialize the string later as well, as lastName we have only declared & initialized later after the userName variable.

Execute the above code will get output like

```
First Name is:  John
Last Name is:  Roy
string
string
string
```

Consider the below example, I am going to declare two variables, at the time of declaration I have initialized them.

```
let firstName: string  = "John";
let lastName: string  = 'Sing';
console.log("First Name: ",firstName);
```

```
console.log("Last Name: ",lastName);
console.log("Access letter using index at position 0: ",firstName[0]);
```

Execute the above code snippet & you will get output like this:

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
First Name:  John
Last Name:  Sing
Access letter using index at position 0:  J
```

**Important Points:**

- The string is an array, it starts with the 0th Index.
- Try to use **Pascal Case** while giving names to the variables, Ex. FirstName, LastName
- Avoid all **upper case** or all **lower case** characters.
- Avoid using **too many special characters** like first_Name, @first_name
- For prefix of the Boolean variable with **Is, Can, Has,** etc. Ex. IsDelete, IsActive

Create a reference to string function which creates an object as below

```
let address = new String("Flat No 303, MG Road");
console.log("Return reference to string function using string constructor", + add
ress.constructor);
```

Executing the above code will get output like this:

```
Return reference to string function using string constructor function String() { [native code] }
```

On the above string object, we can apply the built-in functions like calculate length, make as all characters as upper case, etc.

```
console.log("address value: ",address);
console.log("length of address string: ",address.length);
console.log("address as Upper string: ",address.toUpperCase());
```

Executing the above code will get output like this:

```
address value:  [String: 'Flat No 303, MG Road']
length address string:  20
address as Upper string:  FLAT NO 303, MG ROAD
```

**string Methods:** TypeScript supports some inbuilt methods in the string data types as below.

A. **charCodeAt() -** Returns the Unicode of the character at the specified index

```
let letter: string = "Apple is good for health";
let result = letter.charCodeAt(0);
console.log(result);
```

The output of the above code is

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
65
```

B. **fromCharCode() -** Converts Unicode values to characters. Due to static object it always uses String.fromCharCode();

Syntax - String.fromCharCode(CharNum);
CharNum is a numeric value, users can pass multiple values as well.

Example:

```
let result = String.fromCharCode(72,69,76,76,79);
```

The output of the above code is:

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
HELLO
```

C. **includes() -** This method returns a true or false by checking whether a string contains the specified string/characters

Example:

```
let welcomeNote = "Welcome to the https://www.CoderFunda.com/ => One Stop Solutio
n for All Your Learning Needs";
let result = welcomeNote.includes("Solution");
console.log(result);
```

The output of the above code is

```
true
```

D. **indexOf()** - This function returns the position of the first found occurrence of a specified value in a string

Example:

```
let welcomeNote = "Welcome to the https://www.CoderFunda.com/ => One Stop Solutio
n for All Your Learning Needs";
let result = welcomeNote.indexOf("Stop");
console.log(result);
```

The output of the above code is

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
47
```

E. **lastIndexOf()** - This function returns the position of the last found occurrence of a specified value in a string; if not found then returns -1.

Example:

```
let welcomeNote = "Welcome to the https://www.CoderFunda.com/ => One Stop Solutio
n for All Your Learning Needs";
let result = welcomeNote.lastIndexOf("Stop");
console.log(result);
```

The output of the above code is

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
47
```

F. **localeCompare()** - Compares two strings in the current locale.
Return value: 0 if string matches 100%
1 if string does not match

Example:

```
let wordA = "Sky is blue";
let wordB = "Sky is blue";
console.log(wordA.localeCompare(wordB));
console.log(wordA.localeCompare("Welcome to CoderFunda.com"));
```

The output of the above code is:

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
0
-1
```

G. **replace()** - This function searches a string for a specified value, or a regular expression, and returns a new string where the specified values are replaced.

Example:

```
let myURL = "Do not forget to comment us at www.CoderFunda.com!";
console.log(myURL.replace("comment", "rate"));
```

The output of the above code is
Do not forget to comment on us at www.CoderFunda.com!"

H. **trim()** - Removes white space from both ends of a string.

Example:

```
let mySentence = "   Welcome!!!   "
console.log(mySentence.trim());
```

The output of the above code is

```
Welcome!!!
```

I. **match()** - This function searches a string for a match against a regular expression, and returns the matches

Example:

```
let result = "100,200,300,300,400!";
console.log(result.search("200"));
console.log(result.search("300"));
```

The output of the above code is:

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
4
8
```

J. **toLowerCase()** - Converts a string to lowercase letters
**toUpperCase()**- Converts a string to uppercase letters

Example:

```
let stringWelcome = "Welcome to hello World !!!";
console.log(stringWelcome.toLowerCase());
console.log(stringWelcome.toUpperCase());
```

The output of the above code is

```
welcome to hello world !!!
WELCOME TO HELLO WORLD !!!
```

K. **toString() -** This function returns the value of a String object

Example:

```
let myNUmber: number = 1001;
let myString = myNUmber.toString();
console.log(myString);
console.log(typeof(myString));
```

The output of the above code is:

```
PS G:\Web Development Fundamentals\TypeScript\string> tsc datatypeString.ts
PS G:\Web Development Fundamentals\TypeScript\string> node datatypeString.js
1001
string
```

## 3. Boolean data type in TypeScript

**What is Boolean?**

Primitive data type function is used as a Boolean in TypeScript & JavaScript. Boolean can take true or false values. In general, consider an example of a light bulb, it might be an On/Off, true/false. We can consider if the light is on then true, else false.

**Syntax**

variable_name : data_type = optional_value;

**Example:**

```
let x: boolean = true;
if (x) {
  // this code is executed if value is true
```

```
console.log('  x has a value', x);
}
```

Here in the above code x value is of type Boolean & has the value true.

Run the above code & you will get the output as:

```
 x has a value true
```

**Object:** It is an instance of a data type/object. TypeScript allows us to create instances of data type.

```
let myObject = new Boolean("CoderFunda.com")
console.log("myObject is : ",typeof(myObject));

var x: boolean = false;
var y = new Boolean(false);
if (x == y) {
    console.log("x & y both are same");
}
```

Here in above code operator ==, checks equality in both type and value.

Run the above code & you will get the output as:

myObject is:  object

x & y both are the same.

**Boolean function:** In typescript & javascript Boolean() is a function, it takes a value & returns a result.  In the below example, commented lines return the below  result.

```
console.log(Boolean(false))
// returns false
console.log(Boolean(true))
// returns true
console.log(Boolean("welcome to CoderFunda.com"))
// returns true
console.log(Boolean(100.50))
// returns true
```

```
console.log(Boolean(NaN))
 // returns false
console.log(Boolean(undefined))
// returns false
```

## Properties of Boolean data type:

A. **Boolean.length:** length is a property having a value of 1
B. **Boolean.prototype:** It represents the prototype for the Boolean constructor.

Example

```
console.log(Boolean.length);
console.log(Boolean.prototype.constructor);
```

Run the above code & you will get output as

1

ƒ Boolean() { [native code] }

**Boolean Methods:** TypeScript supports some inbuilt methods in the Boolean data types as below.

**A. toString():** The toString() method returns a string. In the below example for variable stringA.toString() method returns a string.

```
let stringA: Boolean= new Boolean(true);
console.log(stringA.toString());
// returns output: true

let stringB: Boolean= new Boolean(1);
console.log(stringB.toString());
// returns output: true

let stringC: string= "1000";
console.log(stringC.toString());
// returns output: 1000
```

B. **valueOf():** The valueOf() method returns the primitive value of a Boolean object. Consider the below example.

```
var x = new Boolean();
console.log(x.valueOf());
// returns output: false

var y = new Boolean("Hello World");
console.log(y.valueOf());
// returns output: true
```

## 4. Tuples data type in TypeScript

**What is Tuple?**

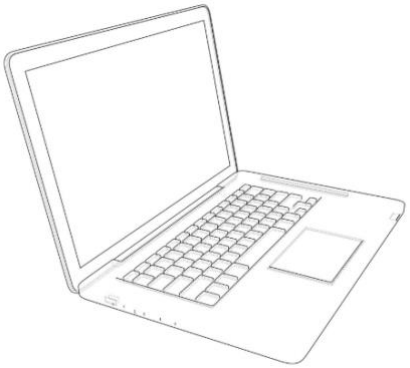
It is a new data type introduced in Typescript. Sometimes the programmer needs to store the data of a different type, so Array does not satisfy this condition. In tuples, we can store multiple types of data like number, string, Boolean, etc.

**Why Tuple?**

Consider a normal variable which stores the single data type value at a time, we can't assign other data type values. To store multiple data type values into a single variable is possible using a tuple.

**Example:**

Consider a computer or laptop, having some configuration like computer/laptop serial number, manufacturer name, processor, hard disk, RAM, etc.



Consider the below code snippet.

```
// normal variable declaration
let numberOfComputers: number = 1000;
let manufacturerName: string = "Dell";

// tuple declaration
let computerDetails: [number, string, boolean] = [1000,"Dell", true];
```

In the above example, computer details are stored in two different variables in a normal variable declaration, since **Tuple** is a new data type. So we can store different types of data into a single variable as above.

**Tuple Array:** Typescript allows us to declare the tuple array, so the user can store the different types of data in a single array.

```
let laptopDetails: [number, string, string][];
laptopDetails = [[5000, "Dell","Intel CORE i3"], [2000, "Lenovo","Intel CORE i5"]
, [1000, "HP","Intel CORE i5 with Graphics"]];
```

**Array Methods**

Consider the whole example.

```
let laptopDetails: [number, string, string][];
laptopDetails = [[5000, "Dell","Intel CORE i3"], [2000, "Lenovo","Intel CORE i5"]
, [1000, "HP","Intel CORE i5 with Graphics"]];

console.log(laptopDetails[0]);
console.log(laptopDetails[1]);
console.log(laptopDetails[2]);
console.log();
console.log(laptopDetails.sort());
console.log();
laptopDetails.forEach(function (value) {
    console.log(value);
  });
```

In the above example, laptopDetails is a tuple array having some values. We can access it by using indexes & also we can print the values in tuples by using the sort method of the array.

Execute the above & we will get output like this:

```
PS F:\Web Development Fundamentals\TypeScript\tuples> tsc tuples.ts
PS F:\Web Development Fundamentals\TypeScript\tuples> node tuples.js
[ 5000, 'Dell', 'Intel CORE i3' ]
[ 2000, 'Lenovo', 'Intel CORE i5' ]
[ 1000, 'HP', 'Intel CORE i5 with Graphics' ]

[
  [ 1000, 'HP', 'Intel CORE i5 with Graphics' ],
  [ 2000, 'Lenovo', 'Intel CORE i5' ],
  [ 5000, 'Dell', 'Intel CORE i3' ]
]

[ 1000, 'HP', 'Intel CORE i5 with Graphics' ]
[ 2000, 'Lenovo', 'Intel CORE i5' ]
[ 5000, 'Dell', 'Intel CORE i3' ]
PS F:\Web Development Fundamentals\TypeScript\tuples>
```

We can't directly push or pop elements:

```
laptopDetails.push(1023);
```

We can push an array element into a tuple:

```
laptopDetails.push([10,"Dell","Intel"]);
console.log(laptopDetails.sort());
```

Execute the above code & will get this output:

```
PS F:\Web Development Fundamentals\TypeScript\tuples> tsc tuples.ts
PS F:\Web Development Fundamentals\TypeScript\tuples> node tuples.js
[
  [ 10, 'Dell', 'Intel' ],
  [ 1000, 'HP', 'Intel CORE i5 with Graphics' ],
  [ 2000, 'Lenovo', 'Intel CORE i5' ],
  [ 5000, 'Dell', 'Intel CORE i3' ]
]
```

## 5. Union data type in TypeScript

### What is the Union data type?

It allows us to create more than one type. By using Union we can easily add values of different types to the same variable. Consider our function is returning values based on conditions, then we can declare the variable using Union. A vertical bar (pipe i.e. symbol **|** ) is used to declare two or more data types into a single variable.

**Syntax:**

variable_Name: dataType1 | dataType2 | dataType3 = [Optional_Value];

**Example:**

```
let retun_Value: number | string | boolean;
retun_Value = 1000;
console.log('Varible has assigned a numeric value', retun_Value);
retun_Value = 'Welcome';
console.log('Varible has assigned a string value', retun_Value);
retun_Value = true;
console.log('Varible has assigned a boolean value', retun_Value);
```

Execute the above code & you will get output like:

```
PS F:\Web Development Fundamentals\TypeScript\union> tsc union.ts
PS F:\Web Development Fundamentals\TypeScript\union> node union.js
Varible has assigned a numeric value 1000
Varible has assigned a string value Welcome
Varible has assigned a boolean value true
```

#### 1) Passing parameter as union type to function.

Consider the below function. In a function ShowValue we are passing parameter input as a union type of number & boolean so it can accept number & union value.

```
let input: number | boolean;
```

```
function ShowValue(input) {
    if(typeof(input) == "boolean") {
       console.log('Varible has assigned a boolean value', input);
       return input;
    } else {
        console.log('Varible has assigned a other value', input);
       return input;
    }
 }
 console.log("Passing a numeric value to function & function returns: ",ShowValue
(5));
 console.log();
 console.log("Passing a boolean value to function & function returns: ",ShowValue
(true));
```

Execute the above code & we will get output like this:

```
PS F:\Web Development Fundamentals\TypeScript\union> tsc union.ts
PS F:\Web Development Fundamentals\TypeScript\union> node union.js
Varible has assigned a other value 5
Passing a numeric value to function & function returns:  5

Varible has assigned a boolean value true
Passing a boolean value to function & function returns:  true
```

**Union Type & Array:** Union also accepts the array. Properties & interface is also valid in the union.

1) **Passing array parameter as union type to function.**

Consider the below function. In function getDetails will pass a parameter as a Boolean & array of number type, function will accept values as a union type of Boolean and numeric array.

```
let parameter: boolean | number[];
function getDetails(parameter)
{
    if(typeof(parameter) == "boolean")
    {
        console.log("Variable has a value: ",parameter)
    }
    else
    {
        for(let i: number  = 0;i<parameter.length;i++) {
```

```
        console.log("Value in Array at position "+ i + " is :",parameter[i]);
         }
    }
}

 console.log(getDetails(true));
 console.log();
 console.log(getDetails([100,200,300]));
```

Execute the above code & you will get this output:

```
PS F:\Web Development Fundamentals\TypeScript\union> tsc union.ts
PS F:\Web Development Fundamentals\TypeScript\union> node union.js
Variable has a value:  true


Value in Array at position 0 is : 100
Value in Array at position 1 is : 200
Value in Array at position 2 is : 300
```

**2) Passing a two-dimensional array of union type as a parameter to function.**

Consider the below function. In function getDetails will pass a parameter of two dimensional array of number & string of union type.

```
let parameter: number[][] | string[][];
function getDetails(parameter)
{
    for(let i: number = 0; i<parameter.length; i ++)
    {
        for(let j: number  = 0;j<parameter.length;j++)
        {
  console.log("Value in Array at position "+ i, j + " is :",parameter[i][j]);
        }
    }
}
parameter = [[0, 1], [2, 3]];
console.log(getDetails(parameter));
parameter = [["Sam", "Amar"], ["Ajit", "Ashish"]];
console.log(getDetails(parameter));
```

Execute the above code & you will get output like this:

```
Value in Array at position 0 0 is : 0
Value in Array at position 0 1 is : 1
Value in Array at position 1 0 is : 2
Value in Array at position 1 1 is : 3

Value in Array at position 0 0 is : Sam
Value in Array at position 0 1 is : Amar
Value in Array at position 1 0 is : Ajit
Value in Array at position 1 1 is : Ashish
```

### 6. any data type in TypeScript

**What is any data type?**

When the user/programmer is not sure about the data type then, in this case, we can define it as any type. It can accept any data type values like a number, boolean, string, etc. Sometimes if we are getting data from any third party application & we are not aware of the return type of that data then, in this case, we can use any data type. It is very powerful to use in Typescript other than Javascript. It is representing values with no constraints. The compiler does not have an idea about its members, it will type check when it will assign or access the members.

**Syntax:**

variable_Name: any = [Optional_Value];

**Example:**

Consider the below example - we have created a variable having type as any. We will assign a different value to it like string, numeric & boolean, etc.

```
let myVariable: any;
myVariable= "Welcome to CoderFunda.com !!!"
console.log("Variable has assigned a string type value is: ", myVariable)

myVariable= 1000;
console.log("Variable has assigned a numeric type value is: ", myVariable)

myVariable= true;
console.log("Variable has assigned a boolean type value is: ", myVariable)
```

Execute the above code & we will get output like this:

Variable has assigned a string type value is: Welcome to CoderFunda.com

Variable has assigned a numeric type value is: 1000

Variable has assigned a boolean type value is: true

We can define the function which is not yet defined so the compiler will transcompile successfully, which means the user is telling the compiler don't worry about it. I will handle it & come back to you. Consider the below code snippet for more information.

```
myVariable.getLatestInfomation();
```

Let's transpire the above code & see the command prompt.

```
PS F:\Web Development Fundamentals\TypeScript\Any> tsc any.ts
```

It transpires successfully.

**Array using any type:**

Consider the below code snippet, we have defined parameter variables as any type & at the time of calling we are passing an array value of the same or different types.

```
let parameter:any;
function getDetails(parameter)
{
  for(let i: number  = 0;i<parameter.length;i++)
  {
        console.log("Value in Array at position "+ i + " is :",parameter[i]);

  }
}
console.log(getDetails([100,200,300]));
console.log();
console.log(getDetails(["Sam","Rohan","Prathmesh"]));
console.log();
console.log(getDetails([true,"Welcome",5000]));
```

Execute the above code & you will get output like this:

```
PS F:\Web Development Fundamentals\TypeScript\Any> tsc any.ts
PS F:\Web Development Fundamentals\TypeScript\Any> node any.js
Value in Array at position 0 is : 100
Value in Array at position 1 is : 200
Value in Array at position 2 is : 300

Value in Array at position 0 is : Sam
Value in Array at position 1 is : Rohan
Value in Array at position 2 is : Prathmesh

Value in Array at position 0 is : true
Value in Array at position 1 is : Welcome
Value in Array at position 2 is : 5000
```

**What is Object?**

The object has its life span & it is an Object interface. Only members inside the interfaces are available in the object. When we write access methods without implementation then we will not transcompile, it will give us an error. Object exposes the properties, functions defined in the Object.

**Syntax:**

variable_Name: Object;

**Example:**

In the below code snippet there are two Objects, result & Person. Person has two properties like Id & firstName having numeric & string types. We can access properties using the object. Property name will be assigned to the result object directly.

```
let result: Object;
let Person = {
    Id: 1001,
    firstName: "Joy"
}
result = Person.Id;
console.log("Value of result object is: ", result);
result = Person.firstName;
console.log("Value of result object is: ", result);
```

Execute the above code & you will get output like this:

```
PS F:\Web Development Fundamentals\TypeScript\Any> tsc any.ts
PS F:\Web Development Fundamentals\TypeScript\Any> node any.js
Value of result object is:  1001
Value of result object is:  Joy
```

If we do not implement any function & try to access it by using Object then it will not trans-compile the code & will give us an error during compile time. Consider the below code snippet:

```
let Employee: Object;
Employee.GetEmployeeDetails();
```

Ww will try to trans-compile the above code & see the terminal

```
PS F:\Web Development Fundamentals\TypeScript\Any> tsc any.ts
any.ts:24:10 - error TS2339: Property 'GetEmployeeDetails' does not exist on type 'Object'.

24 Employee.GetEmployeeDetails();
            ~~~~~~~~~~~~~~~~~~~~
```

### 7. void data type in TypeScript

If we use a void return type of any function, then it means the function does not return any value. The void is the opposite of any data type, which means the void does not have any value at all. any may return any kind of data (including null and undefined) so avoid the use of void data type.

**Example:**

```
function getEmployeeDetails(): void {
    let Id: number = 1000;
    let Name: string = 'Sam';
        console.log("Employee id is: ", Id);
        console.log("Employee name is: ", Name);
    }
    let result = getEmployeeDetails();
    console.log("Variable called");
```

Execute the above code & you will get output like this:

```
Employee id is:  1000
Employee name is:  Sam
Variable called
```

**Variable declaration using void:**

We cannot use void as a data type to a variable due to it being null.

**Example:**

**1)**

```
let firstName : void;
console.log("Variable value is : ", firstName);
```

It will give us a warning.

**2)** Suppose we tried in another way, like consider the below code snippet:

```
let myResult: void = "Jay";
console.log("myResult variable has a type of : ",typeof(myResult));
```

It will also give us a warning at compile time.

Execute the above code & you will get output like this:

Type '"Jay"' is not assignable to type 'void'

## 8. null & undefined in TypeScript:

These are the same as void, and not used too much in TypeScript / JavaScript. These have their types named as null & undefined. We can check at the time of strict type checking. It's better to use any to avoid many errors in our application.

**Example:**

```
let myValue: undefined = undefined;
let myResult: null = null;
console.log("myValue variable has a type of : ",typeof(myValue));
console.log("myResult variable has a type of : ",typeof(myResult));
```

Execute the above code & you will get output like this:

```
myValue variable has a type of :  undefined
myResult variable has a type of :  object
```

### 9. never data type in TypeScript:

It is a new type of typescript. It means the value will never be assigned to variable or return from function. never is a subtype of and assignable to every type. It doesn't return undefined, either.

**Example:**

1) In the below two sentences first id is valid & second name will give us a warning as - Type '"Joy"' is not assignable to type 'never'

```
let id: never;
let name: never = "Joy";
```

2) Consider the below code snippet, the return type of the calculatePrice function is inferred to be never. Here TypeScript infers the never type.

```
const calculatePrice = (message: string) => {
    throw new Error(message);
  };
```

**Difference between never and void**

For void it will allow us to assign undefined, for never it will give us the error. All errors & warnings are as follows

```
let firstName: void = undefined;
let middleName: void = null;
let lastName: never = undefined;
let addressLine: never = null;
```

Type 'null' is not assignable to type 'void'

Type 'undefined' is not assignable to type 'never'

Type 'null' is not assignable to type 'never'.

**Summary:** In this chapter, we covered different data types in TypeScript, how we can declare the variable using different types & some inbuilt methods in data types which are useful while creating some programs. In the next chapter, I will cover different types of statements in TypeScript.

# Chapter 4. Statements in TypeScript

## Introduction:

Whenever we are creating an application, at that time we need to handle some conditional blocks, so statements are very useful in this case. Different types of statements like if, if-else, if – else if –else, etc can be used in TypeScript. In this chapter, we will cover the following

1. What is a Statement?
2. Why do we need a Statement?
3. if statement
4. if-else statement
5. if-else-if

**What is a Statement?**

In computer programming, a statement is a syntactic unit of an imperative programming language that expresses some action to be carried out. A program written in such a language is formed by a sequence of one or more statements. For more information click here

In programming languages, we need to execute the different conditions, so the statements are very useful. We can use multiple statements while creating a program.

**Real-life example:**

In general, suppose we visit a hotel or tea joint and order tea.

Image Source

If the tea is hot, then we will wait for a little bit of cooling, because we cannot drink hot tea directly. Here the condition makes sense and is very important.

**Why do we need statements?**

Programming has many features like conditional statements, conditional expressions, and conditional constructs, etc. that are useful while building small programs or large applications. To print a large amount of data using loops or based on conditions, statements are useful. To handle different cases, we can use conditional statements. To perform logical operations by using operators and different conditions, statements are a very powerful way.

**Selection Statement**

**1) If statement**: If expression/condition is true then it will execute the statement block. In this type of if-statement, the sub-statement will only be executed if the expression is non-zero. Multiple if blocks are allowed. if it is a reserved word in TypeScript, we can't use it as a variable name.

**Syntax:**

```
if ( expression ) statement
```

**Example:**

```
let IsActive: boolean = true;
if(IsActive)
{
console.log("If condition true, so print successfully");
}
console.log();
let Id : number = 10;
if(Id < 20)
{
console.log("Id is less than 20, so If condition satisfied");
}
```

Execute the above code & we will get this output:

If condition true, so print successfully

It is less than 20, so the  If condition is satisfied.

**2) if-else statement**: If-else statement executes based on the condition, if the expression is satisfied then the if statement will execute, else statement will execute. Multiple if-else blocks are allowed. Each else matches up with the closest unmatched if. If & else are the reserved words in TypeScript, we can't use it as a variable name.

**Syntax:**

```
if ( expression )
{
    statement
}else
{
    statement
}
```

**Example:** Consider the below example, if IsActive is true then if block will execute otherwise else block will execute.

```
let IsActive: boolean = true;
if(IsActive)
{
console.log("You are in the If block");
}
else
{
console.log("You are in the else block");
}
```

Execute the above code & we will get this output:

You are in the If block

Example: Consider the below example, we can write multiple if-else conditions per our requirement.

```
let IsActive: boolean = true;
let IsDelete: boolean = false;
if(IsActive)
{
```

```
console.log("You are in the If block of isActive");
}
else
{
console.log("You are in the else block of isActive");
}
if(IsDelete)
{
console.log("You are in the If block of isDelete");
}
else
{
console.log("You are in the else block of isDelete");
}
```

Execute the above code & we will get this output:

You are in the If block of isActive

You are in the else block of isDelete

**3) if-else - if the statement**: If-else-if statement executes based on the multiple conditions. In this statement, we can have 'if' and ''else', also we can have use multiple 'else-if' clause. As soon as the condition meets it will execute that conditional block & rest gets ignored. If none of the condition executes then it will execute the else block.

**Syntax:**

```
 if (expression_1)
   statement1;
 else if(expression_2)
   statement2;
 else if (expression_3)
    statement3;
 else
   statement4;
```

**Example:** Consider the below example, we are going to use if-else-if statements into it.

```
let motion: string = "left";
let strLength: number = motion.length;
if (strLength <=1 && strLength>0) {
      console.log("string length is one.");
    }
else if (strLength <=2 && strLength >1) {
      console.log("string length is two.");
    }
```

```
else if (strLength <=3 && strLength>2) {
      console.log("string length is three.");
    }
else if (strLength <=4 && strLength>3) {
      console.log("string length is four.");
    }
else{
      console.log("string length is greater than four.");
    }
```

Execute the above code & we will get this output:

string length is four.

**Summary:** In this chapter, we covered variables, naming conventions, why to use variables, variable declaration, Re-initializing / re-declaring of variables, blocked scope, use of let & const, Operator & its types in TypeScript. In the next chapter, I will cover enum & types of enum in TypeScript.

# Chapter 5. enum in TypeScript

## Introduction:

We have already learned different data types in the previous chapter. In this chapter we will learn what an enum is, advantages of the enum, types of the enum, reverse mapping, function & interface using enum, const enum, map enum & export enum in TypeScript. In this chapter, we will cover the following:

1. What is enum?
2. Advantages of an enum?
3. Types of enum
4. Reverse mapping
5. Function & interface using enum
6. const enum
7. map enum
8. export enum

### 1. What is enum in TypeScript?

The short form of enumeration is enum. It is used as a type with some set of values called a named constant. Internally it holds numeric type values or in some cases, we can create heterogeneous enums. The enum keyword is used to define the enums. We can create a distinct case so it makes it easy to document.

It forces us to use all possible values. Access the enums using an index or use dot operator after the enum name & named constant. The initial value starts from 0 if we do not set any value. Auto incremental is the best approach to use because we do not need to worry about the values

**Syntax:**

```
enum  enum_Name {
    constant1, constant2, ... constantN
    }
```

**Example:**

Consider the below example, WeekDays are declared as an enum & for Wednesday we are assigning the value as 10 so the next value will be increasingly based on that.

```
enum WeekDays
{
    Monday,
    Tuesday,
    Wednesday = 10,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

console.log("value of enum Monday is: ",WeekDays.Monday);
console.log("value of enum Thursday is: ", WeekDays.Thursday);
```

Execute the above code & we will get this output:

value of enum Monday is:  0

value of enum Thursday is:  11

**2. Advantages of Enums:**

The use of enums makes code more readable and manageable. Enums are a set of named constants. Always starts from 0 if the value is not set. Supports numeric, string & heterogeneous enums.

**3. Types of enums:**

I) Numeric enum

II) String enum

III) Heterogeneous enum

**I) Numeric enum:**

Here enums are numeric based enums, having numeric values in them. The enum keyword is used to declare an enum. Consider the below example having cables like mobile, TV, SetTop box, Landline phone, etc.

**Example:**

A) In this example here are default values assigned to enums.

```
enum Cables
{
    Mobile,
    TV,
    SetTopBox,
    LanlinePhone
}
console.log("value of enum Mobile is: ",Cables.Mobile);
console.log("value of enum TV is: ",Cables.TV);
console.log("value of enum SetTopBox is: ",Cables.SetTopBox);
```

Execute the above code & we will get this output:

value of enum Mobile is:  0

value of enum TV is:  1

value of enum SetTopBox is:  2

B) In this example values are assigned based on our requirement to enums.

```
enum Cables
{
    Mobile = 20,
    TV = 30,
    SetTopBox = 25,
    LanlinePhone = 34
}
console.log("value of enum Mobile is: ",Cables.Mobile);
console.log("value of enum TV is: ",Cables.TV);
console.log("value of enum SetTopBox is: ",Cables.SetTopBox);
```

Execute the above code & we will get this output:

value of enum Mobile is:  20

value of enum TV is:  30

value of enum SetTopBox is:  25

C) **Passing a function in enums:**

**Scenario One:**

Consider the below example. We can pass a function to enum which returns a numeric value as:

```
function calculateLength(): number{
    return 100;
}
enum Cables
{
    Mobile = 20,
    TV = calculateLength(),
    SetTopBox = 25,
    LanlinePhone = 34
}
console.log("value of enum Mobile is: ",Cables.Mobile);
console.log("value of enum TV is: ",Cables.TV);
console.log("value of enum SetTopBox is: ",Cables.SetTopBox);
```

Execute the above code & we will get this output:

value of enum Mobile is:  20

value of enum TV is:  100

value of enum SetTopBox is:  25

**4. Reverse Mapping:** We can access the value of enum from their value directly.

Example: consider the below example for reverse mapping.

```
enum Employee
{
    Name = "Sam",
    Id =  1010,
    Is_Active = 20,

    Is_Delete,
    DeptId
```

```
}
console.log("value of enum Name is: ",Employee.Name);
console.log("value of enum Name is: ",Employee["Name"]);
console.log("value of enum Is_Active is: ", Employee[20]);
```

Execute the above code & we will get this output:

value of enum Name is:  Sam

value of enum Name is:  Sam

value of enum Is_Active is:  Is_Active

### 5. Function & interface using enum

We are going to learn Interface in detail in the next chapter, here we will just consider the below example only for reference purposes. clothSize is an enum which contains some constants. IClothSize is an interface that contains two properties, key & value, having string & number type respectively. The function is taking enum as a parameter & will print the data on the console.

```
enum clothSize {
    small,
    medium,
    large
}
interface IClothSize {
    key: string,
        value: number
}

function getClothSize(size: clothSize): IClothSize {
    switch (size) {
        case clothSize.small:
            return {
                key: clothSize[clothSize.small], value: 10
            };
        case clothSize.medium:
            return {
                key: clothSize[clothSize.medium], value: 20
            };
```

```
        case clothSize.large:
            return {
                key: clothSize[clothSize.large], value: 30
            };
    }
}
console.log("the cloth is " + getClothSize(clothSize.small).key + " & the
value is " + getClothSize(clothSize.small).value);
console.log("the cloth is " + getClothSize(clothSize.medium).key + " & the
value is " + getClothSize(clothSize.medium).value);
```

Execute the above code & we will get this output:

the cloth is small & the value is 10

the cloth is medium & the value is 20

**6. const enum**

Typescript allows us to declare a const enum using a const keyword. It has inlined values. We can access enum using enum['ConstantName']

**Example:**

```
const enum myColor {
    Red = 10,
        White = Red * 4,
        Blue = White + 10,
        Yellow,
}
console.log(myColor.Red)
console.log(myColor.White)
console.log(myColor.Blue)
console.log(myColor['Yellow'])
```

Execute the above code & we will get this output:

10
40
50
51

We will not be able to access enum using index in const enum like myColor[0], it will give us an error.

6. **map enum:** ES6 allows us to use map enum keys.

Example:

```
enum classes {
    I,
    II,
    III,
    IV,
    V
}
const ClassNames = new Map < number,
    string > ([
        [classes.I, '100'],
        [classes.II, '200'],
        [classes.III, '300'],
    ]);
console.log(ClassNames);
```

Execute the above code & we will get this output:

Map(3) {0 => "100", 1 => "200", 2 => "300"}

We will modify the above example, we can use this into a class as below.

Note

Users can modify the code as per their requirements.

Example

```
enum classes {
    I,
    II,
    III,
    IV,
    V
}
const ClassNames = new Map < number,
    string > ([
        [classes.I, '100'],
        [classes.II, '200'],
        [classes.III, '300'],
    ]);
class AllStandards {
    public allNames: object;
```

```
    constructor() {
        this.allNames = ClassNames;
    }
}
let obj: AllStandards = new AllStandards();
console.log(obj.allNames);
```

Map(3) {0 => "100", 1 => "200", 2 => "300"}

**8. export enum**

TypeScript allows us to export the enum. export keyword is used before the enum. Import is used to import that enum file. We can declare the enum as below.

```
export enum sportActivities {Football, Cricket,Badminton, Tennis}
```

To import the enum in .ts,  i.e.; typescript file looks like this:

```
import {sportActivities} from '../enums'
```

**Summary:** In this chapter, we covered what is an enum, advantages of the enum, types of the enum, reverse mapping, function & interface using enum, const enum, map enum & export enum in TypeScript. In the next chapter, I will cover loops & control statements in TypeScript.

# Chapter 6. Loops & control statements in TypeScript

## Introduction:

Many times while creating an application we need to repeat some conditions. We can use loops, and we can control these loops using break, continuous control statement, iterators & generator in TypeScript. In this chapter we will learn what a loop is, why to use loops, real-life examples, types of loops & how we can control the loops. In this chapter, we will cover the following

1. What is the loop?
2. Why  use loops?
3. Real-life example
4. for loop
5. for...of
6. for..in
7. while loop
8. do-while
9. Loop control statements: A) Break B)  Continue
10. Iterators & Generators

**1. What is the loop?**

A loop in a computer program is an instruction that repeats until a specified condition is reached.

**2. Why use loops?**

The purpose of loops is to repeat the same, or similar, code several times. Because we want to repeat something: like Count from 1 to 100. This number of times could be specified to a certain number depending upon the conditions, or the number of times could be dictated by a certain condition being met.

**3. Real-life Example:**

A) For each customer that has an outstanding balance, send out an email reminder that payment is due.

B) For each directory under this one, find music files and add them to the list of known music.

4. **for loop:** for loop executes the code block for a specified number of times. It is executed based on the conditions on a fixed set of values. We need to pass a starting point, termination condition to this loop.

**Syntax:**

```
for (initial_starting_point; termination-condition; step) {
    //statements
 }
```

**Example:**

A) Consider the below example if the user would like to print the numbers less than 10.

```
let i: number;
let result: string = "Output is : ";
for(i = 0;i < 10;i++) {
    result = result + i + "  ";
}
 console.log(result)
```

Execute the above code & we will get this output:

Output is : 0 1 2 3 4 5 6 7 8 9

Example:

B) We can use for loop: Consider the example below. We will create two variables & use two for loops. The second for loop will print the value of j and will execute only three times as our termination condition is 3.

```
let i: number = 1;
let j: number = 1;
for (i; i <= 5; i++){
    for (j; j <= 3; j++){
    console.log(j);
}
}
```

Execute the above code & we will get this output:

1
2
3

Example:
C) Use of if condition: We can use if conditions based on our logic like we would like to exit the loop if the inner for loop condition is satisfied.

```
let i: number = 1;
let j: number = 1;
for (i; i <= 5; i++){
    for (j; j <= 3; j++){
        console.log("j variable : "+ j);
    }
     if (j <= i)
     {
       break;
     }
    else
     {
      console.log("i variable : "+ i);
    }
}
```

Execute the above code & we will get this output:
j variable : 1
j variable : 2
j variable : 3
i variable : 1
i variable : 2
i variable : 3

5. **for..of:** for iterating over iterable collections. The objects that have a Symbol.iterator property.

**Example:** In the below example, we have defined the array having some values & will print it using for..of in console

```
let employeeArray = [1001, "Sam", "true", false];

for (let item of employeeArray) {
    console.log(item);
}
```

Execute the above code & we will get this output:

1001
Sam
true
false

6. **for..in:** for loops iterating through the properties of an object.

**Example:** Consider the below example which contains an employee object having some properties like firstName, lastName, etc & we will access using the for-in loop:

```
let employee = { firstName: "Sam", lastName: "Joy", address1: "Street Road, I
ND", pinCode: 411010 };
let x: string;
for (x in employee) {
    console.log(x + ": " + employee[x]);
}
```

Execute the above code & we will get this output:

firstName: Sam
lastName: Joy
address1: Street Road, IND
pinCode: 411010

7. **while loop:** while loop is iterating through a block of code while a specified condition is true. While condition becomes false the loop will terminate.

Syntax:

```
while (condition) {
    // statements to be executed
  }
```

Example: A) While loop without any other condition.

```
let i: number = 0;
let result: string = "Output is : ";
while (i < 10) {
    result = result + i + "  ";
    i++;
}
console.log(result);
```

Execute the above code & we will get this output:

Output is : 0 1 2 3 4 5 6 7 8 9

B) While loop using array: We can use array and execute the while loop:

```
let employeeNames: string[]= ["Sam", "Amol", "Ashish", "John"];
  let result: string = "";
  let totalCount : number = employeeNames.length;
  while (totalCount--) {
    result += employeeNames[totalCount] + " ";
}
  console.log(result)
```

Execute the above code & we will get this output:

John Ashish Amol Sam

8. **do-while:** This is the same as while loop, the condition will check at the end of the loop; i.e., the loop will always be executed at least once, even if the condition is false.

Syntax:

```
do {
    // statements to be executed
  }
  while (condition);
```

Example:

```
let result: string= "";
let i: number = 0;
do {
  result += " The number is " + i + "\n";
  i++;
}
while (i < 6);
console.log(result);
```

Execute the above code & we will get this output:

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5

Suppose we change the while condition to while(i < 1) then the loop will execute one time at least due to the fact that the while condition is at the end in the do while statement.

```
let result: string= "";
let i: number = 0;
do {
  result += " The number is " + i + "\n";
  i++;
}
while (i < 1);
console.log(result);
```

Execute the above code & we will get this output:

The number is 0

### 9. Loop control statements:

In loops we need to have control over our loops while using them, so break & continue are the best way.

**A) Break:** The break statement jumps out of a loop. Consider in the below example we will add if condition, so if the condition is true then it will break the loop.

**Example:**

```
let i: number = 0;
let result: string = "Output is : ";
while (i < 10) {
    result = result + i + "  ";
    i++;
    if (i == 5) {
        break;
    }
}
console.log(result);
```

Execute the above code & we will get this output:

Output is : 0 1 2 3 4

**B) Continue:** The continue statement jumps over one iteration in the loop. Consider the below example in which we will add continue in the if condition, so if the condition is true then the loop will continue.

Example:

```
let i: number = 0;
let result: string = "Output is : ";
for (i = 0; i < 6; i++)  {
    result = result + i + "  ";
    if (i === 4) {
        continue;
    }
}
console.log(result);
```

Execute the above code & we will get this output:

Output is : 0 1 2 3 4 5

## 10. Iterator & Generator in TypeScript

### What is an Iterator?

It is common in Object-oriented programming languages. An iterator is an object that exposes three methods, next, returns, and throws. Consider the below syntax, an object which implements the interface. Iterator is useful for iterating. The object has an iteration if it has a Symbol. An iterator method returns an Iterator object. In ES2015 there are some features like for..of

**Syntax:**

```
interface Iterator<T> {
next(value?: any): IteratorResult<T>;
return?(value?: any): IteratorResult<T>;
throw?(e?: any): IteratorResult<T>;
}
```

Symbol.iterator property is already implemented in data types like Array, Map, Set, String, Int32Array, Uint32Array, etc.

**Example:**

1) Using **for..of** is used as like, consider the below example having an array & it will print the elements in the array using for..of.

```
let employeeDetails = ["Sam", "John", 1010, true];

for (let item of employeeDetails ) {
    console.log(item);
}
```

Execute the above code & we will get this output:

Sam
John
1010
true

2) Using **Iterator**: Consider the below example of iterator, will create a function & will pass the starting index,

```
function myIteratorFunction(startingIndex = 0, end = Infinity, steps = 0) {
    let nextIndexValue = startingIndex;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndexValue < end) {
               result = { value: nextIndexValue, done: false }
               nextIndexValue += steps;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}

let myResult = myIteratorFunction(1, 15, 3);

let result = myResult.next();
while (!result.done) {
 console.log("Result of value by increment is: " + result.value);
 result = myResult.next();
}
```

Execute the above code & we will get this output:
Result of values by increment is: 1
Result of value by increment is: 4
Result of value by increment is: 7
Result of value by increment is: 10
Result of value by increment is: 13

**AsyncIterator**: It returns a Promise for the iteration result, rather than the result itself. Consider below syntax, an object which implements the interface.

**Syntax:**

```
interface AsyncIterator<T> {
next(value?: any): Promise<IteratorResult<T>>;
return?(value?: any): Promise<IteratorResult<T>>;
throw?(e?: any): Promise<IteratorResult<T>>;
}
```

**Example:** Consider the below example, we have created an array of numbers having some values assigned at the time of initialization. Here asyncIterator() is used & created as an object of this function. The function will check the length of the array & return a promise with myValue& result.

```
function asyncIterator() {
  let myNumbers = [100, 200, 300];
  return {
    next: function() {
      if (myNumbers.length && myNumbers.length  > 0) {
        return Promise.resolve({
          myValue: myNumbers.shift(),
          result: false
        });
      } else {
        return Promise.resolve({
          result: true
        });
      }
    }
  };
}

var iterator = asyncIterator();

(async function() {
    await iterator.next().then(console.log);
    await iterator.next().then(console.log);
    await iterator.next().then(console.log);
})();
```

Execute the above code & we will get this output:

object { myValue: 100, result: false }
object { myValue: 200, result: false }
object { myValue: 300 result: false }

**What are Generators?**

We can create a generator function using function *. It returns a generator object. It follows the iterator interface (i.e. the next, return and throw functions).

**Why use Generators?**

In previous Typescript versions, we could not identify the value of yield or return from the generator, so Generator type is introduced. The generator easily identifies the value from the iterator ; this is the main advantage.

**Syntax**:

Declare a generator function using asterisk after the function keyword & generator name as follows

```
function* myGenerator() {
  // statement block here...
}
```

**What is Yield's expression?**

Use yield keyword to send a value back to the caller in the statement blocks. This creates a lazy iterator. The function body does not execute the next statement unless and until we call next() & only a single function will resume at a time. The function will pause to execute successfully.

**Why use the Yield?**

Yield allows pausing its communication. It also helps to pass control to an external system. We can push the value from the external system into the generator function body. The external system can throw an exception to the generator function body. A yield & yield * expression has data type any. yield * expression is always assignable to iterable<any>.

**Example**:

```
function* generator(){
    console.log('Execution started');
    yield 0;
    console.log('Execution resumed');
    yield 1;
    console.log('Execution resumed');
}
```

```
var iterator = generator();
console.log('Starting iteration');
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

Execute the above code & we will get this output:

done: false, value: 0
done: false, value: 1
done: true, value: undefined

**Summary:** In this chapter, we covered what a loop is, why use loops, real-life examples, types of loops, how we can control the loops, Iterator & generator in TypeScript. In the next chapter, I will cover Function in TypeScript.

# Chapter 7. Function in TypeScript

## Introduction:

While creating large applications we need to write the functions which will be re-used; anyone can extend the functionality. In this chapter we will learn what a function is, Named function, Anonymous Function, Function Type, Inferring the Types, Default parameter, Optional parameter, Rest parameter & Function overloading in TypeScript programming language. In this chapter, we will cover the following

1. What is the function?
2. Named function
3. Anonymous Function
4. Function Type
5. Inferring the Types
6. Default parameter
7. Optional parameter
8. Rest parameter
9. Function overloading

### 1. What is the function?

These are the main building blocks of any programming language or application. Use these to build a strong application, hide complexity, abstract the information, and hide valuable information from an outsider. Functions are used to do some things / perform the operations easily. We can create a named function or anonymous function in Typescript. It's easy to build our application by using different approaches & easy to understand as well.

### 2. Named function

These can be declared using a named identifier, so this can be used to share in other functions as well. These also help in the reusability of code. Users can pass parameters & it returns a data/data type value based on conditions.

**Syntax:**

```
function function_Name(optional_parameter) :return_type {
     // statement blocks
}
```

Here the function keyword is used. function_Name is the function name used to identify it. The parameter list should be comma-separated. return_type is a data type that will be returned by the function. A statement block is the number of states that would like to execute.

**Example:**

Consider the example below, which returns a full name of a person. We are passing first name & last name to our function.

```
function getFullName(firstName, lastName) {
    return firstName + " " +lastName;
}
console.log(getFullName("John","Kent"))
```

Execute the above code & we will get this output:
John Kent

### 3. Anonymous Function:

These can be used as an inline function. Users can use these functions one-time using function names.

Example:

```
let addition = function(x, y)
 {
return x + y;
};
console.log(addition(5,10));
```

Execute the above code & we will get this output:

15

### 4. Function Type:

In function, there are two main important pillars like the type of the arguments & return type. The function may or may not return the value, it depends upon our requirement.

**Example 1)** Consider the below example, we are passing string parameters as arguments to our function & function returns a combined string value.

```
function getFullName(firstName: string, lastName: string): string {
    return firstName + " " +lastName;
}
console.log(getFullName("John","Kent"))
```

```
//we can write the in different way like
let firstName: string, lastName: string;
function getFullName(firstName, lastName) {
    return firstName + " " +lastName;
}
console.log(getFullName("John","Kent"))
```

Execute the above codes & we will get this output:

John Kent

**Example 2)** Consider the below example, we are passing string parameters to our function & we are just printing a message on console. We will get the same result as above.

```
function getFullName(firstName: string, lastName: string) {
    console.log(firstName + " " + lastName);
}
getFullName("John", "Kent");
```

Execute the above code & we will get this output:

John Kent

We can write a function type as like below

```
let toDoAddition: (a: number, b: number) => number =
    function(a: number, b: number): number { return a + b; };
console.log(toDoAddition(5, 10));
```

Execute the above code & we will get this output:

15

### 5. Inferring the Types

The TypeScript compiler will take care of the type even if we used types at one side. No need to worry about the types while using it. This will save the programmer typing effort. If we used such types, then this is called “contextual typing".

**Example:** Consider the  below examples.

```
let getAddition = function (a: number, b: number): number
{
    return a + b;
};
```

```
console.log(getAddition(10, 20));

let doAddition: (firstValue: number, secondValue: number) => number =
    function (a, b)
    {
        return a + b;
    };
console.log(doAddition(20, 30));
```

Execute the above code & we will get this output:

30
50

### 6. Default Parameter:

No need to maintain the sequence of parameters. When the user uses the function at that time, the user must pass the parameter values.

**Example:**

```
function getValues(a: number, b: number){
    console.log("a: " + a + " & "+ "b: " + b);
}
getValues(10, 20);
```

Execute the above code & we will get this output:

a: 10 & b: 20

**Example:** Consider the below example, we can pass the parameter to the number type as number, null & undefined.

```
function getValues(a: number, b: string){
    console.log("a: " + a + " & "+ "b: " + b);
}
getValues(10, "Welcome!!!");
getValues(null, "Welcome!!!");
getValues(undefined, "Welcome!!!");
```

Execute the above code & we will get this output:

a: 10 & b: Welcome!!!
a: null & b: Welcome!!!
a: undefined & b: Welcome!!!

### 7. Optional Parameter:

Users can declare optional parameters using '?'. This means users don't need to pass a value to the parameter. Users can declare it after the required parameter.

**Example**:

```
function getValues(a: number, b?: number){
    console.log("a: " + a + " & "+ "b: " + b);
}
getValues(10, 20);
getValues(1);
```

Execute the above code & we will get this output:

a: 10 & b: 20
a: 1 & b: undefined

**Example**: Consider the below example, we need to take care about the sequence of parameters as well.

```
function getValues(a?: number, b: number){
    console.log("a: " + a + " & "+ "b: " + b);
}
getValues(10, 20);
getValues(1);
```

Will get the error 'An argument for 'b' was not provided'. We must have to follow the sequence; i.e. optional parameter must be declared after the required parameter.

### 8. Rest Parameter:

Typescript allows users to use rest parameters to use n number of parameters used. Users can use these when they are not sure about the parameters list or unknown parameters. Rest parameters can be denoted by an ellipsis (...) This should be used at the last parameter in the parameter list.

**Syntax:**

```
function function_Name(parameter1: datatype, ...parameter2: datatype)
{
// statement blocks
}
```

**Example:** Consider the below example, we will create one function & pass rest parameters to this function as a number of the array.

```
function getCombinedString(string1: string, ...string2: number[]) {
    return string1+ " " + string2.join(", ");
}
console.log(getCombinedString("Hello !!!! ", 10, 20, 30));
```

Execute the above code & we will get this output:

Hello !!!!  10, 20, 30

**Example:** Users can pass the rest parameter as any typed array so the user can pass any data type values like a number, string, Boolean, etc.

```
function getCombinedString(string1: string, ...string2: any[]) {
    return string1+ " " + string2.join(", ");
}
console.log(getCombinedString("Hello !!!! ", 10, 20, 30, "Sam", "John", true));
```

Execute the above code & we will get this output:
Hello !!!!  10, 20, 30, Sam, John, true

**Example:** Rest parameter must be used at the last parameter in the parameter list otherwise it will give us errors:

```
function getCombinedString(...string2: any[], string1: string) {
    return string1+ " " + string2.join(", ");
}
console.log(getCombinedString(10, 20, 30, "Sam", "John", true, "Hello !!!! "));
```

Will get error 'Rest parameter must be last formal parameter'

### 9. Function Overloading:

Typescript allows us to create function overloading. The function name should be the same with different parameters & return types. We can change the sequence of the parameter as well.

**Example:** Consider the below example, getAddition is a function taking two parameters of any data type & return type as well.

```
function getAddition(x:number, y:number): number;
function getAddition(x:string, y:string):string;
function getAddition(x: any, y:any): any {
    return x + y;
}
console.log(getAddition(100, 200));
console.log(getAddition("Welcome to  ", "CoderFunda.com !!!"));
```

Execute the above code & we will get this output:

300
Welcome to  CoderFunda.com !!!

**Example**: Will pass the parameter in different sequences.

```
function getAddition(x:number, y:number): number;
function getAddition(x:string, y:string):string;
function getAddition(x:string, y:number):string;
function getAddition(x:string, y:number):any;
function getAddition(x: any, y:any): any {
    return x + y;
}
console.log(getAddition(100, 200));
console.log(getAddition("Welcome to  ", "CoderFunda.com !!!"));
console.log(getAddition("Hello !! ", 200));
console.log(getAddition("Welcome !! ", 200));
console.log(getAddition(1000, 5000));
```

Execute the above code & we will get this output:

```
300
Welcome to  CoderFunda.com !!!
Hello !! 200
Welcome !! 200
6000
```

**Summary:** In this chapter, we covered what a function is, Named function, Anonymous Function, Function Type, Inferring the Types, Default parameter, Optional parameter, Rest parameter & Function overloading in TypeScript. In the next chapter, I will cover the Interface in TypeScript.

# Chapter 8. Interface in TypeScript

## Introduction:

TypeScript supports object-oriented principles like Classes, Object, Interface, Polymorphism, etc. Object-oriented principles are useful to reuse the code in the application & the developers who are familiar with the OOPS concept can easily learn TypeScript. In this chapter, we will learn what Interface is and Types of Interfaces. In this chapter, we will cover the following

1. What is Interface?
2. Implementation of an Interface
3. Types of Interface
4. Read-Only properties in Interface
5. Optional Property in Interface

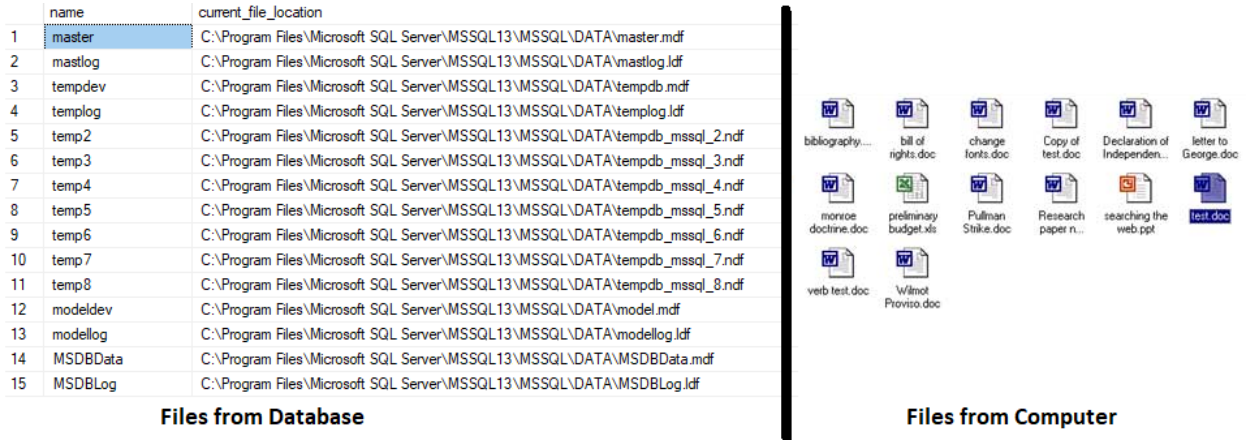
### 1. What is Interface?

An interface is contract specific. It acts as an interaction between two entities. The interface is a description of the function that a class is obliged to do. In interfaces, we can define properties, methods, and events. The properties, methods & events are the members of the interface.

We can only declare the members. Methods are implemented later after the implementation of the interface.

**Real-life example:**

Suppose there is a function to process data, two entities like files stored on the local drive of a computer & files stored in the database. Both the entities are processing a file based on the condition at runtime, so we need an interface that is implemented by both the processors.



| | name | current_file_location |
|---|---|---|
| 1 | master | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\master.mdf |
| 2 | mastlog | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\mastlog.ldf |
| 3 | tempdev | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb.mdf |
| 4 | templog | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\templog.ldf |
| 5 | temp2 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_2.ndf |
| 6 | temp3 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_3.ndf |
| 7 | temp4 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_4.ndf |
| 8 | temp5 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_5.ndf |
| 9 | temp6 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_6.ndf |
| 10 | temp7 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_7.ndf |
| 11 | temp8 | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\tempdb_mssql_8.ndf |
| 12 | modeldev | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\model.mdf |
| 13 | modellog | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\modellog.ldf |
| 14 | MSDBData | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\MSDBData.mdf |
| 15 | MSDBLog | C:\Program Files\Microsoft SQL Server\MSSQL13\MSSQL\DATA\MSDBLog.ldf |



**Files from Database** **Files from Computer**

Important points

- All methods in the interface much have to implement at the time of interface implementation
- The class can implement n number of interfaces at a time.
- It does not have a constructor.

**Syntax**

```
interface interfaceName {
// declaration of properties, function names
}
```

**Example:**

```
interface IEmployee {
    employeeId: number;
    employeeName: string;
}

let emp1: IEmployee = { employeeId:1000, employeeName:"John" };
let emp2: IEmployee = { employeeId:1001, employeeName:"Sam"};

console.log(emp1.employeeId + ", " + emp1.employeeName);
console.log(emp2.employeeId + ", " + emp2.employeeName);
```

Execute the above code & we will get this output:

1000, John
1001, Sam

## 2. Implementation of an Interface

Consider the below example, in programming languages the OOPS concept has an interface and it is implemented using the implements keywords as below, IPerson is an interface & it's implemented in Person class.

The implemented class must have implemented it with the properties & functions with the same data type defined as in the interface. There are properties in an interface like person id, person name & person address. getAddress is a function that takes a numeric parameter & returns a string.

**Example**

```
interface IPerson {
    personId: number;
    personName: string;
    personAddress: string;
    getAddress:(x: number)=>string;
}

class Person implements IPerson {
    personId: number;
    personName: string;
    personAddress: string;

    constructor(id: number, name: string, address: string) {
                this.personId = id;
                this.personName = name;
                this.personAddress = address;
    }

    getAddress(empCode: number):string {
        return this.personAddress;
    }
}

let objPerson = new Person(1001, "Sam","M G Road, Delhi");
console.log("Id: " + objPerson.personId);
console.log("Name: " + objPerson.personName);
console.log("Address: " +objPerson.getAddress(objPerson.personId));
```

Execute the above code & we will get this output:
Id: 1001
Name: Sam
Address: M G Road, Delhi

## 3. Types of Interface

### A. Function Types

Typescript allows us to define the function in Interface & interface is used to define a type of a function. Consider the below example, we have defined some properties & functions in interface & we are accessing it.

**Example:**

```
interface IEmployee {
    employeeId: number;
    employeeName: string;
    getDetails(): any;
}
function getDetails()
{
    return "Hello !!!";
}
let emp1: IEmployee = { employeeId: 1000, employeeName: "John"};
let objForGet: IEmployee = getDetails;

console.log(objForGet());
console.log(emp1.employeeId + ", " + emp1.employeeName );
```

Execute the above code & we will get this output:
Hello !!!
1000, John

We can define the interface used to define a type of a function.

Consider the  below example, Interface contains only the signature of a function having a single parameter & returns a string value. Now we will define the function & will assign that function to the interface variable. While accessing the variable we will pass the parameter.

```
interface IEmployee {
    (employeeId: number): any;
}
function getDetails(employeeId: any)
{
    return "Hello !!!" + " & employee id is: " + employeeId;
}
let emp1: IEmployee = getDetails;

console.log(emp1(1001));
```

Execute the above code & we will get this output:

```
Hello !!!& employee id is: 1001
```

**B. Indexable Types**

We can define the Interfaces as an Indexable Type & it has an Indexable signature. Users can access it via Indexes.

In Typescript, we can access the object via Indexes:

```
let a: object;
a = { x: 1, y: 'true', z: 'Sam'};
console.log(a['x']);
console.log(a['y']);
console.log(a['z']);
```

Execute the above code & we will get this output:

1

true

Sam

Now consider the below example. We will define one Interface IEmployee which contains Indexes & is an object to which we are assigning the values in Indexes. Now we will access the values using Indexes.

```
interface IEnployees {
    [Employees: number]: string;//indexer
}

let s: IEnployees = {1: 'Sam', 2:'John'};
console.log(s);
console.log(s[2]);
```

Execute the above code & we will get this output:

{1: "Sam", 2: "John"}

John

Also we can define the other members & function in above interface as well:

```
interface IEnployees {
    [Employees: number]: string;
    isActive: boolean;
    getEmployeeAddress(): string;
}
```

IEmployee is an interface that has an index signature. IEmployee is indexed with a number & that will return a string. IEmployee also contains property isActive as boolean& a function `getEmployeeAddress` will return a string.

### C. Class Types

**What is the Class?**

Class is a collection of objects. It contains properties and methods. Consider class Person having different properties like personName, personAge, PersonAddress & methods like getPersonDetails().

It is the blueprint of data & behavior, which  means how the data would look & how the functions work.

**Real-life example**

Consider an example. If a person would like to build a house then he first needs to draw a pictorial representation of this house & some functions like  openDoor, useOfLift, etc.

**Syntax:**

```
class ClassName{
// properties
// behaviour
}
```

**What is a Constructor?**

It is used to initialize the members of the class. The constructor keyword is used to define the constructor.

**Example:**

```
class Person{
    id: number;
    constructor(i: number) {
        this.id = i;
    }
}
let p: Person = new Person(100);
console.log(p)
```

Execute the above code & we will get this output:

Person {id: 100}

**Example:**

Consider the below example. interface IEmployee contains some properties & we will declare one class, Employee, which will implement the IEmployee interface. The constructor of class will assign the values to properties.

```
interface IEnployees {
    employeeId: number;
    employeeName: string;
    isActive: boolean;
}
class Employee implements IEnployees{
    eId: number;
    eName: string;
    eActive: boolean;
constructor (eId: number, eName: string, eActive: boolean) {
  this.employeeId = eId;
  this.employeeName = eName;
    this.isActive = eActive;
 }
}
let objEmployee: Employee = new Employee(1001, "Sam Sinha", true);
console.log(objEmployee);
```

Execute the above code & we will get this output:

Employee {employeeId: 1001, employeeName: "Sam Sinha", isActive: true}

### D. Hybrid Types

While creating an application we need to create an object that will act as a function and an object. In some third-party applications, we need to implement such kinds of scenarios.

Consider the below example. We have defined two interfaces, and one function. For the object of INewSearch interface, it will pass the parameters to functions & if string search finds the first substring match in a regular expression search then it returns a boolean value.

```
interface ISearchFunction {
  (a: string, b: string): boolean;
}

interface INewSearch extends ISearchFunction {
    name: string;
}

function getReturnData() : INewSearch {
    let result: any = function (a:string, b:string) : boolean {
        var result = a.search(b);
        if (result == -1) {
            return false;
        }
        else {
            return true;
        }
    };
    return result;
}

let objMySearch: INewSearch = getReturnData();
console.log(objMySearch("input is valid", "put"));
```

Execute the above code & we will get this output:
true

### E. Extend Interface

As we know we can extend the classes, so interfaces also can be extended to each other. Users can copy the members of one interface into another by declaring multiple interfaces. It helps the user to reuse components.

**Example:**
Consider the below example, we have defined the two interfaces IDepartment & IPerson having some properties & will extend IDepartment into aIPersoninterace. So while creating an object of IPerson we will be able to access the property of IDepartment interface.

```
interface IDepartment {
    deptId: number;
}
```

```
interface IPerson extends IDepartment {
    personId: number;
}

let person = {} as IPerson;
person.deptId = 1001;
person.personId = 2001;

console.log(person);
```

Execute the above code & we will get this output:
{deptId: 1001, personId: 2001}

We can extend multiple Interfaces as like below:

```
interface ISector {
    sectorName: string;
}

interface IDepartment {
    deptId: number;
}

interface IPerson extends IDepartment, ISector {
    personId: number;
}

let person = {} as IPerson;
person.deptId = 1001;
person.personId = 2001;
person.sectorName = "IT";
let person1 = {} as IPerson;
person1.deptId = 1002;
person1.personId = 2002;
person1.sectorName = "Govt";

console.log(person);
console.log(person1);
```

Execute the above code & we will get this output:

```
{deptId: 1001, personId: 2001, sectorName: "IT"}
{deptId: 1002, personId: 2002, sectorName: "Govt"}
```

4. **Read-Only properties in Interface**

Read-Only property can't be changed once it is initialized. When the user would like to use some fixed values to the property in this case they can use read-only.

```
interface Circle {
    readonly pieValue: number;
    getString(): string;
}
function getString(): string{
    return "Hello !!!"
}
```

5. **Optional Property in Interface:**

In the interface, we can define the optional properties. So, we need to assign the value.

Consider the below example, we will add one property; i.e. personality, which will be optional. So, no need to assign the value in the constructor.

```
interface IPerson {
    personId: number;
    personName: string;
    personAddress: string;
    personSalary?: number;
    getAddress:(x: number)=>string;
}

class Person implements IPerson {
    personId: number;
    personName: string;
    personAddress: string;


    constructor(id: number, name: string, address: string) {
                this.personId = id;
                this.personName = name;
                this.personAddress = address;
    }

    getAddress(empCode:number):string {
        return this.personAddress;
    }
}

let objPerson = new Person(1001, "Sam","M G Road, Delhi");
console.log("Id: " + objPerson.personId);
```

```
console.log("Name: " + objPerson.personName);
console.log("Address: " + objPerson.getAddress(objPerson.personId));
```

Execute the above code & we will get this output:
Id: 1001
Name: Sam
Address: M G Road, Delhi

**Summary:** In this chapter, we covered the major concepts of Object-oriented programming language which are supported by Typescript; i.e., Interfaces, Types of Interfaces & property declarations in Interface in TypeScript. In the next chapter, I will cover Classes & some object-oriented concepts in TypeScript.

# Chapter 9. Classes in TypeScript

## Introduction:

TypeScript supports OOPS concepts so we can easily define the Classes, Objects, Constructors, Access Modifiers, Accessors, Inheritance, Types of Inheritance, Polymorphism, Property Declarations in TypeScript. Class is the building block of every application that contains properties, methods, how we inherit the class, how to override the methods, how to use access modifiers, etc. In this chapter, we will cover the following

1. Classes
2. Objects
3. Constructors
4. Access Modifiers
5. Accessors
6. Inheritance
7. Type Of Inheritance
8. Polymorphism
9. Property Declarations

### 1. Classes

**What is the Class?**

Class is a collection of objects. In object-oriented programming languages, some fundamental concepts are useful to reuse the code & make an application better. It contains properties and methods. Consider class Person having different properties like personName, personAge, PersonAddress& methods like getPersonDetails(). It is the blueprint of data & behavior means how the data would look & how the functions work. Class object defines what kind of data & which kind of functionality it can contain.

**Real-life example**

1) Light bulb

A light bulb has some characteristics like the coil, cover, below surface etc., and some behavior like bulb ON / OFF.

2) Construct a house

Consider an example, If a person would like to build a house then he first needs to draw a pictorial presentation of this house & some functions like openDoor, useOfLift, etc.

Syntax:

```
class ClassName{
// properties
// behaviour
}
```

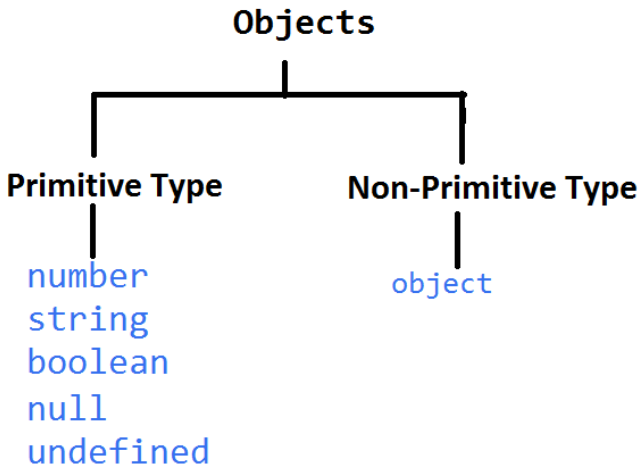
Example: Consider the below example. Person is a class containing id property and a constructor & personDetails is a function/method which returns an id value.

```
class Person{
    id: number;
    constructor(i: number) {
        this.id = i;
    }
personDetails()
{
        return this.id;
}
}
```

## 2. Objects

The object is an instance of the class. Objects have state & behavior. Users can create an instance of a class using a new keyword as well. Consider in real life: A human is an object which has states like name,

age, color, etc & behavior like walk, talk, speak, etc. object (lowercase) is used for non-primitive types & Object (uppercase) used for functionality which is common to use.




The object used in common:

```
interface Object {
    personId: number;
    personName: string;
    personAddress: string;
    getAddress:(x: number)=>string;
}
```

**Empty Type:**

It is used to declare when no member is inside it.

```
let obj = {}
```

When we are trying to assign any value to empty type then the compiler will give us an error like Property 'id' does not exist on type '{}'

**Example:**

```
let obj = {}
obj.id = 100;
```

**Example**: Consider the below example. Person is a class which contains property id & constructor. We have created an object of Person class using the new keyword.

```
class Person{
    id: number;
    constructor(i: number) {
        this.id = i;
    }
}
let p: Person = new Person(100);
```

**Object Literal Notation**

Consider the below example, the employee object contains some properties & we can access these using (dot) . Operator like

```
let employee = {
   id: 1000,
   name:"John Kent",
   address:"US"
};
console.log(employee.id)
console.log(employee.name)
console.log(employee.address)
```

Execute the above code & we will get this output:
1000
John Kent
US

Consider the below example we can access the element using Indexable type like

Example;

```
let a: object;
a = { x: 1, y: 'true', z: 'Sam'};
console.log(a['x']);
console.log(a['y']);
console.log(a['z']);
```

Execute the above code & we will get this output:

1
true
Sam

**Object as a function parameter**

Consider that the below example is an object literal. Function myFunction is taking an object as a parameter while calling the function will pass the employee object.

Example:

```
let employee = {
   id: 1000,
   name:"John Kent",
   address:"US"
};
let myFunction = function(objectType: { id:number, name: string, address :string }) : any {
console.log(objectType.id)
console.log(objectType.name)
console.log(objectType.address)
}
myFunction(employee)
```

Execute the above code & we will get this output:
1000
John Kent
US

## 3. Constructor

**What is a Constructor?**

It is used to initialize the members of the class. The constructor keyword is used to define the constructor.

Example:

**1)** Consider the below example

```
class Person{
    id: number;
    constructor(i: number) {
```

```
        this.id = i;
    }
}
let p: Person = new Person(100);
console.log(p)
```

Execute the above code & we will get this output:
Person {id: 100}

**2)** Consider the below example. Interface contains a method, class, which is having some properties. The constructor will assign values to these properties & after the creation of the object it will print the values into a console.

```
interface IPerson {
   getAddress:(x: number)=>string;
}
class Person implements IPerson {
    personId: number;
    personName: string;
    personAddress: string;
    constructor(id: number, name: string, address: string) {
                this.personId = id;
                this.personName = name;
                this.personAddress = address;
    }
    getAddress(empCode:number):string {
        return this.personAddress;
    }
}
let objPerson = new Person(1001, "Sam","M G Road, Delhi");
console.log("Id: " + objPerson.personId);
console.log("Name: " + objPerson.personName);
console.log("Address: " +objPerson.getAddress(objPerson.personId));
```

**Constructor function**: Below is the construction function for the person object

Example:

```
function Person(name, address, pinCode) {
    this.Name = name;
    this.Address = address;
```

```
    this.PinCode = pinCode;
  }
  var firstPerson = new Person("John", "Pune", 41101,);
  console.log(firstPerson.Name);
  console.log(firstPerson.Address);
  console.log(firstPerson.PinCode);
```

Output:
John
Pune
41001

**Built-in constructors**: TypeScript has built-in constructors for native objects like below.

Example:

```
let A = new Object();
let B = new String();
let C = new Number();
let D = new Boolean();
let E = new Array();

  console.log(A);
  console.log(B);
  console.log(C);
  console.log(D);
  console.log(E);
```

Output:
{}
String {""}
Number {0}
Boolean {false}
[]

## 4. Access Modifiers

Typescript supports access modifiers which is Encapsulation. It is used to make members accessible in the class or outside the class. Below are the access modifiers.

1) Public:

By default, all class members are public in Typescript. These members can be accessible anywhere.

Example:

```
class Person{
    public Id: number;
    public Name: string;
    Address: string;
    constructor(id:number, name: string, address: string)
    {
        this.Id = id;
        this.Name = name;
        this.Address = address;
    }
}
let obj = new Person(120, "Sam","Pune");
console.log(obj.Id)
console.log(obj.Name)
console.log(obj.Address)
```

Execute the above code & we get output like:
120
Sam
Pune

2) Private:

These members are only accessible inside the class. Outside the class, these are not accessible.

3) Protected:

These members are accessible only to the derived type.

Example:

```
class Person{
    public Id: number;
    private Name: string;
    protected Address: string;
    constructor(id:number, name: string, address: string)
    {
        this.Id = id;
        this.Name = name;
        this.Address = address;
    }
}
let obj = new Person(120, "Sam","Pune");
console.log(obj.Id)
console.log(obj.Name)
console.log(obj.Address)
```

Will get an error when we try to access the Name & Address as:
Property 'Name' is private and only accessible within class 'Person'
Property 'Address' is protected and only accessible within class 'Person' and its subclasses.

Will modify the above code as:

We will create one more class as Details and extend the Person class. In detail, the class will write one method which will print the base class member values. We are not able to access the private members in the derived class.

```
class Person{
    public Id: number;
    private Name: string;
    protected Address: string;
    public constructor(id: number, name: string, address: string)
    {
        this.Id = id;
        this.Name = name;
        this.Address = address;
    }

}
class Details extends Person{
    constructor(a: number,b: string, c: string){
        super(a,b,c);
    }
```

```
    getData(){
        console.log(this.Id);
        //console.log(this.Name);
        console.log(this.Address);
    }

}
let obj = new Details(120,"Sam","Pune")
obj.getData();
```

Execute the above code & we will get this output:

120
Pune

## 5. Accessors

Typescript allows us to get & set the members of the value inside the class. getter & setter are two methods

1) **Getter:** Will retrieve the value of the member. get keyword is used to get the value.

Syntax:

```
get PropertyName()
{
    // code
}
```

Example:

```
class Person {
    id: number;
```

```
    private name: string;
    protected address: string;
    constructor(i: number, n: string, a: string){
        this.id = i;
        this.name = n;
        this.address = a;
    }
    get getDetails(): any {
        return this.id + " " + this.name + " " + this.address;
    }
}
let obj = new Person(1001, "Sam","Pune")
console.log(obj.getDetails);
```

Output

1001 Sam Pune

Also we can write as below:

```
class Person {
    id: number = 1002;
    private name: string ="Sam";
    protected address: string = "Pune";

    get getDetails(): any {
        return this.id + " " + this.name + " " + this.address;
    }
}
let obj = new Person()
console.log(obj.getDetails);
```

Output

1002 Sam Pune

2) **Setter:** We can set the value to members using the set keyword.

Syntax

```
set PropertyName()
{
    pName = value
}
```

Example:

```
class Person {
    name: string;

    set setName(n: string) {
        this.name = n;
    }
}
let obj:Person = new Person();
obj.name = "Sam";
console.log(obj.name);
```

Output:

Sam

## 6. Inheritance

TypeScript allows us to use the concept of Inheritance. Users can create new classes from the existing class. The existing class can be called the base class & the new class can be called a derived class. By using 'extends' keyword users can inherit the base class; i.e. all properties and methods will be accessible in derived class & constructors from the base class. In Inheritance users can access the base class properties as well as methods, if the user would like to add more functionality to that base method then he can easily modify or extend the methods.

**Real-life example:**

Consider a real-life example. In the old days there was a watch used to identify the time, so watches had some functions like calculate hours, minutes, etc. Now the same methods & properties are used in a smartwatch with some additional functionality, here Inheritance is used.


Old Watch


Smart Watch

**Why Inheritance:**

It is used to code reusability which means we can add or remove the existing functionality easily. We can override the base methods without changing the parent class. Inheritance hierarchy represents an "is-a" relationship and not a "has-a" relationship. We can easily make global changes to derived classes by changing a base class.

**Syntax:**

```
class newClass extends existingClass
```

Example:

```
class person{
    id: number;
    name: string;
    constructor(i: number, n: string)
    {
        this.id = i;
        this.name = n;
    }
    getDetails(): void {
```

```
      console.log(this.id +" , " +  this.name)
   }
}
class personDetails extends person
{
    let obj = new person(100, "Sam");
    obj.getDetails();
}
```

Output:

100, Sam

### 7. Types of Inheritance

**1) Single Level Inheritance:** In which one class can be extended into another class.

Consider the below example, a base class person having properties & one constructor to initialize these all properties. We will extend the person class into personDetails class & create an object of the base class by passing parameter values.

```
class person{
    public id: number;
    private name: string;
    protected address: string;
    constructor(i: number, n: string, a: string)
    {
        this.id = i;
        this.name = n;
        this.address = a;
    }
    getDetails():void {
      console.log(this.id +" , " +  this.name +" , " +  this.address)
   }
}
class personDetails extends person
{
    let obj = new person(100, "Sam","Pune");
    obj.getDetails();
```

```
}
```

Output:

100, Sam, Pune

**2) Multi Level Inheritance:** In which we can extend more than two levels of Inheritance. In the last derived class, we can access the elements of first base class & we can initialize all.

```
class person{
    id: number = 0;
    name: string = "";
    address: string = "";
    getDetails():void {
      console.log(this.id +" , " +  this.name +" , " +  this.address)
   }
}
class personDetails extends person
{
}

class personAllDetails extends personDetails
{
    let obj = new personAllDetails();
    obj.id = 1001;
    obj.name = "Sam";
    obj.address = "Pune";
    obj.getDetails();
}
```

Output:

1001, Sam, Pune

## 8. Polymorphism

Poly means single name & morphism means many forms. It is one of the important object-oriented concepts. It means function has the same name & multiple meanings. It means the name of the function is the same but its definitions are different; i.e. function overloading. There are different rules for the naming declaration used by the different compilers.

**Function Overloading**

It means we can overload the function using the same function name having a different parameters list. Function overloading means function redefinition. The return type should be different.

Example:

In the below example, function overloading is used, doAdditiontakes two parameters & returns any data type value. First function is taking two string parameters & the second function is taking two numeric parameters.

```
function doAddition(a: string, b: string): any;
function doAddition(a: number, b: number): any;
function doAddition(a: any, b: any): any {
    return a + b;
}
console.log(doAddition("Welcome", " to Typescript !!!"));
console.log(doAddition(10, 20));
```

Output:

Welcome to Typescript !!!

30

We can't achieve the function overloading by fewer parameter lists or more parameter lists as:

```
function doAddition(a: number): any;
```

```
function doAddition(a: string, b: string): any;
function doAddition(a: string, b: string, c: number): number;
function doAddition(a: any, b: any): any {
    return a + b;
}
console.log(doAddition("welcome", " to Typescript !!!"));
console.log(doAddition(10));
console.log(doAddition(10, 20, 30));
```

**Function Override:**

Function override means the base class function can be overridden in the derived class. The function call depends upon the object creation of the class.

Example:

```
class baseClass{
    doTask(): void{
        console.log("Test")
    }
}
class derivedClass extends baseClass{
    doTask(): void{
        console.log("Hello");
    }
}

let obj: derivedClass = new derivedClass();
let obj_1: derivedClass = new baseClass();
console.log(obj.doTask());
console.log(obj_1.doTask());
```

Output:
Hello
Test

## 9. Property Declarations in TypeScript

## A. Optional Properties

Typescript allows us to define the optional property in the class. By using the '?' symbol we can define the optional property. It means there's no need to assign the value to that property, it is optional. Many times users aren't aware of the value.

**Syntax:**

```
propertyName? : datatype;
```

**Example:** Consider the below example, the name is an optional property. In the constructor, we have initialized the class members.

When we create an object at that time, we will decide whether to pass a value or not. For first object creation we are not passing name value to Person constructor, in second object creation we are passing name value to Person constructor.

```
class Person{
    id: number;
    name?: string;
    constructor(i: number, n?: string) {
        this.id = i
        this.name = n;
    }
}
let obj: Person = new Person(1000);
console.log(obj.id)
console.log(obj.name)

let obj_1: Person = new Person(1001,"Sam");
console.log(obj_1.id)
console.log(obj_1.name)
```

Execute the above code & we will get this output:
1000
undefined
1001
Sam

The difference between optional parameter & parameter type is we need to supply value in parameter type.

Optional property can be used in class, construction, function, interfaces etc.

```
function getValueWithOptional(a?: number) {
```

```
    return a;
}
function getValueWithoutOptional(a: number | undefined) {
    return a;
}
getValueWithOptional();      // works fine
getValueWithoutOptional(1);  // works fine
getValueWithoutOptional();  // this function expect at least one parameter
```

## B. Readonly Properties

Typescript allows us to declare the property as readonly. The 'readonly' keyword is used to declare the property. When we create the first object at that time we can assign the value. After that, we can't modify the property.

**Syntax:**

```
readonly propertyName: datatype;
```

**Example:**

In the below example, we have defined the read-only property, while creating the object we are assigning the value to it. After that we can't modify the value and will get an error like "Cannot assign to 'id' because it is a read-only property."

```
class Person{
     readonly id: number;
    constructor(i: number) {
        this.id = i
    }
}
let obj: Person = new Person(1000);
console.log(obj.id)
obj.id = 2000;
```

## C. Static Properties

The static keyword is used to define static property. Static members of the class are accessible without creating an object of the class. We can directly use static members by using. (dot) operator. We can reinitialize the values of static members.

**Syntax:**

```
static propertyName: datatype;
```

**Example:**

```
class Person{
     static id: number;
}
Person.id = 1000;
console.log(Person.id)
Person.id = 2000;
console.log(Person.id)
```

Execute the above code & we will get this output:
1000
2000

**Summary:** In this chapter, we covered major concepts of Object-oriented programming language which are supported by Typescript like Classes, Objects, Constructors, Access Modifiers, Accessors, Inheritance, Types of Inheritance, Polymorphism, Property Declarations in TypeScript. In the next chapter, I will cover Generics in TypeScript.

# Chapter 10. Generics in TypeScript

## Introduction:

While creating large applications, for functions we need to pass the parameter, which has different types & it should be type-safe. These parameters will sometimes have a single value or multiple values; or sometimes will have different values with different data types, So according to that, we need to build the functions. TypeScript supports generic class, generic function, generic interface, etc. In this chapter, we will cover the following

1. What are Generics?
2. Identity function using generic type
3. Key-Value Pair
4. Generic Type
5. Function & properties of generic type
6. Generic Class
7. Generic Constraints
8. Generic Interface

### 1. What are Generics?

Object-oriented programming is used to build in such a way that users can easily reuse the code. While creating an application we need to repeat the code to handle different conditions so we can create a generalized method. Generics have a feature to create reusable components. Typescript allows us to use it in function, interfaces & classes as well. Consider the below examples, suppose we have a function which returns a number & boolean types values.

#### The Hello World of Generics

```
function getData(A: number): number{
        return A;
}

function getData(B: boolean): boolean{
        return B;
}
```

Now above both of the functions are doing the same job.

The identity function is a function which accepts a specific type & returns an argument.

Example:

**Case 1:**

```
function identity(arg: number): number {
    return arg;
}
console.log(identity(1000));
```

Output
1000

**Case 2:**

Suppose we would try to pass a string parameter to the above function then we will get an error like Argument of type '"hello"' is not assignable to parameter of type 'number'.

```
function identity(arg: number): number {
    return arg;
}
console.log(identity(1000));
console.log(identity("hello"));
```

**Case 3:**

Typescript allows us to create a function using identity() which will take a type of parameter & return an argument.

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

2. **Identity function using generic type:**

This is not type-safe, so  make a type-safe use type of variable. This is used when the user must pass multiple parameters to a function.

Here T is used for Type. To pass a generic type we use angled brackets <> after the name of our function.

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

Users can pass the multiple generic types to a function & return type T remains the same as it is.

```
function identities<T, U>(arg1: T, arg2: U): T {
   return arg1;
}
```

Example: In the below identity function we are passing different types of parameters.

```
function identity(arg: any): any {
    return arg;
}
 console.log(identity(1001));
 console.log(identity("Hello world"));
 console.log(identity(true));
let employeeDetails = ["Sam", "John", 1010, true];
console.log(identity(employeeDetails));
```

Execute the code & we will get this output:
1001
Hello world
true
(4) ["Sam", "John", 1010, true]

3. **Key-Value Pair:**

Consider the below example, assignValues is a generic function that takes different types of parameters. The class contains key &val properties that are initialized in function.

```
class myKeyValue<T, U>
{
    public key: T;
    public val: U;

    assignValues(key: T, val: U): void {
        this.key = key;
        this.val = val;
    }
}

let obj1 = new myKeyValue<number, string>();
obj1.assignValues(1, "Sam");
console.log("key is: " + obj1.key + " & values is: " + obj1.val);

let obj2 = new myKeyValue<string, string>();
obj2.assignValues("John", "kent");
console.log("key is: " + obj2.key + " & values is: " + obj2.val);
```

### 4. Generic Type

Non generic functions would be declared by passing the parameter argument which will be returned from the identity function.

```
function identity<T>(arg: T): T {
    return arg;
}
let myIdentity: <T>(arg: T) => T = identity;
console.log(myIdentity(10))
console.log(myIdentity("Hello Word"))
console.log(myIdentity("Hello","Word"))
console.log(myIdentity(["Sam", "John", 1010, true]))
```

Execute the code & we will get this output:

10

Hello Word

Hello

["Sam", "John", 1010, true]

["Sam", "John", 1010, true]

In the above section, we have checked how identity function takes a different type of parameter. Now suppose we would like to print the length on the console; then we will use arg.length

For number data type we will get an error "Property 'length' does not exist on type 'T'" if we calculate the length.

```
function identity<T>(arg: T): T {
    console.log(arg.length);
    return arg;
}
 console.log(identity(1001));
```

Property 'length' does not exist on type 'T'

Now, we will modify the argument type as array & will get the length of the argument.

**Example:**

```
function identity<T>(arg: T[]): T[] {
     console.log(arg.length);
     return arg;
}
let employeeDetails = ["Sam", "John", 1010, true];
console.log(identity(employeeDetails));
```

**Output:**

Identity function returns an argument as:

4

(4) ["Sam", "John", 1010, true]

If we try to pass a number to the above function as argument then we will get the error "Argument of type '1001' is not assignable to parameter of type 'unknown[]'"

```
function identity<T>(arg: T[]): T[] {
     console.log(arg.length);
     return arg;
}
console.log(identity(1001));
```

5. **Function & properties of generic type:**

Typescript allows us to use the built-in functions of properties. Users can pass the parameter to the function & use the built-in functions in generic type functions. For function, the user can pass any type of parameter.

**Example:**

```
let worldString : string = 'Hello, world';

console.log(worldString.blink());
console.log(worldString.bold());
console.log(worldString.italics());
console.log(worldString.strike());
console.log(worldString.lastIndexOf("world"));

function getData<T, U>(id:T, name:U): void {
    console.log(id.toString());
    console.log(name.toString());
    console.log("the id is of " + typeof (id) + " & name is of " + typeof (name) + "
type");
    //console.log(name.toFixed());
    //console.log(name.toUpperCase());
}
getData(2000, "Hi !!!");
```

Execute the above code & we will get this output:

<blink>Hello, world</blink>
Hello, world

Hello, world
<strike>Hello, world</strike>
7
2000
Hi !!!
the id is of number & name is of string type

In the above code for generic function we will get error for the below code:

```
console.log(name.toFixed());
console.log(name.toUpperCase());
```

In the above function we can pass the different parameters as:

```
function getData<T, U>(id: T, name: U): void {
    console.log(id.toString());
    console.log(name.toString());
    console.log("The id is of " + typeof (id) + " & name is of " + typeof (name) + "
type");

}
getData(2000, "Hi !!!");
getData("Welcome to Typescript", true);
getData(1001, 2002);
```

Execute the above code & we will get this output:
2000
Hi !!!
The id is of number & name is of string type
Welcome to Typescript
true
The id is of string & name is of boolean type
1001
2002
The id is of number & name is of number type

## 6. Generic Class

Typescript allows us to create a generic class. Users can pass the parameters in the angular brackets. The class contains properties & methods. Specific data types allow us in the class, so the code can be reusable. Passing a parameter is type-safe & when the class is used in different data types it will be useful.

Consider the below example, employee class is a generic class in which emp is one member & in the constructor will assign the value to it. We will pass the array while creating an object & these contain different data type values.

```
class employee<T> {
    public emp: T[];

    constructor(arg: T[]) {
    this.emp = arg;
  }
}
let obj_1 = new employee(["Sam", "John", 1010, true]);
console.log(obj_1.emp);
let obj_2 = new employee([10,20,30,40,50]);
console.log(obj_2.emp);
```

Execute the above code & we will get this output:
(4) ["Sam", "John", 1010, true]
(5) [10, 20, 30, 40, 50]

Consider the below example, class contains generic members agr1, agr2 & one function which will print the values of the members. The constructor will assign the values to the members. Generic class has parameters passed using angle brackets after the class name.

```
class myGeneric<T,U>
{
    public arg1: T;
    public arg2: U;
    constructor(arg1: T, arg2: U) {
        this.arg1 = arg1;
        this.arg2 = arg2;
    }
```

```
    getValues():void {
        console.log(`arg1 : ${this.arg1}, arg2 : ${this.arg2}`);
    }
}

let obj_1 = new myGeneric<number, string>(1000,"Sam");
obj_1.getValues();
let obj_2 = new myGeneric<string, string>("John","USA");
obj_2.getValues();
let obj_3 = new myGeneric<boolean, number>(true, 2001);
obj_3.getValues();
```

Execute the above code & we will get this output:

arg1 : 1000, arg2 : Sam

arg1 : John, arg2 : USA

arg1 : true, arg2 : 2001

### 7. Generic Constraints:

It checks which type of argument we are passing as an argument. Without constraints, it will take any argument. By using constraints, we can increase the number of operations. The method can be called by the generic class.

In the above two examples, we know the types of data. Now suppose the user would like to print the length of argument, then we will modify the class as below. Typescript will not print the value on the console because every argument does not have a valid length.

```
class employee<T> {
    public emp: T[];

    constructor(arg: T[]) {
    this.emp = arg;
    }

    getData<T>(arg: T[])   {
        console.log(this.T.length);
    }
}
let obj_1 = new employee(["Sam", "John", 1010, true]);
obj_1.getData(obj_1.emp);
let obj_2 = new employee([10,20,30,40,50]);
obj_2.getData(obj_2.emp);
```

We will modify the above code to print the length of the argument.

```
    class employee<T> {
    public emp: T[];

    constructor(arg: T[]) {
    this.emp = arg;
    }

    getData<T>(arg: T[])   {
         console.log("length: ",this.emp.length);
    }
}
let obj_1 = new employee(["Sam", "John", 1010, true]);
obj_1.getData(obj_1.emp);
let obj_2 = new employee([10,20,30,40,50]);
obj_2.getData(obj_2.emp);
```

Execute the above code & we will get this output:
length: 4
length: 5

### 8. Generic Interface

**Interface**

Typescript allows us to use a generic class user interface, we can implement an interface in the class. Generic interface types can also access multiple generic types. Key-value pairs are also allowed in the class & we do not need to worry about it.

**Generic Interface:** Consider the below interface. Identity function will pass the generic parameter which returns the argument.

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

```
}
let myId_1: IGenericInterface = identity;
console.log(myId_1(10))
console.log(myId_1("Hello Word"))
console.log(myId_1("Hello","Word"))
```

Execute the above code & we will get this output:

```
10
Hello Word
Hello
```

Consider the below example, we have an interface IEmployee having two properties & function employeedetails extends that interface. If we try to access the length of an argument, then it will give us an error.

```
interface IEmployee {
    ID: number;
    Name: string;
}

function employeeDetails<T extends IEmployee>(arg1: T): T {
    console.log(arg1.length);
    return arg1;
}
```

For the time being comment console.log(arg1.length); in employeeDetails function & execute the above function:

```
console.log(employeeDetails({ID: 100, Name: "Sam"}))
console.log(employeeDetails({ID: 200, Name: "John"}))
```

Will get the output as:
{ID: 100, Name: "Sam"}
{ID: 200, Name: "John"}

Consider the below example. IEmployee is an interface, Class employee implements an interface. While creating an object  either will pass the type.

```
interface IEmployee<T, U>
{
    identity(arg1: T, arg2: U): void;
};

class Employee<T, U> implements IEmployee<T, U>
{
    identity(arg1:T, arg2:U):void {
        console.log(`arg1 = ${arg1}, arg2 = ${arg2}`);
    }
}

let obj1: IEmployee<number, string> = new Employee();
obj1.identity(1001, 'Sam');
obj1.identity(2002, 'John');
let obj2: IEmployee<string, string> = new Employee();
obj1.identity("Sameer", 'USA');
```

Execute the above code & we will get this output:

arg1 = 1001, arg2 = Sam

arg1 = 2002, arg2 = John

arg1 = Sameer, arg2 = USA

**Summary:** In this chapter, we covered what Generics are, identity function using generic type, Key-Value Pair, Generic Type, function & properties of a generic type, Generic Class, Generic Constraints & Generic Interface in TypeScript. In the next chapter, I will cover Advanced Types in TypeScript.

# Chapter 11. Advanced Types in TypeScript

## Introduction:

TypeScript is a superset of JavaScript, so it supports some advanced types as well. In this chapter, we will learn about Intersection, Union, Type Guards, Nullable Types, Type Aliases & Literal Types in TypeScript. In this chapter, we will cover the following

1. Intersection
2. Union
3. Type Guards
4. Nullable Types
5. Type Aliases
6. Literal Types

### 1. Intersection

It allows us to create intersection types, which means it combines multiple types into a single type. If any one type intersects with another one then this has all the properties from both types. It combines all the properties into one, which means all features are available in a single type. It can be used in classes, interfaces, etc. It allows using optional properties as well.

**Syntax** to define intersection

```
type AB = A & B;
let test: AB;
```

Now the **test** has all properties together from A & B.

**Example**: Consider the below example, having two interfaces with some properties & with an optional property as well. To our function we will pass the intersection of these two interfaces with property values.

```
interface IPerson {
    personId: number;
    personName: string;
    personAddress: string;
```

```
}
interface IDepartment {
    deptId?: number;
    deptName: string;
    deptLocation: string;
}
function getDetails(myInter: IPerson & IDepartment): void {
    console.log(myInter)
}
let myInter: IPerson & IDepartment = {personId: 1001, personName: "Sam", personAddress:
"India", deptName: "Engg", deptLocation: "1st Floor"};
getDetails(myInter);
let myInter1: IPerson & IDepartment = {personId: 1001, personName: "Sam", personAddress:
"India", deptId : 2001, deptName: "Engg", deptLocation: "1st Floor"};
getDetails(myInter1);
```

Output:
{personId: 1001,
personName: "Sam"
personAddress: "India"
deptName: "Engg"
deptLocation: "1st Floor"}

{personId: 1001,
personName: "Sam"
personAddress: "India"
deptId: 2001
deptName: "Engg"
deptLocation: "1st Floor"}

**Example**: Same properties cannot be defined with different / same data types in the same type.

```
interface A {
  name: number;
}
interface B {
  name: string;
  address: string
}
```

```
type AB = A & B;
let p: AB;
p.name = "Sam";
console.log(p.name)
```

Will get an error as: Type '"Sam"' is not assignable to type.

We can have multiple interfaces together in a single new type as:

**Example**:

```
interface A { a: number; }
interface B { b: string; }
interface C { c: boolean; }
interface D { d: string; }

interface X { x: A & B  }
interface Y { y: C & D }

type test = X & Y;

let demo: test = {
    x: {
        a: 1010,
        b: 'Sam'
    },
    y: {
        c: true,
        d: "US"
    }
};
console.log(demo);
```

Output:
{x: {…}, y: {…}}
x: {a: 1010, b: "Sam"}
y: {c: true, d: "US"}
__proto__: Object

## 2. Union

The union is declared by using a pipe symbol / vertical bar i.e. (|) separated by each type. It combines multiple types into one. Variables can contain one or more types. It allows defining the optional properties as well.

**Syntax**

```
let A: type1 | type 2 | type3
```

### A. Variable as Union Type

Consider the below example, the message variable is declared as union type which accepts a string, number, or boolean value.

**Example**

```
let message: string | number | boolean;
message = "Hello world";
console.log(message);
message = 1010;
console.log(message);
message =true;
console.log(message);
```

Output:

Hello world
1010
true

### B. Function Parameter as Union Type

Typescript allows us to pass a parameter as a union type. Consider the below example containing a single parameter as a union type. The function can be declared as below:

**Syntax**

```
myFunction = (arg1: string | number): void => {}
```

**Example 1)**

```
function getValue(message: string | number)
```

```
{
    if (typeof (message) == "string")
    {
        console.log("value is of string type")
    }
    else if (typeof (message) == "number")
    {
        console.log("value is of number type")
    }
}
getValue("hello world");
getValue(2000);
//getValue(true);  //Argument of type 'true' is not assignable to parameter of type
'string | number'
```

Output:

value is of string type

value is of number type

**Example 2)** Consider the below example function having two parameters. The first message is taking any type & the second parameter is taking union type i.e. number or string.

```
function getValue(message: any, arg2: number | string )
{
    if (typeof (arg2) == "number" || typeof (arg2) == "string")
    {
        console.log(message + " "  + arg2)
    }
    else
    {
        console.log("arg2 is differnt type")
    }
}
getValue("hello world", 50);
getValue(20, 30 );
getValue(20, true);
```

Output
hello world 50
20 30

arg2 is a different type

### C. Classes / Interfaces with Union

Typescript allows us to create classes/interfaces using union type. In the interface, we can define the optional properties as well.

**Example 1)**

```
class Employee
{
    public name: string;
    public id: string | number
    constructor(i: any, j: any)
    {
        this.name = i;
        this.id = j;
    }
}
let obj: Employee = new Employee("Sam", 2020);
console.log(obj)
let obj1: Employee = new Employee("John", "1010");
console.log(obj1)
```

Output:
Employee {name: "Sam", id: 2020}
Employee {name: "John", id: "1010"}

**Example 2)** Consider below an example of Interface which contains some properties & type myTypes is of union type

```
interface IEmployee {
    empId: number;
    empName: string;
    empAddress?: string;
}
```

```
interface IDepartment {
    deptId: number;
    deptName: string;
}

type myTypes = IEmployee | IDepartment;

let A: myTypes = {
    empId: 1010,
    empName: "Sam"
};
console.log(A);

let B: myTypes = { deptId: 2020, deptName: "IT" }
console.log(B);

let C: myTypes = {
    empId: 1010,
    empName: "Sam",
    empAddress: "US"
};
console.log(C);
```

Output:
{empId: 1010, empName: "Sam"}
{deptId: 2020, deptName: "IT"}
{empId: 1010, empName: "Sam", empAddress: "US"}

**Example 3)** Consider the below example, in interface function type can be used as a union.

```
type MyFunctionA = (name: string) => string;
type MyFunctionB = (name: number) => number;
interface IClassInterface {
    onMouseOver(): MyFunctionA | MyFunctionB;
    name: string;
}
```

## 3. Type Guards

The conditional block is used to restrict the type in Typescript. It is useful when we already know the type on which we will add the conditional blocks & do the next execution. Conditional blocks are like if, if-else, etc conditions that we can use. It allows accessing the members inside the union type. Operators type, in & instanceof, is used in type guards. User-defined type guards are also used in typescript.

**A) typeof:** It checks the primitive types in typescript. We can add the conditions to check the types like number, string, Boolean, etc.

**Example:**

```
function getValue(message: string | number)
{
    if (typeof (message) == "string")
    {
        console.log("value is of string type")
    }
    else if (typeof (message) == "number")
    {
        console.log("value is of number type")
    }
}
getValue("hello world");
getValue(2000);
```

Output:
value is of string type
value is of number type

**B) instanceof:** We already know how typeof works. instanceof is an operator in Javascript. It is narrowing types using their constructor function.

**Example**: In the below example we are using a type assertion of a class student having property name & class standard having property teacher name. Our function returns an object of class

type depending on the condition. We are passing an argument to our function which will be conditionally used. We know the type of result.

```
class Student {
    name: string;
    constructor(i: string) {
        this.name = i;
   }

}
class Standard {
    teacherName: string;
    constructor(j: string) {
        this.teacherName = j;
   }
}
function getDetails(stdId: number): Student | Standard {
    if (stdId === 1001) {
        return new Student("Sam");
    } else {
        return new Standard("John Sir");
    }
}
let result = getDetails(10);
if (result instanceof Student) {
    console.log(result.name);
} else {
     //result.name(); // will get error.
   console.log(result.teacherName);
}

let result2 = getDetails(1001);
if (result2 instanceof Student) {
    console.log(result2.name);
} else {
   console.log(result2.teacherName);
}
```

Output
John Sir
Sam

**C) in:** It checks if the property exists in the given context or not.

**Syntax**:

```
"string_literal" in the result expression
```

Here string_literal is string literal type & result is the union type. It will check for the optional or required property and if it is true then it executes the block.

**Example**:

```
class Student {
    name: string;
    constructor(i: string) {
        this.name = i;
   }

}
class Standard {
    teacherName: string;
    constructor(j: string) {
        this.teacherName = j;
   }
}
function getDetails(stdId: number): Student | Standard {
    if (stdId === 1001) {
        return new Student("Sam");
    } else {
        return new Standard("John Sir");
    }
}
let result = getDetails(10);
if ("name" in result) {
    console.log(result.name);
} else if ("teacherName" in result) {
   console.log(result.teacherName);
}

let result2 = getDetails(1001);
if ("name" in result2) {
    console.log(result2.name);
} else if ("teacherName" in result2) {
   console.log(result2.teacherName);
}
```

Output:

John Sir

Sam

**Example**: Different ways to **check if property exists** in the given context or not.

```
class Student {
    name: string;
    constructor(i: string) {
        this.name = i;
   }

}
class Standard {
    teacherName: string;
    constructor(j: string) {
        this.teacherName = j;
   }
}
function getDetails(stdId: number): Student | Standard {
    if (stdId === 1001) {
        return new Student("Sam");
    } else {
        return new Standard("John Sir");
    }
}

let result = getDetails(10);

if ((result as Student).name) {
    console.log(result);
} else if ((result as Standard).teacherName) {
  console.log(result);
}

let result1 = getDetails(1001);

if ((result1 as Student).name) {
    console.log(result1);
} else if ((result1 as Standard).teacherName) {
  console.log(result1);
}
```

output:

Standard {teacherName: "John Sir"}
Student {name: "Sam"}

## 4. Nullable Types

Typescript allows us to create two types as null & undefined. These are their typed names. These are not too useful but while creating a large application we need to check if the types are null or undefined to handle errors. We can assign null & undefined to any type. strictNullChecks will protect nulls and undefined. It can enable you to add a strictNullChecks flag as an option. We can add a nullable type using union type.

**Example 1)**

Consider the below example, name variable is of type string. In the third line we cannot assign null to string type.

```
let name: string = "sam";
console.log(name);
name = null; //cannot assign null to name
```

We can modify the code by adding union type during declaration.

```
let name: string | null = "sam";
console.log(name);
name = null;   // Its ok to assign null
```

Typescript considers null & undefined differently to match semantics in JavaScript.

Below union has different types in each.

```
let A: number | null
let A: number | undefined
let A: number | null | undefined
```

**Example**

```
type Employee = {
  name: string;
  phoneNumber: string | undefined | null;
};

let emp1: Employee = { name: "John", phoneNumber: "020256880" };
let emp2: Employee = { name: "Sam", phoneNumber: undefined };
let emp3: Employee = { name: "Ashish", phoneNumber: null };
console.log(emp1);
console.log(emp2);
console.log(emp3);
```

output

{name: "John", phoneNumber: "020256880"}

{name: "Sam", phoneNumber: undefined}

{name: "Ashish", phoneNumber: null}

**Example 2)**

Consider the below example, function is taking one argument of type number & function expects to return number. We can't directly return undefined.

```
function doCalculation(arg1: number): number {
    if (arg1 != null) {
        return arg1 * 100;;
    } else {
         return undefined;   // Type undefined is not assignable to
type number.
    }
}
```

We can modify the return type of function by passing union type which is explicitly defined so function may return number, null & undefined as:

```
function doCalculation(arg1: number): number | null | undefined {
    if (arg1 != null) {
        return arg1 * 100;;
    } else {
         return undefined;
    }
}
```

**Optional properties are allowed in Typescript**.

**Example 3)** Consider the below example function having two-arguments, the second argument is an option, so we may or may not pass the second argument. Null is not assignable to number as the second argument.

```
function addition(A: number, B?: number): number{
    if (typeof (B) == "number")
    {
        return A + B;
    }
    else {
        return A;
    }
}
console.log(addition(10));
console.log(addition(10, 20));
console.log(addition(10, null));
```

**Nullable Types using Type Guard:** Nullable types are achieved using union types so we can use type guard to use the guard  properly.

**Example 4)** Consider the below functions. We can check the argument value as equal to null or not and by typeof we can check the type as well.

```
function checkType(arg1: number | null): number {
    if (arg1 == null) {
        return 0;
    }
    else {
        return arg1;
    }
}
console.log(checkType(10));
console.log(checkType(null));
```

Output
10
0

**Using typeof**

**Example:**

```
function usingTypeof(A: number | null): number{
    if (typeof (A) == "number")
    {
        return A;
    }
    else {
        return 0;
    }
```

```
}
console.log(usingTypeof(10));
console.log(usingTypeof(null));
```

Output
10
0

**We can use terser operators as well.**

**Example:**

```
function getData(arg1: string | null | undefined): string {
     return arg1 || "default";
}
console.log(getData("hello world"));
console.log(getData(null));
console.log(getData(undefined));
```

Output
hello world
default
default

## 5. Type Aliases

It creates a new name that refers to the types. It is a new name for the type. It can be used in intersection, unions, tuples, and any other valid type in typescript.

Syntax

```
type var_name = <valid_type>
```

**Example**: Consider the below example, we have a name, and fullName is of type & completeName refers to these types. Function is taking arguments of type completeName& returning a string.

```
type name = string;
type fullName = string;
type completeName = name | fullName;
function getName(arg1: completeName): string {
    return arg1;
}
console.log(getName("sam"));
```

Output
Sam

We will modify the above code, we will check the type of argument using condition.

```
type name = string;
type fullName = string;
type completeName = name | fullName;
function getName(arg1: completeName): string {
    if (typeof (arg1) === "string") {
        return arg1;
    }
    else {
        return "the argument is not a string";
    }
}
console.log(getName("sam"));
console.log(getName(1010));
```

Output
sam
the argument is not a string

## 6. Literal Types

**A. String Literal Types**: ES6 allows us to use a string in a new way. We need to assign the value to our string. It can be used with the union, type guards & type aliases. We can add the conditions as well while creating an application.

**Example**:

```
type Language = "English-US" | "English-EU";
class AllLanguages {
    getLanguage(country: string, language: Language) {
        if (language === "English-US") {
            return country + " " + language;
        }
        else if (language === "English-EU") {
            return country + " " + language;
        }
        else {
           return "Invalid input !!!";
        }
    }
}

let obj = new AllLanguages();
console.log(obj.getLanguage("America" "English-US"));
console.log(obj.getLanguage("India", "Hindi"));
```

Output

America English-US

Invalid input !!!

**B. Numeric Literal Types:** Typescript allows us to create numeric literal types. Consider the below example, we have a function which will take the argument as parameter & will check odd/even numbers using a switch case statement.

```
function checkNumber(value: number): 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9| 10 {
    switch (value % 2) {
```

```
        case 0:
            console.log("The below number is even number");
            return value;
        case 1:
             console.log("The below number is an odd number");
            return value;
        default:
        console.log("Below is invalid input!!!");
            return value;
}
}
console.log(checkNumber(2));
console.log(checkNumber(5));
console.log(checkNumber(NaN));
```

Output
The below number is even number
2
The below number is an odd number
5
Below is invalid input!!!
NaN

**Example** We can't apply|| with != operator using the conditional clause.

```
function doCheck(arg1: number) {
    if (arg1 !== 1 || arg1 !== 2) {
       console.log("value is " + arg1)
    }
}
doCheck(1);
doCheck(2);
doCheck(3);
```

Modify the above function & we will achieve our goal.

```
function doChecks(arg1: number) {
       console.log( arg1 || "default")
```

```
}
doChecks(1)
doChecks(null)
doChecks(undefined)
```

**C. Enum Member Types:** In Typescript enum member has types. We can have enum members literal-initialized. Consider the below example, we have declared an enum for ASCII Code.

```
const enum ASCII_Codes {
  A = 65,
  B = 66,
  C = 67
}

function getCodes(arg1: ASCII_Codes): "A" | "B"  | "C" {
  switch (arg1) {
      case ASCII_Codes.A:
          console.log("ASCII Code for A is " + arg1);
      return "A";
      case ASCII_Codes.B:
       console.log("ASCII Code for B is " + arg1);
      return "B";
  }
}
getCodes(ASCII_Codes.A)
getCodes(ASCII_Codes.B)
```

Output
ASCII Code for A is 65
ASCII Code for B is 66

### D. Boolean Literal Types

We can declare the boolean literal type like the below constants having values true & false respectively. We can declare a union type as well.

```
const TRUE: true = true;
console.log(TRUE)
```

```
const FALSE: false = false;
console.log(FALSE)
```

**Union type**

```
let result: true | false;
```

**Conditional based:**

We can check the value based on condition as well. Consider the below function which will take one argument & will check if the argument is a valid email or not in the function based on condition. We will return a true or false value.

```
function ValidateEmail(email: string)
{
 if (/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/.test(email))
  {
     return true;
  }

    return false;
}
console.log(ValidateEmail("sam@gmail.com"));
console.log(ValidateEmail("test.com"));
```

### E. Polymorphic Types

We can create a polymorphic type in Typescript. We can use this type in class/interfaces. F-bound polymorphism is used when a subtype contains a class/interface. The below example contains classes having constructors & methods which are used to do the calculation like addition, subtraction, multiplication, division, etc.

```
class DoMathsCalculations {
    constructor(protected arg1: number = 0) { }
```

```
    addition(operand: number): this {
        this.arg1 += operand;
        return this;
    }
    subtraction(operand: number): this {
        this.arg1 -= operand;
        return this;
    }
    multiplication(operand: number): this {
        this.arg1 *= operand;
        return this;
    }
    division(operand: number): this {
        this.arg1 /= operand;
        return this;
    }
}

let obj = new DoMathsCalculations(2);
console.log(obj.addition(10));
console.log(obj. subtraction(5));
console.log(obj.multiplication(6));
console.log(obj.division(10));
```

Output

DoMathsCalculations {arg1: 12}

DoMathsCalculations {arg1: 7}

DoMathsCalculations {arg1: 42}

DoMathsCalculations {arg1: 4.2}

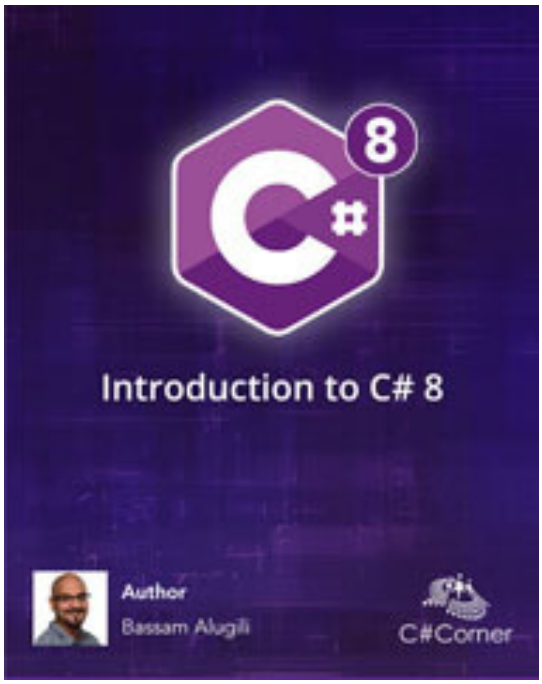
**Summary:** In this chapter, we covered some advanced types of Intersection, Union, Type Guards, Nullable Types, Type Aliases & Literal Types in TypeScript.

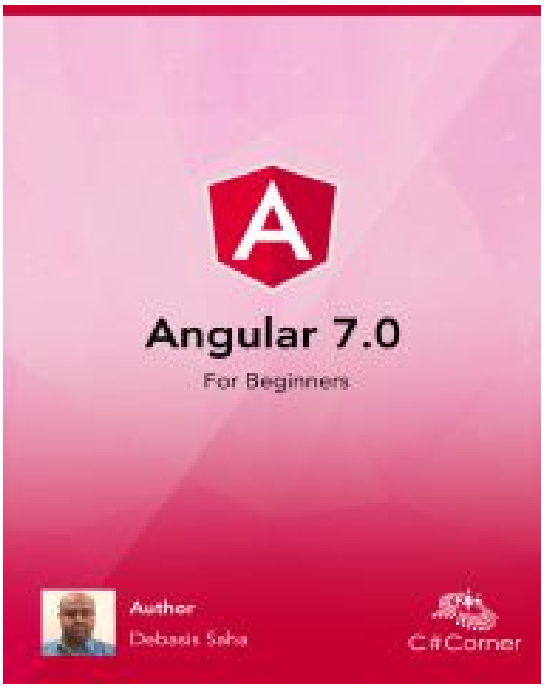
References


1) www.typescriptlang.org
2) https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html


SharePoint
Online & Office 365
Administration
Author
Habibur Rahaman
C#Corner


Build A Full-stack
Web Application
Using
Angular And Firebase
Author
Ankit Sharma
C#Corner


8
Introduction to C# 8
Author
Bassam Alugili
C#Corner


Angular 7.0
For Beginners
Author
Debasis Saha
C#Corner