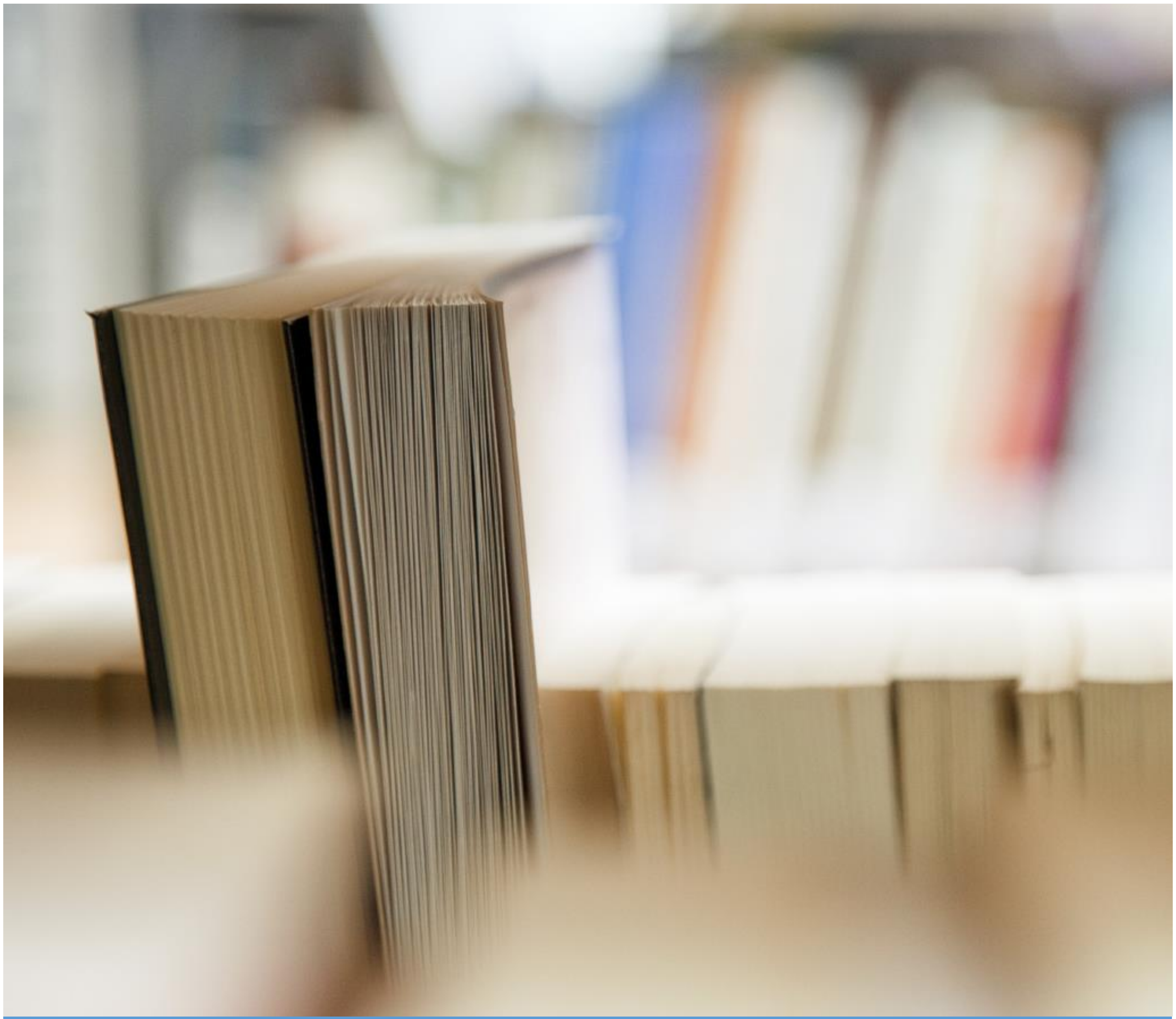


TypeScript ES 6

From Zero to Hero

NAVEEN SAGGAM

4/24/18

UiBrians.com

TypeScript with ES 6 Features

Contents

Introduction:

Typescript is a superset of JavaScript best transpiler that compiles the Object Oriented code to plain JavaScript. This is a pure Object Oriented Language with Classes and Interfaces. This is an Extension of JavaScript where we can write OOPS code and that compiles back to pure / Core JavaScript which runs on any browser , any OS and any Device.

TypeScript is developed and Maintained by Microsoft and is an Open Source Project.

The Popular Front End Framework Angular 5 is also built on TypeScript.

The Current version is 2.8.1 released 27th, March, 2018.

Typescript is one of the best transpiler as of today which supports almost all the modern features of ECMA Script and in future also it will supports all the new upcoming features of ECMA Script.

We will write code in TypeScript and compiles it back to core JavaScript of any version.

As JavaScript is a weakly typed language where in TypeScript is a Strongly Typed Language, in order to avoid unnecessary errors caused by JavaScript at Runtime.

TypeScript Summary:

Type	Details
Developed by	**Microsoft**
License	**Open Source**
Current Version	**2.8.1 released 27th,March,2018**
Official Website	**https://www.typescriptlang.org/**
Practice Online	**https://www.typescriptlang.org/play/index.html**
Known as	**Best Transpiler for JavaScript**
Compatible with	**Any Browser , Any OS , Any Device**

ECMA Script

JavaScript is also called as ECMA Script. ECMA Stands for European Computers Manufacturers Association.

ECMA is the standard body which releases the different versions of ECMA Script / JavaScript. The current version of ECMA Script is **ECMA Script-2017** (8th Edition).

This ECMA Script language features can be used by all the browsers, to implement the support for running various versions of JavaScript/ECMA Script.

As ECMA releases the new features, which are not automatically available by all the browsers and most browsers supported version of ECMA Script is **ES-5** version.

This ECMA Script 5 is the version, which is used by almost all the browsers supports as today.

Here we needs to understand if ECMA releases a new feature / standard, however it does not mean that every feature of the standard body ECMA is the standard and immediately available in all the browsers.

In reality some implementations / features are already available in all the browsers and which are not standardized by ECMA and also the features and some implementations are standardized by ECMA which are not available in all browser.

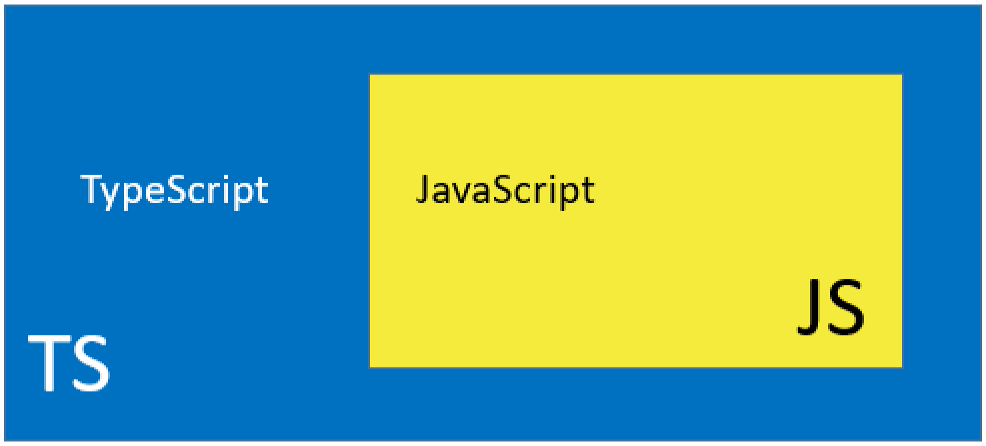
Now we understood that there is a gap between the ECMA Script versions used by all the modern browsers and the version which are released by ECMA.

As a JavaScript developer may get frustrate because there are lot of new features coming in but there is no actual browser support for those features in order to leverage those features.

This is the place where Transpiler comes into picture to fill the gap, where the transpiler compiles and transforms from one standard to another standard.

So TypeScript is one of the best transpiler as of today and we can leverage all the features of ECMA script and even future versions of ECMA Script and transpiles to ES-5 and which is fully supported by all the browsers as of today.

TypeScript with ES 6 Features



There are many Transpilers available in the market along with TypeScript is Babel Script. TypeScript is more popular and even the Angular Framework uses TypeScript.

TypeScript supports all the features of ES6 and also the future versions of ECMAScript and compiles them to any chosen version of JavaScript like ES5.

Differences between TYPESCRIPT and JAVASCRIPT

JAVASCRIPT	TypeScript
JAVASCRIPT is a Dynamic Typed Language	TS is a Static Typed Language
This is most forgiving language and weakly typed language	This is particular about typing and it is a strongly typed language like Java
As this is not a Strict typing, it leads to many unexpected runtime errors during development time	As this is a Strong typing, it avoids almost all unexpected runtime errors during development time, especially unexpected errors seen in JAVASCRIPT
It applies typing in Dynamic runtime and which leads to many runtime errors	It applies type in the development time itself and avoids runtime errors
JAVASCRIPT is great for Web Browser Object Model	It promotes stability and maintainability

NOTE:

JavaScript is a dynamic typing language means we can provide the typing information for any variables at runtime. But for TypeScript we can provide the typing information only at the time of declaration of variables. This is called static typing.

As JavaScript is dynamic typing language, it is best suitable for BOM (browser object model) and with TypeScript adds static typing to the JavaScript to add more stability and more maintainability of the program and avoids unexpected runtime errors for JavaScript.

Instead of writing direct JavaScript, if we use TypeScript to generate JavaScript which compiles and gives the most stable and most maintainable JavaScript and most powerful JavaScript.

So this is the reason why Angular JS is completely redesigned to use TypeScript and renamed to Angular.

TYPESCRIPT Editors

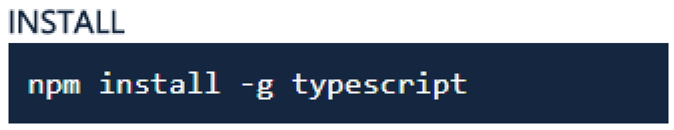
We can use any text editor to work with TypeScript as follows,

1. Any text editor + TS compiler (command prompt)
2. Atom Text Editor by Github - open source
3. Visual Studio Code by Microsoft - open source
4. Brackets Editor by Adobe Systems - open source
5. Sublime Text by Sublime – Semi licensed
6. WebStorm by JetBrains – Fully Licensed

Installing TypeScript

We can install the Type using the Node JS' Node Package Manager (npm).

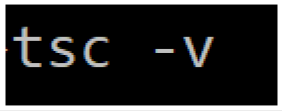
1. Install Node JS
2. Check the versions of **node** and **npm**
3. Install "typescript" using npm



INSTALL

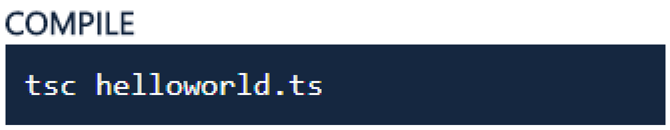
```
npm install -g typescript
```

4. Check the version of TypeScript



```
tsc -v
```

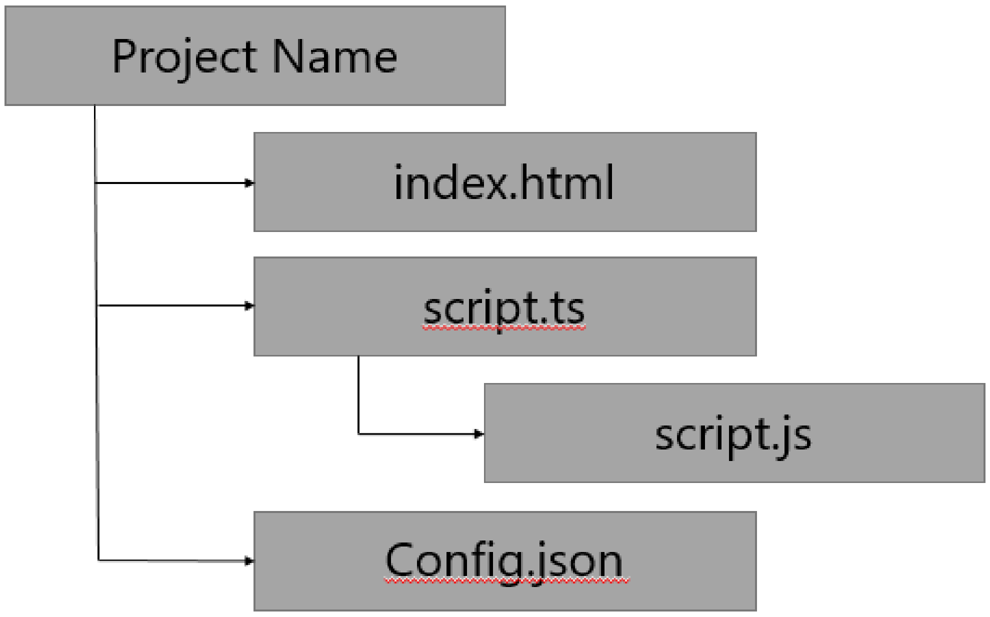
5. Compile the TypeScript using **tsc**


COMPILE

```
tsc helloworld.ts
```

Project Structure of TypeScript

We must follow the below project structure for the entire course.



ECMA Script 6 Features:

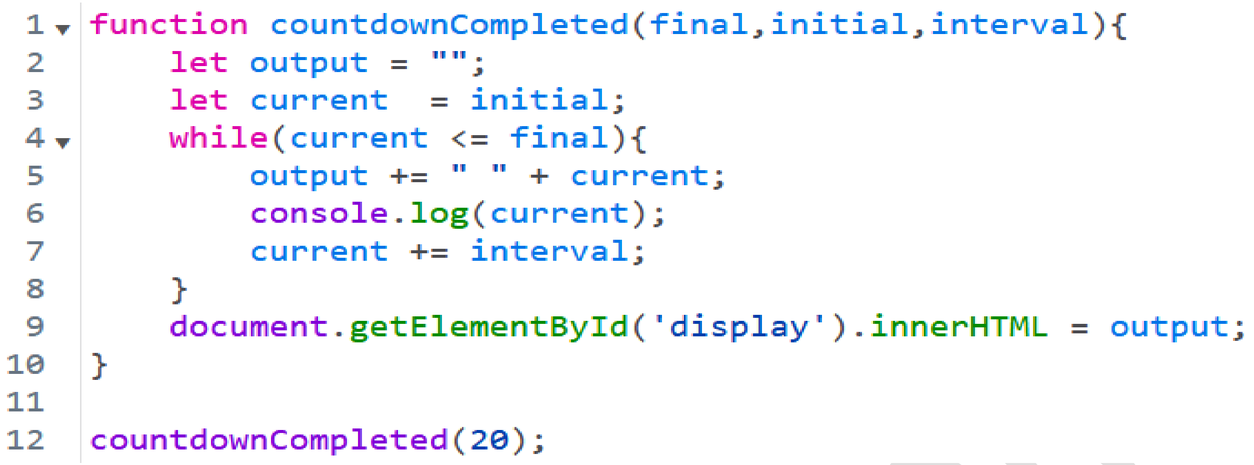
TypeScript leverages to use all the ECMA script features and compile them to 'ESS' plain JavaScript to work on any modern browser.

Here we use all the 'ES6, features in TypeScript and compile them to 'ES5' JavaScript code and execute them in any browser.

Now let's understand the 'ES6' features and their usage.

Default Parameters:

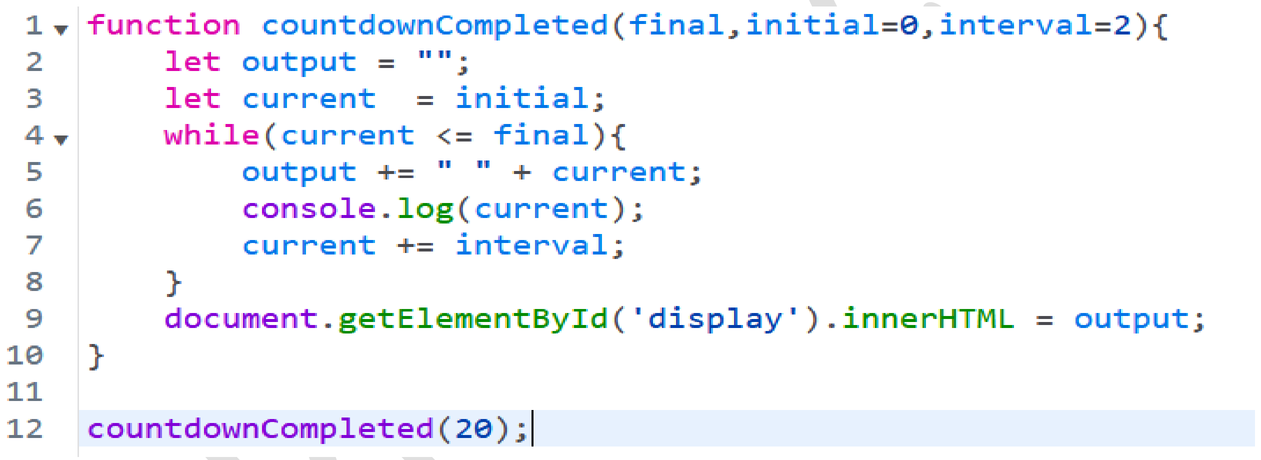
This feature is to provide some default values to optional parameters, if any of the parameter is not supplied, then the program will be using these default parameters.



```
function countdownCompleted(final,initial,interval){
    let output = "";
    let current  = initial;
    while(current <= final){
        output += " " + current;
        console.log(current);
        current += interval;
    }
    document.getElementById('display').innerHTML = output;
}

countdownCompleted(20);
```

In the above example the function takes in the parameters final, initial, interval.

Let's make "initial" and "interval" are optional parameters, we can use in the following way:



```
function countdownCompleted(final,initial=0,interval=2){
    let output = "";
    let current  = initial;
    while(current <= final){
        output += " " + current;
        console.log(current);
        current += interval;
    }
    document.getElementById('display').innerHTML = output;
}

countdownCompleted(20);
```

In the above function, we have provide a default parameters for **initial** and **interval** values,

If that values are not provided, then these default values will take into consideration.

Template Strings

Normally we use String Concatenation operator to combine two or more strings. This approach is more difficult if mix the HTML tags with normal Strings. We resolve this by using Template Strings.

In this template string, we use back tick (`) symbol to combine any kind of strings literals or HTML code or any JavaScript code instead of using string concatenation operator.

By using String Concatenation operator we can mix the HTML code with normal Strings as follows,

```
var stringConcat = "<div>" + "<h1> Name : " + employee.name + "</h1>" +
                             "<h1> Age : " + employee.age + " </h1>" +
                             "<h1> Desg : " + employee.desg + "</h1>" +
                    "</div>";
```

We can replace this by using Template String with back tick (`) operator as follows,

```
var templateString = `<div>
                        <h1>Name : ${employee.name}</h1>
                        <h1>Age : ${employee.age}</h1>
                        <h1>Desg : ${employee.desg}</h1>
                      </div>`;
```

Let and Const

Let is one of the **ECMA Script6** feature which is used to declare variables in TypeScript same as the '**var**' keyword in JavaScript.

The problem with the **var** keyword is, once we create a variable inside if block or for loop, we can still access that variable outside if block or for loop. We can avoid access of the variables which are declared inside if block or for loop using **let** keyword.

```
for(var i=0; i<= 10; i++){

}
var output = " The value of 'i' is : " + i;

//Output : The value of 'i' is : 11
```

We can avoid the access any variable declared inside any block is by using **let** keyword as follows,

```
for(let j=0; j<=10; j++){

}
var output = " The value of 'j' is : " + j;

//Output : ReferenceError: j is not defined
```

Const keyword is used to declare some final variables, once a variable is declared with **const**, we can't able to change the value of it and also it avoid access of that variable outside the declared block.

```
const  MONTH_NAME = 'January';
output = "The Month name is : " + MONTH_NAME;
```

Once a variable is declare using **const**, we can't change the value of it. If we are trying to change the value of it, we may get the Compile Time error as follows,

```
const  MONTH_NAME = 'January';
output = "The Month name is : " + MONTH_NAME;

MONTH_NAME = "February";

// Output : TypeError: invalid assignment to const `MONTH_NAME'
```

For ...of loops

Normally we use **for...in loop** to loop through an array and we will get the index and with the index we can get the actual value of an array as follows,

This is available in ES5 version only.

```
let myArray = ['html','javascript','css','bootstrap'];

let output = "";
for(let index in myArray){
    let value = myArray[index];
    console.log(value);
    output += value + " , ";
}
document.getElementById('display').innerHTML = output;
```

From ES6 version onwards, ECMA has provided a new concept called **for...of loop**.

By using this for...of loop, we can directly get the actual value of an array instead of getting an index of an array as follows,

```
let myArray = ['html','javascript','css','bootstrap'];
let output = "";
for(let value of myArray){
     console.log(value);
     output += value + " , ";
}
document.getElementById('display').innerHTML = output;

// output : html , javascript , css , bootstrap
```

Lambdas: [Arrow function]

A lambda expression is an anonymous functions that replaces the normal function definition with shortcut arrow functions.

This is one of the functional programming feature, we can use this feature to replace the function parameters with arrow functions.

Let's understand a function Expression call without using Arrow function as follows,

```
1  let greet = function():string{
2       return "Good Morning";
3  };
4
5  let output = greet();
6  console.log(output);
```

Now let's replace the function call at line number 5 using Arrow function as follows,

```
1  let greet = function():string{
2       return "Good Morning";
3  };
4
5  let output = () => "Good Morning";
6  console.log(output);
```

In the above example, instead of creating and executing the function greet(); we can directly use the arrow function.

```
() => "Good Morning";
```

Let's understand how to replace an anonymous function expression that takes in few arguments with lambda / Arrow functions as follows,

```
1  let sum = function(a,b){
2    return a + b;
3  };
4
5  let sumArrow = (a,b) => a + b;
6  let output = "The Sum is : " + sumArrow(10,20);
7  console.log(output); // The Sum is : 30
```

In the above example, the sum function we replaced with an arrow function in the line number 5. This is one of the shortcut method to replace function expressions using Arrow Function.

Let's have an another function which takes an array and prints the length of passed in array using arrow functions as follows,

```
1  let len = function (array:Array<number>) {
2    return array.length;
3  };
4
5  let lenArrow = (array) => array.length;
6  let myArray = [10,20,30,40,50];
7  let output = lenArrow(myArray);
```

In the above example, we replaced the len() with an arrow function in the line number 5.

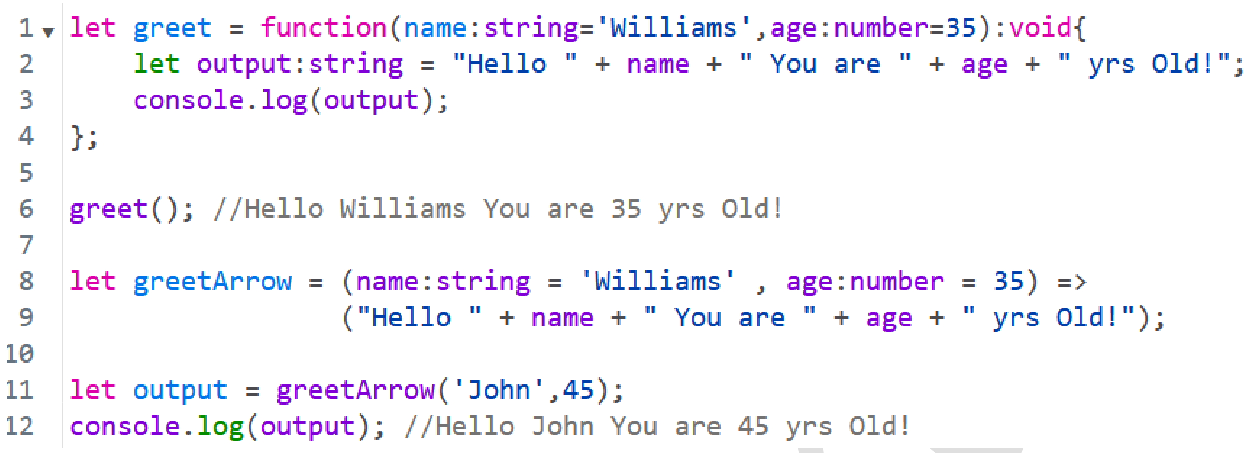
We can also use Arrow functions replace the map() function of an array as follows,

```
1  let movies = ['Baahubali' , 'Dangal' , 'Drushyam'];
2
3  // Normal Function
4  let movieLength = movies.map(function (movie) {
5            return movie.length;
6      }
7  );
8  console.log(movieLength); // [9, 6, 8]
9
10 // Lambda Expressions
11 movieLength = movies.map( movie => movie.length);
12
13 let output = movieLength.join(" - ");
14 console.log(movieLength); // [9, 6, 8]
```

In the above example, we are printing the length of each of the element and creating an array using movies.map() function.

We can replace the parameter of movies.map() using Lambda expressions or Arrow Functions as specified in the line number 11.

We can also mix the default parameters with Arrow functions as follows,



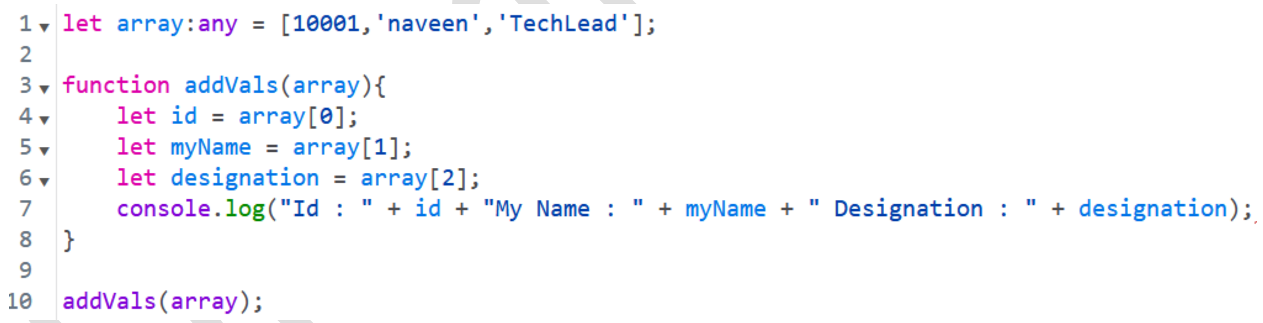
```
let greet = function(name:string='Williams',age:number=35):void{
    let output:string = "Hello " + name + " You are " + age + " yrs Old!";
    console.log(output);
};

greet(); //Hello Williams You are 35 yrs Old!

let greetArrow = (name:string = 'Williams' , age:number = 35) =>
                    ("Hello " + name + " You are " + age + " yrs Old!");

let output = greetArrow('John',45);
console.log(output); //Hello John You are 45 yrs Old!
```

Destructing

This is one of the powerful feature in ES6 in which instead of adding a single value to a variable from an array, we can do it all by once.

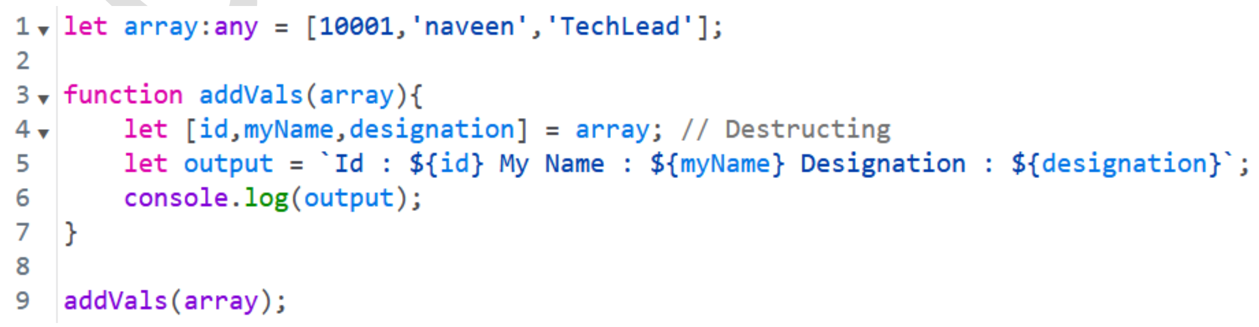
Here Instead of fetching each and every value from an array and adding to each variable created, we use the shortcut notation called **Destructing**.

We normally use the following way to access each element from an array as follows,



```
let array:any = [10001,'naveen','TechLead'];

function addVals(array){
    let id = array[0];
    let myName = array[1];
    let designation = array[2];
    console.log("Id : " + id + "My Name : " + myName + " Designation : " + designation);
}

addVals(array);
```

In shortcut we can achieve this by using the concept called **destructing** as follows,

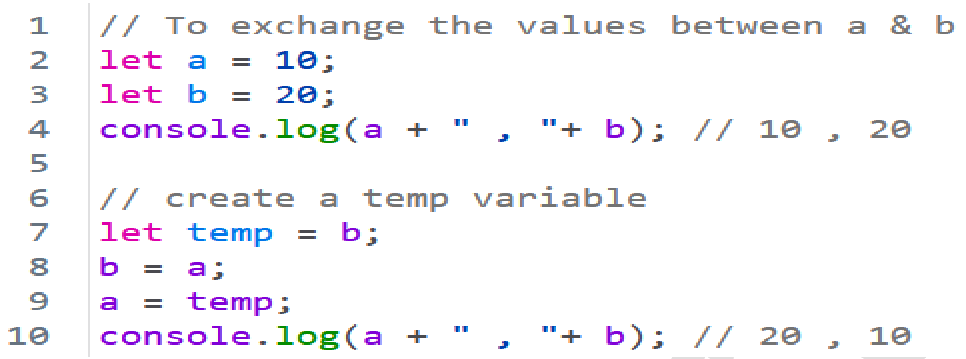


```
let array:any = [10001,'naveen','TechLead'];

function addVals(array){
    let [id,myName,designation] = array; // Destructing
    let output = `Id : ${id} My Name : ${myName} Designation : ${designation}`;
    console.log(output);
}

addVals(array);
```

In the above example, line number 4, we use the concept called **destructing**. At line number 5, we use the concept called **Template String** (back tick operator) for combining various stings and values.

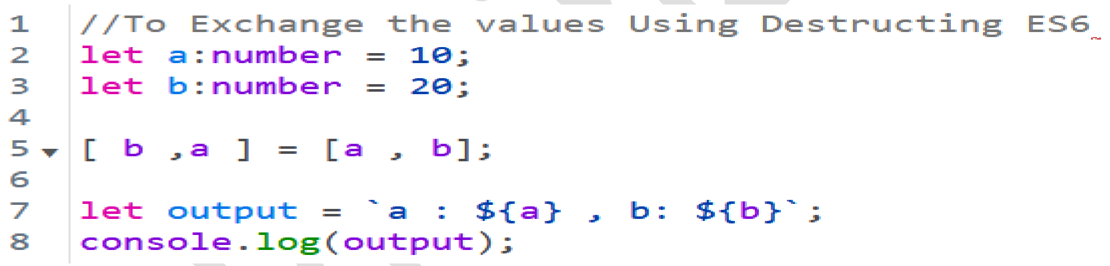
We can also use the concept called destructing to exchange the values between two variables.

Normally to exchange values between two variables, we should require one temp variable as follows,



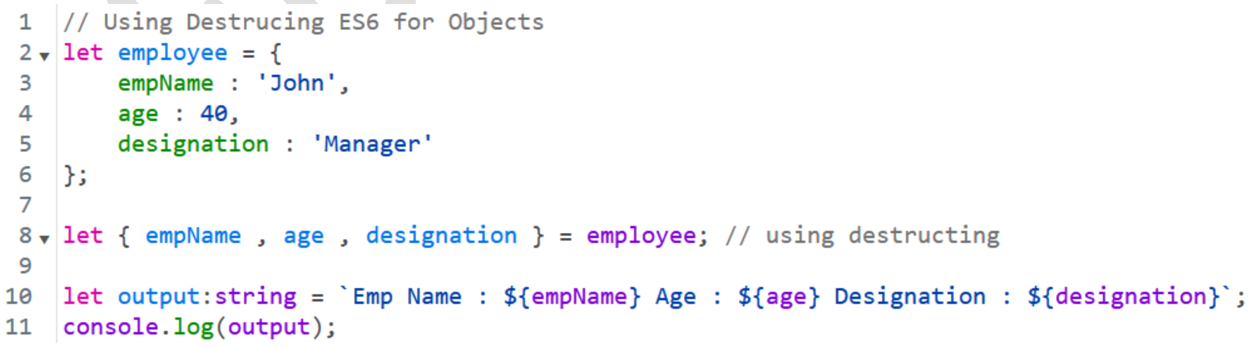
```
// To exchange the values between a & b
let a = 10;
let b = 20;
console.log(a + " , "+ b); // 10 , 20

// create a temp variable
let temp = b;
b = a;
a = temp;
console.log(a + " , "+ b); // 20 , 10
```

In a simple way of exchange the values between two variables is by using the concept called destructing as follows,



```
//To Exchange the values Using Destructing ES6
let a:number = 10;
let b:number = 20;

[ b ,a ] = [a , b];

let output = `a : ${a} , b: ${b}`;
console.log(output);
```

The Destructing concept even works for Objects as follows,

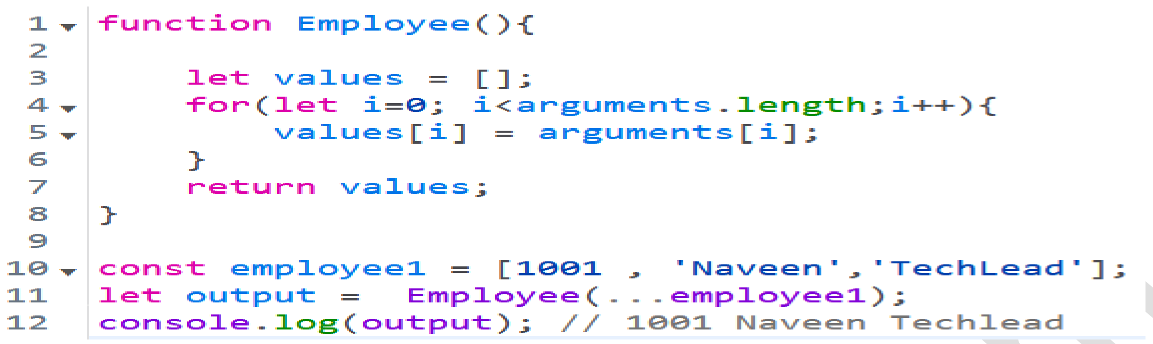


```
// Using Destrucing ES6 for Objects
let employee = {
    empName : 'John',
    age : 40,
    designation : 'Manager'
};

let { empName , age , designation } = employee; // using destructing

let output:string = `Emp Name : ${empName} Age : ${age} Designation : ${designation}`;
console.log(output);
```

The Spread Operator

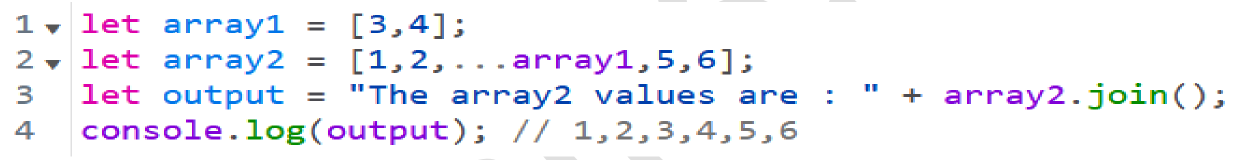
The spread operator is allows an expression to be expanded in places where multiple elements / variables / arguments are expected.

If any function is expected zero or more number of arguments, we can use the spread operator to pass in the complete array to is.


```
function Employee(){

    let values = [];
    for(let i=0; i<arguments.length;i++){
        values[i] = arguments[i];
    }
    return values;
}

const employee1 = [1001 , 'Naveen','TechLead'];
let output =  Employee(...employee1);
console.log(output); // 1001 Naveen Techlead
```

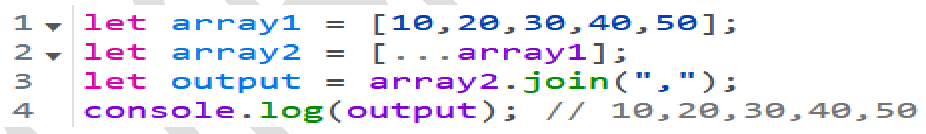
In the above example we can pass in the complete array as a parameter to the Employee() function. Here the complete array will be expanded and pass it as an argument to the above function.

We can use the spread operator to insert an array in the middle of another array using spread operator as follows


```
let array1 = [3,4];
let array2 = [1,2,...array1,5,6];
let output = "The array2 values are : " + array2.join();
console.log(output); // 1,2,3,4,5,6
```

In the above example we have inserted the elements of an array1 into the middle of array2 using spread operator.

We can also use the spread operator to create a copy of an array and assign to another using spread operator, same like slice() function of arrays.


```
let array1 = [10,20,30,40,50];
let array2 = [...array1];
let output = array2.join(",");
console.log(output); // 10,20,30,40,50
```

In the above example, we created a new array with the same copy of array1 using the spread operator.

We can also use spread operator to concatenate two or more arrays as follows,

```
let array1 = [10,20,30,40];
let array2 = [50,60,70,80];

// Without using spread operator
//array1 = array1.concat(array2);
console.log(array1); // 10,20,30,40,50,60,70,80

// Using the spread operator
array1 = [...array1,...array2];
console.log(array1); // 10,20,30,40,50,60,70,80
```

We can also use Spread operator in conjunction with Math Operations as follows,

```
let myArray = [9,20,45,13];
let minVal = " The min value is : " + Math.min(...myArray);
console.log(minVal); // The min value is : 9
```

TypeScript Data Types

TypeScript datatype are used to specify the type of data we pass to any variables. In JavaScript we won't specify the type of data at compile time, at runtime the assigned data will be identified by the JavaScript. By using TypeScript we can add static typing to the defined variables.

In JavaScript we can re-assign the values to any variables defined. This may cause unexpected runtime errors at runtime, in order to avoid these errors by using TypeScript we can add static typing to JavaScript using various TypeScript datatypes.

TypeScript supports the following data types.

Number

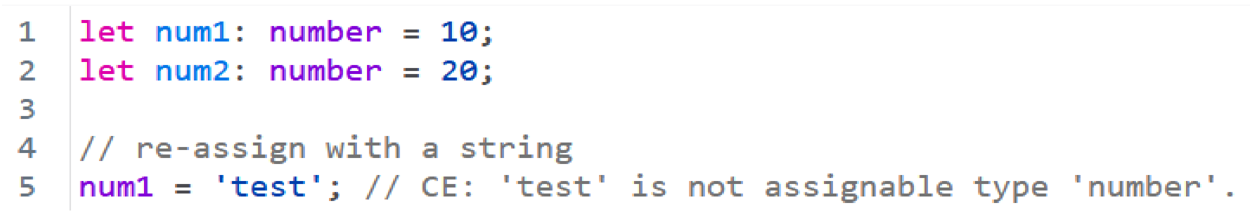
In order to specify any number to a variable, we use the data type number in TypeScript.

Let's understand the creation of variables in TypeScript as follows,

```
let num1: number = 10;
let num2: number = 20;
let sum:number = num1 + num2;
console.log(`The Sum of num1 , num2 is : ${sum}`); //The Sum of num1 , num2 is 30
```

In the above example, we declared two numbers and the sum of two variables assigned to a new variable sum and printed them on the console using Template String of ES6 version.

Note: Once we declare any variable of one type and try to re-assign them to another type, we may get the compile time error in TypeScript.

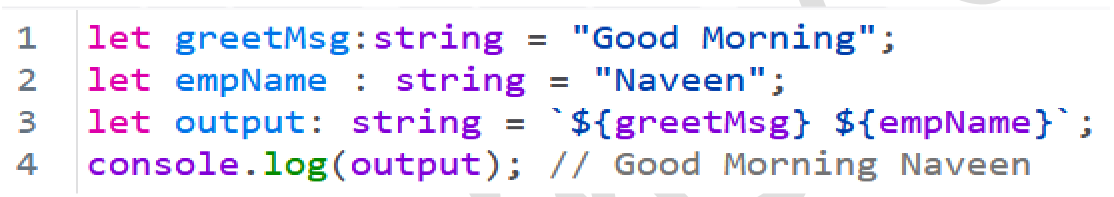


```
let num1: number = 10;
let num2: number = 20;

// re-assign with a string
num1 = 'test'; // CE: 'test' is not assignable type 'number'.
```

String

This is one of the commonly used datatype for textual data of any length.

We use String datatype to assign any textual data to any variable of any length. We can even assign a single character to a variable using this string datatype. In TypeScript once a declare any variable with string datatype, we cannot re-assign that variable with any other type of value to it. If we try to assign any other type of value to the string variable we will get a compile time error.

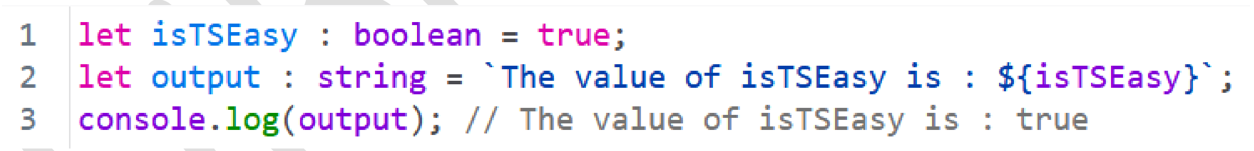


```
let greetMsg:string = "Good Morning";
let empName : string = "Naveen";
let output: string = `${greetMsg} ${empName}`;
console.log(output); // Good Morning Naveen
```

Boolean

We use Boolean datatype to assign true or false value to a variable.

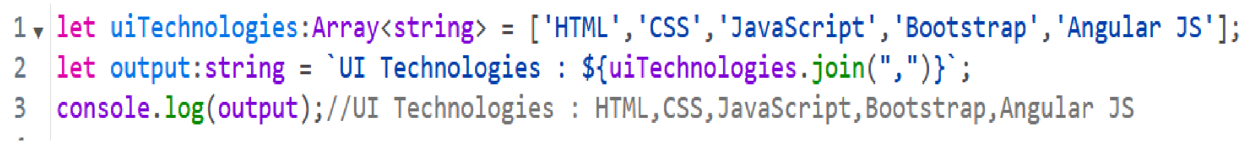
Once we declare any variable with Boolean datatype, we can’t assign any other type of value to the same variable.



```
let isTSEasy : boolean = true;
let output : string = `The value of isTSEasy is : ${isTSEasy}`;
console.log(output); // The value of isTSEasy is : true
```

Array

We use Array datatype is to assign an array to a variable. Normally in JavaScript we can add any type of elements to an array, but in TypeScript we can restrict the type of elements can be used to in an array.

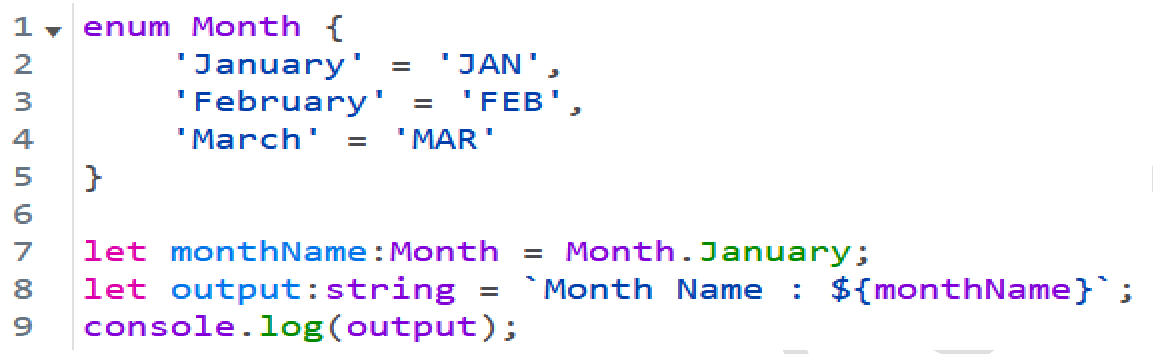


```
let uiTechnologies:Array<string> = ['HTML','CSS','JavaScript','Bootstrap','Angular JS'];
let output:string = `UI Technologies : ${uiTechnologies.join(",")}`;
console.log(output);//UI Technologies : HTML,CSS,JavaScript,Bootstrap,Angular JS
```

Enum

The Enums are used to define a collection of constant values. One an Enum is declared with some value they are final and they cannot be modified.

Let's understand the declaration of Enum and accessing the properties of an Enum as follows,



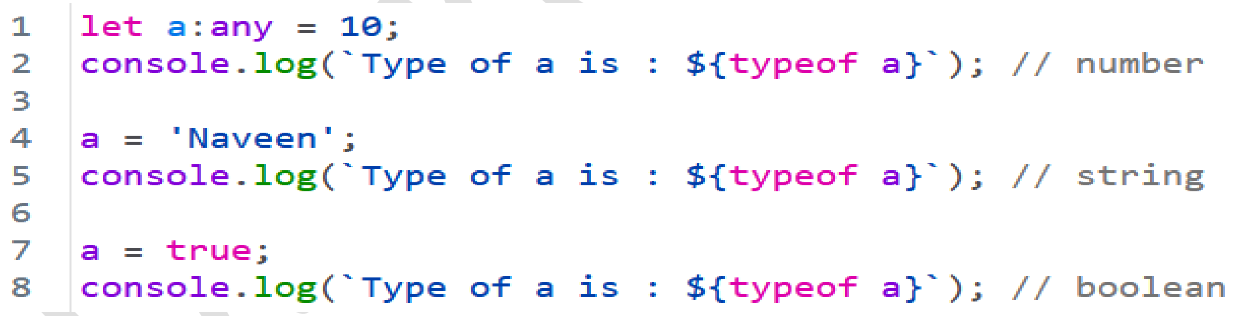
```typescript
enum Month {
    'January' = 'JAN',
    'February' = 'FEB',
    'March' = 'MAR'
}

let monthName:Month = Month.January;
let output:string = `Month Name : ${monthName}`;
console.log(output);
```

In Enum, we can access the properties of it using the name of Enum.

Any

In JavaScript we can assign any type of value to a variable. We can even re-assign different types of values to a variable. We can achieve this in TypeScript by using a datatype called **any**.

Once we declare a variable with **any** datatype, we can assign any type of value to it.



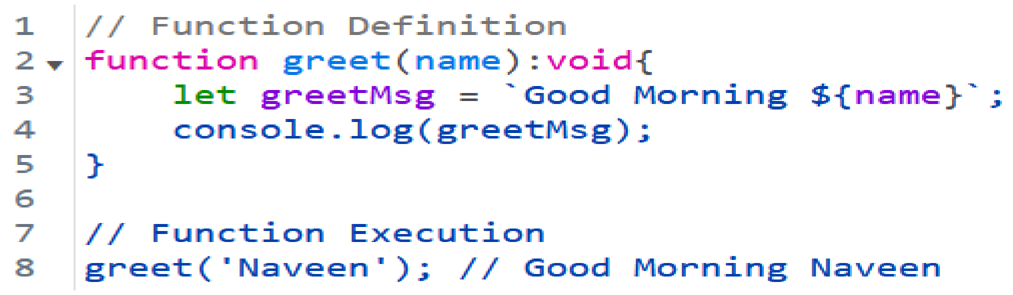
```typescript
let a:any = 10;
console.log(`Type of a is : ${typeof a}`); // number

a = 'Naveen';
console.log(`Type of a is : ${typeof a}`); // string

a = true;
console.log(`Type of a is : ${typeof a}`); // boolean
```

Void

We use void datatype for functions which does not return any return value. Any function with non-return value, we declare it with void datatype.

In JavaScript we may not specify the type of value it returns from a functions, but in TypeScript we must have to specify the return value of any function defined.

Let's understand the usage of void as follows,



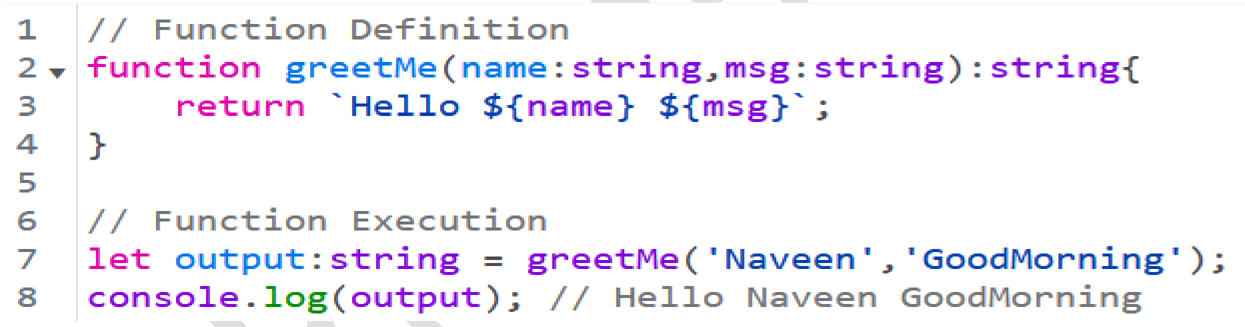
```
// Function Definition
function greet(name):void{
     let greetMsg = `Good Morning ${name}`;
     console.log(greetMsg);
}

// Function Execution
greet('Naveen'); // Good Morning Naveen
```

Functions in TypeScript

A Function is a block of code that takes in some of the arguments and does some processing and returns the results back the called function or variable.

We can define functions in TypeScript same as in JavaScript way, but here in TypeScript we can specify the type of parameters takes in and also specify the type of values may returns from a function.

Let's understand the function declaration in TypeScript as follows,



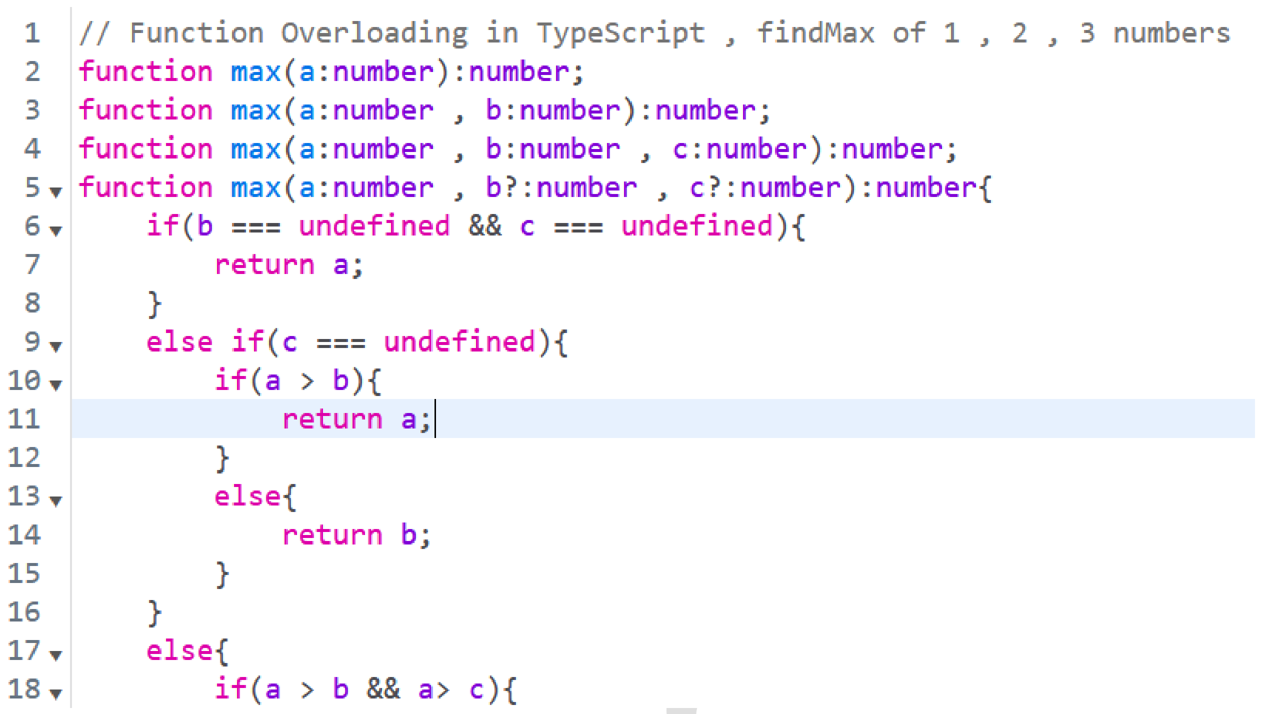
```
// Function Definition
function greetMe(name:string,msg:string):string{
     return `Hello ${name} ${msg}`;
}

// Function Execution
let output:string = greetMe('Naveen','GoodMorning');
console.log(output); // Hello Naveen GoodMorning
```

In JavaScript we can call a function with same number of arguments/parameters or with less number of arguments and also with more number of arguments, but in TypeScript we can call a function with the same number of parameters specified in the function definition.

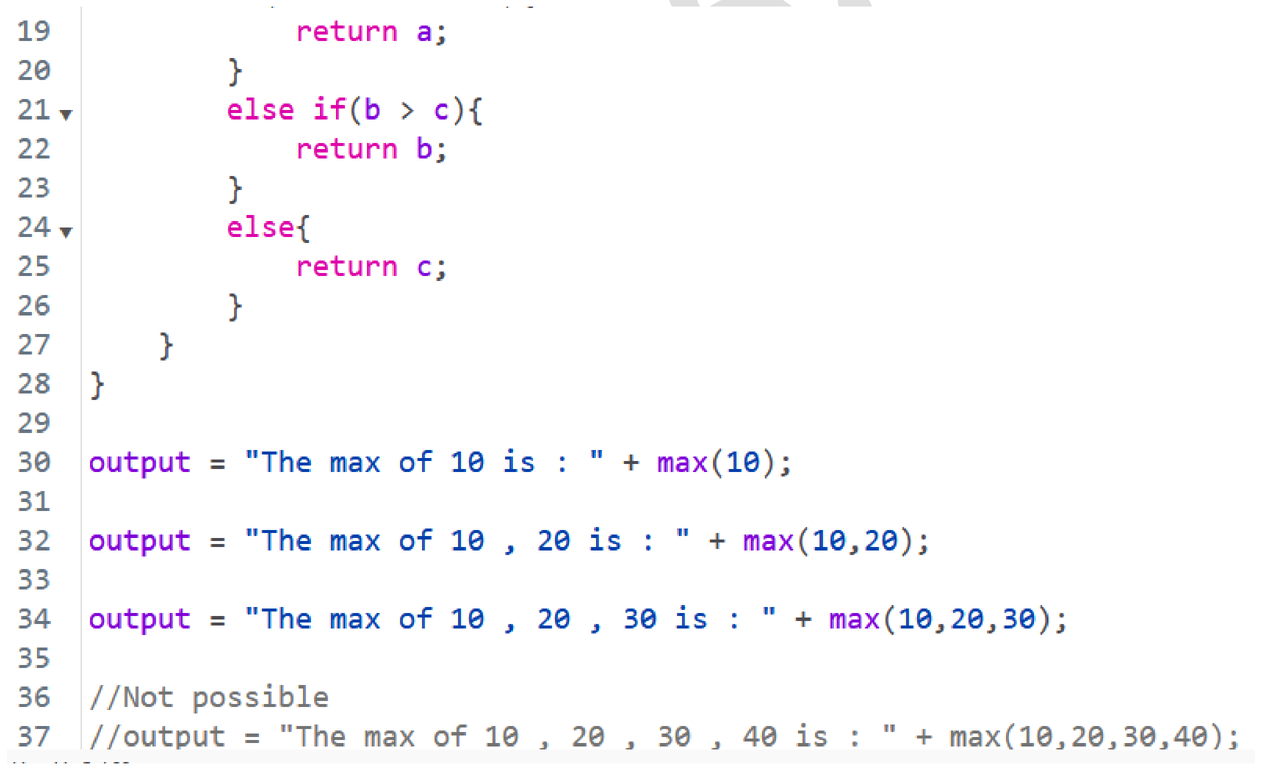
If needs to pass more or less number of arguments, we must have to declare each function with the required number of functions.

We cannot pass more or less number of parameters to a function like JavaScript. This clearly says that the TypeScript supports the concept called function overloading.

Let's understand the concept of function overloading in TypeScript is as follows,



```
// Function Overloading in TypeScript , findMax of 1 , 2 , 3 numbers
function max(a:number):number;
function max(a:number , b:number):number;
function max(a:number , b:number , c:number):number;
function max(a:number , b?:number , c?:number):number{
    if(b === undefined && c === undefined){
        return a;
    }
    else if(c === undefined){
        if(a > b){
            return a;
        }
        else{
            return b;
        }
    }
    else{
        if(a > b && a> c){
```



```
            return a;
        }
        else if(b > c){
            return b;
        }
        else{
            return c;
        }
    }
}

output = "The max of 10 is : " + max(10);

output = "The max of 10 , 20 is : " + max(10,20);

output = "The max of 10 , 20 , 30 is : " + max(10,20,30);

//Not possible
//output = "The max of 10 , 20 , 30 , 40 is : " + max(10,20,30,40);
```

As per the above example, we needs to declare the overloaded function first and then we needs to provide implementation for them.

Interfaces in TypeScript

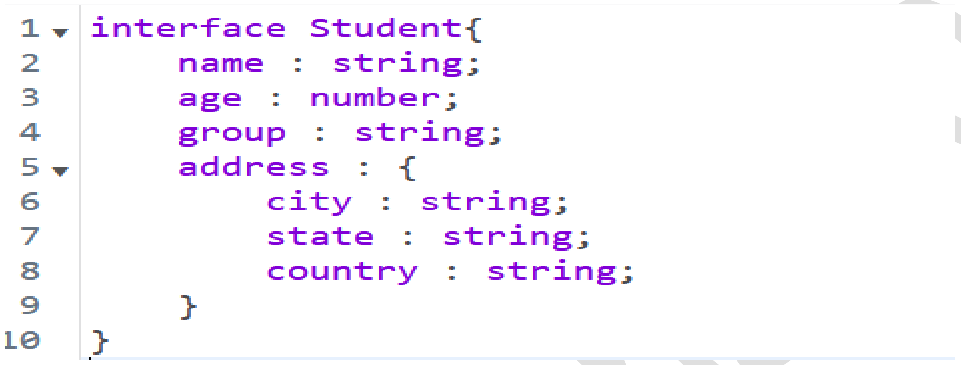
An interface defines a contract between the specification and the actual implementation.

We can just specify the typing information inside an interface, and this can be implemented inside classes for those interfaces.

An interface defines an abstraction and that can be implemented using Classes in TypeScript. An interface just defines the rules which can be implemented in Classes of TypeScript.

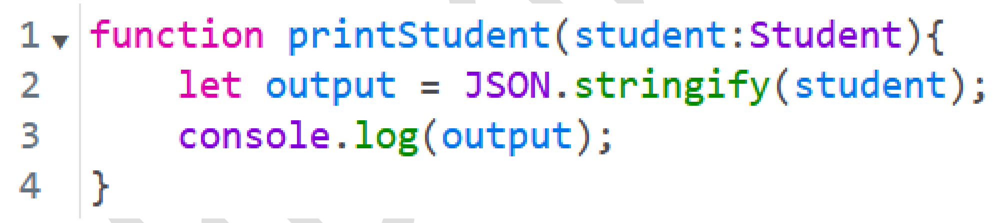
Let's understand the usage of Interface in TypeScript is as follows,

We can define an interface in TypeScript is by using the keyword called 'interface'.



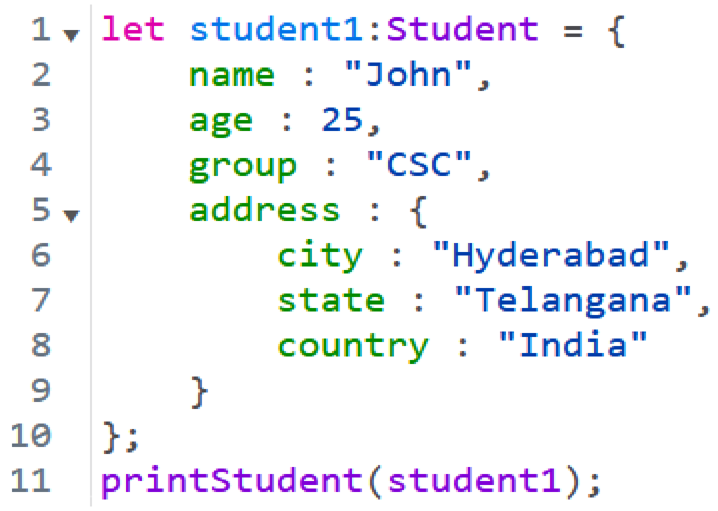
```
interface Student{
    name : string;
    age : number;
    group : string;
    address : {
        city : string;
        state : string;
        country : string;
    }
}
```

We have a function which accepts a student as a parameter and which is of type Student Interface.



```
function printStudent(student:Student){
    let output = JSON.stringify(student);
    console.log(output);
}
```

Here we needs to create a student object same as the structure defined in the Student Interface and pass it to this function.



```
let student1:Student = {
    name : "John",
    age : 25,
    group : "CSC",
    address : {
        city : "Hyderabad",
        state : "Telangana",
        country : "India"
    }
};
printStudent(student1);
```

We use Interface to define the rules and the structure of an Object to be passed in to any function.

Interfaces may contains methods with only declaration and no implementation will be provided inside an interface. We must have to implement an interface to a class and implement those methods.

Classes in TypeScript

In order to develop any software system in object oriented way or approach, we must needs to create few classes to represent a blueprint of various components of a software system.

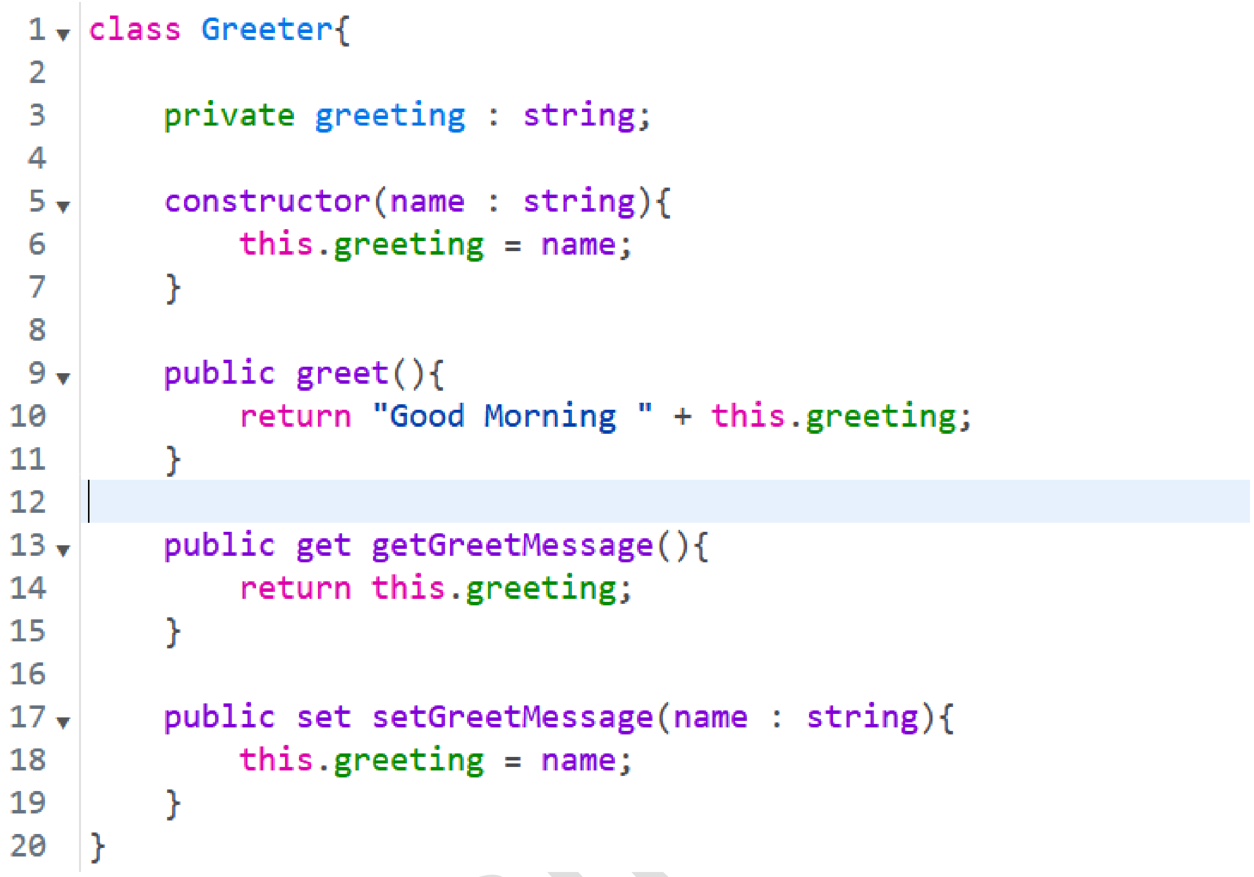
A class is just a template or blue print and allow to create few objects to perform operations on the data. A class contains few properties to holds data of the system and also contains few methods to perform some operations on the data.

We normally create Objects for those blueprint classes and manipulate the data using the properties and methods of that particular class. Once any object is created for any class then all the properties and methods are available to the objects to manipulate the data.

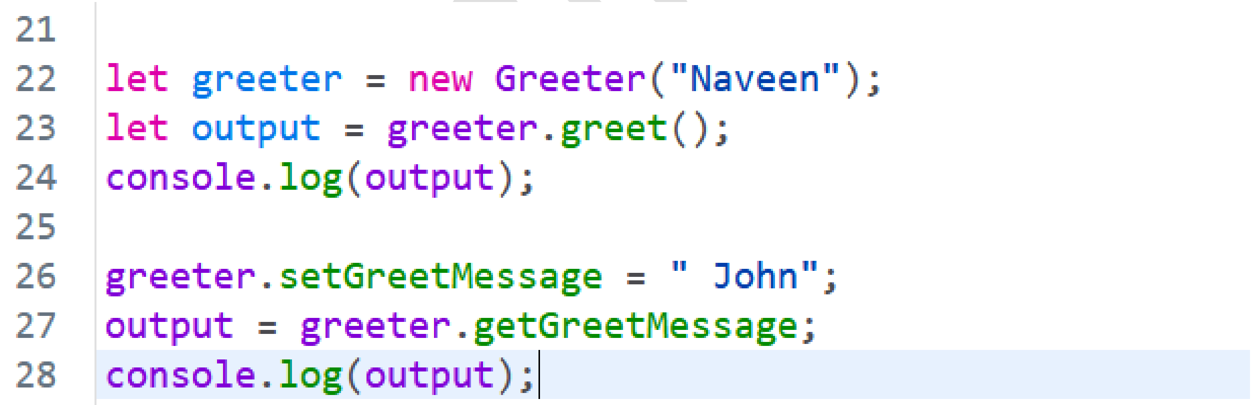
We can also write an independent classes or first we create an interface for providing rules and we may implement those rules of an interface inside a class by implementing an interface.

Let's understand the concept of classes in TypeScript and creation of Objects to manipulate the data of a particular class.

We can create a class by using the keyword called 'class'.



```
1  class Greeter{
2
3      private greeting : string;
4
5      constructor(name : string){
6          this.greeting = name;
7      }
8
9      public greet(){
10         return "Good Morning " + this.greeting;
11     }
12
13     public get getGreetMessage(){
14         return this.greeting;
15     }
16
17     public set setGreetMessage(name : string){
18         this.greeting = name;
19     }
20 }
```



```
21
22 let greeter = new Greeter("Naveen");
23 let output = greeter.greet();
24 console.log(output);
25
26 greeter.setGreetMessage = " John";
27 output = greeter.getGreetMessage;
28 console.log(output);
```

In the Above example, line number specifies the creation of a class in TypeScript.

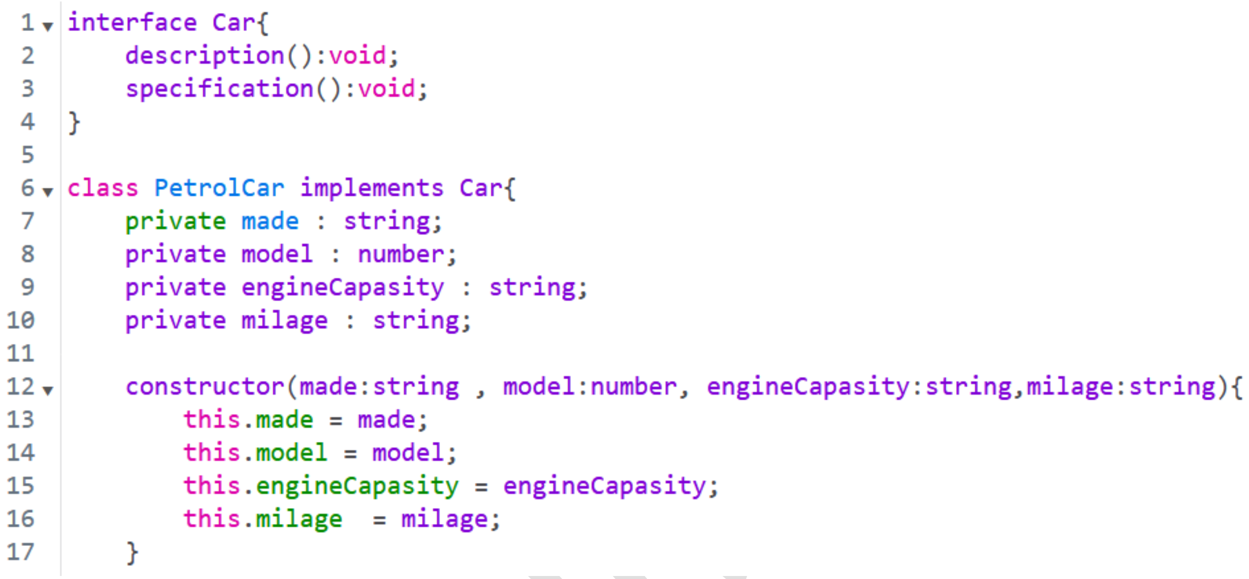
Line number 3, specifies creation of properties of a class.

Line number 5-7 indicates a constructor, a constructor is just like a normal function / method which is executed whenever we create an object to a class and to provide a default implementation or data to an object.

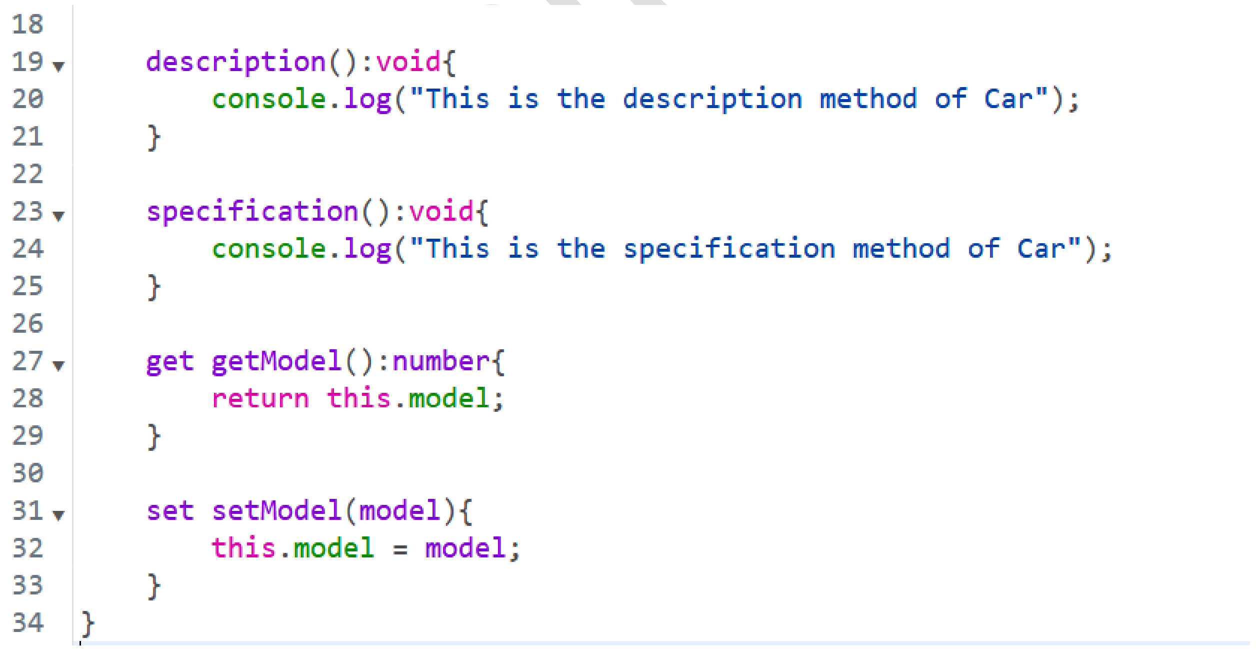
Line number 13 – 16, we call it as getter methods, which are used to access the properties of an Object. In the concept called encapsulation we should not have direct access to any variables or properties of a classes. We can only access the properties using getter methods.

Line number 17 – 19 indicates a setter methods to set values to the properties of a class.

We can even create a class by implementing an interface as follows,



```
interface Car{
    description():void;
    specification():void;
}

class PetrolCar implements Car{
    private made : string;
    private model : number;
    private engineCapasity : string;
    private milage : string;

    constructor(made:string , model:number, engineCapasity:string,milage:string){
        this.made = made;
        this.model = model;
        this.engineCapasity = engineCapasity;
        this.milage  = milage;
    }
```



```

    description():void{
        console.log("This is the description method of Car");
    }

    specification():void{
        console.log("This is the specification method of Car");
    }

    get getModel():number{
        return this.model;
    }

    set setModel(model){
        this.model = model;
    }
}
```

```
let petrolCar = new PetrolCar('Marthi Suziki',2018,'1300CC','20kmpl');
let out:number = petrolCar.getModel;
let output = "The car model is : " + petrolCar.getModel;
console.log(output);
document.getElementById('display').innerHTML = output;
```

Usage of Access Modifiers

The mainly supported modifiers in TypeScript are as follows,

1. Private
2. Protected
3. Public

1) **Private**: This is the most restricted modifier.

This we can use for properties or methods.

Any member which is declared with the private property, it will not be accessed outside of the class where it is declared.

```
class greet{
     private greetMsg:string;
     private welcomeMsg:string;
}
```

NOTE: For getter and setter methods, the access modifier should be same. If we try to change it, we may get compile time error.

2) **Protected**:

This is a least restricted modifier than Private.

This can be used only in the base class.

The protected modifiers can be access in the same class and its implemented/child class only.

```
class greet{
     protected greetMsg:string;
     protected welcomeMsg:string;
}
```

3) **Public**: This is the lease restricted modifier among all the other two modifiers.

Any member declared with this modifier even outside if the class also we can access them.

```
class greet{
    public greetMsg:string;
    public welcomeMsg:string;
}
```

Generics

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

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

Hello World of Generics

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the echo command.

Without generics, we would either have to give the identity function a specific type:

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

This function takes in a number and do same calculations and returns a number back again.

In future if this same method/component has to accept any other type as well.

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

We can define this 'any' type but here actually we are missing the behavior. So it takes in 'any' type so that can be string/number/object. But we cannot guarantee that whatever comes in we are returning the same.

In order to avoid this, we will be using 'generics' concepts.

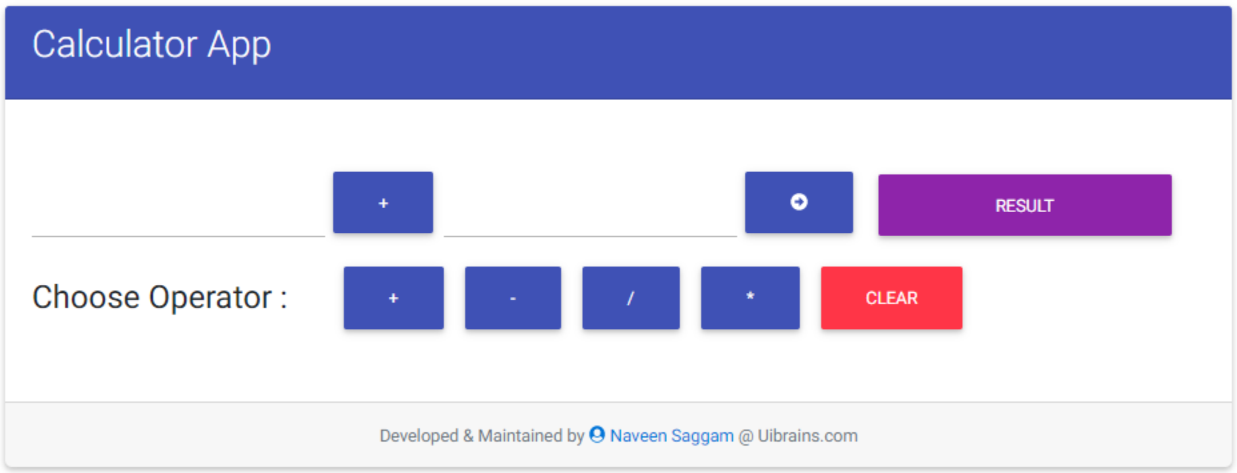
```
function identity<T>(arg: T): T {
     return arg;
}
```

In the above, function takes in a 'generic' type and do same calculations inside and returns the same type we passed in. This gives the guarantee that provided type is same as returned type.

Mini Project

In this Mini Project, we can create an app called Simple Calculator App by using all the Object Oriented concepts we learned as of now.

The App looks as follows,


Calculator App
+
RESULT
Choose Operator :
+
-
/
*
CLEAR
Developed & Maintained by Naveen Saggam @ Uibrains.com