What is TypeScript?

- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- TypeScript is pure object oriented with classes, interfaces and statically typed like C# or Java.
- It is projected as scalable JavaScript
- The preferred language to build Angular 2 applications is Typescript.
- Angular 2 itself was written in TypeScript.
- It was designed by **Anders Hejlsberg** (designer of C#) at Microsoft.

Features of TypeScript

- TypeScript is transpiled to JavaScript
- It can use any JavaScript code and library
- It is portable as it runs on any browser, any host and device. All that is needs is support for JavaScript and nothing else.
- Based on ECMAScript5 ,ECMAScript6, ECMAScript7 and ECMAScript8
- Provides data types and strong typing

Getting Started

Use one of the following ways to write, compile and run typescript code.

If you want to try typescript code online then use https://www.typescriptlang.org/play

If you want to type TypeScript code in your system then install NodeJS, followed by TypeScript into local systems as follows:

1. Install Node.js by downloading .msi file from https://nodejs.org/en.
2. Install TypeScript as follows:

```
npm install –g typescript
```

Identifiers in TypeScript

Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are:

I. Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
II. Identifiers cannot include special symbols except for underscore (_) or a dollar sign ($).
III. Identifiers cannot be keywords.

IV. Identifiers are case-sensitive.

TypeScript – Keywords

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

break	As	Any	switch
case	If	Throw	else
var	Number	String	get
module	Type	instanceof	typeof
public	Private	Enum	export
finally	For	While	void
null	Super	This	new
in	Return	True	false
any	Extends	Static	let
package	Implements	interface	function
new	Try	Yield	const
continue	Do	Catch	

Data types in TypeScript

Data types can be divided into 3 types – **Any type, built-in types and user-defined types**.

Any type

The **any** data type is the super type of all types in TypeScript. It denotes a dynamic type. Using the **any** type is equivalent to opting out of type checking for a variable.

Built-in types

The following table illustrates all the built-in types in TypeScript:

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

Declaring variables

The following syntax is used to create variables in TypeScript.

```
var identifier = [type-annotation] = value ;
```

Examples:

```
var name     : string = “Aspire”;
var   name   = “Aspire” // variable’s type is inferred from value
var   name;       // its type is any and value is undefined.
```

Type Assertion

- Type assertion is the process of converting variable from one type to another.
- We need to put target type in <> in front of source variable or expression.

```
var v1 : any = "Aspire";
var len = (<string>v1).length;
console.log(typeof(len));
console.log(typeof(v1));
console.log(len);
```

Compiler decides the data type of the variable using type inference. Once data type of a variable is decided, its type cannot be changed.

var v2 = 10;	type of v2 is number
v2 = "20";	ERROR as string cannot be set to number type

Variable Scope

The scope of a variable specifies where the variable is defined and available

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

Ex:

```
var g : number = 1;
class Test {
    static sv : number = 2;
    iv : number = 3;

    print1(): void {
        var i : number = 4;
        console.log("Local : " + i);
        console.log("Instance variable : " + this.iv);
```

```
            console.log("Static variable: " + Test.sv);
            console.log("Global Variable: " + g);
        }
  }
  var obj = new Test();
  obj.print1();
```

Loop

TypeScript supports the following looping structures.

1) while
2) do while
3) for
4) for .. in
5) for .. of

```
var marks: number[] = [ 10,30,40 ];
var i: number;

i = 1;
while (i <= 10) {
      console.log(i);
      i++;
}

i = 1;
do {
      console.log(i);
      i++;
}
while (i <= 10);

for (i = 1; i <= 10; i++){
      console.log(i);
}

for (var idx in marks) {
      console.log(`Marks for student ${ idx+1} are ${marks[idx]}`);
}
for (var m of marks){
      console.log(m);
}
```

Functions

The following are important features related to functions in TypeScript.

- ❖ Optional parameters
- ❖ Rest Parameters
- ❖ Default values to parameters

General syntax to create a function is as follows:

```
function  function_name(param[?] : datatype [=value],
param[?]:datatype [=value]) : returntype{
}
```

- Number of actual parameters and formal parameter must match in TypeScript.
- It is possible to explicitly mention that a parameter is optional by using ? after parameter name.
- It is possible to assign default value to formal parameter by giving = followed by value after parameter. Then this parameter acts as optional parameter.

Function definition

```
function add ( n1: number, n2:number) : number{
        return n1 + n2;
 }
console.log( add(10,20));

// Another way to define function
let   sub = function( n1: number, n2:number) : number{
         return n1 - n2;
 }
console.log(sub(50,20));
```

Lambda Functions

Anonymous functions can be represented using lambda expressions or lambda statements.

A lambda function contains the following components:

- Parameters
- Fat arrow
- Statements or expression

```
// Lambda Expression

var nextEven = (n : number) => n % 2 == 0 ? n + 2 : n + 1;
```

```
console.log(nextEven(10));

// Lambda block
var nextOdd = (n: number) => {

        console.log("Before : " + n);

        n = n % 2 != 0 ? n + 2 : n + 1;

        console.log("After : " + n);

}

nextOdd(10);

var calculateSquare = (num:number)=> {
  num = num * num;
  console.log(num);
}
calculateSquare(10);
```

Optional Parameter

```
// Optional parameter - n2 declared with ? after parameter name

function mul(n1 : number, n2? : number) : number {
        if (n2) // if parameter is passed return n1 * n2;
        else
                return n1 * 10;
}

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

Default Parameter

```
//Setting second parameter n2 to default value
function div(n1 : number, n2 : number = 10) : number
{
        return n1 / n2;
}
console.log( div(100,5));
console.log( div(100));
```

Rest Parameters

It is possible for a function to take variable number of parameter by declaring a formal parameter with … (three dots) before name as the following example show.

```
// Rest parameters
function print( message : string , ... names : string[])
{
        for(let n of names)
           console.log( message + " " + n);
}

print("Hello", "Ben","Joe");
print("Hi ", "Scott","Anders","Tom");
```

Function Overloading

Here is an example for function overloading.

```
// declare functions to be overloaded

function f1(x:number): void;
function f1(s: string): void;
function f1(x: number, s: string): void;

function f1( n: any, s? : any ): void {
      console.log(`value is ${n}. Type is ${typeof (n)}`);
      if (s){
            console.log(`Second parameter is ${s}`);
            }
}
f1("Abc");
f1(10);
f1(100, "PQR");
```

Numbers Data Type

Number data type represents a number.

```
// Number demo

console.log(Number.MAX_VALUE);
var n = new Number(10.7867);
console.log(n.toFixed(2));
console.log(n.toLocaleString());
console.log(n.toLocaleString("es"));

var qty = 255;

console.log(qty.toString(16));
```

Arrays

An array is a collection of items of same type. In TypeScript an array is an object.

```
//Arrays Demo

var marks: number[] = [60, 70, 66];

console.log(marks.length)
console.log(marks[0]);

var subjects: string[] = ["Java", "TypeScript", "Angular"];

for (var i = 0; i < subjects.length; i++)

      console.log(subjects[i]);

// Use iterator
for (var sub of subjects)
      console.log(sub);

//   Array Methods subjects.push("jQuery");
console.log("Top Element  : " + subjects.pop());

//   Print all elements
subjects.forEach((v,idx,a) => console.log(v));
```

Tuple

- A tuple is a heterogeneous collection of values.
- Individual elements are called as items.
- Tuples are index based and index starts at 0.
- Tuples are mutable, so we can manipulate them using methods and simple indexed access.
- It is possible to deconstruct a tuple – copy value to individual elements.

```
// Tuple demo

var tup1 = [10, "Abc", true];

console.log(tup1[0]);
console.log(tup1.length);

//   change an item in tuple tup1[2] = false;

//   destructuring tuple

var [i1, i2, i3] = tup1;

console.log("Second Item : " + i2);
for (var v of tup1)
console.log(v);
```

Classes

- TypeScript supports classes, which were introduced in ES6.
- A class may contains fields, constructors, and methods.
- We instantiate objects using new keyword followed by classname.
- Fields and methods are accessed using dot operator (.).
- Constructors are defined using keyword **constructor**.
- Keyword **extends** is used to specify super class.
- TypeScript does NOT support multiple inheritance.
- Super class is accessed using **super** keyword.
- Classes can have static members that represent data and operations related to class and declared using **static** keyword. Static members are accessed through classname.
- Classes can implement interfaces using **implements** keyword.

```
class  class_name {
    // Members
}
```

Access Specifiers

The following specifiers are allowed for members in class.

public

A public data member has universal accessibility. Data members in a class are public by default.

private

Private data members are accessible only within the class that defines these members.

protected

A protected data member is accessible by the members within the same class and also by the members of the sub classes.

```
class Product
{
    readonly  name =’srini’;
     protected name :string;
     protected price : number;
     constructor(name :string, price : number) {
            this.name = name;
            this.price = price;
     }
     print():void {
            console.log(this.name);
            console.log(this.price);
     }
}

var per1 = new Product("iPhone7 Plus", 70000);
per1.print();
console.log(per1.name);
per1.name=””;//not possible because name is readonly

class TaxProduct extends Product {
     protected tax : number;

     constructor(name :string, price : number, tax:number) {
              super(name,price);
              this.tax = tax;
     }
     print():void {
            super.print();
            console.log(this.tax);
     }
     getNetPrice(): number {
            return this.price + this.price * this.tax / 100;
     }
}

var tp = new TaxProduct("Dell Laptop",65000,12);
tp.print();
console.log("Net Price : " + tp.getNetPrice());
```

Enum:

To read constants

Enums allow us to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

Numeric enums:

```
enum Daysofweek{
   SUN=100,MON,TUE,WED,THR,FRI,SAT
   }
   let day: Daysofweek;
   console.log(Daysofweek.MON);
   if(day ==Daysofweek.MON){
      console.log(day);
   }
```

An Enum is a datatype consisting of a set of named values. The names are usually identifiers that behave as constants. Enums were introduced in ES6.

```
enum Direction {
        Up,
        Down,
        Left,
        Right
}
let go: Direction;
go = Direction.Up;
```

```
enum Color {
  Red = '#ff0000',
  Green = '#00ff00',
  Blue = '#0000ff'
}
const myFavoriteColor = Color.Green;
let chosenColor = Color.Red;
```

In this case myFavoriteColor will be of the value '#00ff00' during runtime and it has the type of Color.Green since it's a constant. For the chosenColor, TypeScript will automatically walk one step up and assign it the type Color since it could change over time. The value assignned during runtime will be '#ff0000' as expected.

Generics:

With *generics* we can dynamically generate a new type by passing into the Post type a *type variable*,like so:

```
class Audio {}
class Video {}
class Link {}
class Text {}

class Post<T> {
        content: any;
}
```

<T> above is the *generic* syntax and T is the *type variable*. We could name it anything but the convention if there is only one is to call it just T.

Then we can use T wherever we would use a type, like so:

```
class Post<T> {
        content: T;
}
```

Ex2:

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

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

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

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

      }
      var argstr1:string=abc(1);//error

      var argstr =abc(1);//ok
```

We've now added a type variable T to the identity function.
This T allows us to capture the type the user provides (e.g. number), so that we can use that information later.

Interface

- An interface contains a collection of methods, properties and events.
- Interface contains only declarations and implementing classes provide definition.
- Interfaces are TypeScript only constructs. They are not converted to JavaScript.

```
interface interfacename {
      // members
      }
```

The following example shows how to use interface and inheritance in interfaces.

```
interface Person {
        name : string;
        age : number;
        toString : () => string;
```

```
}
//   Inheritance in Interface
interface Student extends Person
{
    course : string;
}

let p1: Person = {name : "Richards", age : 40 , toString : function(){
                return this.name + ":" + this.age;
        }
};

function print(v : Person)
{
      console.log(v.toString());
}

print(p1);

let s1: Student = {name : "Mark", age : 20 ,course :"Angular",
        toString : function()
        {
                return this.name + ":" + this.age + ":" + this.course;
        }
};

print(s1);
```

Duck-typing

In duck-typing, two objects are considered to be of the same type if both share the same set of properties. Duck-typing verifies the presence of certain properties in the objects, rather than their actual type, to check their suitability.

The TypeScript compiler implements the duck-typing system that allows object creation on the fly while keeping type safety.

```
interface IPoint {
  x:number
  y:number
}
function addPoints(p1:IPoint,p2:IPoint):IPoint {
  var x = p1.x + p2.x
  var y = p1.y + p2.y
  return {x:x,y:y}
}

//Valid
```

```
var newPoint = addPoints({x:3,y:4},{x:5,y:1})
console.log(newPoint);
//Error
//var newPoint2 = addPoints({x:1},{x:4,y:3})
```

VAR Vs LET Difference:
var is scoped to the nearest function block and let is scoped to the nearest enclosing block, which can be smaller than a function block. Both are global if outside any block.

Also, variables declared with let are not accessible before they are declared in their enclosing block.

```
function abc(){
  let a =10;
  console.log(a);  // output 10
  if(true){
    let a=20;
   console.log(a); // output 20
  }
  console.log(a);  // output 10
}

abc();
```

Object →is basically anything that is not undefined or null.
object →represents anything that isn't a primitive type (string, number, boolean, etc.) plus undefined.

```
const srini: Object = 'srini';
const hai: object = 'hai'; // ==> This is invalid
const cdate: object = new Date();
```

Modules:

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

"Internal modules" are now "namespaces". "External modules" are now simply "modules", as to align with ECMAScript 2015's terminology, (namely that module X { is equivalent to the now-preferred namespace X {)

Decorators:

A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.