# TypeScript

- Strong Typing
- Object Oriented features
- Catch errors at compile time
- Access to great tools

> TypeScript ====(Trasnpile)==> JavaScript

## Hello World in TypeScript :

- First Install TypeScript

```
$ npm install -g typescript
```

- To check the Version of Typescript and confirm your installtion type:

```
$ tsc --version
```

- First create a hello.ts file after that transpile that file

```
$ tsc hello.ts
```

- Now you have a hello.js in the same directory so we will execute this file with node

```
$ node hello.js
Hello World
```

> You can also combine above two steps using pipe
> $ tsc hello.ts | node hello.js

### JavaScript Versions

- ES5 (ECMAScript5) : Supported by all browsers
- ES6 (2015) : (Let keyword was introduced)
- ES2016
- ES2017

### Declaring a variable

#### Let Vs Var

- Variables Declared with 'var' keyword is scoped to the nearest function.
- Variables Declared with 'let' are are scoped to nearest block.
- Though we will get error during compilation but Transpilation will be successful and generate a functioning Js file in ES2015.
  Instead of 'let' that Js File will have the var keyword and it will work. But this may give some logical error.
  So better go with 'let' while declaring variable from now on.

```
//main.ts
function letVsVar() {
    //using the var keyword
    for (var j = 0; j < 5; j++) {
        console.log(j);
    }
    console.log('Finally' + j);  //(J is scoped to nearest fucntion i.e. letVsVar here
so it gives no errors)

    //using the let keyword
    for (let i = 5; i < 10; i++) {
        console.log(i);
    }
    console.log('Finally ' + i); // (i is scoped to for block so it gives "Error Can't
find name 'i'")

    //But in Javascript file the let will be replaced by var as per transpilation to
ES2015 and JS will run smoothly.
}
```

On compilation it gives an error like

```
$ tsc main.ts
main.ts(12,27): error TS2304: Cannot find name 'i'.
```

But it successfully transpile to a JavaScript file

```
//main.js after transpilation
function letVsVar() {
    //using the var keyword
    for (var j = 0; j < 5; j++) {
        console.log(j);
    }
    console.log('Finally' + j); //(J is scoped to nearest fucntion i.e. letVsVar here so
it gives no errors)
    //using the let keyword
    for (var i = 5; i < 10; i++) {
        console.log(i);
    }
    console.log('Finally ' + i); // (i is scoped to for block so it gives "Error Can't
find name 'i'")
    //But in Javascript file the let will be replaced by var as per transpilation to
ES2015 and JS will run smoothly.
}
```

## Types in Typescript

We have basic variable types like number, boolean, string, enum etc.

```
let a:number;
let b:boolean;
let c: string;
let d:any; //Will behave like var d;
let e:number[];
let f:number[]=[1,2,3]
let g:any[]=[1, true, 'a'];

//Enum
enum Color {Red,Green,Blue};
//Red will be automatically assigned 0 and so on for green and blue but we should
statically give them value for unexpected future errors.

enum Color{Red=0, Green=1, Blue=2}
//so in future if someone add Purple in between our enum by mistake our code will not be
impacted
```

---

## Type Assertion

```
let message;
message='abc';
//let noIntellisense= message. //We will get nothing.
let assertTypeAsString =(<string>message).endsWith('c');
let anotherWayToAssert =(message as string).endsWith('c');
```

> This type assertion doesn't change the type of variable at run time. It's just a way to tell TypeScript compiler about variable's type so that features like Intellisense can work.

---

## Arrow Functions

```
let log =function(message){
    console.log(message);
}

//Using arrow function
let dolog= (message)=>{
    console.log(message);
}

//We can remove {} when its just one line and () when there is only one argument
//but it makes code less readable
let dologMinified= message => console.log(message);

//With no parameters
let dologWithoutParameter = () => console.log("hello");
```

---

## Using Custom type in TypeScript [Interface]

### In line Annotations

```
let drawPoint =( point:{x: number, y: number})=>{
    // ...
}

drawPoint({
    x: 1,
    y: 2
})
```

In line annotation works well with simple cases but it is a little bit verbose. Chances are there may be some other function expecting the same point object. We don't need to repeat it. So to tackle that we will use interface.

```
interface Point{
    x: number,
    y: number
}
let drawPoint =(point: Point) => {
    //...
}
let getDistance = (pointA: Point, pointB: Point)=>{
    //...
}
}
//When using interface use Pascal Naming convention for interface Name
```

> **Cohesion** Things that are related should go together

> We are violating **Cohesion** in above example as that interface and that standalone drawPoint

> function are highly related with each other. We use Class to solve this issue.

## Class

Groups Variables (properties) and functions (methods) that are highly related.

```
class Point{
    x: number;
    y: number;
    //draw: ()=> void No need to pass Point as parameter to draw as it can access x and
y as they are part of same unit.

    draw() {
        console.log('X: '+ this.x + ', Y: ' + this.y);
    }

    getDistance(another: Point){
        //...
    }
}

let point: Point; // It will give error as While declaring a custom object we need to
explicitly allocate memory to it.
let point = new Point();
point.x=1,
point.y=2
point.draw();
```

## Constructor

```
class Point{
    x: number;
    y: number;

    constructor(x?: number, y?:number){
        this.x=x;
        this.y=y;
    }
    //We made the parameters optional as JS has no constructor overloading

    draw() {
        console.log('X: '+ this.x + ', Y: ' + this.y);
    }
}
let point()= new Point();
point.draw();

let point2= new Point(1,2);
point2.draw();
```

## Access Modifiers

- Public
- Private
- Protected

> By default all members are public. You don't need to add public before function or member that you don't want private. As that is just redundant code.

### Access Modifiers in Constructor parameter

In our last code a lot of thing is redundant. We can remove those redundancy by prefixing access modifier in constructor parameter whether public or private.

```
class Point{
   constructor(private x?: number,private y?:number){
   }
   //TypeScript compiler generate a field with exact same name and also initialize this field with value of this argument.
   //If we would have applied Public access modifier then we would have been able to access that field using object outside the class.
   draw() {
       console.log('X: '+ this.x + ', Y: ' + this.y);
   }
}
let point()= new Point();
point.draw();

let point2= new Point(1,2);
point2.draw();
```

## Properties

In our last example we have a tiny problem, we can set initial coordinate of a point object as well as we can call the draw function but we can't get the value of x and y.
We can simply define a method like

```
getX(){
    return this.x;
}
```

We want to give user freedome to set new value of x after doing some validation.

```
setX(value){
    if(value<0)
        throw new Error('value cannot be less than 0.');
    this.x=value;
    }
}
```

Is there anything else we can do?

Yes, We can use property. let's define a property.

```
class Point{
    constructor(private _x?: number,private _y?:number){
    }
    draw() {
        console.log('X: '+ this._x + ', Y: ' + this._y);
    }
}
let point()= new Point();

//Setting value of a private field x on basis of some validation
set x(value){
    if(value<0)
        throw new Error('value cannot be less than 0.');
    this._x=value;
}
set y(value){
    if(value<0)
        throw new Error('value cannot be less than 0.');
    this._y=value;
}

//Getting value of private field x
get x(){
    return this._x;
}

point.draw();
```

Property looks like a field from outside but it's more of a combination of getter and setter function.

## Modules

A simple pragmatic explaination
In the below example Point is a module and we exported it from point.ts and imported in main.ts and used it there.
**point.ts**

```
export class Point{
    constructor(private _x?: number, private _y?:number){

    }
    draw(){
        console.log('X: '+this._x + ', Y: '+this._y);
    }
}
```

**main.ts**

```
import {Point} from './point'

let point = new Point(1,2);
point.draw();
```

In typeScript we divide our file in multiple files in each file we export one or more types, These types can be classes, function or simple objects.
When we have a import or export statemenet on top of a file, that file is a module from typescript's point of view.
In Angular we also have the concept of modules but angular modules are a little bit diffrent they are not about organization of code in different files. They are aboout organization of your application into smaller functional area.
For Now this much info is enough about modules. We will learn more about Angualar module later.