TypeScript

What is typescript

JavaScript became one of the most popular and important languages today. Javascript was adopted as the programming language by every modern browser today and it's the only language that browsers understand. JavaScript is an interpreted language, weakly typed and dynamic by nature which has his pros and cons. Due to its nature other languages that compile to JavaScript were created and typescript is one of them. TypeScript is a strongly typed language that comes with a compiler. After we finish writing our typescript files we can compile them and the compilation will output a js file of the code we wrote, **browsers can't run typescript file directly, so we need to compile the files in order to run them on the browser**. TypeScript was developed by microsoft and has strong support for types and easy to use OOP features. You can also add a configuration file which you can use to specify how the JS file will be created, when writing typescript you can also use old javascript which the typescript compiler will understand.
TypeScript is a superset of JavaScript which means everything that works on JavaScript, will work on typescript (not exactly true but that's what they claim).
We will use angular2 with typescript so we will cover here the main features of typescript that you need to know in order to start with angular2.

Installing the environment

- First you need to install node and npm by following this link
- Now you need to install typescript as a global package by typing:
  > npm install -g typescript
- Open a new directory where we will experiment with typescript
  > mkdir typescript-tutorial
  > cd typescript-tutorial
- Create a file called **tsconfig.json** - this will hold the configuration for our typescript compiler
- Let's explain on main **tsconfig.json** options

tsconfig.json

The tsconfig file indicate that this is a root folder of a typescript project, and also it allows us to add configuration to the typescript compiler.

If you add a **tsconfig.json** file you can run compilation without specifying which file to compile than typescript will compile all ts files in the root folder and all sub folders.
You can also specify in the tsconfig which files to compile by adding the **files** array.

```
{
 "files": [
   "person.ts",
   "pokemon.ts"
 ]
}
```

Now when you run:
> tsc
It will compile only the files specified in the array.
If you don't specify the files it will compile all the **ts** files in the root and subdirectory.
You can also specify **exclude** array if you want to compile all files except the ones specified.
It's common to put 3rd party packages in the exclude array. For example:

```
{
   "exclude": [
      node_modules
   ]
}
```

You can't put the **files** and **exclude** together and if they are together typescript will only check the **files** array.
You can also specify configuration for the compiler in the key **compilerOptions.**
Example:

```
{
 "compilerOptions": {
    "removeComments": true,
    "declaration": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
 }
}
```

You have the full specs of the **compilerOptions** in this link
Here are some of the more popular options

- declaration - set to true if you want the compiler to create a declaration file with the api of the file compiled
- emitDecoratorMetadata - important to set to true on angular projects, will remove the design type metadata added on decorators, angular sets it's own metadata with decorators so it's important to add this option with angular projects

- experimentalDecorators - set to true to use decorators, must be set to true with angular projects since angular uses decorators.
- removeComments - will remove the comments from the compiled files

The file can return an empty object and it will use the default values
For now leave the **tsconfig.json** to look like this:

```
{
"compilerOptions": {
   "removeComments": true,
   "declaration": true,
   "emitDecoratorMetadata": true,
   "experimentalDecorators": true
}
}
```

Declaring Variables

Var Scope

Until now we defined variables by typing **var**
Let's try and examine the scope of a variable defined with **var.**
What will the following code print:

```
function scopeVar(isBoolean){
  if(isBoolean){
     var message = 'hello world';
  }
  console.log(message);
}
scopeVar(true);
scopeVar(false);
```

And what will this code print?

```
for(var i=0; i<10; i++){
  for(var i=0; i<10; i++){
     console.log(i);
  }
}
```

Variables declared with **var** if they are in a function their scope is the function (even if they are in an if block or a for loop)

If the variable is declared outside a function the variable scope is global.
So the first code will run the function two times:

- The first time it will enter the if block and define the variable then print the message defined in the if block
- The second run of the function it wont enter the if so the variable will not be defined and the print will return **undefined**

For the loop the **i** variable is defined once globally so on the inner loop it will print 0,…, 9 and then exit the inner and outer loop

ES6 Variables
When using typescript we can use some **ES6** among them are variable declaration using **let** and **const**

let

The **let** statement declares a block scope local variable, and optionally initializing it to a value. Meaning if we are in an **if** block and define a **let** variable, then the scope of that variable will be that **if** block. So if looking at the previous code and replacing the declaration with **let:**

```
function scopeVar(isBoolean){
  if(isBoolean){
     let message = 'hello world';
  }
  console.log(message);
}
scopeVar(true);
scopeVar(false);
```

This time since the variable is declared with **let** it will be local to the if block. So in the two runs of the function **undefined** will be printed.

What will be printed here?

```
for(let i=0; i<10; i++){
  for(let i=0; i<10; i++){
     console.log(i);
  }
}
```

This time both of the loops will run 10 times, so the numbers 0, … 9
Will be printed 10 times.

const

The const declaration creates a read only reference to a value. Notice that it doesn't mean the value is immutable, we can place an object or an array to a **const** variable and change the object or array, what we can't do is assign a new value to that variable after it has been initialized. That means that when you define a **const** var you have to initialize it's value.
Example:
const num = 20;
num = 15; //error
const obj = {'hello': 'world'};
obj['hello'] = 'world2'; //change the value correctly

Arrow Functions

Also called lambda function allows us to define a function without writing **function.** This feature was adopted in **ES6**
Problem with JS functions is that sometime they lose the meaning of this. For example consider the following code.

```
function Person(age){
  this.age = age;
  this.birthday = function(){
      this.age++;
  }
}

var yariv = new Person(31);
setTimeout(yariv.birthday, 1000);
setTimeout(function(){
  console.log(yariv.age);
}, 2000);
```

We define a **class** here named **Person**, our **Person** class constructor get's the age of the person, and we save that age in the **age** property.
We also defined in our class a **birthday** method which increases the **age** by one.
In JS this sometime tends to get lost, in the following example we are instantiating a **Person** with age of 31, we then call the **birthday** method in a **setTimeout**, when we call the instance in the context of **setTimeout** the **this** gets lost and become the **setTimeout** method, which means our birthday **this.age++** didn't quite do anything since **this.age** will be undefined. So the second **setTimeout** will print 31 and not 32 as we expect.
The arrow function preserves the original this where the function was defined, so changing our class to this:

```
function Person(age){
  this.age = age;
  this.birthday = () => {
     this.age++;
  }
}
```

Will fix our problem.
The arrow function also allows us to write a one line function is the following syntax.

```
function Person(age, name){
  this.age = age;
  this.name = name;
  this.birthday = () => {
     this.age++;
  }
  this.getName = () => this.name;
}
```

Notice the **getName** method where we return **this.name** without curly brackets and without the return keyword.

Modules

Starting with ES6 JavaScript has a concept of modules which TypeScript adopted. Modules are executed with their own scope so a variable declared in file **person1.ts** is not available in file **person2.ts** unless it's explicitly export.
It's common in angular2 projects to split the project sections to multiple files and import and export them when needed, we are also using modules when working with angular library so this subject is really important for angular2 development
Let's try to do a small example.
Create a file called **person-constants.ts** which will hold constants to use in the **person.ts** file:

```
export const GREETING = 'hello world';
```

Create another file called: **person.ts**

```
import {GREETING} from './person-constants';

console.log(GREETING);
```

Make sure your **tsconfig.json** file has:

```
{
"compilerOptions": {
  "target": "ES5"
}
```

```
}
```

In your terminal type:
> tsc
> node person.js

You should see the const was exported from the **person-constants.ts** file and printed in the terminal.

export can be written to **variables, classes, interface, function.**
Another example in your **person-constants.ts** add the following:

```
export const GREETING = 'hello world';

export function sayHello(message){
  console.log(message);
}
```

We are exporting here a constant and a function. Now in our **person.ts** add the following:

```
import * as Constants from './person-constants';

Constants.sayHello(Constants.GREETING);
```

We exported everything in the **person-contants.ts** file by using the * and naming the things we exported as the **Constants** namespace.

You can also add in a file the **default** keyword for exporting without specifying the exact name.
In **person-constants.ts**:

```
export default function sayHello(message){
  console.log(message);
}
```

And to use the function in **person.ts:**

```
import mainFunction from './person-constants';

mainFunction('wat');
```

Notice that you no longer need to put the curly braces and specify the exact name of the function.

Type Annotation

Type annotation in TypeScript allows us to specify the intended type of a variable, a return function value, or function arguments. TypeScript will report an error if the wrong type is passed.

The general syntax for providing type is:
<var name> : <TypeAnnotation>
For example, let's try and define a function which returns a boolean value and let's return an integer and try to compile that file.
Create a new file called **type-annotation.ts** and place the following:

```
function isTrue() : boolean {
  return 1; //this should fail the compilation
}
isTrue();
```

Try compiling this file and you should see the following error:
type-annotation.ts(2,12): error TS2322: Type 'number' is not assignable to type 'boolean'.

We can also define types to arguments in our function declaration.
Change **type-annotation.ts** file to look like this:

```
function sayHello(message : string) : void {
  console.log(message);
}
sayHello(1);
```

Try and compile the new file, you should see the following error:
type-annotation.ts(4,10): error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'.

You can also specify type of variables:

```
let message : string = 'hello world';
```

If you assign a number value to the above variable then TypeScript will complain.

Primitive Types

The primitive types are:

- string
- number
- boolean

Example:

```
let myString : string = 'hello world';
let myNumber : number = 10;
let myBoolean : boolean = true;
```

Array Type

You can define a variable of type array. The general syntax is:
<variable name> : Array<Type Annotation>;
<variable name> : <type anotation>[];
Example of defining an array:

```
let myBooleanArray : Array<boolean> = [];
let myNumberArray : number[];
let myStringArray : Array<string> = [];
```

TypeScript will complain if you try to put a wrong value type in the array.

Interface

The interface allows us to define custom types and to create a contract between our types and what they expect to contain.
For example let's say we have a function which expect a dictionary object that has to contain the key **firstName** defining interface for our object will look like this:

```
interface Person{
  firstName: string
}

function sayMyName(person : Person){
  console.log(person.firstName);
}
```

If you call the above function without an object that contains a **firstName** key, we will get an error.

You can also define an interface to a **class** example:

```
interface Pokemon{
  name : string,
  sayHello() : string
}

class SeaPokemon implements Pokemon{
  public name : string;
  sayHello(){
     return 'hello';
  }
}
```

We defined here an interface called **Pokemon** every class that implements this **interface** should have a property called **name** and a method called **sayHello** which returns a string

Special Types

Typescript contains some special types:

- **any -** a variable of this type can contain any type
- **Object** - a bit more restrictive the **any** it doesn't allow to call functions on types defined as **Object**
- **null/undefined -** we can put these values to all type annotation
- **void -** can be used to specify function that doesn't return anything

Casting

Sometimes you know better what a type should be than the TypeScript compiler, for this reason TypeScript allows us to use casting from type to type.
For example let's say you have an object which implements an interface, we want our object to contain additional field which is not in the interface. Example:

```
interface Pokemon{
  name : string,
  power : number
}

let pokObject : Pokemon = {name: 'pikachu', power: 3};
pokObject.strength = 10; //this will produce error
(<any>pokObject).strength = 10;
```

We are defining an interface of an object with two keys and we are trying to add a third key, unless we use casting the compiler will complain about the third key.

Class

Although JavaScript is an object oriented scripting language, the mechanism to create classes by using functions is a bit counter intuitive. TypeScript embraced the sugar coat syntax in ES6 for creating classes so we can now create a class by using the keyword **class** our class can have a special function called **constructor** used for creating class instances.you can also defined class properties as private and public. Let's give a small example of creating a **Pokemon** class

```
class Pokemon{
```

```
  constructor(public name : string, public power? : number){}
  sayHello(){
      console.log(this.name + ' said hello');
  }
}

let pikachu = new Pokemon('pikachu');
pikachu.sayHello();
```

We created a class called **Pokemon** with a constructor that gets two arguments: **name** and **power.** The question mark next to **power** marks it as an optional argument. Notice that in the **constructor** we put **public** next to the argument which maps the arguments as public class properties. We also defined a method called **sayHello**. We are creating an instance of our class and calling the **sayHello.**

Decorators

Decorators are a way to add functionality to a class without changing the code or structure of the class.
Decorators are commonly used in Angular2 to attach metadata information to angular modules, components and directives.
Decorators can be attached to **class declaration, method, accessor, property, or parameter.**
The syntax of using a **decorator** is: **@expression** where expression is the decorator name, and need to evaluate to a function. Its also possible to apply multiple decorators.
In Angular it's very common to use **class decorators and property decorator.** The class decorator gets the constructor as argument, it's called when the class is declared and not when an object is instantiated, in the decorator function we are getting a param of the class prototype.
To use decorators in typescript you have to set the **experimentalDecorators** to **true** in **compilerOptions** in the **tsconfig.json** file.
In angular2 the decorators are usually applied with a decorator factory. To create a decorator factory you have to create a function that returns a function in the following syntax:

```
function component(metadata : any){
   return function(target){
      ...
   }
}
```

To apply the decorator above to a class you can simply:

```
@component({
   selector: 'app'
})
export class AppComponent {}
```

The decorator gets the class constructor and it can return a new constructor or modify the existing one.
Let's create a small example of a decorator that adds a name property with a default value:

```
function nameDecorator(name : string) : Function{
  return function(originalConstructor){
     var newConstructor = function newConstructor(...args){
        this.name = name;
        originalConstructor.apply(this, args);
     }
     return newConstructor;
  }
}

@nameDecorator('yariv')
class Pokemon{
  public name : string;
  constructor(public power : number = 10){
  }
}

var pikachu = new Pokemon();
console.log(pikachu.name);
console.log(pikachu.power);
```

In this example we are creating a class called **Pokemon,** our class contains a property called **power** and a property called **name,** and we want to create a decorator that will populate the name with a default value.
We created a decorator factory which returns a function that gets the **originalConstructor**, we are adding the new name and call the original constructor.