What is typescript

Typescript is a superset of Javascript, which is designed to develop large Javascript applications.

TS provides highly productive development tools for JS IDEs and practices, like static checking.

With TS, we can make a huge improvement over plain JS as TS gives us all the benefits of ES6 (ECMAScript 6), plus more productivity.

Why you should use Typescript?

Code is easier to understand

Usually when you work on a piece of code, e.g. function, you usually need to answer these questions:

1. What arguments does it accept?
2. What value does it return?
3. What external data does it require?

In dynamically typed languages, it is offten difficult to answer the first 3 questions, but in Typescript, you get answers to all of the above questions immediately from your IDE and compiler.

Other reasons are:

1. Code is easier and faster to implement,
2. Code is easier to refactor,
3. Less bugs,
4. Less boilerplate tests,
5. Code is easier to merge,
6. Aids the developer in having the correct workflow;

TYPESCRIPT CHEATSHEET

Type declarations

TS makes JS more strongly-typed, object-oriented language.

The strong typing also means that the language is precompiled and that variables cannot be assigned values that are out of their declared range.

For instance, when a TypeScript variable is declared as a number, you cannot assign a text value to it:

```
let age: number = 5;
age = 'I like dogs';
```

Error: Type 'string' is not assignable to type 'number'.

Basic types

String

As in other languages, we use the type string to refer to these textual datatypes. Just like JS, TS also uses double quotes (") or single quotes (') to surround string data.

```
const myName: string = 'Robert';
```

Number

As in JS, all numbers in TS are floating point values. These floating point numbers get the type number.

```
const myAge: number = 24;
```

Boolean

The most basic datatype is the simple true/false value, which JS and TS call a boolean value.

```
const isDone: boolean = true;
```

Defaults to undefined when not explicitly set

```
const isDone: boolean;
```

Can also be set to null as well

```
const isDone: boolean = null;
```

Void

Void is used where there is no data type.
For example, in return type of functions that do not return any value.

```
function sayHi( ): void {
        console.log( 'Hi!' )
}
let speech: void = sayHi( );
console.log(speech);
```

Output is undefined

! There is no meaning to assign void to a variable, as only null or undefined is assignable to void.

```
let nothing: void = undefined;
let num: void = 1;
```

Error!

Tuple

Tuple types allow you to express an array with a fixed number of elements whose types are known, but not the same. For example, you may want to represent a value as a pair of a string and a number:

```
let x: [string, number];
Initialize it
x = [«hello», 10]; <-- This is OK
x = [10, «hello»]; <-- Error, because of wrong order.
```

Array

TS, like JS, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
const hobbies: string[] = ['Programming', 'Cooking'];
```

The second way uses a generic array type, Array<elemType>.

```
let list: Array<number> = [1, 2, 3];
```

Any

Permits its any type. Expressions involving any are not type checked.
!! It is not recommended to use any, as it can cause unexpected problems later in your code.

Null and Undefined

By default null and undefined are subtypes of all other types.

That means you can assign null and undefined to something like number.

```
const myAge: number = undefined;
```

{} Object <-- you can define its properties or indexer object with name and age attributes

```
{name: string, age: number}
```

A dictionary of numbers indexed by string

```
{ [ key: string ]: number }
```

Enum

By default all **enum** values are resolved to numbers.
Enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {
Red,
Green,
Blue
};
let c: Color = Color.Green;
```

Real value behind Color.Green will be 1!

By default all **enums** start with a value of **0**

Or You can manually set the values in enum.

```
enum Color {
Red = 1,
Green = 35,
Blue = 100
}
let c: Color = Color.Green;
```

Now the value will be 35!

Never

The **never** type is used when you are sure that something is never going to occur.

```
function throwError(errorMsg: string): never {
        throw new Error(errorMsg);
}
```

Function returning never has an unreachable end point

Function Parameters

Parameters are values or arguments passed to a function.
In TS, the compiler expects a function to receive the exact number and type of arguments as defined in the function signature.

!In TypeScript, every parameter is assumed to be required by the function !

Optional parameters

TS has an optional parameter functionality.

The parameters that may or may not receive a value can be appended with a '?' to mark them as optional.

```
function dogProfile(dogName: string, dogAge?: number)
```

! All optional parameters must follow required parameters and should be at the end !

Default parameters

If the user does not provide a value to an argument, TS will initialize the parameter with the default value.

If a value is not passed for the default parameter in a function call, the default parameter must follow the required parameters in the function signature.

```
function dogProfile(dogAge: number, dogName = «Sprinkles»)
dogProfile(4); // dogAge == 4, dogName == 'Sprinkles'
```

! Default parameters have the same behaviour as optional parameters !

Rest parameters

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters which is denoted by ellipsis «...» .

We can pass zero or more arguments to the rest parameter.

```
function Greet(greeting: string, ...names: string[ ]) {
        return greeting + « « + names.join(«, «) + «!»;
};
Greet(«Hello», «Steve», «Bill»); // returns «Hello Steve, Bill!»
```

! Remember: Rest parameters must come last in the function defination, otherwise the TS compiler will show an error. !

Function types

Named function

A named function is one where you declare and call a function by its given name.

```
function multiply(a: number, b: number): number {
        return a * b;
};
```

Anonymous Function

An anonymous function is one which is defined as an expression. This expression is stored in a variable. So, the function itself does not have a name.

```
let multiply = function(a: number, b: number): number {
        return a * b;
};
```

Arrow function

Fat arrow notations are used for anonymous functions i.e for function expressions.

```
let multiply = (a: number, b: number): number => {
        return a * b;
};
```

Tip: if the function body consists of only one statement then no need for the curly brackets!

Interfaces

Interface is a structure that defines the contract in your application.
It defines the syntax for classes to follow.
Classes that are derived from an interface must follow the structure provided by their interface.
Interfaces have **zero** runtime JS impact. It uses interface for type checking.

```
interface Dogs {
        dogAge: number;
        dogName: string;
        getDogAge: (number) => number;
        getDogName(number): string;
        dogOwner?: string;
        readonly price: number;
}
```

- An interface is defined with the keyword interface
- In this example, the Dogs interface includes two properties dogAge and dogName.
- It also includes a method declaration getDogAge using an arrow function which includes one number parameter and a number return type.
- The getDogName method is declared using a normal function.
- **! Any object of type Dogs must define the three (or four) properties and two methods. !**
- We can have optional properties, marked with a «?».
  In such cases, objects of the interface may or may not define these properties.
- TS provides a way to mark a property as read only.
  This means that once a property is assigned a value, it cannot be changed!

Interface for Array Type

An interface can also define the type of an array where you can define the type of index as well as values.

```
interface DogList {
        [index: number]: number
}
```

Interface DogList defines a type of array with index as number and value as number type.

```
interface DogNameList {
        [index: string]: string
}
```

In the same way, DogNameList defines a string array with index as string and value as string.

Extending interfaces

Interfaces can extend one or more interfaces.
This makes writing interfaces flexible and reusable.

```
interface Dogs {
        name: string;
        owner: string;
}
interface DogData extends Dogs {
        dogData: number;
}
let dogExample: DogData = {
        dogData: 1,
        name "Sprinkles",
        owner: "Bob"
}
```

The DogData interface extends the Dogs interface.

So, objects of DogData must include all the properties and methods of the Dogs interface otherwise, the compiler will show an error.

The class support in TS is similar to that of languages like Java and C#, in that classes may inherit from other classes, while objects are instantiated as class instances. Also similar to those languages, TS classes may implement interfaces or make use of generics.

Simple class

We declare a simple class Plane.

```
class Plane {
        public altitude: number = 1500;
        private speed: number = 100;

        fly() {
        this.altitude += this.speed;
 }
}
```

The class has three members:

a **public** property altitude,

a **private** property speed and a

public method fly.

Each member is public by default.

```
let plane = new Plane( );
plane.fly( );
console.log(plane.altitude)
```

Creating a new instance of Plane.

Here we call the method.

Here we access a public property.

Basic inheritance

```
class Plane {
        public altitude: number = 1500;
        protected speed: number = 100;
        fly( ) {
         this.altitude += this.speed;
 }
}

class SelfFlyingPlane extends Plane {
        fly( ) {
        super.fly();
        super.fly();
 }
}
```

The SelfFlyingPlane class extends the Plane class using extends keyword.

Now SelfyFlyingPlane class includes all members of Plane class.

The SelfFlyingPlane class overrides the move( ) method and uses the base class implementation using super keyword.

Constructor

The constructor is a special type of method which is called when creating an object.
Constructor method is always defined with "constructor" name.

```
class Dog {
dogAge: number;
dogName: string;

        constructor(age: number, name: string ) {
        this.dogAge = age; this.dogName = name;
 }
}
```

The **Dog** class includes a constructor with the parameters **age** and **name**.
In the constructor, members of the class can be accessed using this keyword.
It is not necessary for a class to have a constructor.

The implementation of generics in TS give us the ability to pass in a range of types to a component, adding an extra layer of abstraction and re-usability to your code.

Generics can be applied to functions, interfaces and classes in TS.

The type variable **T** is specified with the function in the angle brackets **getArray<T>.**
The type variable **T** is also used to specify the type of the arguments and the return value.

```
function getArray<T>(items : T[] ) : T[] {
        return new Array<T>( ).concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400);                    This is OK!
myStrArr.push("Hello TypeScript");     This is OK!

myNumArr.push("Hi");                   Error
 myStrArr.push(500);                   Error
```

We call generic function **getArray()** and pass the numbers array and the strings array.

For example, calling the function as **getArray<number>** ([100, 200, 300]) will replace T with the number

So now, the compiler will show an error if you try to add a string in **myNumArr** or a number in **myStrArr** array.

Data modifiers

In object-oriented programming, the concept of 'Encapsulation' is used to make class members public or private (class can control the visibility of its data members).
This is done using access modifiers.
There are three types of access modifiers in TS.

Public

All the public members can be accessed anywhere without any restrictions.

```
class DogsList {
        public dogName: string;
        dogOwner: string;
}

let dogs = new DogsList( );
dogs.dogName = «Sprinkles»;
dogs.dogOwner = «Bob»;
```

dogName and **dogOwner** are declared as public.

So, they can be accessible outside of the class using an object of the class.

Remember: By default, all members of a class in TypeScript are public.

Private

The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class DogsList {
private dogName: string;
dogOwner: string;
}

let dogs = new DogsList( );
dogs.dogName = «Sprinkles»;     <--- Error!
dogs.dogOwner = «Bob»;          <--- Ok!
```

We have marked the member **dogName** private.

When we create an object dogs and try to access the dogs. dogName member, it will give an error.

Protected

The members protected access modifier is similar to the private access modifier, except that protected can be accessed using their deriving classes.

```
class DogList {
        public dogName: string;
        protected dogAge: number;

        constructor(name: string, age: number){
        this.dogName = name;
        this.dogAge = age;
 }
}

class Dog extends DogList{
        private dogOwner: string;

        constructor(name: string, age: number, dogOwner: string) {
        super(name, age);
        this.dogOwner = dogOwner;
}
}
```

We have a class DogList with two members, public dogName and protected property dogAge.

We create a subclass Dog that extends from the parent class DogList.

```
let dog = new Dog(«Sprinkles», 4, «Bob»);     OK!
dog.dogAge;                                   Error!
```

If we try to access the protected member from outside the class, as dog.dogAge, we get the following compilation error:

error TS2445: Property 'dogAge' is protected and only accessible within class 'DogList' and its subclasses.

What is a module?

The TS code we write is in the global scope by default.

If we have multiple files in a project, the variables, functions, etc. written in one file are accessible in all the other files.

We have to files: dog1.ts & dog2.ts

dog1.ts

```
var dog: string = "Sprinkles"
```

dog2.ts

```
console.log(dog); Prints Sprinkles!

dog = "Rex" This is allowed!
```

The variable **dog**, declared in **dog1.ts** is accessible in **dog2.ts** as well.

Not only it is accessible but also it is open to modifications.

Anybody can easily override variables declared in the global scope without even knowing they are doing so!

To prevent overriding variables in global scope, we use modules.

Module can be created using keyword export and can be used in another module using import.

dog1.ts

```
export var dog: string = "Sprinkles";
```

In dog1.ts, we used the keyword export before the variable.

dog2.ts

```
console.log(dog); Error: cannot find 'dog'

dog = "Rex"; Declared a new variable
```

Now, accessing a variable in dog2.ts will give an error, because dog is no longer in the global scope.

Import

A module can be used in another module using an import statement.

Syntax:

```
Import { export name } from "file path without extension"
```

Other examples:

Importing a Single export from a Module:

```
import { dog } from "./dog1";
```

Importing the Entire Module into a Variable:

```
import * as dog from "./dog1"
```

Renaming an Export from a Module:

```
import { dog as cat } from "./dog1"
```