TypeScript

TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions.

Types provide a way to describe the shape of an object, providing better documentation, and allowing TypeScript to validate that your code is working correctly.

TypeScript is typed super set of JavaScript. It supports " static type checking". . Determining what's an error and what's not based on the kinds of values being operated on is known as static *type* checking.

TypeScript extends JavaScript by adding types.

By understanding JavaScript, TypeScript saves you time catching errors and providing fixes before you run code.

Any browser, any OS, anywhere JavaScript runs. Entirely Open Source.

Installing TypeScript

We can install typescript via "npm"

```
npm install  -g typescript
```

To transpile TypeScript to JavaScript,

```
tsc   file-name.ts
```

Basic types

The following are data types supported by TypeScript.

- Boolean
- Number
- String
- Array
- Tuple
- Unknown
- Any
- Void
- Null and undefined
- Object

Declaration of variable

Syntax:

variable-name : datatype

Ex:

```
n:number;
name:string=”John”
student:object={name:”kiran”,age:20}
numbers:number[]=[10,20,30]
```

Declaration of function

syntax:

```
function function-name ( parameters ): return-type
    {
    }
```

Ex:

```
Function test(): number
    {
        return 100;
    }
```

Optional & default parameters of a function

Optional parameters can be represented by adding “?” to the its end.

Default parameters can be assigned with a value as default one

Ex:

```
function test (firstName?:string,lastName:string=”Ravi)
    {
    }
```

Interfaces

One of TypeScript’s core principles is that type checking focuses on the *shape* that values have. This is sometimes called “duck typing” or “structural subtyping”.

In TypeScript, interfaces fill the role of naming these types, and are a powerful way of defining contracts within your code as well as contracts with code outside of your project.

Creating a type using interface:

```
Interface interface-name{
        Property : datatype;
        Property:datatype;
        …………………
}
```

Ex:

```
Interface Student
        {
                name:string;
                age:number;
        }
```

It creates a type Student .

Using interface:

Let us write a function which can accept argument of type “Student”

```
function test( std:Student)
        {
        }
```

Then, while calling the function, we should pass argument of type “Student” like below

```
test({name:”xxxxx”,age:20})
```

Optional parameters in interface:

Interfaces do a great job in making sure the entities are implemented as expected. However, there would be cases when it is not necessary to have all of the properties as defined in the interface. These are called optional properties and are represented in the interface like this:

Syntax:

```
Interface Interface-name
        {
                Propert1:datatype;
                Property2 ?:datatype;
        }
```

Here, the “property2” is optional .

Read-only properties:

Read-only properties cannot be changed once they are initialized.

Syntax:

```
Interface Interface-name
        {
                readonly Propert1:datatype;
                Property2 ?:datatype;
        }
```

Class

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Syntax:

```
class Class-name
    {
        Instance variables
        Constructor
        Methods
    }
```

Ex:

```
class Greeter {
    greeting: string;

     constructor(message: string) {
                    this.greeting = message;
          }

     greet() {
          return "Hello, " + this.greeting;
          }
    }



let greeter = new Greeter("world");
```

Access modifiers

The following are three access modifiers are supported by TypeScript to decide accessible premises of class content.

public

--This default one. Public content is freely accessible

private

--Private content is accessible within the class

protected

--protected content is accessible with in same class &its child class

Creating instance variable using access modifiers

Access modifiers can be applied only to instance & static content of a class.Local variables and method parameters cannot be applied with access modifiers.

But, in TypeScript ,to simplify the process of creation and initialization of instance variables, “access modifiers” can be applied to parameters of constructor.

Ex:

```
class Test{
        constructor(public a:number,public b:number){}
}
```

The above class definition is equal to

```
class Test{
        a:number;
        b:number;
        constructor(a:number,b:number)
                {
                        this.a=a;
                        this.b=b;
                }   }
```