Typescript.md

Source: Codevolution

About Typescript

- Open source programming language from Microsoft
- Typed superset of Javascript
- Compiles down to Javascript

Advantages of Typescript

- Relation to Javascript (Extended Javascript)
- Optional static typing and type inference
- IDE support
- Rapid growth and use
- Specifying types is completely optional

Installation

```
npm install -g typescript
```

Version

```
tsc -v
```

Compile ts file

```
tsc main.ts
tsc main
```

Generates a file \main.js\

Run js file

```
node main.js
node main
```

Scopes

Scripts have global scope Module has its own scope

Treat a ts file as a module

At the top:

```
export {}
```

To automatically recompile

```
tsc main --watch
```

Variable Declarations

```
let x = 10;
const y = 20;
```

Cannot redeclare `let x = 30;`

let declaration => can be done without initialization

const declaration => cannot be done without initialization

Cannot declare 'const title;'

Never reassign => const else use let

Eg: sum of 2 numbers => let, title of application => const

Variable Types

Specifying types is optional

Basic types:

- boolean
- number
- string

Examples:

```
let isEmpty: boolean = true;
let total:number = 5;
let lang:string = "Typescript";

let sentence: string = `
        Hello ${name}
        ${lang}
`;

let n:null = null;
let u:undefined = undefined;

let isNew: boolean = null;
let myName: string = undefined;
```

Arrays

```
let list1: number[] = [1,2,3];
let list2: Array<number> = [1,2,3];
```

Tuples

```
let person: [string, number] = ["Vinit", 22];
```

Enum

```
enum Color {Red, Green, Blue};
let c:Color = Color.Green;
console.log(c);
```

Prints value 1 (index of Green)

```
enum Color {Red=5, Green, Blue};
let c:Color = Color.Green;
console.log(c);
```

Prints value 6 as index starts from 5

Any type

```
let randomValue:any = 10;
randomValue = true;
randomValue = "Vinit";
```

- Variable can take value of any type
- No intellisense support for any type
- Variable can call any property on it `randomValue.name`
- Variable can be called as a function `randomValue()`
- Functions can be called on variable `randomValue.toUpperCase()`

Unknown type

```
let randomValue:unknown = 10;
randomValue = true;
randomValue = "Vinit";
```

- Variable can take value of any type
- Variable cannot call any property on it `randomValue.name`
- Variable cannot be called as a function `randomValue()`
- Functions cannot be called on variable `randomValue.toUpperCase()`

Type assertion

Same as typecasting in other languages

```
(randomValue as string).toUpperCase();
```

User defined type guard

```
function hasName(obj:any): obj is {name: string}{
    return !!obj && typeof obj === "object"  && "name" in obj;
}
if(hasName(randomValue)){
    console.log(randomValue.name);
}
```

- Input: obj of type any
- Returns: obj having name as type string
- Description:
  - convert obj to boolean
  - check if obj is of type object
  - check if obj has `name` property

Type inference

Works only on variable initialization

```
let a;
a = 10;
a = true; //valid, no type inference


let b = 10;
b = true; // invalid, type inference performed
```

Union of types

```
let multiType: number | boolean;
multiType = 10;
multiType = true;
```

- Union type => restrict values to specified type, any type => no restriction
- Union type => intellisense support based on type, any type => no intellisense support

Functions

```
//optional parameters
function add(num1: number, num2?: number): number{
    if(num2)    return num1 + num2;
    else return num1;
}
add(2,3);
add(2);
```

- Takes 1 or 2 arguments of type number and return a number
- Here, num2 is optional parameter
- Optional parameters must always be after required parameters
- Optional parameters have value undefined if no value is passed

```
//default parameters
function sub(num1: number, num2: number = 20): number{
    if(num2)    return num1 - num2;
    else return num1;
}
sub(50,10);
sub(50);
```

- Takes 1 or 2 arguments of type number and return a number
- Here, num2 has default value of 20

Interface

```
interface Person {
    firstName: string;
    lastName?: string;
}

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

let p = {
    firstName: "Vinit"
}
```

```
fullName(p);
```

Class

Basic class in typescript:

```
class Employee {
    employeeName: string;

    constructor(name: string){
        this.employeeName = name;
    }

    greet(){
        console.log(`Hello ${this.employeeName}`);
    }
}

let emp1 = new Employee("Vinit");
console.log(emp1.employeeName);
emp1.greet()
```

Inheritance

```
class Manager extends Employee {

    constructor(managerName: string){
        super(managerName);
    }

    delegateWork(){
        console.log(`Manager delegating tasks`);
    }
}

let m1 = new Manager('Bruce');
m1.delegateWork();
m1.greet();
console.log(m1.employeeName);
```

Access Modifiers

By default, each class member is public

Public access modifier

Can be accessed anywhere, default

Private access modifier

Can be accessed only inside same class, cannot be accessed even in subclass

```
class Employee {
    private employeeName: string;

    constructor(name: string){
        this.employeeName = name;
    }

    greet(){
        console.log(`Hello ${this.employeeName}`);
    }
```

```
}

let emp1 = new Employee("Vinit");
// console.log(emp1.employeeName); causes error

class Manager extends Employee {

    constructor(managerName: string){
        super(managerName);
    }

    delegateWork(){
        //console.log(`Manager delegating tasks ${this.employeeName}`); // causes error
    }
}
```

Protected access modifier

Can be accessed in same class and subclass, not outside

```
class Employee {
    protected employeeName: string;

    constructor(name: string){
        this.employeeName = name; //accessible here
    }

    greet(){
        console.log(`Hello ${this.employeeName}`);
    }
}

let emp1 = new Employee("Vinit");
// console.log(emp1.employeeName); causes error

class Manager extends Employee {

    constructor(managerName: string){
        super(managerName);
    }

    delegateWork(){
        console.log(`Manager delegating tasks ${this.employeeName}`); //accessible here
    }
}
```