**Typescript Demonstrations**

**Sample Project Location**

C:\Users\robma\Userdata\Personal\Training\npm and typescript\sample typescript project

Much of the tutorial is based on the following

https://www.typescripttutorial.net/

**Setup project**

1. npm init (older versions of npm –save-dev for development dependencies)

2 – node -version– install typescript

will go into package.json

3 – configure the typescript config tsconfig.json –

npx tsc –init

Add typescript globally.

npm install -g typescript

Then compile a simple hello world

npm install -g typescript

**tsc vs ts-node vs node**

* tsc will generate the javascript - compiler.
* Will create a simple Java script file - like .js
* node xx.js will run the java script
* ts-node will execute the file – not a compiler.

**How to set up a typescript general project**

**Prerequisites**

Assume that node js and tsc are installed

*node --version*

*C:\Users\robma\Userdata\Personal\Training>npm list -g*

*C:\Users\robma\AppData\Roaming\npm*

*+-- @angular/cli@11.2.11*

*+-- newman@5.3.2*

*+-- ts-node@10.9.1*

*`--* [*typescript@5.2.2*](mailto:typescript@5.2.2)

**Step 1** Create a new folder – say node ts

**Step 2** Create a build and src folder

Directory of C:\Users\robma\Userdata\Personal\Training\npm and typescript\demo-project-structure\nodets

06/10/2023 11:07 AM <DIR> .

06/10/2023 11:07 AM <DIR> ..

06/10/2023 11:07 AM <DIR> build

06/10/2023 11:07 AM <DIR> src

**Step 3** run tsc –init

Inside that folder

… demo-project-structure\nodets

Creates the tscconfig.json

**Step 4** examine and updated the tsconfig.json and change the follwign lines -for the src and build files - uncomment and rember to add the , at the end

For the outDir option:

"outDir": "./build"Code language: JavaScript (javascript)

And for the rootDir option:

"rootDir": "./src"

**Step 5** verify setup

Create a basic file - say app.ts

console.log('Node.js TypeScript');

by running tsc in the nodets folder - show that a app.js is created

**How to configure nodemon**

This recognises that the when a project has ben updated and then updated the code appropriately. This is taken from here .. https://www.typescripttutorial.net/typescript-tutorial/nodejs-typescript/

The nodemon module allows you to automatically restart the application when you change the JavaScript source code.

The concurrently module runs multiple commands concurrently.

First, execute the npm init command from the root directory of the project:

npm init --yes

Next, install the nodemon and concurrently module:

npm install --g nodemon concurrently

It’ll take some time to install. Once the installation completes.

Note that the -g flag will instruct npm to install these two modules globally. This allows you to use them in other projects.

Then, open the package.json file, and you’ll something like this in the scripts option:

...

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

...Code language: JavaScript (javascript)

After that, change the scripts option to the following:

...

"scripts": {

"start:build": "tsc -w",

"start:run": "nodemon build/app.js",

"start": "concurrently npm:start:\*"

},

...Code language: JavaScript (javascript)

This "start:build": "tsc -w" will watch for changes in the ./src directory and compile them automatically.

This "start:run": "nodemon build/app.js" will automatically run the app.js in the ./build directory whenever the new file is generated.

This "start": "concurrently npm:start:\*" runs all the commands that start with npm:start:\*, which executes both start:build and start:run commands above.

Since the app.js will be the entry point for the Node.js program, you also need to change the following option in the package.json file to app.js:

From:

"main": "index.js"Code language: JavaScript (javascript)

To:

"main": "app.js"Code language: JavaScript (javascript)

Finally, execute the following command:

npm start

To verify the configuration, you change some code in the app.ts

**Step 6** run the app.js using node

Like

node app.js

**To set up the linter** –

e.g. ESLint

***Step 1 Install the packages***

npm install --save-dev @typescript-eslint/parser @typescript-eslint/eslint-plugin eslint typescript

***Step 2 Create the config file. -*** .eslintrc.cjs

/\* eslint-env node \*/

module.exports = {

extends: ['eslint:recommended', 'plugin:@typescript-eslint/recommended'],

parser: '@typescript-eslint/parser',

plugins: ['@typescript-eslint'],

root: true,

};

***Step 3 -in VSCODE***

Install eslint

npx eslint --init

In the project directory

**To run a typescript file**

Install ts-node //install globally to be used on other projects

npm install -g ts-node

npm list -g

+-- [ts-node@10.9.1](mailto:ts-node@10.9.1) // or other typescript files

`-- typescript@5.2.2

like

ts-node simpleTypeSafeDemoApp.ts

ts-node will create the .js and then execute the file.

**Typescript and project modules**

How to set up the various modules and the allow the import and export of those functions.

There are 3 files

* TypeSource.ts is the original source – it has a
  + class
  + an abstract class
  + some types – numberbools and alphanumeric
  + it exports these
* Types.ts then imports these – an creates 3 validators
* One happens =to suethe clas form tyepssource 9pruely for demo
* One is defined as the default export

**Overview of type script syntax**

**Embed into simple web application**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>TypeScript: Hello, World!</title>

</head>

<body>

    <script src="simpleApp.js"></script>

</body>

</html>

The run using liveserver.

Primitives are as listed here

* boolean - true or false values
* number - whole numbers and floating point values
* string - text values like "TypeScript Rocks"
* bigint - whole numbers and floating point values, but allows larger negative and positive numbers than the number type.
* symbol are used to create a globally unique identifier.

Plus object types

|  |  |
| --- | --- |
| null | has one value: null |
| undefined | has one value: undefined. It is a default value of an uninitialized variable |
| symbol | represents a unique constant value |

Unknown and any – Unknown is safer.

Object types =arrays, classes, functions

Type annotations are with a : type

Apply this to Arrays / objects / classes

let names: string[] = ['John', 'Jane', 'Peter', 'David', 'Mary'];

or

let person: {

name: string;

age: number

};

***Type inference*** - we can miss out the return type (the red bit)

function increment(counter: number) : number {

return counter++;

}

An array can have a union to types

Numbers

let price: number;

let counter: number = 0; // decimal number

let bin = 0b100; // binary number - with ‘b’

let octal: number = 0o10; // octal number ‘o’

let hexadecimal: number = 0XA; // hexadecimal ‘X’

let big: bigint = 9007199254740991n; // n at the end

***Strings***

Double or single quotes

Across multiple lines

Interpolation

let profile: string = `I'm ${firstName}. I'm a ${title}`;

***Boolean***

let completed: boolean = true;

with || or && or ! – standard syntax

***Object***

Define the properties

let employee: {

firstName: string;

lastName: string;

age: number;

jobTitle: string;

};

Then create the object

employee = {

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

};

Or in 1 go

let employee: {

firstName: string;

lastName: string;

age: number;

jobTitle: string;

} = {

firstName: 'John',

lastName: 'Doe',

age: 25,

jobTitle: 'Web Developer'

};

Object describes functionality of all objects

Empty {} - no property on its own.

let vacant: {};

***Arrays***

let arrayName: type[];

we can then add using the index or push and pop

let series = [1, 2, 3];

let doubleIt = series.map(e => e\* 2);

console.log(doubleIt);

We can mix them.

let scores : (string | number)[];

scores = ['Programming', 5, 'Software Design', 4];

***tuple***

A tuple is an array with a fixed number of elements whose types are known

* A tuple is an array with a fixed number of elements whose types are known.

let skill: [string, number];  
can have optional elements - with a ? question mark

let bgColor, headerColor: [number, number, number, number?];

***enum***

a limited set of choices

enum Month {

Jan,

Feb,

Mar,

Then used in a switch statement – like below

function getSeason(month: Month) {

  let season: string;

  switch (month) {

      case Month.Jun:

      case Month.Jul:

      case Month.Aug:

          season = 'winter';

          break;

Typescript – use of any

We can choose to turn off the type checking – here current location does not have a type associated with it – so it becomes any.

const json = `{"latitude": 10.11, "longitude":12.12}`;

// parse JSON to find location

const currentLocation = JSON.parse(json);

console.log(currentLocation);

***never case***

* Never for a function that throws an error
* The never type contains no value.
* The never type represents the return type of a function that always throws an error or a function that contains an indefinite loop.

**A union type with the |**

function add(a: number | string, b: number | string) {

if (typeof a === 'number' && typeof b === 'number') {

return a + b;

}

if (typeof a === 'string' && typeof b === 'string') {

return a.concat(b);

}

throw new Error('Parameters must be numbers or strings');

}

**Alias type** – e.g. for alphanumeric

type alphanumeric = string | number;

**String literals**

These can limit the types – then with a type alias we can have set of allowed values – bit like an enum

type Season: 'Spring' | 'Summer' | 'Autumn' | 'Winter';

let season: Season;

season = 'Spring'

season = 'Summer'

season = 'Autumn'

season = 'Winter'’

// compile error season = 'Holidays'

***Functions***

We can define a function type

let addNumbers: (x: number, y: number) => number;

and then use it

addNumbers = function(a: number, b: number): number {

return a + b;

}

***Optional parameters***

// or with a function type

let multiplyAgain: (x: number, y: number, z?: number) => number;

multiplyAgain = function (a: number, b: number, c?: number): number {

if (typeof c !== 'undefined') {

return a \* b \* c;

}

return a \* b;

}

***Default values***

An example of this is as below

function multiplyThirdTime(a: number, b: number, c: number = 2): number {

return a \* b \* c;

}

***Rest parameters*** – like varags - see the ‘…’ notation

function getMultiplicationProduct(greetingMessage: string, ...numbers: number[]): number {

let total = 1;

console.log(`Greeting is ${greetingMessage}.`)

if (numbers.length === 0) {

return 0;

}

numbers.forEach((num) => total \*= num);

return total;

}

***Function overloading***

Using the ‘type of’ we can vary the implementation

function addingThings(x: number, y: number): number;

function addingThings(x: string, y: string): string;

function addingThings(x: any, y: any): any {

if (typeof x === 'number' && typeof y === 'number') {

return x + y;

}

if (typeof x === 'string' && typeof y === 'string') {

return x + " --- " + y;

}

}

***Typescript and classes***

Adds type annotation to the properties and methods of a class - the bold bit below

class AUPerson {

taxFileNumber: **string**;

firstName: **string**;

lastName: **string;**

constructor(taxFileNumber: string, firstName: string, lastName: string) {

this.taxFileNumber = taxFileNumber;

this.firstName = firstName;

this.lastName = lastName;

}

getFullName(): **string** {

return `${this.firstName} ${this.lastName}`;

}

printTFN(): string {

return `Tax File Number ${this.taxFileNumber}`;

}

}

const auperson = new AUPerson('ABC\_123\_DEF','Roberto','Allan');

console.log(auperson.getFullName());

console.log(auperson.printTFN());

// we can add access modifiers –

* to the properties
* or in a constructor (in this case this declares the properties at the same time.

class AUPersonAccessModified {

**// no additional property definitions**

constructor(**protected ssn: string, private firstName: string, private lastName: string**) {

this.ssn = ssn;

this.firstName = firstName;

this.lastName = lastName;

}

getFullName(): string {

return `${this.firstName} ${this.lastName}`;

}

}

const auAMPerson = new AUPersonAccessModified('ABC\_123\_DEF','Roberto','Allan');

console.log(auAMPerson.getFullName());

We can add a read only (and then protect that with access modifiers such as private). We can then intialise this in the constructor. See the readonly below.

class AUPersonAccessModified {

**private readonly birthDate: Date;**

constructor(protected ssn: string, private firstName: string, private lastName: string**, birthDate: Date**) {

this.ssn = ssn;

**Inheritance**

Uses the extends syntax

Call super to get to super constructor

We can override methods – here with a call tt he super class method definition – to add more details.

return super.describe() + `I'm a ${this.jobTitle}.`;

class AUEmployee extends AUPerson {

    constructor(

        taxFileNumber: string,

        firstName: string,

        lastName: string,

        private jobTitle: string) {

        // call the constructor of the Person class:

        super(taxFileNumber, firstName, lastName);

Statics members

class AUPerson {

…

private static populationCount: number = 0;

constructor(taxFileNumber: string, firstName: string, lastName: string) {

.

AUPerson.populationCount++; // **Incrementing the numbers**

}

public static getPopulationcount() { **//static method**

return AUPerson.populationCount;

}

}

Abstract classes typically have 1 or more abstract methods

abstract class Employee {

…

abstract getSalary(): number

**2nd example**

**Note the getHomeLanguage()**

abstract class human {

    constructor(protected firstName: string, protected lastName: string) {

    }

    abstract getHomeLanguage(): string

    get capitalisedName(): string {

        return `${this.firstName.toUpperCase()} ${this.lastName.toUpperCase()}`;

    }

}

class AUPerson extends human{

**Interfaces**

Interface in typescript can be applied to

* Properties
* Functions
* Class

1 - it can be used to describe a functions and the arguments

2 - to force functions to have a signature

3 - to force behaviour on a class

1 - it can be used to describe a functions and the arguments

interface Person {

firstName: string;

lastName: string;

}

function getFullName(**person: Person**) {

return `${person.firstName} ${person.lastName}`;

}

let john = {

firstName: 'John',

lastName: 'Doe'

};

console.log(getFullName(john));

2 - to force functions to have a signature

let format: StringFormat;

3 to force behaviour on a class

class Person implements Json {

interfaces can be extended – for example we can have deliver interface

* with an address
* and a deliver later

Interfaces can extend multiple interfaces – any required method must have identical arguments and return type.

Interface can extend a class – but so the private and protected parameters are available, any class implementing that interface must extend from the same base class (direct or as a sub class) - to get the members.

**Types**

We can define a set of properties

interface HumanBeing {

firstname: string;

lastname: string;

dateOfBirth: date;

}

interface Employment {

id: number;

department: string;

}

interface Contact {

email: string;

phone: string;

}

type Employee = HumanBeing & Employment & Contact

type Customer = HumanBeing & Contact

**Type guards**

We can use type guards to determine how to handle logic – we might support passing multiple types into a method.

* typeof
* instanceof
* in - check the existence of property on an object

**Casting**

let a: typeA;

let b = a as typeB;

using the <> operator as well

Type assertions tell the compiler to treat the value as a type

Example below returns type as a union of string or number

function getNetPrice(price: number, discount: number, format: boolean): number | string {

let netPrice = price \* (1 - discount);

return format ? `$${netPrice}` : netPrice;

}

Using the as determines what it should be treated as

let netPrice = getNetPrice(100, 0.05, true) as string;

console.log(netPrice);

let netPrice = getNetPrice(100, 0.05, false) as number;

console.log(netPrice);

We can use the diamond operator as well <>.

Generics

We can create a type gernic method and then pass in type that we want get back.

function getRandomElement<T>(items: T[]): T {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

let numbers = [1, 5, 7, 4, 2, 9];

let randomEle = getRandomElement<number>(numbers);

console.log(randomEle);

It makes it typesafe that we get back what we expect – the following will fail to compile.

let returnElem: string;

returnElem = getRandomElement(numbers);

***Constraints - method 1*** We can implement constraints of generics – 1 using the extends

function getRandomPetFeedingCycle<T extends Animal, U extends FeedingTime>(items1stSet: T[], items2ndSet: U[]): string

the first element better be an animal and the better be a feeding time.The we can pass in 2 sets when we invoke the operation.

const randomStringAboutFoodsAndPets = getRandomPetFeedingCycle<Animal, FeedingTime>(setOfPets,setOfDinners);

***Constraints - method 2***

We can use key off to only get the available properties – this also allows us to get a property dynamically.

// Define a simple object type

type Purchaser = {

    name: string;

    age: number;

    company: string;

  };

  // Create a function that accepts a generic type T constrained to keys of Person

  function getProperty<T extends keyof Purchaser>(obj: Purchaser, key: T): Purchaser[T] {

    return obj[key];

  }

  // Create an instance of the Person object

  const purchaser: Purchaser = {

    name: "Tommy",

    age: 20,

    company: "BCC Procurement",

  };

  // Access properties using getProperty function

  const purchaserName: string = getProperty(purchaser, "name"); // Valid

  const purchaserAge: number = getProperty(purchaser, "age"); // Valid

  const purchaserCompany: string = getProperty(purchaser, "company"); // Valid

  //const purchaserDiscount: string = getProperty(purchaser, "discount"); // Valid

  console.log(purchaserName + " " + purchaserAge + " " + purchaserCompany)

We can have ***generic interfaces*** -

*Case 1 - for a type.*

interface Pair<K, V> {

key: K;

value: V;

}

let month: Pair<string, number> = {

key: 'Jan',

value: 1

};

*Case 2 for a set of methods – all use the same type.*

interface Collection<T> {

add(o: T): void;

remove(o: T): void;

}

*Case 3 define an index type*

interface Options<T> {

[name: string]: T

}

let inputOptions: Options<boolean> = {

'disabled': false,

'visible': true

};

We can have a ***generic class*** - not just for method. Say we have a collection and this can hold many types of object, but we instantiate with one type – for numbers XOR strings XOR people.

 class Stack<T> {

    private elements: T[] = [];

    constructor(private size: number) {

    }

    isEmpty(): boolean {

        return this.elements.length === 0;

    }

    isFull(): boolean {

        return this.elements.length === this.size;

    }

    push(element: T): void {

        if (this.elements.length === this.size) {

            throw new Error('The stack is overflow!');

        }

        this.elements.push(element);

    }

    pop(): T | undefined {

        if (this.elements.length == 0) {

            throw new Error('The stack is empty!');

        }

        return this.elements.pop();

    }

}

const numbersStack = new Stack<number>(5);

//numbersStack.push('rabbit')

numbersStack.push(1)

numbersStack.push(2)

numbersStack.push(3)

numbersStack.push(4)

numbersStack.push(5)

// numbersStack.push(3) - will throw an error

console.log("is the stack full? - should be " + numbersStack.isFull())

numbersStack.pop()

console.log("is the stack full? - should no longer be " + numbersStack.isFull())

while (!numbersStack.isEmpty()) {

    let n = numbersStack.pop();

    console.log(`Pop ${n} from the stack. - is it empty yet ` + numbersStack.isEmpty());

}

const stringStack = new Stack<string>(2);

stringStack.push("Hello")

stringStack.push("there")

while (!stringStack.isEmpty()) {

    let n = stringStack.pop();

    console.log(`Pop String ${n} from the stack. - is it empty yet ` + stringStack.isEmpty());

}