Introduction to Angular 2

Amin Meyghani

Contents

1 Installing Node

- Use nvm to install and manage Node on the machine. Copy the install script and run it:

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.30.1/install.sh | bash
```

- After installed, make sure that it is installed, by running:

```
nvm --help
```

- Then use `nvm` to install node version `0.12.9` by running:

```
nvm install 0.12.9
```

- Confirm that it is installed by running `node -v`.
- You can load any node version in the current shell with `nvm use 0.x.y` after of course installing it.
- You can make `0.12.9` the default version by making an alias for the default node:

```
nvm alias default 0.12.9
```

1.1 Permissions

- Never use `sudo` to install packages, never do `sudo npm install <package>`. If you get permission errors, you can own the folders by the current user. So for example, if you get an error like:

```
Error: EACCES, mkdir '/usr/local'
```

- you can own the folder with:

```
sudo chown -R `whoami` /usr/local
```

You can own folders until node doesn't complain.

1.2 Install a Package

- Install a package to verify that node is installed and everything is wired up correctly. We are going to use `live-server` through the course. So let's install that:

```
npm i -g live-server
```

- Then, you should be able to run `live-server` in any folder to server the content of that folder:

```
mdkir ~/Desktop/sample && cd $_
live-server .
```

2 Visual Studio Code

- Install Visual Studio Code from: https://code.visualstudio.com/
- You can open new projects by going to the `File > Open` tag, to etierh open a folder containing your project or a single file
- Some useful keyboard shortcuts are:
  - `command + b`: to close/open the file navigator
  - `command + shift + p`: to open the prompt
- To install extensions open the prompt with `command + shift + p` and type:
  - `> install extension`
- Open the shortcuts settings from `Preferences > Keyboard Shortcuts`, and then you can add your own shortcuts:

```
// Place your key bindings in this file to overwrite the defaults
[
  {
    "key": "cmd+t",
    "command": "workbench.action.quickOpen"
  },
  {
    "key": "shift+cmd+r",
    "command": "editor.action.format",
    "when": "editorTextFocus"
  }
]
```

3 TypeScript Crash-course

3.1 Installing TypeScript

You can install the TypeScript compiler with node:

```
npm i typescript -g
```

Then to verify that it is installed, run `tsc -v` to see the version of the compiler. You will get an output like this:

```
message TS6029: Version 1.7.5
```

In addition to the compiler, we also need to install the TypeScript Definition manager for DefinitelyTyped (tsd). You can install tsd with:

```
npm i tsd -g
```

Using TSD, you can search and install TypeScript definition files directly from the community driven DefinitelyTyped repository. To verify that tsd is installed, run tsd with the `version` flag:

```
tsd --version
```

You should get an output like this:

```
>> tsd 0.6.5
```

After `tsd` and `tsc` are installed, we can compile a hello world program:

- make a file called `hello.ts` on your desktop:

```
touch ~/Desktop/hello.ts
```

- Then, put some TypeScript code in the file:

```
echo "const adder = (a: number, b: number): number => a + b;" > ~/Desktop/hello.ts
```

- Then you can compile the file to JavaScript:

```
tsc ~/Desktop/hello.ts
```

- It should output a file in `Desktop/hello.js`:

```
var adder = function (a, b) { return a + b; };
```

Now that your TypeScript compiler setup, we can move on to configuring Visual Studio Code.

3.2 Setting up TypeScript for VSCode

You can set up Visual Studio Code to compile your TypeScript code as your work.

- First, open Visual Studio Code
- Make a new window: `File > New Window`
- Then, make a folder on your desktop for a new project: `mkdir ~/Desktop/vscode-demo`
- The, open the folder in VSCode: `File > open` and select the `vscode-demo` folder on your desktop.
- Now we need to make three configuration files:
    1. `tsconfig.json`: configuration for the TypeScript compiler

2. `tasks.json`: Task configuration for VSCode to watch and compile files
3. `launch.json`: Configuration for the debugger

- The `tsconfig.json` file should be in the root of the project. Let's make the file and put the following in it:

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true,
    "outDir": "output",
    "watch": true
  }
}
```

- Now to make the `tasks.json` file, open the prompt with `command + shift + p` and type:

```
> configure task runner
```

- Then put the following in the file and save the file:

```
{
  "version": "0.1.0",
  "command": "tsc",
  "showOutput": "silent",
  "isShellCommand": true,
  "problemMatcher": "$tsc"
}
```

- The last thing that we need to set up is the debugger, i.e.`launch.json` file. Right click on the `.vscode` folder in the file navigator and make a new file called `launch.json` and put in the following:

```
{
  "version": "0.1.0",
  "configurations": [
    {
      "name": "TS Debugger",
      "type": "node",
      "program": "main.ts",
      "stopOnEntry": false,
      "sourceMaps": true,
      "outDir": "output"
    }
  ]
}
```

- After you save the file, you should be able to see the debugger in the debugger dropdown options.
- Now, we are ready to make the `main.ts` file in the root of the project:

  main.ts

```
console.log('hello');
```

- Now you can start the task to watch the files and compile as you work. Open the prompt with `command + shift + p` and type:

  ```
  > run build tasks
  ```

  you can also use the `command + shift + b` keyboard shortcut instead. This will start the debugger and watch the files. After making a change to `main.ts`, you should be able to see the output in the `output` folder.
- Now that the build task is running, we can put a breakpoint anywhere in our typescript code. Lets add some more code to the main file and use the debugger:

```
let a = 2;
let b = 3;
let c = 4;
```

- Then click on the margin of line two for example to add a breakpoint. Then open the debugger tab to run the debugger and you should see that the program will stop at the breakpoint and you can step over or into the line.
- To stop the task you can terminate it. Open the prompt and type:

  ```
  > terminate running task
  ```

3.3 Types and the Basics

There are 7 types in TypeScript:

- boolean: `var isDone: boolean = false;`
- number: `var height: number = 6;`
- string: `var name: string = "bob";`
- array: `var list:number[] = [1, 2, 3];` also `var list:Array<number> = [1, 2, 3];`
- enum: `enum Color {Red, Green, Blue};`
- any: `var notSure: any = 4;`
- void: `function hello(): void { console.log('hello'); }`

3.4 Interface

- An Interface is defined using the `interface` keyword
- Interfaces are used only during compilation time to check types
- By convention, interface definitions start with an `I`, e.g. : `IPoint`
- Interfaces are used in classical object oriented programming as a design tool
- Interfaces don't contain implementations
- They provide definitions only
- When an object implements an interface, it must adhere to the contract defined by the interface
- An interface defines what properties and methods an object must implement
- If an object implements an interface, it must adhere to the contract. If it doesn't the compiler will let us know.
- Interfaces also define custom types

3.4.1 Basic Interface

Below is an example of an Interface that defines two properties and three methods that implementers should provide implementations for:

```
interface IMyInterface {
  // some properties
  id: number;
  name: string;

  // some methods
  method(): void;
  methodWithReturnVal():number;
  sum(nums: number[]):number;
}
```

Using the interface above we can create an object that adheres to the interface:

```
let myObj: IMyInterface = {
  id: 2,
  name: 'some name',

  method() { console.log('hello'); },
  methodWithReturnVal () { return 2; },
  sum(numbers) {
    return numbers.reduce( (a,b) => { return a + b } );
  }
};
```

Notice that we had to provide values to **all** the properties defined by the Interface, and the implementations for **all** the methods defined by the Interface.

And then of course you can use your object methods to perform operations:

```
let sum = myObj.sum([1,2,3,4,5]); // -> 15
```

3.5 Classes

- Classes are heavily used in classical object oriented programming
- It defines what an object is and what it can do
- A class is defined using the `class` keyword followed by a name
- By convention, the name of the class start with an uppercase letter
- A class can be used to create multiple objects (instances) of the same class
- An object is created from a class using the `new` keyword
- A class can have a `constructor` which is called when an object is made from the class
- Properties of a class are called instance variables and its functions are called the class methods
- Access modifiers can be used to make them public or private
- The instance variables are attached to the instance itself but not the prototype
- Methods however are attached to the prototype object as opposed to the instance itself
- Classes can inherit functionality from other classes, but you should favor composition over inheritance or make sure you know when to use it
- Classes can implement interfaces

Let's make a class definition for a car and incrementally add more things to it.

3.5.1 Adding an Instance Variable

The `Car` class definition can be very simple and can define only a single instance variable that all cars can have:

```
class Car {
  distance: number;
}
```

- `Car` is the name of the class, which also defines the custom type `Car`
- `distance` is a property that tracks the distance that car has traveled
- Distance is of type `number` and only accepts `number` type.

Now that we have the definition for a car, we can create a car from the definition:

```
let myCar:Car = new Car();
myCar.distance = 0;
```

- `myCar:Car` means that `myCar` is of type `Car`
- `new Car()` creates an instance from the `Car` definition.
- `myCar.distance = 0` sets the initial value of the `distance` to 0 for the newly created `car`

3.5.2 Adding a Method

So far our car doesn't have any definitions for any actions. Let's define a `move` method that all the cars can have:

```
class Car {
  distance: number;
  move():void {
    this.distance += 1;
  };
}
```

- `move():void` means that `move` is a method that does not return any value, hence `void`.
- The body of the method is defined in `{ }`
- `this` refers to the instance, therefore `this.distance` points to the `distance` property defined on the car instance.
- Now you can call the `move` method on the car instance to increment the `distance` value by 1:

```
myCar.move();
console.log(myCar.distance) // -> 1
```

3.5.3 Adding a constructor

A `constructor` is a special method that gets called when an instance is created from a class. Let's add a constructor to the `Car` class that initializes the `distance` value to 0. This means that all the cars that are crated from this class, will have their `distance` set to 0 automatically:

```
class Car {
  distance: number;
  constructor () {
    this.distance = 0;
  };
  move():void {
    this.distance += 1;
  };
}
```

- `constructor()` is called automatically when a new car is created
- The body of the constructor is defined in the `{ }`

So now when we create a car, the `distance` property is automatically set to 0.

3.5.4 Using Access Modifiers

If you wanted to tell the compiler that the `distance` variable is private and can only be used by the object itself, you can use the `private` modifier before the name of the property:

```
class Car {
  private distance: number;
  constructor () {
    ...
  };
  ...
}
```

Access modifiers can be used in different places. Check out the access modifiers chapter for more details.

3.5.5 Implementing an Interface

Classes can implement one or multiple interfaces. We can make the `Car` class implement two interfaces:

interfaces

```
interface ICarProps {
  distance: number;
}
interface ICarMethods {
  move():void;
}
```

Making the `Car` class implement the interfaces:

```
class Car implements ICarProps, ICarMethods {
  distance: number;
  constructor () {
    this.distance = 5;
  };
  move():void {
    this.distance += 1;
  };
}
```

The above example is silly, but it shows the point that a class can implement one or more interfaces. Now if the class does not provide implementations for any of the interfaces, the compiler will complain. For example, if we leave out the `distance` instance variable, the compiler will print out the following error:

> error TS2420: Class 'Car' incorrectly implements interface 'ICarProps'. Property 'distance' is missing in type 'Car'.