


TypeScript Quick Labs

Quick Labs Environment Set-Up	2
Code Editing	2
NodeJS	2
Quick Labs Alternative Logging Out of Results.....	2
Quick Lab 1a - "Hello World" TypeScript.....	3
Quick Lab 1b - TypeScript Dev Environment	4
Quick Lab 2a - TypeScript Tuples	6
Quick Lab 2b - Type Assertion and Unknown	7
Quick Lab 3 - TypeScript Classes	8
Quick Lab 4 - TypeScript Interfaces	9
Quick Lab 5 - TypeScript Generics	10
Quick Lab 6 - Modules.....	12
Quick Lab 7 - Decorators.....	13

Quick Labs Environment Set-Up

Code Editing

1. Open VSCode (or download and install if not present).
2. Use the desktop shortcut to open the VSCode download page:
3. For Windows users download the 64-bit System Installer.
4. Check for updates and download and install if necessary:
5. For Windows Users click Help - Check for updates;
6. For MacOS Users click Code - Check for updates.
7. Using File - Open, navigate to the QuickLabs folder and click Open. This will give you access to all of the QuickLab files and solutions needed to complete the QuickLabs.
8. Using **CTRL + '** on the keyboard (**CTRL + `** on *MacOS*) or by using click-path **View → Terminal**, open **VSCode's** *integrated terminal* or click the terminal icon  on the bottom bar

NodeJS

1. Use the desktop shortcut to open the NodeJS download page.
2. Download and install the LTS version for the operating system you are working in:
3. For Windows users, download the Installer file (.msi);
4. For MacOS users, download the Installer file (.pkg).

Quick Labs Alternative Logging Out of Results

Webpack has been included in the set-up of the Quick Labs to demonstrate a realistic development environment, so the bundled JavaScript output can be viewed using the browser console.

If you want to manually compile the files you create you can use the following pattern to see the output:

1. Point the terminal at the TypeScript file location.
2. Run the TypeScript compiler on the file using:

```
tsc <filename>.ts
```

3. From the same location, use Node to run the compiled JavaScript file using:

```
node <filename>.js
```

The output of any `console.log`s will appear in the terminal, as will any error messages.

Quick Lab 1a - "Hello World" TypeScript

Objectives

- To be able to write and compile a simple TypeScript file and run the outputted JavaScript.

Activity

1. On the command line, using the **cd** command, navigate to the **QuickLabs/01a_HelloWorld/starter** folder.
2. Install **TypeScript**, globally on your machine using the command:

```
npm i -g typescript
```

3. In the starter folder, create a new file called **helloWorld.ts**.
4. Add the following lines of code into it:

```
let world = `world`;  
console.log(`Hello ${world}`);
```

5. Save the file.
6. Back on the command line, compile the TypeScript file to JavaScript using:

```
tsc helloWorld.ts
```

Notice that, if the compiler is able to run error free, it appears nothing has happened.

7. Check the starter folder, you should see a new **file helloWorld.js** - refresh the view if not.
8. Examine the contents of the file **helloWorld.js**.

Notice how all of the *ES2015+* syntax has been replaced by older syntax? This is because part of the compilation process deals with the transpilation to ES3!

9. Run the file by typing the following command into the command line:

```
node helloWorld.js
```

10. Run the same command but use **helloWorld.ts**.

The file should run the same - this is because Node has found only JavaScript in the file.

11. Change the declaration of world to the following:

```
let world: string = `world`;
```

12. Run the compiler again and check the outputted JavaScript file - you should see that the type declaration has been removed.
13. Run both files with node again - the TypeScript file should now fail as it is not pure JavaScript.

This is the end of Quick Lab 1a

Quick Lab 1b - TypeScript Dev Environment

Objectives

- To be able to create a project environment for TypeScript development.

Activity

1. On the command line, using the **cd** command, navigate to the **QuickLabs/01b_DeveloperEnvironment/starter** folder.
2. Initialise an npm project, accepting all defaults by using the command:

```
npm init -y
```

3. Install typescript and the webpack plugin ts-loader for the project using:

```
npm i --save-dev typescript ts-loader
```

4. Install Webpack, its CLI and the development server using:

```
npm i --save-dev webpack webpack-cli webpack-dev-server
```

5. Amend the **package.json** file scripts section so that the following replace the **"test"** script:

```
"build": "webpack --mode production",  
"dev": "webpack --mode development",  
"start": "webpack serve --open \"Chrome\"",  
"check-types": "tsc"
```

Note: On MacOS the start command needs to be:

```
"start": "webpack serve --open \"Google Chrome\"",
```

6. Configure webpack by creating the file webpack.config.js and putting the following code inside it:

```
module.exports = {  
  entry: './src/index.ts',  
  resolve: {  
    extensions: ['.ts', '.js'],  
  },  
  module: {  
    rules: [  
      {  
        test: /\.ts$/,  
        use: {  
          loader: "ts-loader"  
        }  
      }  
    ]  
  }  
}
```

```
    ]  
  }  
};
```

The entry part of this file tells Webpack how to get into our application. The resolve object tells Webpack to use both .ts and .js files imported. The module object tells Webpack to use the ts-loader when bundling .ts files.

When the build or dev commands are used, Webpack will create a bundled JS file called main.js (minified for build because of --mode production) and place it in a folder called dist. When using the development server, a virtual file is created and held on it. The check-types script will simply run the compiler and highlight any TypeScript errors.

7. Next, configure TypeScript by creating a **tsconfig.json** file and putting the following inside it:

```
{  
  "compilerOptions": {  
    "sourceMap": true,  
    "target": "es5"  
  }  
}
```

This config enables TypeScript and Webpack to allow you to debug the TypeScript files in the browser (use the Sources tab and then **webpack://** → . → **src** and then debug from the **.ts** file). It also means that the outputted JavaScript is only ES5 compliant. If you need to support ES3 browsers, change this and look into Polyfills!!!

8. Create the entry file, **index.ts** in a new folder called **src**
9. Re-write or copy the contents from QuickLab 1's TypeScript file in **index.ts** and save.
10. Create a new file in the **starter** folder called **index.html**.
11. Add a skeleton HTML page that as a script tag with a source of **main.js**.

This file **src** will need to be modified when going into a production environment. Presently, it enables the development server to run the bundled file.

12. On the command line, run the project using:

```
npm start
```

Your browser should spin up now - check the console and also find the file to debug.

13. Experiment with the build and dev commands (insert **run** after the npm command if it doesn't work out of the box).
14. Run the check-types script and see the results - i.e. the new files in the src folder!

This is the end of Quick Lab 1b

Quick Lab 2a - TypeScript Tuples

Objectives

- To experiment with TypeScript's Tuples.

Activity

1. On the command line, navigate to **the QuickLabs/02_TypeScriptTypes/starter** folder and initiate the **npm install**.
2. In **VSCode**, open the file **index.ts** from the **QuickLabs/02_TypeScriptTuples/starter/src** folder.
3. Define a *tuple* called **person** that has 3 types allowed: **string**, **number** and **boolean**.
4. Try to define the person with values in the wrong order - note the errors that are given
5. Log out the array and verify that the valid JS you have written is executed.
6. Define a **person** correctly and check the logging.
7. Try to add another element using different types to the array and note the errors.
8. Check the output of logging.

This is the end of Quick Lab 2a

Quick Lab 2b - Type Assertion and Unknown

Objectives

- To use the unknown and type assertion when working with variables.

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/02_TypeScriptTypes/starter/src** folder.
2. Make sure that the development server is still running.
3. It may help you to comment out Part 2a.
4. Define a variable called **something** with a type of **unknown** and initially set it to the **string 1234**.
5. Log out the result of the Boolean expression `something == 1234`.
6. Log out the result of the Boolean expression `something === 1234`.
7. Log out the result of the Boolean expression `something !== 1234`.
8. Log out the result of the Boolean expression `something >= 1000`.
9. Log out the result of asking for the `length` of `something`.
Note the error here - check the output of the compiled JavaScript!
10. Add the **as number** type assertion to all of the expressions above.

What do you notice about the output of `something as number === 1234`? Can you explain this?

This is the end of Quick Lab 2b

Quick Lab 3 - TypeScript Classes

Objectives

- To experiment with classes and access modifiers

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/03_TypeScriptClasses/starter/src** folder.
2. Run **npm install** on the command line and start the application running.
3. Declare a **class** called **Vehicle** and set it to have **private** properties of **make** and **model**, set as **strings** and a **number private** property of **speed** set initially to **0**.
4. Provide **get** methods for **make**, **model** and **speed**.
5. Provide a **set** method for **speed** that:
 - Takes a parameter **delta** of type **number** to represent the change in speed;
 - Checks to see if the new speed (by applying the change speed to the current speed) is *greater than 0* - if it is set the new speed to the *calculated value*, otherwise set speed as **0**.
6. Make an *instance* of a **Vehicle** and ensure that the *methods* and *modifiers* work as expected.
7. **Extend** the **Vehicle** with a class called **RoadVehicle** that has its own **private** property of **wheels** (it should be a **number**).
8. Provide a **getter** for **wheels**.
9. Create an *instance* of a **RoadVehicle** and check that all of its properties can be accessed in the expected way.
10. Make the **Vehicle** class and the **get** and **set** for **speed** **abstract**.

Notice that you are no longer allowed to make an instance of the **Vehicle** class.

Notice that getters and setters can be **abstract** - but only as a pair.

Notice that the implementation of the **RoadVehicle** class is now incorrect.

11. Fix the **RoadVehicle** class by providing the concrete implementations of the **get** and **set** methods for **speed**. You may notice that **_speed** is *not accessible* - *without* making it **public**, what should you do?
12. Check that your new implementation works.

This is the end of Quick Lab 3

Quick Lab 4 - TypeScript Interfaces

Objectives

- To use an Interface with multiple classes

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/04_TypeScriptInterfaces/starter/src** folder.
2. Under the provided code, create an **interface** called **HasPassengers**, it should specify:
 - a) A **readonly** property of **passengerSeats**;
 - b) A method called **makeStop** that takes *2 numeric arguments* of **numberOn** and **numberOff** and specifies that it *does not return anything*.
3. Create a **class** called **SingleDeckerBus** that **extends** the **RoadVehicle** class and **implements** the **HasPassengers** interface:
 - a) The constructor should accept *all parameters* needed for the **RoadVehicle** class and the **HasPassengers** interface, along with a **private** property **passengersOnBoard**, initially set to **0**.
 - b) In the class, implement a **getter** for **passengersOnBoard** and the required **makeStop** method.
 - The **makeStop** method can be as simple as you like - we implemented it so that the bus never has a negative number of passengers or more passengers than there are seats!
4. Create an *instance* of **SingleDeckerBus** and check that the methods work and that the properties are as expected.

If you feel the need...

5. Create a **class** called **Train** that **extends** **Vehicle** and **implements** **HasPassengers**, adding any properties or methods that are needed to make the class function.

This is the end of Quick Lab 4

Quick Lab 5 - TypeScript Generics

Objectives

- To experiment with Generic Classes, functions and constraints

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/05_TypeScriptGenerics/starter/src** folder.
2. Examine the supplied code - you should see a Circle class, an abstract Vehicle class, a RoadVehicle interface, a Car class, a Bus class and a Plane class.
3. Under the Plane class, create a Generic Class called Garage:

```
class Garage<T> {}
```

4. Give the **class** a property called **garage** that is an *empty array* of type **T**.
5. Write a **park** method that takes **aThing** of type **T** as an *argument* and *pushes aThing* to the **garage** array.
6. Try to make the method log out the **speed** of the *first item* in the **garage**:

```
console.log(this.garage[0].speed);
```

Why is this?

7. Create another Generic Class called **VehicleGarage** that has a type of **T** that **extends Vehicle**.
8. Give the **class** a property called **vehicleGarage** that is an *empty array* of type **T** and a method called **park** that takes a **vehicle** of type **T** and *pushes* the **vehicle** to the **vehicleGarage**.
9. Try to make the method log out the **speed** of the first item in the **vehicleGarage** and then its **taxed** property.
10. Create a third Generic Class called **RoadVehicleGarage** that has a type of **T** that extends **RoadVehicle**.
11. Give the class a property called **roadVehicleGarage** that is an *empty array* of type **T** and a **park** method that *pushes* the **roadVehicle** supplied to the **roadVehicleGarage** and then logs out the **speed** and **taxed** property value of the *first item* in the array.
12. Make *instances* of a Circle, Car, Bus, Plane, Garage, VehicleGarage and RoadVehicleGarage.
13. Try to **'park'** each of the objects you have made in each of the garages.

What do you notice?

14. Create a **function** called **logVehicleGarage** - it should be Generic, accepting **T** that **extends Vehicle** and an argument of **anyVehicleGarage** that is an *array* of type **T**, explicitly *returning nothing*.

```
function logVehicleGarage<T extends Vehicle>(  
  anyVehicleGarage: T[]): void {}
```

15. Make the body of the function simply log out the garage property of passed arrays.
16. Call the function with the three garages you created earlier.

What do you notice?

This is the end of Quick Lab 5

Quick Lab 6 - Modules

There is no QuickLab for this module.

This is the end of Quick Lab 6

Quick Lab 7 - Decorators

Objectives

- To experiment with a simple set of decorators

Activity

1. In **VSCode**, open the file **index.ts** from the **QuickLabs/07_Decorators/starter/src** folder.
2. Write a **function** called **simpleDecorator** that takes an argument **target** of type **any**.
3. Make the body of the function *log out a simple message*.
4. Create a **class** called **DecoratedClass** and decorate it with the **simpleDecorator**.

```
@simpleDecorator
class DecoratedClass {}
```

5. Save the file and view the result on the console.

The compilation may fail at this point as decorators are still an experimental feature. The message suggests to add experimental decorators option to the **tsconfig.json** file. Here's how:

6. Open **tsconfig.json** for the project.
7. At the end of the line for **target**, add a **comma**.
8. Add a new key of **experimentalDecorators** with a value of **true**.
9. Save the file and recompile - you may need to restart the process.

This should clear the error.

10. Write a **function** called **factoryDecorator** that takes **name** of type **string** as an argument.
11. Make the body of the function **return** a **function** that takes **target** of type **Function** as an argument and *logs out name* and ' **decorator was called**'.
12. Create a class called **FactoryDecoratedClass**, decorating it with the **decoratorFactory**, passing in the string **"factory"**.
13. Save the file and view the output on the console.
14. Create a **function** called **merge** which takes an argument **toMerge** of type **Object**.
15. Make the body of the function **return** a **function** that:
 - a) Takes an argument **target** of **type** any
 - b) Uses a *for in loop* to examine each **prop** in **toMerge** setting **target.prototype[prop]** to **toMerge[prop]**.
16. Declare a variable called **user** with **name** (string), **age** (number) and **instructor** (boolean) *key/value pairs*.
17. Create a **class**, *decorated* with **merge** and **user** as a parameter, called **AnotherDecoratedClass**.
18. *Instantiate* the class calling it **thing**.
19. *Log out* the **name** property on **thing**, asserting that **thing** is of type **any**.
20. Save the file and examine the output.
21. Write a **function** called **readOnly** that has arguments of:
 - a) **target** of type any;

- b) **methodName** of type **string**;
 - c) **descriptor**, *optional* of type **PropertyDescriptor**
22. Make the body of the function set the **writable** and **enumerable** properties of **descriptor** to be **false**.
 23. Create a **class** call **HoldingClass**.
 24. In **HoldingClass** add a function called **sayHello** that is decorated with **readOnly** and *logs out* the message '**Hello**'.
 25. *Instantiate* **HoldingClass** calling it **newThing**.
 26. Call **sayHello** on **newThing**.
 27. Try to change **newThing.sayHello** to a value of **false**.
 28. Save the file and observe the output.

This is the end of Quick Lab 7 and of all QuickLabs for
TypeScript
