TypeScript
Lab 1

1 Lab setup

Make sure you have the following items installed

- Latest version of Node and NPM (note: labs are tested with Node v16.13 and NPM v8.1 but should work with later versions with no or minimal changes)
- Latest version of TypeScript (note: labs are tested with TypeScript v4.5 but should work with later versions with no or minimal changes)
- Visual Studio Code (or a suitable alternative IDE for TypeScript)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

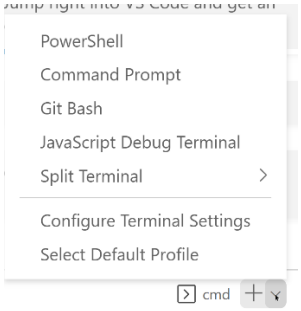
2 Initializing and compiling a TypeScript project

Create an empty folder on a suitable location on your local drive to use as the root folder for a project.

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`simple.ts`

Open a terminal shell (or command prompt in Windows) and navigate to this folder.
You can open a standard Windows command prompt (type `cmd` in Search) or else use the embedded terminal in Visual Studio Code. To access this, select Terminal -> New Terminal from the main menu or use the context menu in the lower right hand corner



At the terminal, type this to transpile (compile) the TypeScript file to its JavaScript equivalent:

```
tsc simple.ts
```

Successful compilation will produce a JavaScript file with the same name (`simple.js`). The actual version of JavaScript (ES 4, 5 or higher) produced from this compilation can be configured.

Open `simple.js` in VS Code. Notice that the syntax is slightly different from TypeScript

Note: VS Code will flag an error in the `ts` file if you have both the `ts` and `js` files open simultaneously, as they both have identical variable and function definitions and this is not allowed in the current scope. You can ignore this for now: closing the `js` file will remove the error.

We can run the transpiled `js` file at the command prompt using the Node runtime with:

```
node simple.js
```

Verify that the output is as expected.

We will need to repeat the process of transpiling everytime we make some changes to the `ts` file. To speed things up, we can configure the TypeScript compiler (`tsc`) to watch for changes to the `ts` file and automatically transpile it to JavaScript.

In the terminal window, type:

```
tsc --watch simple.ts
```

You should see output similar to the one below indicating that `tsc` is now actively monitoring the specified file for changes:

```
[7:34:08 pm] Starting compilation in watch mode...
[7:34:09 pm] Found 0 errors. Watching for file changes.
```

Make some changes to `simple.ts`. For e.g. change the two numbers being added. Verify that `tsc` notices this change and makes an incremental compilation, resulting in a new `js` file.

You can now open a new terminal window (either a normal command prompt or a terminal within VS Code) to run the `js` file in the usual manner.

This approach only works for a single file. If you have more than one file that you wish to have tsc monitor and compile incrementally, you will have to configure the current folder as a TypeScript project

Stop the incremental watch mode of `tsc` by typing Ctrl-C in the window that it is running in.

Type:

```
tsc --init
```

This generates a new TypeScript project configuration file: `tsconfig.json` in the current folder.

tsc uses tsconfig.json to provide configuration options for its compilation process whenever you type tsc in the terminal shell without explicitly specifying a file (or files) to compile. The presence of this file in a directory indicates that this directory is the root folder of a TypeScript project.

Some of the more popular compiler options that can be configured are explained below:

https://howtodoinjava.com/typescript/tsconfig-json/
https://www.typescriptlang.org/docs/handbook/tsconfig-json.html
https://www.kunal-chowdhury.com/2018/05/typescript-tutorial-tsconfig-json.html

All of the compiler options available in `compilerOptions` in `tsconfig.json` are also available at the command line via `tsc` (check with `tsc --help`).

The complete list of options and their uses is documented at:
https://www.typescriptlang.org/tsconfig

You can now type this to monitor and autocompile all `ts` files in the project root folder:

```
tsc --watch
```

Note that this command above requires a `tsconfig.json` to work; without it, you will need to explicitly specify the name of the single file that you want to monitor and autocompile.

References:

https://dev.to/arikaturika/how-to-install-and-run-typescript-on-windows-beginner-s-guide-3dpe

3 Basic primitive types

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

```
DemoBasicTypes.ts
```

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#the-primitives-string-number-and-boolean

In TypeScript, we will typically declare variable using the `let` keyword (ES6 and onwards), but sometimes you may see the `var` keyword used in older code samples and tutorials on the Internet:

https://www.javascripttutorial.net/es6/javascript-let/
https://www.javascripttutorial.net/es6/difference-between-var-and-let/

It is recommended to use the primitive type equivalents to JavaScript (`boolean`, `number` and `string`) rather than the `String`, `Boolean` and `Number` standard built-in object types because while JavaScript coerces an object to its primitive type, the TypeScript type system does not. TypeScript treats it like an object type. This can result in unexpected results when performing equality comparisons.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

However, just as in the case with JavaScript, you can access prototype methods on the Number object via autoboxing of primitive types. This means if you attempt to use the dot operator to access a method or property on a `string` or `number` primitive, JavaScript will automatically wrap (or box) it within a `String` or `Number` object.

When variables are declared without annotation, TypeScript will infer its type.

You can use the `const` keyword to declare a constant whose value cannot be changed after its initialization.

4 TypeScript specific types

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoOtherTypes.ts`

TypeScript provides static compile time type-checking. Occasionally, we may need to opt out of this behavior and allow the dynamic typing behavior of JavaScript instead. This may be the case when we are dealing with user input or values from third party libraries of unknown type or when working with older JavaScript code bases. In such cases, we need a provision that can deal with dynamic content. The Any type comes in handy here.

You can use the `typeof` operator to determine the type of any variable

The union type allows you to combine multiple types into one type.

The Array type has access to all the standard JavaScript array methods and properties
https://www.w3schools.com/js/js_array_methods.asp

An enum type is a group of named constant values which are matched internally to numbers in JavaScript.

If you want exact representation of the constant value type (instead of internal matching to numbers as in enum), use union and string literals.

We can use type literals to specify the shape of an object that can be assigned to a variable.

Type aliases are used to make union types and type literals easier for reuse.

5 Functions

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoFunctions.ts`

Functions must have their parameters types annotated. It is optional to annotate return types; if they are not, Typescript will infer them.

Parameters can be made optional via the ? symbol. Functions with optional parameters typically require a type guard to determine the specific operation to be performed. The optional parameters must appear after the required parameters in the parameter list.

An alternative to optional parameters is to provide default parameters

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters via the ellipsis ....
We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us.

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoArrowFunctions.ts`

Arrow functions are a new ES6 feature and are typically used in many places to abbreviate function declaration and usage as well as with various functions of the array class that require a function as a parameter (such as `map` and `filter`).

https://www.warambil.com/javascript-arrays-and-arrow-functions
https://www.javascripttutorial.net/javascript-array-map/
https://www.javascripttutorial.net/javascript-array-filter/

6 Classes

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoClasses.ts`

Functions in classes are technically known as methods. Methods within a class do not have the keyword `function` preceding their declaration, as they do for standard functions

The `readonly` modifier is used to specify constant properties whose values can no longer be changed after initialization.

Parameter properties provide a shortcut for declaring properties and initializing them in a constructor

There are 3 access modifiers which can be applied to properties and methods in a class:

- `private` - allows access within the same class.
- `protected` - allows access within the same class and subclasses.
- `public` (default, if none specified) - allows access from any location.

Inheritance is achieved using the `extends` keyword. Child classes will inherit all `public` and `protected` properties from their parent class. We can use the `super` keyword to access methods (including the constructor) of the parent class. We can also override normal methods from the parent class if necessary.

7 Interfaces

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoInterfaces.ts`

Interfaces can define properties (some of which can be optional or readonly) as well as declare methods. Classes can implement interfaces and must provide definitions for the methods in the interfaces as well as include their properties.

8 Control flow

Copy the following files from `TypeScript-Demo` into the TypeScript project folder:

`DemoControlFlow.ts`

TypeScript expressions are simply JavaScript expressions and these work on the basis of truthy / falsy values when the loose equality operator (==) is used. The recommendation is therefore to use the strict equality operator (===) whenever possible to avoid subtle code errors.

https://www.sitepoint.com/javascript-truthy-falsy/
https://javascript.plainenglish.io/javascript-comparison-operators-loose-equality-vs-strict-equality-explained-w-3d4004625c7f

Most of the other control flow structures are nearly identical to those in other major programming languages.