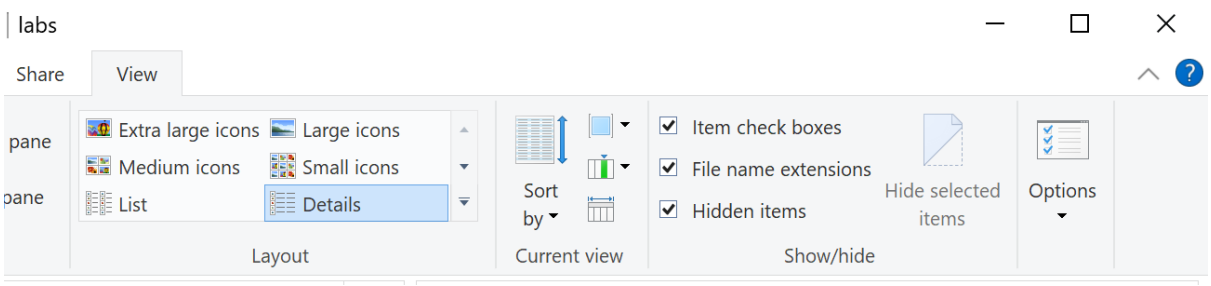
TypeScript Overview Lab

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)

We will need to be able to view file name extensions directly in order to be able to directly manipulate TypeScript source code files. If you are using Windows, make sure that you have checked the File name extensions in your File Explorer in order for us to be able to directly manipulate the file extensions when creating or modifying files.



For MacOs:
Show or hide file name extensions - Article 1
Show or hide file name extensions - Article 2

2 Initializing and compiling a TypeScript project

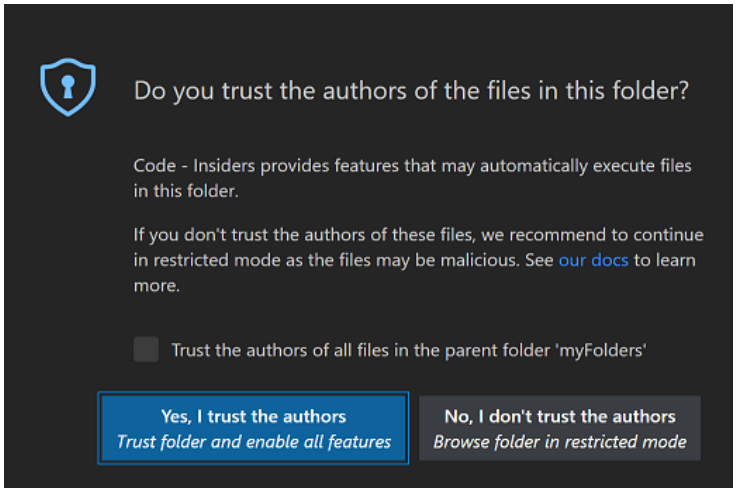
Create an empty folder on a suitable location on your local drive to use as the root folder for TypeScript project for this lab. Copy all the files from `lab-code/TypeScript-Demo` into this new project folder.

To use transpile and execute the TypeScript source code in this new project folder, we need to open a command prompt shell in Windows (or terminal in MacOs) and navigate to this folder

Working with MacOS
Opening terminal on Mac
Opening and using terminal on Mac
Navigating files and folders in Terminal

Working with Windows
Opening a command prompt shell directly from File Explorer
Different ways of opening the command prompt in Windows 10
Navigating to different directories in command prompt in Windows 10 - Article 1

Start up VS Code. From the main menu, select File -> Open Folder, and then navigate to your project folder and open it. You may obtain a message similar to the following:

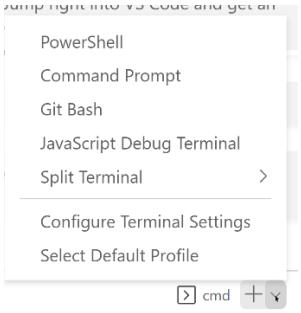


Click in the checkbox next to the sentence `Trust the authors of all files in the ……` and then click on the button `Yes, I trust the authors`.

You will see numerous files with the extension `.ts` in the Explorer pane on the left. These are the TypeScript source code files.

You can then click on the any of the files with the extension `.ts` in the explorer pane on the left. This will open up that file for editing in the main editor view. You can practice opening and closing these files in VS Code.

VS Code also offers an embedded terminal from which we can type the commands. To access this, select Terminal -> New Terminal from the main menu or use the context menu in the lower right hand corner



All the commands demonstrated in this lab can be run in either an independent command prompt / shell terminal or inside the VS Code embedded shell 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 / compiling 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
```

Notice now the new JavaScript source code files corresponding to the transpilation of the TS files.

Using a new command prompt or VS code embedded terminal, you can run any of these JavaScript files using Node.js in the usual manner:

```
node name-of-file
```

You don't have to specify the extension .js, Node automatically assumes this to be the case.
Randomly select a few JavaScript files in the folder that start with the name `Demo…` and run it in this manner.

Some useful tips when working from the command prompt / VS Code embedded terminal:

1. You can use the Tab key to auto complete the name of the files that you are typing.
2. You can use the up and down arrow key to scroll through the history of commands that you have typed in the terminal.
3. Use the command `cls` to clear the screen when it becomes too cluttered with text

Note that the 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 types

Files to use: `BasicTypes.ts`
Exercises: `BasicTypesExercises.ts`

The most commonly used primitive types are string, number and boolean.

In TypeScript, we will typically declare variables 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. There is a difference between these two keywords but for most purposes they are minor.

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.

The ES6 string template allows us to work with strings in a more flexible way.

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.

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 Array type has access to all the standard JavaScript array methods and properties

4 Objects

Files to use: `Objects.ts`
Exercises: `ObjectsExercises.ts`

Objects in TypeScript are identical to in JavaScript. However in TypeScript, most of the time we will be using classes to create objects, rather than creating them explicitly as in the case with JavaScript.

5 Functions

Files to use: `Functions.ts`
Exercises: `FunctionsExercises.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.

5.1 Object methods

Files to use: `ObjectMethods.ts`

Objects can have functions as their properties, and these are invoked in the same way as ordinary functions.
An Array is an object and has access to all the standard JavaScript array methods and properties
There is a String and Number object that provides a number of useful methods to work with strings and numbers in TypeScript.
The Math object provides a large number of useful methods for useful numerical operations.

In both TypeScript and JavaScript, when you 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. This is called autoboxing of primitives.

5.2 Arrow functions

Files to use: `ArrowFunctions.ts`
Exercises: `ArrowFunctionsExercises.ts`

Arrow functions are a new language feature introduced in ES6 that provide you with an alternative way to write a shorter syntax compared to the function expression. Arrow function syntax is widely used throughout JavaScript applications, particularly by experienced developers.
They are typically used with the wide variety of higher order array methods which accept another function as their parameter. The most widely used ones are map and filter.

6 Classes

Files to use: `Classes.ts`
Files to use: `ClassesExercises.ts`

The concept of classes in TypeScript is similar to that of classes in other pure OOP like Java, C#, C++, etc.

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

Files to use: `Interfaces.ts`
Files to use: `InterfacesExercises.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

Files to use: `ControlFlow.ts`

The structures to control program execution such as if-else-if, switch, for-loop, while-loop are identical to that in JavaScript as well as other languages.

9 Implicit type coercion and strict / loose equality operators

File to use: `ImplicitConversion.ts`

Explicit type conversion / coercion is where the specified type to be converted to is explicitly specified programmatically via code. For e.g. explicitly converting a number to a string, or vice versa. Implicit coercion is where the type conversion is performed automatically in the background by the JavaScript engine.

For the case of implicit Boolean conversion, the result of using the Boolean function to convert any particular value will result in either true or false. Thus, if a given value when converted via Boolean gives a true, it is said to have a truthy value, and a falsy value otherwise.

If you are using a non-boolean operand directly within a logical context (such as within an if , if else, if else if statement without any comparison operators involved) or when Logical operators are used, for e.g.

```
if (x) {
}

if (x || y)
}
```

then implicit coercion of the operand to a boolean value occurs

Files to use: `StrictLoose.js`

JavaScript provides two types of equality operators: the loose equality operator (==) and the strict equality operator (===).

The loose equality operator (==) allows comparison between operands of different types (for e.g. string and number, where the string number will be implicitly converted to a number). This provides greater flexibility but results in possible complications.
The strict equality operator (===) does not perform implicit conversion. If both operands are of the same type and have the same value, then and only then is the comparison equal to true.

TypeScript prevents comparison between operands of different types, but you should still use the === operator to make sure.

10 TypeScript specific types

Files to use: `OtherTypes.ts`

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

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.