PER SCHOLAS

ALAB 317.1.1:
Working with TypeScript

Version 1.0, 6/9/23
Click here to open in a separate window.

Introduction

This lab will have you work through a list of TypeScript problems to create a TypeScript program that includes classes, unions, literal types, generics, and more. Working through this process is intended to familiarize you with basic TypeScript syntax and techniques, while also providing insights into the similarities between TypeScript and standard JavaScript code. This will enable you to confidently move forward with TypeScript in your future projects.

Objectives

- Convert an existing JavaScript file into TypeScript.
- Fix existing but unidentified bugs in JavaScript code using TypeScript's editor integration.
- Write TypeScript code from scratch, including:
  - Classes (with Generics).
  - Methods and Functions.
  - Variables.
- Test TypeScript code through simple implementations and basic console logging.

Resources

This lab uses CodeSandbox. If you are unfamiliar with CodeSandbox, or need a refresher, please visit our reference guide on CodeSandbox for instructions on:

- Creating an Account.
- Making a Sandbox.
- Navigating a Sandbox.
- Submitting a Sandbox link to Canvas.

Submission

Submit your completed lab using the **Start Assignment** button on the assignment page in Canvas.

Instructions

You will begin with a pre-configured CodeSandbox that contains a single JavaScript file: **index.js**.

Part 1: Converting JavaScript to TypeScript

The provided CodeSandbox has a variety of errors that are not immediately apparent, and fixing them at runtime could prove tedious. Bugs may not immediately present themselves, and we do not want the end user to be the one that finds them!

Convert the provided index.js into a TypeScript file, index.ts simply by renaming it.

TypeScript has already been installed and configured in this Sandbox, so this should be a simple process. When complete, change the reference from **index.js** to **index.ts** within the **index.html** file, and include the **@ts-check** comment at the top of **index.ts** to enable error checking.

Part 2: Fixing Existing Errors

With error checking enabled, fix all of the errors that are present in the base file by fixing syntax or adding type annotations where relevant, as follows:

- Within the Vehicle class:
  - Add appropriate types for all current Vehicle properties and method parameters.
  - For "status," use a union of literals to declare valid status options:
    - "started" or "stopped"
- Adjust the Car and MotorCycle classes, as needed according to any TypeScript errors.
- Change the printStatus function to accept a parameter of type Vehicle.
  - Fix any errors that reveal themselves due to this type check.
- Fix errors in the output statements below the function definitions.

Already, TypeScript has revealed many errors that existed in the code and allowed us to fix them before runtime with minimal additional code.

Part 3: Creating a Generic Class

Create a new class following the existing code, called NCycle.

- Copy the existing Vehicle class definition as a starting point for NCycle.
- Modify NCycle to accept a generic type.
- Allow make and model to have *either* the generic type or an array of the generic type.
  - Adjust the constructor parameters accordingly.
- Create a new method print, which returns nothing and has a single number parameter (either optional or defaulted to **0**).
  - Use type guards and other appropriate techniques to implement print such that it logs:
    - "This is a <make> <model> NCycle." if make and model are not arrays.
    - "This NCycle has a <make> <model> at <parameter>." if make and model are arrays and the index of parameter exists in each.

        - "This NCycle was not created properly." if neither of the above are true.
- Create a new method printAll, which returns nothing and has no parameters.
    - Use type guards and appropriate techniques to implement printAll such that it logs the same statements as print, but for all matching pairs in the make and model arrays, if applicable.

Part 4: Testing

Test your code in whatever ways you see fit. Below, we have included a block of rudimentary testing code that you can use, but you must compare this with your own expected output.

```
// Rudimentary testing Code, not part of the assignment
const testCycle1 = new NCycle<number>(1, 2, 3);
testCycle1.print();
testCycle1.printAll();

const testCycle2 = new NCycle<string>("This", "That", 4);
testCycle2.print();
testCycle2.printAll();

const testCycle3 = new NCycle<string>("Make", 10, 4);
testCycle3.print(4);
testCycle3.printAll();

const makes4 = ["Volkswagon", "Tesla", "Audi"];
const models4 = ["Passat", "Model X", "A4"];
const testCycle4 = new NCycle<string[]>(makes4, models4, 4);
testCycle4.print(2);
testCycle4.printAll();

const makes5 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
const models5 = [1, 1, 2, 3, 5];
const testCycle5 = new NCycle<number[]>(makes5, models5, 0);
testCycle5.print(7);
testCycle5.printAll();

function add(x: number, y: number): number {
  return x + y;
}
add(testCycle1.make, testCycle5.model[1]);
// Error expected here
add(testCycle2.make, testCycle4.model[1]);
```

Part 5: Conclusion

With your code thoroughly tested, do not forget to submit the link to your CodeSandbox, according to the submission instructions at the beginning of this document.

This code serves no immediate purpose, but should have demonstrated how simple it is to write arbitrary TypeScript code (almost as simple as writing arbitrary JavaScript code!).

Likewise, this exercise highlights the power of TypeScript's editor integrations and ability to identify potential bugs before they occur in runtime. Without extensive testing, bugs in JavaScript code remain well hidden. With TypeScript, we can preempt those bugs with well-structured code and type safety.