Lab 4: Type Systems

Fall Semester 2021

Due: 25 October, 8:00 a.m. Eastern Time

Corresponding Lecture: Lesson 8 (Type Systems)

Objective

The goal of this lab is to experience the difference between untyped and strongly typed languages. Typescript is a strongly typed language built on top of the weakly typed Javascript language. Typescript compiles to Javascript, producing Javascript code that is not typed itself, but has many of the advantages of strongly typed code as long as it is not edited after compilation. Since Javascript is widely supported, Typescript can be used to produce better quality Javascript code without needing to be concerned if Typescript is supported.

In this lab, you will be taking provided Javascript code and converting it to Typescript. You will use the properties of strongly typed Typescript to fix errors in the Javascript code. You will submit your Typescript code, which we will compile to Javascript and run unit tests to determine if you have fixed all the issues.

We will be testing your code with the popular Javascript testing framework, Mocha in conjunction with the assertion library, Chai. You will not need to understand the details of these frameworks for the purposes of this lab. However, if you are interested in learning more, we've included documentation links below.

Resources

1. Typescript Site:
   https://www.typescriptlang.org/index.html
2. Typescript Tutorials:
https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html
3. Typescript Documentation:
   https://www.typescriptlang.org/docs/home.html
   https://www.typescriptlang.org/docs/handbook/interfaces.html
4. Mocha Documentation:
   https://mochajs.org/
5. Chai Documentation:
   https://www.chaijs.com/

Setup

In this lab, we will be running the Typescript compiler on the course VM. Confirm that Typescript is successfully installed by running the following command:

```
tsc -v
```

This command will output the version of Typescript that was installed. Currently 3.8.3

Demonstration Part 1 - Compiling Simple Javascript into Typescript

In this demonstration, we will run the Typescript compiler against some simple Javascript. The first example shows what you can expect during a successful compilation of code with no issues. The second example shows what you can expect when the Typescript compiler finds an error in your code.

Both of the simple examples can be found in the provided `typescript.zip` archive. Place the zip file in the home directory in your VM and extract the files with the following command:

```
~$ unzip typescript.zip
```

Navigate to the Typescript examples directory that now exists in your home directory

```
~$ cd typescript/examples/
```

We will be working with two files in this directory: `sample.js` and `sample-convert.ts`. Every valid Javascript program is a valid Typescript program, so if we make a copy of a valid Javascript file with a Typescript extension (`.ts`), compile it using the Typescript compiler, and compare it to the source file, we should get identical file contents.

```
~/typescript/examples$ cp sample.js sample-ts.ts
~/typescript/examples$ tsc sample-ts.ts
~/typescript/examples$ diff sample.js sample-ts.js
```

You will notice that the diff command produces no output because the files are identical.

Our next example is a file that a developer began converting from Javascript to Typescript. This file had an issue initially where text strings and numbers were used interchangeably for values that were added together, resulting in incorrect math. Specifically, the string `"42"` had `8` added to it, with the result being `428` instead of the expected `50`. The provided `sample-convert.ts` file has the variable declarations converted to Typescript syntax, where the data type is specified. We have also provided the original `sample-convert.js` file and a `sample-convert.html` file you can use to call the `sample-convert.js` code. This conversion has exposed two errors when we compile it:

```
~/typescript/examples$ tsc sample-convert.ts
sample-convert.ts(3,9): error TS2322: Type '"42"' is not assignable to
type 'number'.
```

```
sample-convert.ts(6,39): error TS2345: Argument of type 'number' is not
assignable to parameter of type 'string'.
```

The first error is because we are trying to use a string when we declare a number variable. We can correct this error by removing the quotes around the value `42`. The second error is because we are calling a function that expects a string as its input with a variable that is the number data type. If we look at the function's logic, it appears that the developer only added the call to `parseInt()` to compensate (incorrectly) for the type error introduced by treating a number as text, which we already corrected. A good fix here would be to remove the call to `parseInt()` and simply return the value `combined` instead of `converted`. Here is cleaned up code that will compile without error and produce a correct result when executed:

```
function convert() {
      let message : number = 42;
      let count : number = 8;
      let combined : number = message + count;
      return combined;
}
```

Demonstration Part 2 - Incomplete Specifications

In software engineering, a "specification" is a description of the software to be developed. If the specification isn't strong enough there can be some confusion on what the software is supposed to do, in particular with error cases and other abnormal operations. Incomplete specifications can cause unexpected behavior and bugs when the program is executed on edge cases. In this demonstration we will explore an example of unexpected behavior due to an incomplete specification.

Let's start by opening the file `find.js`. Here, we have a function to find an element in an array. If the element exists in the array, we return the index that it resides in. However, the specification does not describe what happens in the case where the element does not exist in the array. Run the program and observe the output in the success and error cases:

```
~typescript/examples$ node find.js
```

It is probably clear to you what the bug is, but we will go ahead and add type annotations as an exercise. Create a Typescript file as follows:

```
~/typescript/examples$ cp find.js find.ts
```

Let's start by adding type annotations to the parameters and return type of find:

```
function find(elem: string, arr: string[]): number {
```

Now, when we compile with the following command:

```
~/typescript/examples$ tsc --noImplicitReturns find.ts
```

The Typescript compiler reveals an error on the function find. To fix the bug, ensure that find returns a number in all possible paths of execution. There are a few ways we could fix this, but for this exercise, let's `return -1;` in the case where elem does not exist in arr.

Demonstration Part 3 - Interfaces in Typescript

Typescript employs Duck Typing: a type checking principle that focuses on the shape of values. The name is inspired from the "Duck Test"- "If it walks like a duck and it quacks like a duck, then it must be a duck." So, Typescript's type checking focuses on the shape of an object rather than its name. Interfaces provide a way to name these types, creating a stronger contract.

We will explore Typescript's interfaces in the next example, `interfaces.js`. In this example we have a function `printVolume()` that computes the volume of a right rectangular prism and outputs it to the console. However, the function is called with an argument that is not a right rectangular prism, violating the specification.

Run the program with the following command:

```
~/typescript/examples$ node interfaces.js
```

You will see that the first call to `printVolume()` executes successfully. However, the second call results in the output `NaN`. We will now demonstrate how Typescript interfaces can catch this bug at compile time.

Begin by creating a Typescript file as in the previous demonstration:

```
~/typescript/examples$ cp interfaces.js interfaces.ts
```

Now, we will edit interfaces.ts to create an interface for right rectangular prism:

```
interface RightRectPrism {
};
```

Right Rectangular prisms have a length, width, and height. Add these fields to the interface as follows:

```
interface RightRectPrism {
      length: number,
      width: number,
      height: number
};
```

Lastly, add a type annotation to the argument, obj:

```
function printVolume(obj: RightRectPrism){
```

Save interfaces.ts. Now, when we compile:

```
~/typescript/examples$ tsc interfaces.ts
```

The Typescript compiler reveals a Type Error on the incorrect argument, cone. To keep progressing, comment out or remove the line that calls printVolume on a cone.

```
//printVolume(cone);
```

For the purposes of the demonstration, let's add a few more fields to learn more about Typescript's type system. Suppose that we would like a naming system for our shapes so we can print a more clear output with "`name, volume`." Add a field name with type `string`. However, it doesn't necessarily make sense to name all of our shapes with alphanumeric titles. Now, suppose that we want to name some of our shapes with numbers. We would like our name field to be of type string OR number. This is called a **Union Type** and is written with a vertical bar (`|`). Now, let's make the name field optional by adding the (`?`) operator:

```
interface RightRectPrism {
      length: number,
      width: number,
      height: number,
      name?: string | number
};
```

Lab Instructions

Part 1 - Preventing Unexpected Behavior Due To Type Coercion

In the first part of this lab, you will add Typescript annotations to find a bug in an extension for the Brackets IDE code base. Brackets is an open source text editor primarily used for web design. This extension shows a preview when the cursor is hovered over certain items.

This bug is tricky to find in Javascript as it does not cause a crash. Javascript silently coerces a value of type A to type B causing unexpected behavior. Take a look at the `HoverPreview/main.js` file in your favorite text editor. After studying the code, you will likely not be able to easily find the bug.

Your task is to convert the Javascript code to Typescript and reveal the bug by adding type annotations.

We will begin by editing the `main.ts` file. Now, start by adding type annotations to the helper functions at the top of the file. Typescript employs **gradual typing**, meaning that not every identifier needs to have a type. You will notice that as you add type annotations, you can incrementally compile to see if the bug is revealed:

```
~/typescript/HoverPreview$ tsc main.ts
```

Once the bug is revealed by the typescript compiler and your newly added annotations, you need to modify the `main.ts` file to fix the bug.

To test your fix, install the provided test suite with the following commands:

```
~/typescript/HoverPreview$ npm install typescript
~/typescript/HoverPreview$ npm install mocha
~/typescript/HoverPreview$ npm install @types/mocha
~/typescript/HoverPreview$ npm install chai
~/typescript/HoverPreview$ npm install ts-node
```

Now, run the test suite to check your work:

```
~/typescript/HoverPreview$ npm run test
```

Note that if the test suite fails, you may get additional errors at the bottom of the console output below the test case specific output about not having the latest version of node.js. You can ignore these errors and focus on the failing test cases.

Grading

- We will be compiling your Typescript program with the command `tsc main.ts`
- We will be validating that your submitted code passes all provided test cases
- We will be validating that your submitted code uses meaningful type annotations
- We may run more test cases than the provided test suite as part of grading

Part 2 - Clarifying the Specification through Type Annotations

In this part of the lab, we will use type annotations to refine an incomplete specification in the Spring Roll code base. Spring Roll is a tool set for building accessible HTML games. There is a bug in the Container file that control page visibility. This bug likely arose due to an incomplete specification.

Your task is to add the necessary type annotations to find and fix the bug. Begin by editing `SpringRoll/container.ts`. As you add type annotations, incrementally compile with the command:

```
~/typescript/SpringRoll$ tsc container.ts
```

Once the bug is revealed by the Typescript compiler and your newly added annotations, modify the `container.ts` file to fix the bug.

To test your fix, install the provided test suite with the following commands:

```
~/typescript/SpringRoll$ npm install typescript
~/typescript/SpringRoll$ npm install mocha
~/typescript/SpringRoll$ npm install @types/mocha
~/typescript/SpringRoll$ npm install chai
~/typescript/SpringRoll$ npm install ts-node
```

Now, run the test suite to check your work:

```
~/typescript/SpringRoll$ npm run test
```

Note that if the test suite fails, you may get additional errors at the bottom of the console output below the test case specific output about not having the latest version of node.js. You can ignore these errors and focus on the failing test cases.

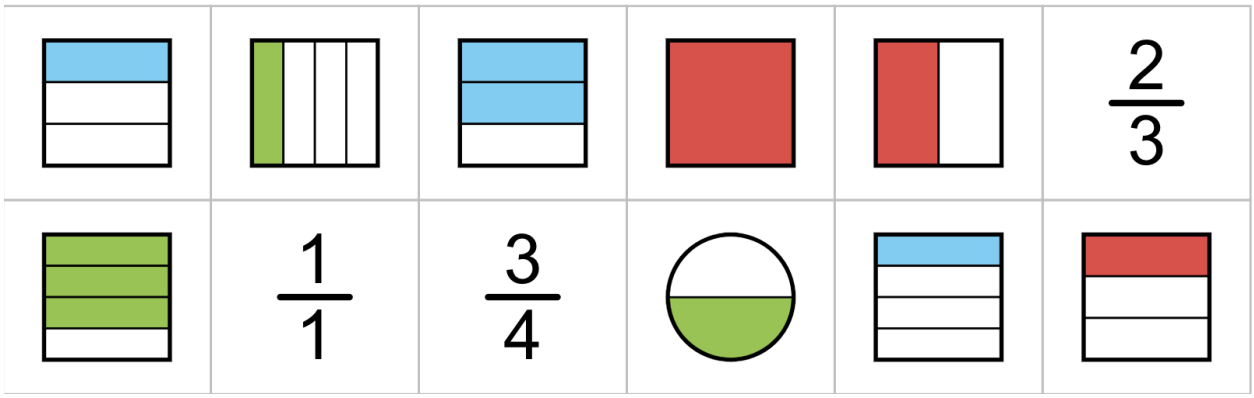
Grading

- We will be compiling your Typescript program with the command `tsc container.ts`
- We will be validating that your submitted code passes all provided test cases
- We will be validating that your submitted code uses meaningful type annotations
- We may run more test cases than the provided test suite as part of grading

Part 3 - Null and Undefined Checks through Type Annotations

In this part of the lab, you will find a bug in real world code for an educational fraction game: https://phet.colorado.edu/sims/html/fraction-matcher/latest/fraction-matcher_en.html

In this game, the user drags images of shapes to find matches. For example, in level one, the following shapes are presented



The fraction 1/1 would match with the filled red square, the half green circle would match with the half red square, and so on.

In order to match the shapes, the user must drag and drop them into a "matching zone." We've given you the code for this functionality. Unfortunately, this code has a crashing bug. Your task will be to convert the Javascript code to Typescript in order to find the bug.

This part of the lab will be an exercise in interfaces. You will need to create interfaces in the Typescript file to provide appropriate types to objects. All shape objects have a `numerator` and `denominator` in order to compare with other shape objects for equality. Additionally, they have a field `dropZone` that holds a numeric value indicating which (if any) drop zone the shape is currently in. Lastly, each shape has a `view`. The view itself has properties `indexShape`, `height`, and `moveToFront`. Since every shape is not displayed on every level, the view may be undefined.

Recall that due to Typescript's **gradual typing,** not every field in our interface needs to have an explicitly annotated type. However, the more types we specify, the stronger the contract, and the greater chance there is of catching bugs at compile time. For fields without an explicit static type, we can add the type annotation `any`. This allows us to "opt-out" of compile time type checks on this identifier.

Your task is to add the necessary type annotations outlined above. After adding the type annotations, compile the file with the command:

```
~/typescript/FractionMatcher$ tsc --strictNullChecks LevelNode.ts
```

Use the errors to find the bug(s) and fix them.

Here, we only use one compiler flag. However, Typescript has many compiler options. If you were working with a project with many files, or using many different compiler options, this could get quite verbose. In order to prevent an unnecessarily long compiling command, Typescript offers a framework to save your configurations in a JSON file. Start by creating a file tsconfig.json

```
~/typescript/FractionMatcher$ touch tsconfig.json
```

Now, open up `tsconfig.json` in your favorite text editor and add:

```
{
	"compilerOptions": {
		"strictNullChecks": true
	},
	"files": [
		"LevelNode.ts"
	]
}
```

Now we can compile with the command:

```
~/typescript/FractionMatcher$ tsc
```

Once you have fixed the bug(s) revealed by the Typescript compiler and your newly added annotations install the provided test suite with the following commands:

```
~/typescript/FractionMatcher$ npm install typescript
~/typescript/FractionMatcher$ npm install mocha
~/typescript/FractionMatcher$ npm install @types/mocha
~/typescript/FractionMatcher$ npm install chai
~/typescript/FractionMatcher$ npm install ts-node
```

Now, run the test suite to check your work:

```
~/typescript/FractionMatcher$ npm run test
```

Note that if the test suite fails, you may get additional errors at the bottom of the console output below the test case specific output about not having the latest version of node.js. You can ignore these errors and focus on the failing test cases.

Grading

- We will be compiling your Typescript program with the command `tsc --strictNullChecks LevelNode.ts`
- We will be validating that your submitted code passes all provided test cases
- We will be validating that your submitted code uses meaningful type annotations
- We may run more test cases than the provided test suite as part of grading

Items to Submit

For this lab, we will be using Gradescope to submit and grade your assignment. For full credit, the files must be properly named and there must not be any subfolders or additional files or folders. You must include meaningful type annotations, and all public and hidden tests must pass. Submit the following files through Gradescope.

- Submit `main.ts` (30 points)
    - Submit your modified version of `main.ts` for HoverPreview. Do **not** include your `node_modules` folder or your `package-lock.json` file
- Submit `container.ts` (35 points)
    - Submit your modified version of `container.ts` for SpringRoll.
- Submit `LevelNode.ts` (35 points total)
    - Submit your modified version of `LevelNode.ts` for FractionMatcher.

Do not submit anything else. Make sure all of your code modifications are in the files listed above as your other files will not be submitted. In particular, past students have modified header files to import additional headers, leading to a compile error in their submission.

Through Gradescope, you will have immediate feedback from the autograder as to whether the submission was structured correctly as well as verification that your code successfully runs the public tests on the grading machine. The public tests are the same tests provided to you as part of the lab, but these tests won't (and are not intended to) catch everything. You will get your final score after the assignment due date.

How Do We Use It?

TypeScript was first publicly available in 2013. Since that time, its use has expanded beyond Microsoft to many well known companies, including Salesforce, Oracle, Capital One, and

Slack. While JavaScript is still as of Q4 2020 the most used language on GitHub (18.8%), TypeScript is now the 7th most used language (6.6%) and is rapidly growing.

Slack shares about their experience converting from JavaScript to TypeScript: “To improve our situation, we decided to give static type checking a shot. A static type checker does not modify how your code behaves at runtime—instead, it analyzes your code and attempts to infer types wherever possible, warning the developer before code ships. A static type checker understands that Math.random() returns a number, which does not contain the string method toLowerCase() … A smart static type checker increases our confidence in our code, catches easily made mistakes before they are committed, and makes the code base more self-documenting.”