Programming Languages Term Paper: TypeScript

Prepared by,

Trevor Spitzley

Computer Science Undergrad

Esther Padnos College of Engineering and Computing

Prepared for,

Dr. Nathan Bowman

Computer Science Legend/Professor

Esther Padnos College of Engineering and Computing

December 11, 2020

Tables of Contents

Page

Introduction

Typescript, sometimes spelled TypeScript, is an open-source programming language developed and maintained by Microsoft. It is a superset of the popular language JavaScript, but with added capabilities most notably being optional static typing, which will be explained later, and class-based object-oriented programming which has been studied for years. TypeScript was first published to the website GitHub on September 17th of 2014 with their first version being v1.1 (Microsoft, 2014). Since then, TypeScript had been updated almost weekly with new stable versions coming out every couple of months. As of the writing of this paper at 3pm on Friday, December 11th, there had been an update as recent as 10 hours ago (Microsoft, 2014). Having been an improvement on the already widely used JavaScript, TypeScript was bound to grow in popularity. The current number of TypeScript users is unknown and would be hard to calculate, but currently on the website *npmjs.com* it shows TypeScript reaching weekly download levels of over 15 million downloads (Npmjs TypeScript, 2020). TypeScript is a popular and extremely successful open-source language, which has made its way into the industry by storm being used for many kinds of development.

One of TypeScript biggest non-coding benefits is its ability to be used in multiple different paradigms of computer science. Given the addition of class-based OOP to the language, it can be successfully used in object-oriented programming. Another few paradigms this language can fall into depending on implementation are imperative, generic, and functional (Typed JavaScript at Any Scale). A functional language can be thought of as a language used primarily off the creation of functions, like Scheme. A generic language is used more so when you are writing functions where you do not know what type of data you may be passed. Languages like Python, Java, and even C++ can implement this. An imperative language is, in

easier terms, a language that uses code to change the current program's state. This paradigm is used a lot in older languages, such as COLBOL, FORTRAN, and ALGOL. TypeScript is a very flexible language being able to implement and be used in many of these different styles, which is why it is growing in popularity for many different industry level projects today.

For the most part, TypeScript was primarily used for Web Development having been the main language used for the front-end framework Angular after their switch from Angular.js. After this switch, many users began learning the language given its vast capabilities discussed earlier. Since then, TypeScript is often being used as an out-right replacement for JavaScript in certain situations. These situations include but are not limited to front-end Web Development with Angular, back-end Web Development through Node.js or Deno, and even some Data Science projects (*TypeScript* 2016). The biggest benefit of switching the TypeScript is the added security when passing data and the ease of debugging. Both processes become easier and less necessary when you have static typing because there are less chances of blowing up your program by passing a string instead of an integer, when you did not know you were doing so based on the like of static typing (*TypeScript* 2016).

Data Abstractions

Being that TypeScript is a superset of JavaScript with static typing, it opens a whole new world for what kinds of data types one can expect. TypeScript supports some of the more common data types seen in many other popular languages, data types like *String, Number* (used for integers, and floating points), as well as *Booleans,* and *Arrays*. But some of the data types TypeScript also implements that some languages do not are data types like *Enums (enumerators), Tuples, Unions, Any, Void,* and *Never* (TypeScript Tutorials). While the data

types *Enums, Tuples,* and *Unions* may be seen in other languages, the average programming might not have knowledge of the others.

Any data type is somewhat self-explanatory, it is a data type that we do not specifically know what it will be until runtime. Think of an *Any* type as a normal variable in Python. You can set it to a string, then later set it to an integer, etc., and it will all run without issues.

Void data type has a specific kind of usage, like its usage in Java, where a data type or function returns a *Void* variable when there is no data being made or used.

The *Never* type is used in a function where you do not expect to reach the functions end, or for a function that is always expected to throw an error. It is used with variables when the value will never occur or exist (TypeScript Tutorials).

Given that TypeScript is a superset of JavaScript, it uses dynamic typing. You can also say it like the language is dynamically typed. Dynamic typing is the idea that your variables are somewhat "trusted" to be the type that you set it to and only checked at run-time. This can of course cause problems in code as it often does but speeds up a lot of code and saves on memory. When it comes to variables in TypeScript, given the static typing referred to earlier it is guaranteed that TS variables are almost always explicitly defined. The word almost is used because of the previously defined data type of *Any*, and the fact that TypeScript can be compiled into JavaScript so having a variable that is not defined will not throw any errors unless used improperly. Because of TypeScript's explicit variables and overall dynamic typing, it is also a strongly typed language. This means that the TypeScript compiled will check the validity of certain assignments, not allowing you to define a variable as a string and store a floating-point value in it, unless it is declared as an *Any* variable. Coercion in computer science is similar to

type casting, which in TypeScript is even more similar to type assertion. Type assertion allows you to set the type of a value and tell the compiler not to read or infer it (TypeScript Tutorials). For a better explanation look at the code snippet below.

```
let code: any = 123;
let employeeCode = <number> code;
console.log(typeof(employeeCode)); //Output: number
```

While the first line shows a variable called code of type any, we know that it is a number given that we can see its assignment. When using type assertion, if we wanted to place the data in 'code' into the data of another variable we can do so by using the <> operators. Inside of it having whatever type we plan on casting to. After doing this successfully, we can print off the type definition of the newly made variable and see it is what we wanted it to be.

Control Abstractions

TypeScript is a language that reads like a lot of other popular languages, for good reason. It supports a lot of the typical control flow aspects that other languages do like if-statements, looping (for loop and while loop), switch statements, etc. Making its readability out of the box quite nice because of most programmer's prior experience with those things. Within those expressions there are many things to consider like operator precedence, function/parameters passing techniques, scope rules, and things like module capabilities and error handling.

Operator precedence is a lot like the math concept of PEMDAS. The code needs some way to process an expression and know exactly which steps to take to get through it. While some languages may only lean one way, left-associative or right-associative, many modern languages like TypeScript use a combination of both. TypeScript uses a predefined table of operators that are sectioned off by importance and precedence level (*Operator Precedence in TypeScript,*

2020). This is how the language chooses which part of a statement to process first. TypeScript can not only be used as a scripting language, but as discussed earlier can also be adapted to using functions and parameters for more complicated processes. This involves the passing of parameters to functions and the return of object/variables from functions. One way many languages accomplish this is by passing parameters by values or passing by reference. TypeScript, like C++, uses both options. For larger amounts of data like arrays or objects, TypeScript tends to pass by reference to not use too much memory and remain efficient. When passing primitive data types, TypeScript will pass by value as it allows a true representation of the parameter passed.

TypeScript also has similar scope rules to languages like Python and Java. If you create a variable outside of a bracket-enclosed chunk of code, you can reference and reassign that variable inside the code as well. This is not allowed vice versa. Creating a variable inside a chunk of code does not allow for that variable to be referenced anywhere else in the code given the language's scope rules. The same rules go for functions as well.

Packages, modules, and namespaces are all roughly referencing the same things. This is a way for a language to import a previously written portion of code and use it in one's program, so that the programmer does not have to re-write code that has already been written and tested over the years. The specific term that TypeScript uses for this is called modules. Some of the more popular modules people have heard of are JavaScript frameworks like React, Vue, or Angular. They all require an import statement at the top of the file to be used, and all behave relatively in the same way. You will see this in my sample calculator program. The import statement looks something like this below.

```
const readline = require('readline')
```

What this code snippet does is import the module called ‘readline’ into the program so we can then create an object from it and use the object’s properties in our code. Specifically, this module is used with Node.js to read input from places like the console, the command line, and even files.

Error handling in TypeScript is similar to error handling in Java but involves a few extra options to evade errors entirely that are similar to guard statements in newer languages like Swift and Kotlin. The most common way of handling errors in TypeScript is by catching them and either throwing them or catching them and printing of a graceful message that does not involve your program blowing up. One common way of doing this is by using a try-catch block. What this does is “try” to execute the code in the try block, but if encountering an error, it will catch the error before it blows up the program and allow the programmer to handle it in whatever way they would like. The way TypeScript allows the programmer to sometimes evade errors entirely is through type-guarding/type-predicates. These are used only in certain situations but come in handy given TypeScript’s *Any* data type being able to be anything. To boil it down, a type guard is an expression that performs a run-time check on a variable that guarantees the data type of that variable given the scope and usage (Typed JavaScript at Any Scale). For a better example check out the code snippet below.

```
function isFish(pet: Fish | Bird): pet is Fish {
  // Some code here
}
```

The statement ‘pet is Fish’ is the type-guard/type-predicate here. Only if the pet object that you are passing is of type fish will the inner code execute. Otherwise, it will not run at all.

Support for OOP

TypeScript began adding implementation for class-based object-oriented programming with the release of ECMA Script 2015 better known as ECMA Script 6. Their structure is similar to Python classes, with the class header being the word “class” followed by the name of the

class. Inside of the class you will find constructor(s), properties, and methods as well. The constructor is clearly labeled as such using the keyword constructor, and the methods are labeled with the word function.

There is some flexibility in the definition of these classes, but when objects are made from a class interface those classes are bound by the rules of the interface (TypeScript Tutorials). This is shown in the below code snippet.

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK

let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in type 'KeyPair'

let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```

The last two objects create errors given that they are not following the defined architecture of the KeyPair class. The kv2 object fails because it tries to set the property “val”, when that does not exist and must be named “value”. The kv3 object fails because of TypeScript explicit typing. The constructor is looking for specifically a string in the “value” property, so when it is passed a number it throws an error because that is not what the interface is expecting.

TypeScript offer data protection mechanics when it comes to these classes as well. This comes from the option of setting properties of a class to be able to be read only and not written to. One does this by simply typing “readonly” before the property. While read-only properties can be accessed outside of the class, attempting to write to them will produce an error stating that it is read-only or a constant value. Because of this, read-only properties of a class or interface

need to be either initialized at the time of declaration or initialized inside of the class/interface constructor (TypeScript Tutorials).

References

Microsoft. (2014, September 17). Microsoft/TypeScript. Retrieved December 12, 2020, from https://github.com/microsoft/TypeScript

Operator Precedence in Typescript. (2020, September 26). Retrieved December 12, 2020, from https://www.tektutorialshub.com/typescript/operator-precedence-in-typescript/

Typed JavaScript at Any Scale. (n.d.). Retrieved December 12, 2020, from https://www.typescriptlang.org/

TypeScript Tutorials. (n.d.). Retrieved December 12, 2020, from https://www.tutorialsteacher.com/typescript

TypeScript. (2016, August 11). Retrieved December 12, 2020, from https://www.cleverism.com/skills-and-tools/typescript/

Typescript. (2020, November 19). Retrieved December 12, 2020, from https://www.npmjs.com/package/typescript

Appendix

**Explanation of code: Given that TypeScript doesn't have it's own development environment that I know of currently, this was written in TS then compiled into JS using typescriptcompiler on the command line. Then the JS file was ran using Node.js through the command line. Because Node.js is different than the console on a browser, it required different ways of accepting user input. Also given Node.js's asynchronous nature, this was quite hard getting the code to wait for user input and not just skip to the next chunk in code. But I did it! If you have Node and the TypeScript compiler installed, run these commands to run the file (I will also paste the JS version below but I genuinely did do this in TS first). The only problem found with the program through my own testing was I couldn't figure out how to get floating point exponentiation properly. For example, if you have a running total of 2 and then input "^ 2.5", you will get 4 which is incorrect.

Compilation and running commands:
$ tsc calculator.ts
$ node calculator.js

TypeScript Version

```
const readline = require('readline')

const rl = readline.createInterface({
   input: process.stdin,
   output: process.stdout
})

console.log("Welcome to the calculator app!")
console.log("To use this, you start with a running total of 0 and modify it using the following operators")
console.log("+ X, this is used to add X to the running total")
console.log("* X, this is used to multiply X by the running total")
console.log("- X, this is used to subtract X from the running total")
console.log("/ X, this is used to divide the running total by X")
console.log("^ X, this is used to raise the running total to the power of X")
console.log("")
console.log("You can start with a valid input and press enter, or type \'exit\' to end program")

let runTot: number = 0.0
let choice: string = ""
let line: string[] = []

function addNum(a: number){
   runTot += a
}
function multiNum(a: number){
```

```
    runTot *= a
}
function subNum(a: number){
    runTot -= a
}
function divNum(a: number){
    if (a == 0){
        console.log("You cannot divide by 0. Nice try, but you'll have to be quicker than that!")
    } else {
        runTot /= a
    }
}
function powNum(a: number){
    runTot **= a
}



var recursiveAsyncReadLine = function () {
  rl.question('Please enter a valid input: ', function (answer) {
    if (answer == 'exit') //we need some base case, for recursion
      return rl.close(); //closing RL and returning from function.
    else if (answer != "DONE"){
        line = answer.split(" ")
        if (line.length > 1){
            //Addition
            if (line[0] === "+"){
                addNum(parseInt(line[1]))
            }
            //Multiplication
            if (line[0] === "*"){
                multiNum(parseInt(line[1]))
            }
            //Subtraction
            if (line[0] === "-"){
                subNum(parseInt(line[1]))
            }
            //Division
            if (line[0] === "/"){
                divNum(parseInt(line[1]))
            }
            //Exponeniation
            if (line[0] === "^"){
                powNum(parseInt(line[1]))
            }
            console.log("Your running total is: " + runTot)
```

```
        }
    }
    recursiveAsyncReadLine() //Calling this function again to ask new question
  });
}

recursiveAsyncReadLine()
```

JavaScript Version

```
var readline = require('readline');
var rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
console.log("Welcome to the calculator app!");
console.log("To use this, you start with a running total of 0 and modify it using the following
operators");
console.log("+ X, this is used to add X to the running total");
console.log("* X, this is used to multiply X by the running total");
console.log("- X, this is used to subtract X from the running total");
console.log("/ X, this is used to divide the running total by X");
console.log("^ X, this is used to raise the running total to the power of X");
console.log("");
console.log("You can start with a valid input and press enter, or type \'exit\' to end program");
var runTot = 0.0;
var choice = "";
var line = [];
function addNum(a) {
    runTot += a;
}
function multiNum(a) {
    runTot *= a;
}
function subNum(a) {
    runTot -= a;
}
function divNum(a) {
    if (a == 0) {
        console.log("You cannot divide by 0. Nice try, but you'll have to be quicker than that!");
    }
    else {
        runTot /= a;
    }
}
```

```
function powNum(a) {
   runTot = Math.pow(runTot, a);
}
function printTotal() {
   console.log("Your running total is: ${runTot}");
}
function printX(a) {
   console.log(a);
}
var recursiveAsyncReadLine = function () {
   rl.question('Please enter a valid input: ', function (answer) {
      if (answer == 'exit') //we need some base case, for recursion
         return rl.close(); //closing RL and returning from function.
      else if (answer != "DONE") {
         line = answer.split(" ");
         if (line.length > 1) {
            //Addition
            if (line[0] === "+") {
               addNum(parseInt(line[1]));
            }
            //Multiplication
            if (line[0] === "*") {
               multiNum(parseInt(line[1]));
            }
            //Subtraction
            if (line[0] === "-") {
               subNum(parseInt(line[1]));
            }
            //Division
            if (line[0] === "/") {
               divNum(parseInt(line[1]));
            }
            //Exponeniation
            if (line[0] === "^") {
               powNum(parseInt(line[1]));
            }
            console.log("Your running total is: " + runTot);
         }
      }
      recursiveAsyncReadLine(); //Calling this function again to ask new question
   });
};
recursiveAsyncReadLine();
```