TypeScript:

Written report

By Samuel Imlig

CSIS 420: Structures of Programming Languages

Dr. Orr

2022-11-15

Table of Contents

The History of TypeScript

In 2010, Microsoft's software developers were having difficulty maintaining and creating new software written in the programming language JavaScript. This was primarily because JavaScript lacked key features that assist with program scalability, such as the ability to assign a static type to a variable [0]. In order to work around this issue, countless solutions were proposed. One particular idea sought to create a new product called Script#. Script# would utilize Microsoft's popular programming language C# and transpile its code into JavaScript [0]. From a corporate perspective, not only would this help fix the scalability issues, but it would also capitalize on the massive market for web-based applications at the time. However, during Script#'s developmental process, Anders Hejlsberg, the mind behind TurboPascal and C#, thought that a different solution was necessary [0]. This was primarily because in order to fix the scalability concerns, all of the current and legacy code of JavaScript would have to be completely rewritten in C#. This criticism helped lead the team towards a different tactic: a superset of JavaScript that could fix the scalability concerns through the use of an erasable type system. After a period of discussion, the Chief Architect for JavaScript runtime and tools at Microsoft, Steve Lucco, would create the TypeScript team [0].

As the TypeScript project continued to progress, the developers found themselves with a difficult question to address: should it be open-source? TypeScript would be appealing to the JavaScript audience; a passionate open-sourced community, but at the time, Microsoft was very blatantly against the open source paradigm. There are many reasons why this might have been the case, but the most logical reason is that they simply feared it would hurt their business [1]. From this mindset, Script# was significantly more appealing, as money could not only be made from the product, but they could have complete control over the property. However, the TypeScript team knew that this would be an extremely difficult sell to the aforementioned community, so they advocated for the use of the open-source paradigm. This led to fierce debates among executives at Microsoft, but surprisingly, the open-source paradigm was approved for TypeScript [1]. After two years in internal development, TypeScript version 0.8 was officially released to the public during the month of October, 2012.

With the official release of TypeScript, Microsoft began a shift to more open-sourced projects and support. Starting with the release of TypeScript 1.0 in 2014, TypeScript was moved to GitHub to encourage open-sourced development and community involvement with future updates [2]. In addition to more open-sourced involvement, the updates between version 1.0 and 2.0 (2014-2016 respectively) added major support for JavaScript libraries, a completely re-written compiler that was 4 times more performant than its predecessor and a partnership with the web application builder Angular [2]. Between version 2.0 and 3.0 (2016-2018 respectively), the TypeScript team introduced mixins, the non-primitive *object* type, expanding and spreading type parameter lists and a concept called conditional types, which allowed for decisions to be statically represented based upon type [3]. Between version 3.0 and 4.0 (2018-2020 respectively), immutable data-structures were given increased support, an optional *--incremental* flag was added to decrease recompile times and auto imports were significantly improved [4]. Finally, between version 4.0 and 4.9 beta (2020- 2022 respectively), template literal types, key remapping in mapped types and recursive conditional types were all introduced [5]. The main takeaway is that since its creation in 2012, the TypeScript team has been incredibly active in maintaining the language and interacting with the community to improve its functionality.

TypeScript Basics

TypeScript is an object-oriented, open-source scripting language that was developed as a superset to JavaScript. Superset in this context means that TypeScript is effectively an add-on of JavaScript that fulfills the same role; the front end development of web-applications. Its key add-on is the introduction of type annotations and object-oriented paradigms in order to increase productivity and consistency when scaling complex applications [6]. How this is achieved will be discussed shortly, but the basic syntactic and semantic qualities of the language should be conveyed first.

Variables

In TypeScript there are three different scopes that can be given to variables: *const, var* and *let.* The *const* keyword, much like C, is for declaring constant variables that cannot change once they are assigned a particular value. Besides this immutability, a *const* variable is identical to a *let* variable. A *let* variable is confined to the nearest code block and cannot be accessed outside of that block. Finally, a *var* variable allows for the initialization of a variable that can be accessed outside of a code block [7]. For example:

```
if (true) {
    const ten = 10;
    let twenty = 20;
    var thirty = 30;
}
console.log(ten);           // Outputs error: Cannot find name 'ten'
console.log(twenty);        // Outputs error: Cannot find name 'twenty'
console.log(thirty);        // Outputs 30
```

Type Annotation

In TypeScript, type annotation is unnecessary. In fact, a developer could program in pure JavaScript and TypeScript's compiler would still function correctly. However, even if this were to transpire, TypeScript's compiler would try to infer the type of each variable in a program. The compiler accomplishes this by examining the value of each variable upon its first initialization or through the values contained in function parameters and function returns [8]. This might seem simple when examining a variable with a single type, but it becomes more complicated when multiple types can be inferred from a single variable, such as this array:

```
var array = [1, 2, 3, "Hello", null, true];
```

In this array, TypeScript, instead of attempting to pick a single type to constrain to the array, assigns the type: (*number* | *string* | *boolean* | *null*). Inferred types are typically smart enough to decipher what type a particular variable should be, but there is a better way to improve correctness and readability. Formally, this way is referred to as explicit static type checking, but is more colloquially referred to as type annotation [9]. Essentially, type annotation allows a developer to specify the precise type a variable should contain and the compiler verifies that the variable only carries values of the permitted type throughout the runtime of the program. In fact, when developing TypeScript in Microsoft Visual Studio Code, error checking is automatically performed during development time. Consider this trivial example:

```
var array: Array<string> = new Array();
array[0] = "Hello";    // Adds "Hello" to the first index
array[1] = 1;          // Type 'number' not assignable to type 'string'
```

In the example, the programmer is warned about the mistake before compilation, but if they were to compile their TypeScript program, they would see the same error. This ability to see typing errors may not seem terribly useful on a miniscule amount of code,, but when there are thousands and thousands of lines of code in a single program, this service cannot be overstated. JavaScript, unlike TypeScript, does not support this feature. Its errors are only viewable at runtime and it doesn't support static typing at all [10]. Although it may have added flexibility, JavaScript's hindrance of readability and increase of recurring bugs make it ill suited for scalable and sustainable infrastructure. TypeScript's type annotation solves these fundamental issues in design, while still offering plenty of flexibility for the average developer.

Primitive Data Types

TypeScript has several primitive data types that should be familiar to users of Java. For example, it possesses *boolean*, *string* and *null*. The *boolean* and *null* are semantically identical to Java and the only difference between strings in Java and TypeScript lies in syntax: strings can be created using single quotations ('), as well as double quotations ("). However, unlike those languages, TypeScript does not contain the *int*, *float, double* or *long* data types. All of these types are encompassed by the *number* data type [11]. For example:

```
let decimal: number = 13;
let float: number = 3.14;        // Floating point number
let hex: number = 0xFFFF;       // Hexadecimal number starts with 0x
let binary: number = 0b1100;    // Binary number starts with 0b
let octal: number = 0o363;      // Octal number starts with 0c
```

That being said, there is a special purpose type that was borrowed from JavaScript: *bigInt*. This type is used primarily for numbers bigger than $2^{53} - 1$, which are the numbers that are captured by the *number* type [12]. It should also be noted that variables of *number* and variables of *bigInt* cannot perform arithmetic operations without converting to the same type [13]. Another data type borrowed from JavaScript is the *symbol* data type. This data type functions like an alias for a particular value, with some slight differences [14]. Consider this following example:

```
var newSymbol1: symbol = Symbol("key");
var newSymbol2: symbol = Symbol("key");
if (newSymbol1 == newSymbol2)            // Always false
```

Although these symbols might appear to refer to the same string, they are not equal because symbols are completely unique. They are primarily used as keys for objects created in TypeScript, such as maps, or to enable the functionality of the built-in replace*(), search(), split()* and *toString()* methods [14]. Another newer data type to Python, Java and C users is the *undefined* type. This type is given to declared, but uninitialized variables and is a legacy data type from JavaScript. In JavaScript, it is critically important to check for *undefined* values, as this value is used instead of sending runtime errors [9]. TypeScript, however, dramatically alleviates this problem as discussed shortly.

The final two new data types for Java users are the *any* and the *void* data types. The first allows for TypeScript code to function identically to JavaScript when it comes to types, as the compiler will accept any type of value that is assigned to an *any* variable. The second data type is essentially the opposite of an *any* type, as the only types allowed for a *void* type are *null* or *undefined*. It is typically only utilized as a return value (or the absence of a return value) in functions [15][16].

Rethinking How Types Work

Unlike C or Java, which tend to prioritize the notion that each variable should only have a singular type, TypeScript likes to think of types as conjunctions [9]. For example, when taking an argument from the command line in C or Java, validation often must be performed to decipher the input's type. In TypeScript, although this validation can still be crucial, it can be lessened by acknowledging that a particular value could belong to multiple data types and still accomplish its purpose. This feature is created through a mechanism called a type union. A type union allows for a variable to be any one data type from a selection of different data types [9]. Here's an example:

```
var userInput: string | number = 12;   // The Union
console.log(userInput);                // Outputs: 12
userInput = "Hello World";
console.log(userInput);                // Outputs: "Hello World"
```

Not only do these unions work with primitive types, but they can also be used in objects and data-structures, such as arrays or maps. This allows for a developer to have additional flexibility for writing their programs and for the ability to create more type agnostic code. Although this lingers closer to JavaScript's paradigm of flexibility over accuracy, it still has the increased reliability of verifying that the values provided are of the types permitted. This makes type unions easy to code with, as the error checking only needs to compensate for the permitted types; any other differing type errors will be caught during development time [9].

Additionally, if a developer wanted to have a tighter constraint on what was allowed in their function, they could utilize a union with literals [9]. For example, perhaps a function's primary purpose is to utilize user confirmation, such as the string literals "y" or "n", to either continue or terminate the program. In this scenario, a developer could simply make the parameter to this function a union of the literal values: "y" | "n". This way, if they incorrectly validated the user's input previously, they will be warned upon compilation.

To help with creating unions from the myriad of different types in TypeScript, the developers created the ability to assign an alias for any given type. For example, the primitive type *string* could be given the alias *character,* if a developer wanted to increase readability for their program that only utilized strings of length one:

```
type character = string;
```

Besides the readability, this also helps writability, as it can be used on unions to reduce the amount of redundant code required [9].

Compiler

TypeScript is a compiled language, or more accurately, a transpiled language. This is because all written TypeScript is converted into JavaScript at compile time. Ever since its existence, TypeScript has had a close relationship to JavaScript and this similarity continues in its compiler settings. TypeScript is very personalizable to the individual needs of its developers and typically offers some sort of flexibility at the cost of accuracy. One such example is with the ability to turn *strictNullChecking* off. By default, *strictNullChecking* is on for the compiler, which means that it will warn the developer if a specific variable could possibly be *null* or *undefined* [17]. If this option is turned off, the language will effectively ignore the possibility that a variable could contain *null* or *undefined* values. This makes the language a lot more like C or Java, but could come at a significant cost if these variables aren't properly handled [17]. That being said, even with *strictNullChecking* on, it can still be a good idea to check for *null* and *undefined* variables, even though their ability to wreak havoc is significantly reduced.

Garbage Collection

Garbage collection is performed automatically in TypeScript by a background process called the garbage collector. This process continuously searches for unreachable code and removes it from memory once found. Unreachable code in TypeScript is any reference or data that can no longer be accessed by a variable, data-structure, object, etc. [18]. This phenomena can occur in a myriad of ways. One possible way is displayed below:

```
var person: {name: string} = {
    name: "John"
};
person = {
    name: "Julie"
};
```

In the previous example, all references to *"John"* become unreachable because *"person"* is reassigned to the name *"Julie"* and *"person"* was the last reference to *"John."* Once this occurs, the loose reference to *"John"* is quickly deleted by the garbage collector. It's important to note that unreachable code is completely contingent upon incoming connections. This means that if there was an object that references another object, but subsequently loses all references to itself, it would be counted as unreachable and subsequently deleted [18].

Structuring in a Program

Depending on what a developer wishes to accomplish, a TypeScript program can be remarkably simple. A main function or focal point of a program isn't required for compilation, so a variety of programs can be accomplished in a few lines of code with miniscule effort. For example, subprograms can be negated entirely and a single-line program could easily resemble the following:

```
console.log("Hello World");         // Outputs "Hello World"
```

That being said, subprograms are often encouraged in TypeScript and share two main commonalities. First, all subprograms utilize curly braces to designate scope, instead of the indentation common in other programming languages. Second, subprograms, as well as nearly every other design choice in TypeScript, should be designed to offer some form of structuring around variables and their types. The main subprogram structures of note are: functions, classes, interfaces and enums.

Functions

A function in TypeScript can be defined nearly identically to Python, except the *def* keyword is replaced with the *function* keyword. To create a "Hello World" program in TypeScript utilizing a *main* function, the following can be written:

```
function main() {
    console.log("Hello World");     // Outputs "Hello World"
}
main();                             // Don't forget to call main!
```

This trivial example doesn't utilize what we've learned about types, so a more useful example would be the following, which verifies that the provided parameter is of the *number* type and the return variable is also a *number* type:

```
function main() {
    console.log(incrementor(2));         // Outputs 3, a number type
}
function incrementor(providedNumber: number): number {
    return providedNumber += 1;
}
main();
```

Besides these helpful tools, TypeScript supports a type of function called an anonymous function. These are functions whose parameters and return values aren't given an explicit type, yet whose types are inferred [9]. This is accomplished through a phenomena referred to as contextual typing. Essentially, the type for the parameters and returns are inferred based on how the function is utilized in the program [9]. As the reader might recall, this is the direct result of inferred types deciphering each variable's type by examining the values utilized throughout the program. As per the purpose of TypeScript, this helps to make the program easier to debug and more scalable.

Classes

Classes in TypeScript possess extraordinarily similar qualities to Java, such as the inclusion of constructors, getters, setters, class heritage, member visibility and generics. A feature that readers might not be aware of is a class expression. In TypeScript, a class expression is essentially the same as a declaration of a class, but it doesn't require a name for the class and instead utilizes a variable [19]. Here's an example:

```
class example {                         // Normal class declaration
}
const example = class<Type> {          // Class expression
}
```

Depending on what a developer wishes to accomplish, a class expression can greatly assist with the writability of a program, because a constructor is entirely optional [20].

Interfaces

In addition to classes, TypeScript has the functionality to implement interfaces. Interfaces in TypeScript are also significantly similar to their use in Java, but have the additional function of describing typed information for the object constructs of JavaScript [21]. Here's an example using the interface *muffins* on a JavaScript object:

```
interface Muffins {
    quantity: number;
    color: string;
}
let muffin: Muffins = {
    quantity: 5,
    color: "blue"
}
```

In the example, the variable *muffin* is an implementation of the interface *muffins* and utilizes the typing specified in *muffins* to guide the type of its own values. This means that the *quantity* and the *color* variables can only possess values of type *number* and of type *string* respectively.

Enums

Enums are defined as a set of named constants that typically contain numeric values or string values [21]. Essentially, the purpose of an enum in TypeScript is to contain all of the constants that correspond to a specific type. For a demonstration on how they work, see the following numeric enum:

```
enum Numbers {
        One = 10,
        Two,
        Three,
}
```

In this example, there are three constants, *One, Two* and *Three.* Notice how only the first enum is actually given a constant value; this is because the remaining values will be auto-incremented based on the starting value provided. In addition to this, note how each of the constants have one capitalized letter and the rest are lowercase. This is the standard style when writing enum constants in TypeScript [21]. Enums can also be written with strings, although each constant must be provided a value, as they lose the auto-incrementing behavior of numeric enums. It should be briefly noted that enums can technically be written with a mixture of strings and numbers for constants, but this is considered to be a poor use of an enum [21]. In practice, enums are usually combined with interfaces to guarantee that one or more of an interface's values match the given enum value(s).

Abstract Data Types

TypeScript, because of its compatibility with modules and user defined types, can possess nearly every abstract data type present in other languages, but there are two main types that are provided inherently: the array and the tuple.

Arrays

Arrays in TypeScript function very similarly to C, C++ or Java. This is because they are static in their allocation, so once they are given a specific size, they cannot deviate from that size [22]. However, there is a way to use TypeScript's array object constructor to create a heap dynamic array, much like a *list* in Python. This type of array can change its length without any additional developer work and can be declared through either of the two following ways:

```
var arrayOfNumbers: number[] = new Array(0);
var arrayOfNumbers: Array<number> = new Array(0);
```

In TypeScript, the first declaration is actually just a short-hand for the latter, so either syntax is correct and up to developer preference. The latter form will be used for the remaining examples, such as the different basic operations that can be accomplished with an array:

```
var arrayOfNumbers: Array<number> = new Array(0);
arrayOfNumbers[0] = 1;       // Assigns 1 to the first index
arrayOfNumbers.push(2);      // Adds 2 as the second element
```

Note how an array of size zero actually has enough space for a singular element and the *push* function allocates more space for the array and appends *2* to the end of the array. Compared to C or C++, which would require the developer to deallocate the memory for this type of array, TypeScript will take care of this deallocation automatically through the garbage collector. Besides the *push* function, TypeScript's arrays come with other useful functions such as *pop, concat, slice, sort, shift, unshift* and *reverse*. It should be noted that if an array is allocated without a data-type, then it is automatically assumed to be of type *any* [22]. This can actually be advantageous when working with arrays, as it allows for operations that wouldn’t normally be available when working strictly with a singular designated type.

Tuples

A tuple is essentially just a typed array that knows the exact type for each individual element in the array [23]. They are best understood via example, so here’s a simple demonstration:

```
var tuple: [number, boolean, string];
tuple = [1, true, “Hello World”];
```

As the reader can see, each individual value of the array holds a designated type and any attempt to deviate from this type will result in an error.

```
var tuple: [number, boolean, string];
tuple = [true, true, “Hello World”];  // boolean not of type number
```

Much like an array in TypeScript, a tuple is accessed via subscripting and is completely mutable [24]. This means that there are some useful operations that can be performed with tuples, such as the following:

```
tuple.push(true);            // Adds “true” to the tuple
tuple.pop();                 // Removes “true” from the tuple
```

At this point, the tuple will allow for a new value and correctly decipher the type, but surprisingly won’t allow for the new values to be accessed in any way. In our previous example, we have a tuple with 3 elements. If another element was added to the tuple and accessed via subscripting its index, the tuple will throw an error stating that it is out of bounds, even though it will show the new value when printed out to the console. This may initially seem like a mistake, but this is an intentional design so that tuples don’t become conflated with arrays [24]. Another interesting functionality that tuples possess is deconstruction. This phenomena helps developer writability and is best understood through example:

```
var tuple: [string, number, boolean]
tuple = [“Hello World”, 10, false]
var [index0, index1, index2] = tuple;
```

In the example, *index0* is assigned to *“Hello World”, index1* is assigned to *10* and *index2* is assigned *false*. Depending on how many variables need to be assigned, this can be an incredibly useful alternative to several, space consuming variable assignments.

Flow of Control

Conditional Branching

Control statements in TypeScript include *if, if else, else* and *switch*. To anyone with experience in the programming languages of C or Java, the syntax and semantic meaning of these statements will be instantly recognizable. Each of these statements behaves identically to the aforementioned languages and features the same syntax for the basic comparison operators, such as ==*, !=* and *&&* [25]. A new concept introduced in TypeScript is the conditional type. This was briefly mentioned earlier in the form of a union for primitive types, but conditional types are fundamental to providing different outputs based upon differing inputs [26]. Essentially, a conditional type is a ternary operator, but instead of values being chosen, types are chosen. Here's the main structure of this type:

```
<Type> extends <Another Type> ? <Type if true> : <Type if false>;
```

To help better understand this, consider an example where there exists four interfaces: *Mammal, Reptile, Cat* and *Tortoise.* The *Cat* interface *extends Mammal* and the *Tortoise* interface *extends Reptile*. With this knowledge in mind, evaluate the following conditional types:

```
type example1 = Cat extends Mammal ? string : number;
type example2 = Tortoise extends Mammal ? string : number;
```

Because *Cat extends Mammal*, the type of *example1* evaluates to *string*. However, because *Tortoise* does not extend *Mammal,* the type of example2 is assigned *number.* Now, after understanding how this works, a reader might consider this to be a neat, but ultimately useless addon. This is not the case, because this feature is incredibly useful when combined with generic types and significantly reduces redundant code [26]. Consider a function that must perform different tasks based upon the variable type provided. To solve this issue, a developer could use a union to only allow for the appropriate types and then check the type of each argument inside of the function through a collection of *if, if else* and *else* statements to provide the appropriate output. The following function declaration would be an example of this implementation, without the required conditional logic:

```
function createDate(date: number | string): numericDate |
    stringDate;
```

This option will work, but is not nearly as elegant or space optimized as the next solution. Instead, a developer could create multiple overloaded functions to deal with each particular type.

```
function createDate(date: number): numericDate;
function createDate(date: string): stringDate;
```

This alternative is also viable, but it's still not the best solution in TypeScript. The best solution would be to utilize a conditional type with a generic type, *T*, such as the following:

```
type NumberOrString<T extends number | string> = T extends number ?
```

```
        numericDate : stringDate;
```

This is a lot to unpack, so I'll go over each part. As we learned earlier with aliases, the *NumberOrString* variable will be an alias for one of the following types: *numericDate* or *stringDate*. However, it's also creating the generic type, *T*, that can be either a *number* or a *string*. Now, consider the following implementation of the function:

```
function createDate<T extends number | string>(date: T):
NumberOrString<T>
{
    throw "Incorrect type provided!";
}
```

In this version, all of the function overloads are taken care of by this single function. Basically, by utilizing the previously created conditional type, the return type of the function will be exactly what we'd like it to be without having to do the unnecessary work of declaring more overloaded functions [26].

Iterative Statements

Much like conditional branching, most of the looping techniques of TypeScript will be recognizable by developers familiar with C, Java or Python. It features a *for, do* and *while* loop. The standard *for* loop functions nearly identically to that of Java, except that *i* is initialized with *var* or *let* instead of an *int* or other data type. The *do* and *while* loops, however, are identical to that of Java [27]. That being said, there are some differences when it comes to more enhanced loops. Consider the following:

```
var cities: Array<string> = ["Portland", "Newberg", "Tigard"];
for (let i in cities) {
    console.log(i);
}
```

Those familiar with Python will see the following loop and find it nearly identical to an enhanced for loop. However, to assume that this loop functions identically to Python's would be a mistake. The variable *i* does not become each element of the array, rather, it is simply a shorthand for creating a standard for loop. This means that the loop will iterate until it reaches the end of the array and *i* is an index, rather than an element. In order to access each element, the value inside of *console.log* must be changed to the following:

```
console.log(cities[i]);
```

In order to use an actual enhanced *for* loop in TypeScript, the following code can be written:

```
for (let i of cities)
```

Note the inclusion of the *of* keyword, instead of the *in* keyword. This means that *i* is an element contained inside of *cities*, instead of the index.

Personal Experience

I used Microsoft VSCode (Visual Studio Code) for my IDE (Integrated Development Environment) when I programmed in TypeScript. There are other IDEs, but because VSCode and TypeScript were both created by Microsoft, VSCode has the best built in compatibility. This made procuring niceties such as syntax highlighting and debugger support a trivial task. However, because I had never programmed JavaScript in my VSCode environment, I needed some dependencies before I could program in TypeScript. The first of these was NodeJS itself. NodeJS is needed in order to run the transpiled JavaScript code that is created when TypeScript is compiled. The second was NPM, the Node Package Manager. This would allow me to import any libraries I would need and download the TypeScript compiler itself. Links to all of these resources can be found in Appendix D. After acquiring these dependencies, I was able to create a simple *HelloWorld.ts* file, compile and run the program.

Feeling confident, I moved onto programming the factorial and the person programs. It quickly became apparent that there were still some difficulties that had to be navigated. Chief among these was the lack of built in features to grasp command line arguments and to read from files. This led me to research libraries that could fulfill these requirements and becoming dismayed when the required libraries wouldn't import. After a significant period of time trying to solve this issue, I eventually discovered that I needed to install the libraries in the same directory that contained my TypeScript source code. This fixed my issue and I was able to both receive command line arguments and read files.

One final issue I experienced was that in the particular library I used to read files, extraneous data, such as carriage returns and newlines, were transferred into my program instead of automatically deleted. To fix this problem, I used the power of global regex to replace these values with empty spaces.

Once all of the issues were resolved, it became a joy to code in TypeScript. I've found through my journey that TypeScript's error statements are incredibly useful compared to the cryptic messages of other languages. Errors provide the line number, the exact number of characters into the line, a copy of the code contained within that line and the reason why the parser sent an error. As a result of explicit static typing, I've also found that the source code is extremely readable, even after not viewing the code for several weeks. There are countless other benefits, but these features work well together to provide a fantastic debugging experience where problems can be solved nearly instantaneously.

Programming Language Assessment

Although I have not personally used TypeScript to create a massively complex web-application, I believe that it has successfully accomplished this goal. After programming in both JavaScript and TypeScript, I find that TypeScript has significantly superior code readability and reliability. As mentioned above, the increased readability is a direct result of the explicit static types, but these types also help with program reliability, because a developer can be assured that a variable will only hold a particular type throughout the entire program.

In addition to readability, TypeScript possesses extremely high writability, both in its approachability for new programmers and in its power for more experienced programmers. To begin, TypeScript's syntax, combined with its automatic garbage collection and memory management, makes it a fantastic first programming language for new programmers. In addition to this, many of its built-in functions are syntactically similar to English, such as the functions *in* or *includes* to check for values in strings.

For more experienced developers, TypeScript possesses regex compatibility, recursive conditional types, threading and countless abstract data-types. But these features, although incredibly useful to possess, are available in many other programming languages. TypeScript's true power lies in its guarantee that simple, untracked typing bugs will not proliferate an application.

If someone desires to create a complex web-application, they should look no further than the highly sustainable and easy to use programming language of TypeScript. In fact, TypeScript is a fantastic choice for any web-application and comes with the added benefit of not having to rewrite a single line of code if a user changes their mind and switches to JavaScript. The disadvantages are non-existent and the advantages are astronomical, so the next time you need to create a web-application, give TypeScript a chance and you won't be disappointed.

Bibliography

[0] M. Foley. "Who Built Microsoft TypeScript and Why". ZDNET. https://www.zdnet.com/article/who-built-microsoft-typescript-and-why/. (accessed Oct. 2, 2022).

[1] T. Warren. "Microsoft: We Were Wrong About Open Source". TheVerge. https://www.theverge.com/2020/5/18/21262103/microsoft-open-source-linux-history-wrong-statement. (accessed Oct. 2, 2022).

[2] D. Rossenwasser. "Announcing TypeScript 2.0". Microsoft. https://devblogs.microsoft.com/typescript/announcing-typescript-2-0/. (accessed Oct. 2, 2022).

[3] D. Rossenwasser. "Announcing TypeScript 3.0". Microsoft. https://devblogs.microsoft.com/typescript/announcing-typescript-3-0/. (accessed Oct. 2, 2022).

[4] D. Rossenwasser. "Announcing TypeScript 4.0". Microsoft. https://devblogs.microsoft.com/typescript/announcing-typescript-4-0/. (accessed Oct. 2, 2022).

[5] D. Rossenwasser. "Announcing TypeScript 4.9 Beta". Microsoft. https://devblogs.microsoft.com/typescript/announcing-typescript-4-9-beta/. (accessed Oct. 2, 2022).

[6] "TypeScript Documentation: Why Create TypeScript". Microsoft. https://www.typescriptlang.org/why-create-typescript. (accessed Oct. 3, 2022).

[7] "Difference Between Var and Let". W3schools. https://www.w3schools.blog/var-vs-let-difference-typescript-javascript. (accessed Oct. 3, 2022).

[8] “TypeScript Documentation: Type Inference”. Microsoft. https://www.typescriptlang.org/docs/handbook/type-inference.html. (accessed Oct. 3, 2022).

[9] “TypeScript Documentation: Every Day Types”. Microsoft. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html. (accessed Oct. 3, 2022)

[10] C. Deshpande. “TypeScript Vs. JavaScript: Which One Is Better?”. Simplilearn. https://www.simplilearn.com/tutorials/typescript-tutorial/typescript-vs-javascript. (accessed Oct. 3, 2022).

[11] “TypeScript Data Types". TutorialsTeacher. https://www.tutorialsteacher.com/typescript/typescript-number. (accessed Oct. 6, 2022).

[12] “Max Safe Integer: JavaScript”. Mozilla Corporation. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER. (accessed Oct. 8, 2022).

[13] “TypeScript Documentation: TypeScript 3.2”. Microsoft. https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-2.html#bigint. (accessed Oct. 8, 2022).

[14] “TypeScript Documentation: Symbols”. Microsoft. https://www.typescriptlang.org/docs/handbook/symbols.html. (accessed Oct. 10, 2022).

[15] R. Enoch. “An Overview of Programming Language: TypeScript”. CSSDeck. https://cssdeck.com/blog/an-overview-of-programming-language-typescript/. (accessed Oct. 10. 2022).

[16] “Primitive Data Types in TypeScript”. Code Topology. https://codetopology.com/scripts/typescript/primitive-data-types-in-typescript/. (accessed Oct. 10. 2022).

[17] “TypeScript Documentation: Compiler Options“. Microsoft. https://www.typescriptlang.org/tsconfig#strictNullChecks. (accessed Oct. 10. 2022).

[18] “Garbage Collection”. JavaScript Info. https://javascript.info/garbage-collection. (accessed Oct. 10. 2022).

[19] “TypeScript Documentation: Classes”. Microsoft. https://www.typescriptlang.org/docs/handbook/2/classes.html#class-expressions. (accessed Oct. 12. 2022).

[20] “Class expression - JavaScript”. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/class. (accessed Oct. 14. 2022).

[21] R. Cavanaugh. “Walkthrough: Interfaces”. Microsoft. https://devblogs.microsoft.com/typescript/walkthrough-interfaces/. (accessed Oct. 16. 2022).

[21] “TypeScript Documentation: Enums”. Microsoft. https://www.typescriptlang.org/docs/handbook/enums.html. (accessed Oct. 18th, 2022).

[22] “TypeScript - Arrays”. TutorialsPoint. https://www.tutorialspoint.com/typescript/typescript_arrays.htm. (accessed Oct. 18th, 2022).

[23] “TypeScript Tuples” W3schools. https://www.w3schools.com/typescript/typescript_tuples.php. (accessed Oct. 18th, 2022).

[24] “TypeScript - Tuples”. TutorialsPoint. https://www.tutorialspoint.com/typescript/typescript_tuples.htm. (accessed Oct. 19th, 2022).

[25] “TypeScript if else”. TutorialsTeacher. https://www.tutorialsteacher.com/typescript/typescript-if-else. (accessed Oct. 19th, 2022).

[26] “TypeScript Documentation: Conditional Types”. Microsoft. https://www.typescriptlang.org/docs/handbook/2/conditional-types.html. (accessed Oct. 19th, 2022).

[27] “TypeScript - Loops” TutorialsPoint. https://www.tutorialspoint.com/typescript/typescript_loops.htm. (accessed Oct. 20th, 2022).

Appendix A: Factorial

```
/**
 * This function calculates the factorial of the provided integer via command
 * line arguments
 */
function main(): void
{
    var providedNumber: number;

    providedNumber = parseInt(process.argv[2]);
    console.log("Factorial of " + providedNumber + " is: " +
         factorial(providedNumber));
}

/**
 * This function calculates the factorial of the provided number
 *
 * @param providedNumber An integer that was passed into the program via
 * the command line
 * @returns The factorial of the provided number
 */
function factorial(providedNumber: number): number
{
    if (providedNumber == 1 || providedNumber == 0)
    {
        return 1;
    }

    else
    {
        return providedNumber * factorial(providedNumber - 1);
    }
}
main();
```

Appendix B: Structured Information Sorter

```
import * as fs from 'fs';

/**
 * This program reads from a file and creates Person objects from that
 * file, sorts the Persons based upon their names and prints out the
 * average age
 */
function main(): void
{
    // This variable will hold the string containing the entire written file
    var fileContent: string;

    // This variable will hold an array of strings split on new lines
    var split: Array<string>;

    // This variable will hold an array of Person objects
    var housing: Array<Person>;

    // Reads the entire file and puts it into a string
    fileContent = fs.readFileSync("Person.dat.txt", 'utf8');

    // Splits that string into an array separated on newLines
    split = fileContent.split("\n");
    housing = [];

    addNewPersons(housing, split);

    // Sorts the array by name in ascending order
    housing = housing.sort((a, b) => (a.name < b.name ? -1 : 1));

    calcAverageAndPrint(housing);
}


/**
 * This is a class created for a Person. Each person has a name and an age
 */
class Person
{
    readonly name: string;
    readonly age: number;

    // Person Constructor
    constructor(name: string, age: number)
    {
        this.name = name;
        this.age = age;
    }
}
```

```
/**
 * Calculates the average age of all the persons and prints out their names
 * in sorted order and the total average
 *
 * @param housing An array containing the created Persons
 */
function calcAverageAndPrint(housing: Array<Person>): void
{
    var average: number;
    average = 0;

    // Iterates over each Person
    for (var person of housing)
    {
        // Prints out the name of the Person, since we are in sorted order
        console.log(person.name)

        // Adds the Person's age to calculate the average age
        average += person.age;
    }

    // Calculates the average age
    average /= housing.length;

    // Prints the average again
    console.log("The average age is:", average);
}

/**
 * This function iterates through the provided array and creates
 * new Persons from the names and ages inside the file
 *
 * @param housing An array containing the created Persons
 * @param split An array containing the entire read file separated by newLines
 */
function addNewPersons(housing: Array<Person>, split: Array<any>): void
{
    // This variable will be used to get a Person's name from the file
    var personName: string;

    // This variable will be used to get a Person's age from the file
    var personAge: string;

    // This variable will be used to denote whether the file is currently
    // on a name or an age. If this value is true, then it is on a name, else
    // it is at an age.
    var isName: boolean;

    // Set to blank
    personName = "";
    personAge = "";
    isName = true;

    for (var i in split)
    {
        for (var j in split[i])
        {
            // Checks that the current character isn't a comma, isn't a space
            // and is currently supposed to be a name
            if (split[i][j] != "," && split[i][j] != " " && isName)
            {
                personName = personName.concat(split[i][j].toString());
            }

            // Checks that the current character isn't a comma, isn't a space
            // and is currently supposed to be an age
            else if (split[i][j] != "," && split[i][j] != " " && isName == false)
            {
                personAge = personAge.concat(split[i][j].toString())
            }

            // If its a comma, then we switch from name to age or vice versa
            else if (split[i][j] == ",")
            {
                if (isName)
                {
                    isName = false;
                }
                else
                {
                    isName = true;
                }
            }
        }

        // Create a new Person
        if (personName != "")
        {
            housing.push(new Person(personName, parseInt(personAge)));
        }

        // Reset our variables to be used again
        personName = "";
        personAge = "";
        isName = true;
    }
}

main()
```

Appendix C: Random Writer

```
import * as fs from 'fs';

/**
 * This program creates a dictionary from the provided grammar file
 * and creates a sentence from the given terminals and non_terminals
 */
function main(): void
{
    // Create a new array to hold the dictionary values
    let dictionary: Array<any>[2];

    dictionary = create_dictionary()
    produceText(dictionary[0], dictionary[1])
}

/**
 * This function iterates over the provided grammar file and creates a
 * dictionary linking every non-terminal to its designated terminals
 */
function create_dictionary(): any
{
    // A check to see if we should avoid adding to another non_terminal
    var skipoverNonTerminal: boolean;

    // A boolean to see if we should currently be adding to the dictionary
    var bracketSensor: boolean;

    // A variable to hold the current non_terminal being examined
    var currentNonTerminal: string;

    // A string that will be used to concat the other delimited strings
    // into the proper form
    var concatWords: string;

    // A variable used to hold the first non_terminal in the file,
    // most likely called something similar to <start>
    var initialNonTerminal: string;

    // This variable will be used to
    var fileContent: string;

    // A map that creates a link between each non_terminal and its
    // corresponding terminals
    var nonTerminalMap: Map<string, Array<string>>;

    // This variable will be used to hold each word in the parsed file,
    // delimited by a space
    var words: Array<string>;
```

```
// This variable will hold the values we need to return from this function
var returnArray: Array<any>;


// Sets initial values used for parsing
skipoverNonTerminal = false;
bracketSensor = false;
currentNonTerminal = '';
concatWords = '';
initialNonTerminal = '';

returnArray = new Array();

// Initializes map to be used for creating the links between non terminals
// and terminals
nonTerminalMap = new Map();

// Reads the file into a single string
fileContent = fs.readFileSync(process.argv[2], "utf-8");

// Removes all of the metadata from the string, so that we have just what
// we need and so that everything is properly sorted for the parser
fileContent = fileContent.replace(/\r\n/g, " ").replace(/\t/, "").replace(
    /  /g, "").replace(/}{/g, "} {").replace(
    /([;])/g, "; ").replace(/  /g, " ").replace(/>;/, "> ;");

words = fileContent.split(" ");

// Creates an array of strings delimited by spaces
for (var word of words)
{
    // A simple check to see that we need to start adding to the dictionary
    // again, as everything else is just comments
    if (word == "{")
    {
        bracketSensor = true;
    }

    // A check to see if we need to stop adding to the dictionary, as
    // comments can be added again
    else if (word == "}")
    {
        currentNonTerminal = "";
        bracketSensor = false;
        skipoverNonTerminal = false;
    }

    // If we are in a bracket, then we check if we are looking at a
    // non_terminal.
    else if (bracketSensor == true)
    {
        //If we are, then we can add that as our current
        // non_terminal.
        if (word.includes("<") && word.includes(">") &&
            skipoverNonTerminal == false)
        {
            currentNonTerminal = word;
            skipoverNonTerminal = true;

            // Additionally, if we don't have any non_terminals
            // yet, then this non_terminal is set to our first non_terminal
            if (initialNonTerminal == '')
            {
                initialNonTerminal = word;
            }
        }

        else
        {
            // Basically either adds starts the string of terminals and
            // non_terminals or adds to the string
            if (word != ";")
            {
                if (concatWords == "")
                {
                    concatWords = word;
                }
                else
                {
                    concatWords = concatWords + " " + word;
                }
            }

            // If we are dealing with a semicolon, then we add the string
            // of terminals and non_terminals to the dictionary
            // under the appropriate non_terminal
            if (word == ";")
            {
                if (nonTerminalMap.has(currentNonTerminal))
                {
                    nonTerminalMap.get(
                        currentNonTerminal)?.push(concatWords);
                }
                else
                {
                    nonTerminalMap.set(currentNonTerminal, [concatWords]);
                }
                concatWords = "";
            }
        }
    }
}

// Add the map and the starting terminal to our return variable
```

```
    returnArray.push(nonTerminalMap, initialNonTerminal);

    return returnArray;
}

/**
 * This function iterates through the provided non_terminals and prints
 * out the terminals it encounters or calls another function to recursively
 * call the terminals from the nonTerminal it encountered
 *
 * @param nonTerminalMap This is a dictionary that links every nonTerminal
 * to its designated terminal
 * @param initialNonTerminal This is the first nonTerminal that was
 * encountered in the grammar file
 */
function produceText(nonTerminalMap: Map<string, Array<string>>,
    initialNonTerminal: string): void
{
    // Contains the first non terminal
    var startNonTerminal: Array<string>;

    // Gets the first non terminal
    startNonTerminal = nonTerminalMap.get(initialNonTerminal)!

    for (var i of startNonTerminal)
    {
        for (var j of i.split(" "))

            // Checks if j is a terminal
            if (!j.includes("<") && !j.includes(">"))
            {
                process.stdout.write(j);
            }

            // Since it isn't, we need to dive deeper to find the terminals
            else
            {
                generator_helper(nonTerminalMap, j);
            }
    }
}

/**
 * This is a recursive function that solves each non-terminal before
 * moving onto the next
 *
 * @param nonTerminalMap The dictionary containing all of the non-terminals and
 * their corresponding terminals
 * @param nonTerminal The non-terminal symbol being passed into the function
 */
function generator_helper(nonTerminalMap: Map<string, Array<string>>,
    nonTerminal: string)
```

```
{
    // Compiler notes that it could be null, but it will never be null if given
    // a correct file
    var innerArray: Array<string> = nonTerminalMap.get(nonTerminal)!

    // Gets a random value from the array
    var nextValue: string = innerArray[Math.floor(Math.random() *
        innerArray.length)];

    // An array of strings that will be iterated over in pursuit of terminals
    // and non terminals
    var nestedInnerArray: Array<string> = nextValue.split(" ");

    for (var i of nestedInnerArray)
    {
        // If it isn't a non terminal, print it
        if (!i.includes("<"))
        {
            process.stdout.write(i);
            process.stdout.write(" ");
        }

        // Recurse till we find terminals
        else
        {
            generator_helper(nonTerminalMap, i)
        }
    }
}

main();
```

Appendix D: Resources Utilized

TypeScript Compiler
https://code.visualstudio.com/docs/typescript/typescript-compiling

NodeJS
https://nodejs.org/en/download/

Node Package Manager
https://www.npmjs.com/package/download

Microsoft Visual Studio Code
https://code.visualstudio.com/