Features:

* Type script is a strongly typed language.
* Typescript is then converted to the normal JavaScript
* Typescript can be used in the client-side application as well as the server-side applications

Typescript Project Files:

* The typescript project file is an simple json text file (**tsconfig.json**)which contains the information about the typescript.
* We can store the different compiler options when we run the compiler.
* Specify files which should be included & excluded in the file
* We can also create sub project files inside an project. (**Configuration Inheritance**)

About Ts Config:

* compilerOptions is an key for the typescript project.
* Target is theversion for the javascript
* Nounusedlocals will generate Error, if the variables present in the code are not used
* outFile is name of the javascript file the program will output
* The files key is used to include all the files which are required for the compilation
* If there is no tsconfig in the directory then it will go up the directory to find the **tsconfig.json** file

Compiler Options:

* Source Map :
  + The source map is used to debug the javascript
* Output Dir:
  + The output directory is where the javascript files will be stored
* Strict:
  + It enables all the strict checking options. It checks everything in the code.

* Watch:
  + The watch option is used to check all the different changes that have occurred an it will take place

Inherited Config:

* We can have inherited config files which can inherit the compiler options from the parent file and then we can just have the files present inside the config files.
* In the inherited config, we have to use the **extends** key to inherit the parent compiler options to the child options

Webpack Config :

* This config is only affected, when we bundle the whole code
* Entry:
  + The entry point is the beginning of the application,
  + entry:’source file’
* devtool:
  + The devtool is handling the source map files
  + devtool:’inline-source-map’. This makes the source map files inline within the bundle.
* resolve:
  + The resolve property is used to specify the extensions for the files.
* output:
  + The output will be the bundle the webpack will produce in the end.
* devServer:
  + The dev server is handling the server with the provided options inside it.
  + The **inline** option is used to stop the browser from refreshing whenever there is an new bundle

Within the webpack, we have many plugins and one of the most used plugin is ts-loader

We can add the ts-loader as an dependency to the project.

Below is an example of the adding webpack plugin to the project

* First, add the dependency to the project
* Second, configure webpack config file to handle the loader added as an dependency.

Example:

* module:*{* rules: *[  
   {* test: /\.tsx?$/, //This is reg exp for all .ts and .tsx files in the folder  
   use: 'ts-loader', // Use the loader which was added as an dependency   
   exclude: 'node\_modules' //Exclude the ts & tsx files in the node\_modules folder  
   *}  
   ]  
  }*,

The output provided by the loader will then be put on to the **output bundle file.**

As, we will be using the bundle file, instead of the normal js file. We can delete the JS files present in the JS folder.

We also have to update the script file in the html file to map it to the bundle file instead of the js file.

The **bundle.js**  file will not be created locally but it will be created inside the memory

Datatypes:

* array,
* Number
* Boolean
* String
* In javascript, all of the variable declarations will be moved to the top of the block code during the execution. So, we can assign the variables anywhere in the code.
* In the typescript, we cannot do the above thing as the typescript compiler will generate an error.

We can create two variables using two things and they are **let & const.**

let:

* The let is used when we know the value will be changed during the code.

const:

* The const will be used when the variable is constant throughout the whole process.

**Annotations and Inference:**

let x: **string** = “Hello World”;

Here, the string is the annotation for the program.

Method Datatypes:

**Void:**

**Null:**

**Undefined**

**Never**

**Union Types:**

**The union types will contain a combination of the two datatypes which are number and string.**

**We can combine as many types as we want**

**StrictNullChecks:**

* **If we want to assign null to a data type then we have to add as an annotation to the union type, then only then we can assign null to it.**
* **If we want to assign undefined to the variable, we have to add it as annotation to the union type.**

**Type Assertions:**

There are two ways in which we can assert the variables and they are the following:

Suppose, we have value but we do not know the datatype of the variable then we can use the **any** datatype for the variable

Ex: let value:any = 5;

1. <number>value:

* let fixedString: string = (<number>value).toFixed(4);
* console.log(fixedString) //5.0000

1. let fixedString: string = (value as number ).toFixed(4);

**Control Flow Based Type Analysis:**

The control flow based type analysis helps us to handle the annotations very easily in the project.

**Annotations Functions:**

In the annotation function, we will be providing annotations even to the function which will be very helpful. We can know the return value directly at the beginning of the function.

**Function Parameters:**

In the JavaScript, every function parameter is optional inside the function but in the typescript, if an parameter is optional then we have to use the **?** after the **parameter to make it as optional.**

Function():string{}

* We can also use the noImplcitAny compiler option, which will generate errors in the code.

Default Initialized Parameters

* The parameters will have some initial values as the parameters in function which will not lead to issues such as the null issue and undefined issue.

Arrow Functions:

The arrow functions are very similar to the normal functions but they use arrow instead of the normal functions

let scores: number[]= [70,125,85,110]

let highScores: number[];

highScores = scores.filter((element, index, array) =>{

if(element>100){

return true;

}

});

Typescript Article Part 1 :

This is the first part of a series of articles I’ll be posting on the fundamentals of TypeScript. If you’re new to TypeScript and you’re not sure where to get started, then you’ve landed on the right page.

Bring your JavaScript knowledge and prepare to be enlightened and well-equipped to learn more on your own.

We’re going to take it slow and start off by discussing the very basics of TypeScript. In Part Two of this series, we’ll set up VS Code and our coding environment to work with TypeScript and apply some of the knowledge in this article. We’ll also discuss some additional basic types.

We’re not really going to be writing any code in this article outside of some examples, but we will cover the basic types and prepare you for the next lesson where we *will* write some code.

When we discuss what TypeScript is and how it works, I will approach this from a beginner’s perspective on a need-to-know basis. In other words, I’m not going to overload you with information.

One of the most difficult things about learning a new language is understanding what’s relevant and how to apply it in a useful manner. Because of this, we’ll only cover introductory concepts to get you started on your TypeScript journey. I’ll cover more advanced topics in Parts Three and Four.

Let’s get started.

**Pre-Flight Checklist**

As I said above, we won’t be writing any serious code in this article, but if you’d like to follow along, you can install VS Code if you haven’t already. We’ll be using it in the upcoming lessons in this series, and basic TypeScript functionality is supported out of the box in VS Code.

To install VS Code on your computer, head over to [Microsoft’s official download page](https://code.visualstudio.com/) and install the **stable build** for your operating system.

It’s necessary for you to have some basic knowledge of JavaScript programming to understand TypeScript. I won’t be digging into the details of how JavaScript works or explaining the intricacies of how things like variables, functions, or classes work, except with respect to how they work in TypeScript.

I’ll cover the key concepts in TypeScript and how they apply specifically to someone *just starting to learn the language.* There’s a lot more than what I can cover in a few articles, though, so I encourage you to [read the docs](https://typescriptlang.org).

**What is TypeScript?**

This is a loaded question. Just Google it and you’ll see a lot of copy-and-paste answers, most very (suspiciously) close to [this answer from StackOverflow](https://stackoverflow.com/questions/12694530/what-is-typescript-and-why-would-i-use-it-in-place-of-javascript):

“TypeScript is a superset of JavaScript which primarily provides optional static typing, classes and interfaces. One of the big benefits is to enable IDEs to provide a richer environment for spotting common errors as you type the code.”  
— Paul Dixon on Stack Overflow and just about every tutorial on TypeScript

Right, okay, that’s great and all. But the real question from an existing JavaScript developer’s perspective is, *“Why should I care?”* Alsomaybe, *“How do I use it to improve my JavaScript skills?”* This answer doesn’t really address those concerns unless you already know something about TypeScript and what **static typing** means.

I personally found a lot of the answers I ran into when I first began learning TypeScript to be more confusing and off-putting. They seemed like they were meant to make the user answering sound like they knew more than they really did. That isn’t helpful to a beginner in any subject, much less a programming language.

I’m not going to wrap this in a neat little bow and try to shove a buzzword driven definition down your throat because it’s useless to learning. Instead, I’ll do my best to make this more approachable.

**Why Do We Care About Types?**

Let’s first talk about what types are in JavaScript. A type really means a ***data type*** *—*also referred to as a ***primitive data type*** *—*whose value describes the type of data that something else contains. I’ll break this down in a few definitions in simple terms:

* Primitive — Data that is not an object and has no methods. The word *primitive* refers to the fact that these are the most low-level fundamentals of data; like primitive humans, they are the simplest form.
* Data — The value of a variable, object, or method (this is intentionally generic). This can be almost anything. We all know what data is — let’s not complicate this.
* Type — A primitive data descriptor representing the kind of a piece of data. This refers to when we describe what a piece of data actually *is* — a string, number, etc. More on this in a bit.

Types exist within JavaScript, but it’s fairly rare that you might actually feel like you’re interacting with them, unless you’re checking the typeof something, such as verifying it’s a number and not a string. Most of what happens with types is in the background.

Allow me to illustrate this with some code.

Code:

|  |
| --- |
| let myNumber = 10; |
|  |

|  |
| --- |
| let myString = 'this is a string'; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| if (typeof myNumber === 'string') { |
|  |

|  |
| --- |
| console.log('This is a string, not a number.') |
|  |

|  |
| --- |
| } else { |
|  |

|  |
| --- |
| console.log('This is a number'); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| const badMath = (x, y) => { |
|  |

|  |
| --- |
| return x \* y; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
|  |
|  |

badMath(myNumber, myString) // Output is NaN because myString isn't a number.

As we can see in the code above, when we check the typeof value of a variable whose data is a string, our if else statement will log “This is a number.” This is because the value of myNumber is, indeed, a number.

But we don’t usually do this in JavaScript, do we? We don’t write unit tests for every line of code; it’s inefficient and unrealistic. There must be a better way.

We’ve introduced a possible bug that may have failed silently with JavaScript (our code would run, but it wouldn’t work as expected). In JavaScript, we may not have detected this error. NaN is a *valid data value*, even though it’s not what we want.

Although our code isn’t giving us a useful value, the fact is in most cases it wouldn’t even cause our app to failand we’d be none the wiser.

This is where TypeScript comes in.

Had we written badMath(myNumber, myString) when using TypeScript, our code editor would have told us it was going to be faulty and that myString is not a number.

This is the first, and arguably the most important thing, you should understand about the value of TypeScript: with TypeScript, we can see these errors before we hit runtime. We can find errors in our code while we are writing it and prevent hours of troubleshooting and frustration.

Now let’s discuss some basic types and their meanings.

**Basic Types and What They Mean**

We’ll go over this in more detail in future parts of this series. For now, though, the types you need to know you should *already* know as part of using JavaScript. However, you may not really think about them much when you’re using them. I’ll outline those here along with some basic definitions.

* string — This is effectively anything in quotes, though [template literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) (a.k.a. template **strings**) are also valid values for strings.
* number — This is, well, a number. Any number that is valid in JavaScript is valid as this type, such as integers, floating points, and etc.
* boolean — A boolean is any value that evaluates to exactly true or exactly false. A valid boolean will always be the keyword true or the keyword false.
* Array — An array is a set of values. Arrays in TypeScript are very similar to those in JavaScript, but you must provide a type for your array which indicates the type of values the array contains. For example, Array<string> would indicate an array of strings. We’ll dig deeper in the next part of this series.
* void and undefined — These values are commonly used to specifically denote that something will return no value or is literally undefined. This topic is best left for our next lesson.
* any — This is literally any and all types as a possibility. For example, you will commonly assign any when you’re starting to type a new object and you haven’t determined all the types. The rule of thumb is you probably shouldn’t use this in a permanent fashion.

If you’ve skimmed the documentation at [TypeScriptLang.org](https://typescriptlang.org), you’ve likely noticed there are more types listed as “Basic.” I’m not covering those here intentionally to both focus our efforts as well as avoid confusion. Fear not, we will get to those in due time.

Now that we know a few types, let’s discuss what we mean when we say “static types.”

**Static Types vs. Dynamic Types**

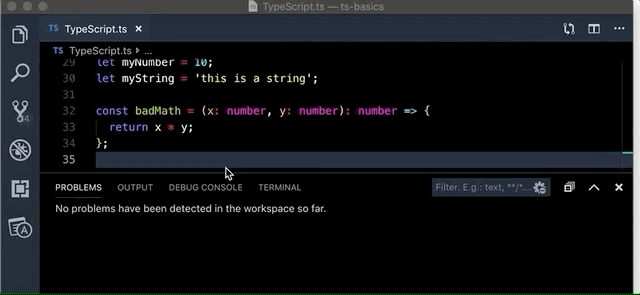
You may have heard people describe JavaScript as being dynamically typed. This means that the JavaScript interpreter (also generically referred to as the JavaScript “engine”) takes care of inferring and assigning types at runtime. This essentially means that our code has no hard-coded types until it’s executed.

TypeScript, while extremely similar and clearly related to JavaScript, is effectively the polar opposite of dynamically typed; it’s *statically typed* so types are checked and assigned at compile time.

However, with TypeScript and VS Code, this is only partially correct.

As we’ll learn in the next lesson when we configure VS Code to use TypeScript, we can actually type check our code as we write it, without compiling first! This allows us to see errors as we make them.





Mo’ code, mo’ problems.

If you’ve ever used the Chrome JavaScript console (a.k.a. the “web dev” console) to quickly test some JavaScript, you’ve seen this kind of behavior in action. The console shows any errors almost immediately as you type the code. In Chrome, this is limited to functions and assignments you’ve already created, but it’s similar in fashion to what VS Code can do with TypeScript.

**TypeScript at Work**

We’re going to take a look at our variables and the badMath function from the code we used earlier, but written using types to give us an idea of how this can be extremely useful to us and help us write better code.

Let’s break down the error we see above:

* Error TS2345 — Don’t worry about this; this is an error code associated with the specific kind of error we’re getting. While they may be useful as a shortcut to reading the error, you’re not likely to use this when you’re first starting with TypeScript.
* “Argument of type ‘string’” — This is referencing the myString argument we passed to our badMath function. There’s nothing in TypeScript (yet) to tell us *which* of the arguments are being referenced, but I’ll show you some tools when we setup VS Code that can help you at least isolate the function throwing this error.
* “…is not assignable to parameter of type ‘number’” — TypeScript is telling us that the “argument of type string” is not valid for the **parameter** it is meant to be passed as (our original parameters are x and y which we statically typed as numbers). In other words, TypeScript is saying, “When you wrote this function, you said you would take two **parameters** and that those would be numbers, but you’re giving me a string — what’s up with that?”

Every time we get an error in TypeScript, we are learning to write our code more accurately and efficiently. As we can see in the code gist above, even the simplest of functions can have bugs in JavaScript. When we rewrite the same code with TypeScript, we’ve avoided ever writing that bug.

It’s important to understand we can see these errors **before we even run our code.** TypeScript can help us catch bugs before we hit runtime, and that’s huge.

**Is TypeScript a Compiled Language?**

You’ve probably heard by now that TypeScript is compiled, though some will argue this point. It’s not a subject we need to discuss in detail, but just know that your TypeScript is converted to JavaScript before it’s executed. It’s during this process that TypeScript performs its type checking, which is where the magic happens.

VS Code is able to type check your code in near real-time, as you saw in the GIF earlier, and it hadn’t been compiled at that point. In the next lesson, we’ll see how we can take advantage of this to make this functionality more robust for our uses.

**What Is TypeScript, Again?**

Let’s ask that question again, and think to yourself if you can answer it on your own now — what is TypeScript?

It’s a bug-catcher, and it’s a toolbelt to help us verify the kind of data we’re passing around is indeed what we intend and expect it to be. It’s also a way to make us better programmers by making us think about how we are handling data. Specifically, it allows us to reduce errors and bugs by effectively turning JavaScript into a statically typed language.

We’ve only scratched the surface with this example, but I hope you can already see how powerful TypeScript can be.

**What’s Next?**

We’ve just skimmed the surface and taken the ten thousand foot view of TypeScript.

In the next lesson, we’ll start with configuring a project in VS Code to work with TypeScript. We’ll also configure some settings and extensions in VS Code to make the TypeScript coding experience a lot smoother and more beneficial to learning. I think you’ll be pleasantly surprised at how much TypeScript can add to your coding experience.

We’ll also dig deeper on basic types and apply them in a meaningful way so you can get a solid understanding of the core principals of TypeScript. We’ll practice using the basic types and start discussing some more advanced topics. I’ll give you some pointers on where you can learn more with TypeScript and how you can avoid some of the headaches that can come with being a newcomer to the language.