/\*

Links: [Book:TypeScript](https://basarat.gitbooks.io/typescript/content/), [Blog:GettingStarted](http://blog.teamtreehouse.com/getting-started-typescript), [Blog:TypeScriptHot](http://blog.teamtreehouse.com/typescript-hot-now-looking-forward), [Blog:AngularTS](https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8)

Notes:

1. Functional Programing(basic definition, some code examples in TS, show the regular logic operators, etc.) Kris WIP
2. Transpiling (Translate+Compile) Fico
3. Installation process Fico
4. OOP Programming in TypeScript Kris
5. Pitfalls of JavaScript
6. When should you use TypeScript

Additional notes:

1. Outline
2. Why use TypeScript Fico
3. When to use TypeScript Fico
4. Getting Started with TypeScript Fico
5. Major Projects Using TypeScript (Angular, Ionic etc.)
6. Comparing TypeScript to JavaScript
7. Advantages and disadvantages to using TypeScript
8. The basic types in TypeScript Kris

\*/

KrisToFeR part:

Functional Programming

There are many types of paradigms in programming, including imperative, declarative, structured, procedural, object-oriented, functional etc. A paradigm simply means a way or style of programming, some languages make it easier to write in a certain paradigms rather than others.

Specifically, we are going to focus on the functional programing paradigm which is a style of coding that treats all computation as  an evaluation of mathematical functions and avoids changing-state and mutable data, which means that it doesn’t use data which can be altered after its creation and also the state of the program cannot change. Functional programming is a type of declarative programing, which uses expressions or declarations to program instead of statements.

([source](https://en.wikipedia.org/wiki/Functional_programming))

Object-oriented Programming in TypeScript

Object-Oriented Programming(OOP) is a type of programing paradigm which focuses more on problem solving, rather than coding itself. OOP is generally used with classes and object which are meant to represent real-life problems. It stores data in the form of attributes, while it codes in the form of procedures, and these procedures often interact with other objects and access and modify the data within those objects. One big advantage of OOP is encapsulation. Encapsulation is a concept that inds data together with the functions that manipulate said data and keep them safe from outside interference and misuse. In Typescript, this is simply done by using the keywords public, private and protected before the name of the variable,for example:

private name: type.

TypeScript allows for a much simple Object-oriented style of programming as opposed to JavaScript because it doesn’t use the strange syntax of prototyping in JavaScript. In TypeScript defining classes is immediately familiar and straightforward as you can see from the following code example:

code\*\*

class A{

private x: number;

private y: string;

private z: any;

constructor(x: number, y: string, z: any){

this.x = x;

this.y = y;

this.z = z;

}

function foo(){  
 alert(x + y + z);

}

}

\*\*

While in contrast, the same code, but in JavaScript, would be translated as:

code\*\*

var A = (function () {

   function A(x, y, z) {

       this.x = x;

       this.y = y;

       this.z = z;

   }

   return A;

}());

function foo() {

   alert(x + y + z);

}

As we can see, JavaScript creates classes using various functions, and does so in a fairly roundabout way because of its functional nature and the fact that JavaScript was never meant to be an Object-Oriented programing language.

Creating members of a certain class in TypeScript is pretty simple, we need only to specify whether it’s private, protected or public, then typing the name of the member, and finally, specify the type of the member preceded by colons. For example:

code\*\*class class\_name{

private name: string;

protected name2: number;

public name3: any;

…

}

In TypeScript, as shown in the example above, classes are made by using the keyword class followed by the name of said class e.g. code\*\* class A{}. Then, the class must have a constructor which is used every time an object of the class is instantiated in order to construct said object. The syntax for creating a constructor is the following:

code\*\*

constructor(arg1: type, arg2: type, arg3: type...){

this.member1 = arg1;

this.member2 = arg2;

this.member3 = arg3;

...

}

You may notice the conspicuous absence of any destructor in the first example of classes in Typescript, that is because it doesn’t have destructors, because any object that is no longer referenced will be deleted by the garbage collector that JavaScript uses.

Another important concept of OOP is inheritance and polymorphism, and TypeScript handles in it’s typical succinct style. Consider the following example

code\*\*

Class Person{

Private name: string;

Private age: number;

constructor(name:string, age:number){

This.name = name;

This.age = age;

}

Function greet(){

alert("Hello, my name is  " + this.name + ", and I am " + this.age.toString()                 + " years old.");

}

}

Class Employee extends Person{

Private title:string;

Private salary: number;

constructor(name: string, age: number, title: string, salary: number){

super(name, age);

This.title = title;

This,salary = salary;

}

Function greet(){

super.greet();

alert("And also I am a " + this.title + " and make $" + this.salary.toString() + " per month");

}

}

The above example represents one parent class: Person, and one inherited, child class: Employee. The inherited class has access to all, except the private data members of the parent class. A child class is defined in TypeScript by using the keyword extends followed by the name of it's parent class i.e class Child extends Parent. By using the function super() you can access the functions and non-private data members of the parent class as well as its constructor. We can see an example of polymorphism by examining the function greet() in both of the classes. In the child class, the function greet() is overridden and has a different definition than the greet() function in the parent class. This means that the greet() function can have different outputs depending on which class object it’s called from.

Basic Types in TypeScript

Programming is all about data, and manipulating the simplest, most primitive forms of data is what makes the program function. TypeScript has all the same datatypes as JavaScript but with an addition of a useful enumeration type.

Boolean

The boolean type is the most datatype and it’s a simple false or true value.

let boolean\_var : boolean = true;

Number

All numbers in TypeScript are floating point numbers and their type in TypeScript is number. TypeScript supports decimal, hexadecimal, binary and octal literals.

let decimal: number = 10;

let hex: number = 0x00F0;

let binary: number = 0b1010;

let octal: number = 0o744;

String

TypeScript, like in many other languages, handles textual data using the type string. TypeScript can use either double quotes (“”) or single quotes (‘’) to surround string data.

let name: string = "John";

let color: string = 'blue';

You could also use backtick/backquote to surround what are called **template strings** which can span multiple lines and have embedded expressions, and these expressions have the form ${ expression }.

Let age: number = 34;

Let name: string = “John Doe”;

Let line: string = `Hello, I am ${ name } and I will be ${ age + 1 } years old next week`;

Array

TypeScript, like most languages, allows you to use arrays of multiple values. Arrays can be written in two ways, you can either write the type of the array followed by square brackets [] or you can use a generic array type, i.e. Array<type>.

let list1 : number[] = [1,2,3];

let list2: Array<number> = [1,2,3];

Tuple

Tuples are arrays with a fixed number of elements, however you can use different types in the same tuple.

let tuple1: [string, number];

tuple1 = ["typescript", 6]; //correct

tuple1 = [6, "typescript"]; //incorrect

When accessing an element with an index that is bigger than the index given when initializing the tuple, you create a union type.

tuple1[3] = "javascript"; //correct, since the type is either string or number.

tuple1[4] = false; //incorrect, since the type is not number or string.

Union types are an advanced types and will be covered later on.

Enum

Enum is a type in TypeScript that is not found in JavaScript, and is an array of numeric values which have specific names.

enum Color {Red, Green, Blue}

let c: Color = Color.Green;

By default, enums will number their elements starting with 0. You can change this by manually changing the value of one of the members. We can change the starting value to 1 by doing this:

enum Color {Red = 1, Green, Blue}

let c: Color = Color.Green;

You can even set all the values to a specific number:

enum Color {Red = 1, Green = 2, Blue = 4}

let c: Color = Color.Green;

What is Transpiling ([source](https://www.stevefenton.co.uk/2012/11/compiling-vs-transpiling/))

Even though not many people know about transpiling, the term and the process have been around for quite some time now. To avoid any confusion that there might be regarding the difference between the terms transpiling and compiling is we will firstly define the two terms. In short: Compiling is, in a general sense, transforming source code written in one language into code written in another language, with the main difference being that the two languages used differ in the level of abstraction that they hold. On the other hand, transpiling is a more specific term which means taking source code written in one language and transforming it into another language but with the same or similar level of abstraction.

For the sake of example, when we do

code\*\* tsc index.ts

The TypeScript transpiler is actually taking our TypeScript code written in the file index.ts and transforming it into a JavaScript file with the same name but different extension, e.g. index.js.

Both compilers and transpilers can optimise the code as part of the process.

Other common combinations that can be dubbed as transpiling include C++ to C, CoffeeScript to JavaScript, Dart to JavaScript and PHP to C++.

Compilation context

The compilation context refers to the grouping of the files that TypeScript will parse and analyze to determine what is valid and what isn't. Along with the information about which files, the compilation context contains information about *which compiler options*. A great way to define this logical grouping (we also like to use the term *project*) is using a tsconfig.json file.

You have two options when it comes to the tsconfig.json file, you can either write it yourself or you can generate it using the command tsc --init which will initialize the folder in which we are present as a TypeScript project and will generate the tsconfig.json. Of course, we can just have an open and closing bracket to serve as our tsconfig.json after all the is technically an empty JSON object.

{}

The compiler, or rather the transpiler will choose to follow its default options such as transpiling all TypeScript documents, meaning all files with the .ts extension, it will transpile them into the default target which is ECMAScript 5, it will allow plain old JavaScript to be seen as valid if written in the given TypeScript files et cetera.

To customize or specify the compiler options we can use the compilerOptions which should be written in the tsconfig.json file. For example:

{

 "compilerOptions": {

   /\* Basic Options \*/

   "target": "es5",                       /\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. \*/

   "module": "commonjs",                  /\* Specify module code generation: 'commonjs', 'amd', 'system', 'umd' or 'es2015'. \*/

   "lib": [],                             /\* Specify library files to be included in the compilation:  \*/

   "allowJs": true,                       /\* Allow javascript files to be compiled. \*/

   "checkJs": true,                       /\* Report errors in .js files. \*/

   "jsx": "preserve",                     /\* Specify JSX code generation: 'preserve', 'react-native', or 'react'. \*/

   "declaration": true,                   /\* Generates corresponding '.d.ts' file. \*/

   "sourceMap": true,                     /\* Generates corresponding '.map' file. \*/

   "outFile": "./",                       /\* Concatenate and emit output to single file. \*/

   "outDir": "./",                        /\* Redirect output structure to the directory. \*/

   "rootDir": "./",                       /\* Specify the root directory of input files. Use to control the output directory structure with --outDir. \*/

   "removeComments": true,                /\* Do not emit comments to output. \*/

   "noEmit": true,                        /\* Do not emit outputs. \*/

   "importHelpers": true,                 /\* Import emit helpers from 'tslib'. \*/

   "downlevelIteration": true,            /\* Provide full support for iterables in 'for-of', spread, and destructuring when targeting 'ES5' or 'ES3'. \*/

   "isolatedModules": true,               /\* Transpile each file as a separate module (similar to 'ts.transpileModule'). \*/

   /\* Strict Type-Checking Options \*/

   "strict": true,                        /\* Enable all strict type-checking options. \*/

   "noImplicitAny": true,                 /\* Raise error on expressions and declarations with an implied 'any' type. \*/

   "strictNullChecks": true,              /\* Enable strict null checks. \*/

   "noImplicitThis": true,                /\* Raise error on 'this' expressions with an implied 'any' type. \*/

   "alwaysStrict": true,                  /\* Parse in strict mode and emit "use strict" for each source file. \*/

   /\* Additional Checks \*/

   "noUnusedLocals": true,                /\* Report errors on unused locals. \*/

   "noUnusedParameters": true,            /\* Report errors on unused parameters. \*/

   "noImplicitReturns": true,             /\* Report error when not all code paths in function return a value. \*/

   "noFallthroughCasesInSwitch": true,    /\* Report errors for fallthrough cases in switch statement. \*/

   /\* Module Resolution Options \*/

   "moduleResolution": "node",            /\* Specify module resolution strategy: 'node' (Node.js) or 'classic' (TypeScript pre-1.6). \*/

   "baseUrl": "./",                       /\* Base directory to resolve non-absolute module names. \*/

   "paths": {},                           /\* A series of entries which re-map imports to lookup locations relative to the 'baseUrl'. \*/

   "rootDirs": [],                        /\* List of root folders whose combined content represents the structure of the project at runtime. \*/

   "typeRoots": [],                       /\* List of folders to include type definitions from. \*/

   "types": [],                           /\* Type declaration files to be included in compilation. \*/

   "allowSyntheticDefaultImports": true,  /\* Allow default imports from modules with no default export. This does not affect code emit, just typechecking. \*/

   /\* Source Map Options \*/

   "sourceRoot": "./",                    /\* Specify the location where debugger should locate TypeScript files instead of source locations. \*/

   "mapRoot": "./",                       /\* Specify the location where debugger should locate map files instead of generated locations. \*/

   "inlineSourceMap": true,               /\* Emit a single file with source maps instead of having a separate file. \*/

   "inlineSources": true,                 /\* Emit the source alongside the sourcemaps within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. \*/

   /\* Experimental Options \*/

   "experimentalDecorators": true,        /\* Enables experimental support for ES7 decorators. \*/

   "emitDecoratorMetadata": true          /\* Enables experimental support for emitting type metadata for decorators. \*/

 }

}

Additionally we can be explicit when telling the transpiler which files we want to transpile by telling so in “files”, for example:

{

   "files":[

       "./some/file.ts"

   ]

}

Or we can use “include” and “exclude” to be even more specific in our file choice:

{

   "include":[

       "./folder"

   ],

   "exclude":[

       "./folder/\*\*/\*.spec.ts",

       "./folder/someSubFolder"

   ]

}

Default options are to transpile all TypeScript files.

Installation Process

There are two main ways to get the TypeScript tools:

1. Via npm (the Node.js package manager)
2. By installing TypeScript's Visual Studio plugins

For option 1, you will need to have NodeJS installed and alongside it npm, a package manager. You can easily install them both by following this [link](https://nodejs.org/en/download/), where we recommend that you choose to install the LTS version, recommended for most users. After NodeJS is installed you can verify your installation by typing

code\*\* node -v

Or

code\*\* node --version

For which you will get something like “v6.10.0” as output. You can also run “npm -v” or “npm --version” to verify that the package manager has also been installed.

Assuming you have node and npm and you’ve chosen to take on the first installation option (using npm) simply write this line in your command line:

code\*\* npm install -g typescript

Where the -g is the global flag meaning that it’ll install TypeScript on the whole machine, if we didn’t have the -g global flag then npm will automatically assume that we want TypeScript just in the current project that we’re working in, which is actually the folder we are in at that moment inside the command line.

For option 2, Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default. If you didn’t install TypeScript with Visual Studio, you can still [download it](https://www.typescriptlang.org/#download-links). In other IDEs such as WebStorm or PHPStorm TypeScript comes built-in, so you can easily just incorporate TypeScript into your projects assuming your project is suited for its use.

Making you first TypeScript file

Assuming you’ve went through with the installation process, using theeditor of your choice type in the following JavaScript code in a file named “greeter.ts”, notice that the extensions is “.ts” instead of “.js”.

**function** **greeter**(person) {  
   **return** "Hello, " + person;  
}  
  
**var** user = "Jane User";  
  
document.body.innerHTML = greeter(user);

At the command line, run the TypeScript compiler:

code\*\* tsc greeter.ts

The result will be a file greeter.js which contains the same JavaScript that we fed in.

Additional notes on transpiling, joining files and watcher. ([source](http://blog.teamtreehouse.com/getting-started-typescript))

The following command will compile a single .ts file into a .js file:

tsc app.ts

This will result in an app.js file being created.

To compile multiple .ts files:

tsc app.ts another.ts someMore.ts

This will result in 3 files, app.js, another.js and someMore.js.

You can also use wildcards too. The following command will compile all TypeScript files in the current folder.

tsc \*.ts

All TypeScript files will compile to their corresponding JavaScript files.

You can also compile all your TypeScript files down to a single JavaScript file. This can reduce the number of HTTP requests a browser has to make and improve performance on HTTP 1.x sites. To do this use the --out option like so:

tsc \*.ts --out app.js

Instead of running the tsc command all the time you can use the option --watch.

tsc \*.ts --out app.js --watch

Every time there’s an update to a TypeScript file it’ll recompile the source files to JavaScript.

If you’re using a wildcard like this, any new files created since running the tsc command won’t get compiled, you need to stop the watcher and start again.

Now we can start taking advantage of some of the new tools that TypeScript offers. Add a : string type annotation to the ‘person’ function argument as shown here:

function greeter(person: string) {  
   return "Hello, " + person;  
}  
  
var user = "Jane User";  
  
document.body.innerHTML = greeter(user);

Type annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the greeter function to be called with a single string parameter. We can try changing the call greeter to pass an array instead:

function greeter(person: string) {  
   return "Hello, " + person;  
}  
  
var user = [0, 1, 2];  
  
document.body.innerHTML = greeter(user);

Re-compiling, you’ll now see an error:

greeter.ts(7,26): error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'string'.

Similarly, try removing all the arguments to the greeter call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the greeter.js file is still created. You can use TypeScript even if there are errors in your code. But in this case, TypeScript is warning that your code will likely not run as expected.

By doing this example we understand that TypeScript is still in essence JavaScript but with added safety. By using types, by forcing the programer to be more explicit we achieve a more strict way of coding, in which the chances of making faults or bugs is reduced. This forces even other programmers which examine or use the code to use it properly because they are forced to follow convention and input appropriate objects in functions that only accept those given types of objects, thus TypeScript is in a sense JavaScript with a debugging safety net. TypeScript in this manner allows for greater safety when working on projects with twenty, thirty or more people, it allows for division of labour amongst the workers in which every person can safely contribute and safely use the work of others. Clearly we see that TypeScript buys in the whole concept of object-orientated programming. But there will be other segments in which we will elaborate more on certain points introduced here. For now let’s develop our sample a little further. Here we use an interface that describes objects that have a firstName and lastName field. In TypeScript, two types are compatible if their internal structure is compatible. This allows us to implement an interface just by having the shape the interface requires, without an explicit implements clause.

interface Person {

   firstName: string;

   lastName: string;

}

function greeter(person: Person) {

   return "Hello, " + person.firstName + " " + person.lastName;

}

var user = { firstName: "Jane", lastName: "User" };

document.body.innerHTML = greeter(user);

Classes

Finally, let’s extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

Here we’re going to create a Student class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of public on arguments to the constructor is a shorthand that allows us to automatically create properties with that name.

class Student {

   fullName: string;

   constructor(public firstName, public middleInitial, public lastName) {

       this.fullName = firstName + " " + middleInitial + " " + lastName;

   }

}

interface Person {

   firstName: string;

   lastName: string;

}

function greeter(person : Person) {

   return "Hello, " + person.firstName + " " + person.lastName;

}

var user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);

Re-run tsc greeter.ts and you’ll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand for the same prototype-based OO that is frequently used in JavaScript. We will dwell deep into classes in another segment.

Running your TypeScript web app

Now type the following in a file greeter.html:

<!DOCTYPE html>

<html>

   <head><title>TypeScript Greeter</title></head>

   <body>

       <script src="greeter.js"></script>

   </body>

</html>

Open greeter.html in the browser to run your first simple TypeScript web application!

We believe that it’s important to note that in the greeter.html file when we add our script we are actually adding the compiled JavaScript version of our code and not the actual code that we typed in written in TypeScript, additionally we recommend that you add all of your scripts above the ending body tag </body> the reason being that when the page is being loading in the browser the person viewing the webpage actually gets to see the visual part of the web page first, its appearance. While the person viewing actually clicks or uses our page the scripts will have already loaded.

Why use TypeScript? ([source](https://vsavkin.com/writing-angular-2-in-typescript-1fa77c78d8e8))

[TypeScript](http://typescriptlang.org/) is ECMAScript 6 (ES6) with optional typing.

[ECMAScript](http://en.wikipedia.org/wiki/ECMAScript) is the standardized specification for the JavaScript language. It’s sometimes referred as ECMAScript Harmony or ES.next. At the time of this writing, JavaScript is currently at ECMAScript 8, which is not yet finished ([source](https://en.wikipedia.org/wiki/ECMAScript#8th_Edition_.28not_yet_finished.29)). Since ES6 is backwardly compatible with the ES5 syntax, you can start writing TypeScript without changing any of your existing code. The good thing is that learning TypeScript doesn’t require that the programer learn a whole new syntax, as one may would using CoffeeScript for example.

TypeScript Has Great Tools

The biggest selling point of TypeScript is tooling. It provides advanced autocompletion, navigation, and refactoring. Having such tools is almost a requirement for large projects. Without them the fear changing the code puts the code base in a semi-read-only state, and makes large-scale refactorings very risky and costly.

TypeScript is not the only typed language that compiles to JavaScript. There are other languages with stronger type systems that in theory can provide absolutely phenomenal tooling. But in practice most of them do not have anything other than a compiler. This is because building rich developer tools has to be an explicit goal from day one, which it has been for the TypeScript team. That is why they built language services that can be used by editors to provide type checking and autocompletion. If you have wondered why there are so many editors with great TypeScript supports, the answer is the language services.

The fact that intellisense and basic refactorings (e.g., rename a symbol) are reliable makes a huge impact on the process of writing and especially refactoring code.

TypeScript Makes Abstractions Explicit

A good design is all about well-defined interfaces. And it is much easier to express the idea of an interface in a language that supports them.

As you can see, both classes play the role of a purchaser. Despite being extremely important for the application, the notion of a purchaser is not clearly expressed in the code. There is no file named purchaser.js. And as a result, it is possible for someone modifying the code to miss the fact that this role even exists.

function processPurchase(purchaser, details){ }

class User { }

class ExternalSystem { }

It is hard, just by looking at the code to tell what objects can play the role of a purchaser, and what methods this role has. We do not know for sure, and we will not get much help from our tools. We have to infer this information manually, which is slow and error-prone.

Now, compare it with a version where we explicitly define the Purchaser interface.

interface Purchaser {id: int; bankAccount: Account;}

class User implements Purchaser {}

class ExternalSystem implements Purchaser {}

The typed version clearly states that we have the Purchaser interface, and the User and ExternalSystem classes implement it. So TypeScript interfaces allow us to define abstractions/protocols/roles.

It is important to realize that TypeScript does not force us to introduce extra abstractions. The Purchaser abstraction is present in the JavaScript version of the code, but it is not explicitly defined.

In a statically-typed language, boundaries between subsystems are defined using interfaces. Since JavaScript lacks interfaces, boundaries are not well expressed in plain JavaScript. Not being able to clearly see the boundaries, developers start depending on concrete types instead of abstract interfaces, which leads to tight coupling.

My experience of working on Angular before and after our transition to TypeScript reinforced this belief. Defining an interface forces me to think about the API boundaries, helps me define the public interfaces of subsystems, and exposes incidental coupling.

Optional Typing

Where TypeScript comes into its own is in its optional typing. A typed language like Java, C# or Objective-C requires you to specify the type of the variable when declaring it.

Declaring a string in Java would look like: String name = "Andrew";, in C# string name = "Andrew"; and in Objective-C it’s NSString name = "Andrew".

In TypeScript it would be: var name: string = "Andrew". But with it being optional, it can be just plain var name = "Andrew". In strictly typed languages you have to declare everything all the time, with TypeScript you don’t!

Having types in your script allows text editors and IDEs to give you intelligent hints quickly without having to run your code.

It also helps with autocompletion of your code too. Other text editors autocomplete based on the text written in your project files, it doesn’t know anything about the type of each variable so you may end up passing a similarly named variable into a method call but it’s the wrong type of object. However with TypeScript, code editors can have a more intelligent approach and suggest more appropriate variables to pass into a function call. It’s a real productivity boost. It’s almost as if the code is writing itself.

It also reduces the need to look up documentation so frequently since the code is annotated with the types needed for a method call or what will be returned from a method call.

Major Projects using TypeScript

Because of these productivity wins from cleaner ES6 code, autocompletion, and hinting from optional typing, TypeScript is being adopted into major projects, like the dynamic web application framework, which can also be used on mobile, AngularJS 2.0 and the open source framework for building amazing mobile applications, Ionic Framework 2.0, the text editor Visual Studio Code etc..

You can also leverage TypeScript in JavaScript projects not written in TypeScript in the first place. TypeScript has definition files that allow you to get these productivity boosts when coding almost in any environment from the Browser to Node.js. You get all the autocompletion and hinting as if they were written in TypeScript.

Why use Types?

The term dynamic typing refers to the practise where a programmer needn’t declare the type of a variable when programing his code, because the variable will dynamically be assigned a type when needed. This is the practise that JavaScript follows as well as popular languages such as PHP and Ruby. On the other hand we have static typing, which is found in all major and famous languages such as a Java, C++/C, C# etc. It is when the programmer is forced into declaring the type for each variable.

What does static typing offer?

Static typing most of the time leads the programmer to think along the lines of software structure, meaning he needs to understand what variable and more specifically what kind of variable can use a certain function and which cannot. Because of this reason TypeScript has been adopted and used in major programing projects as mentioned. Some projects have completely been rewritten into TypeScript because TypeScript conveys the object-oriented programing concept much more clearly than using certain hacks or workarounds when typing plain old JavaScript. Of course, TypeScript also uses those same workarounds and hacks when transpiling the code to JavaScript but the main thing is that while the program is being created the programmer needs only to think about solving the problem and less about how to encapsulate or compose information. Of course the static typing in TypeScript is optional, but of course most programmers will use it since they they want their abstractions to be explicit.

TypeScript Makes Code Easier to Read and Understand

Let’s start off with an example using the function jQuery.ajax() from the JavaScript library jQuery. What kind of information can we get from its signature?

code\*\*jQuery.ajax(url, settings)

The only thing we can tell for sure is that the function takes two arguments. We can guess the types. Maybe the first one is a string and the second one is a configuration object. But it is just a guess, and we might be wrong. We have no idea what options go into the settings object (neither their names nor their types), or what this function returns.

There is no way we can call this function without checking the source code or the documentation. Checking the source code is not a good option — the point of having functions and classes is to be able to use them without knowing how they are implemented. In other words, we should rely on their interfaces, not on their implementation. We can check the documentation, but it is not the best developer experience — it takes additional time, and the docs are often out-of-date.

So although it is easy to read jQuery.ajax(url, settings), to really understand how to call this function we need to either read its implementation or its docs.

Now, contrast it with a typed version.

code\*\*ajax(url: string, settings?: JQueryAjaxSettings): JQueryXHR;

interface JQueryAjaxSettings {

 async?: boolean;

 cache?: boolean;

 contentType?: any;

 headers?: { [key: string]: any; };

 //...

}

interface JQueryXHR {

 responseJSON?: any; //...

}

It gives us a lot more information.

1. The first argument of this function is a string.
2. The settings argument is optional. We can see all the options that can be passed into the function, and not only their names, but also their types. Note: the question mark operator specifies that the given field is optional, e.g. async?: boolean; meaning this field isn’t specifically required.
3. The function returns a JQueryXHR object, and we can see its properties and functions.

The typed signature is certainly longer than the untyped one, but :string, :JQueryAjaxSettings, and JQueryXHR are not clutter. They are important documentation that improves the comprehensibility of the code. We can understand the code to a much greater degree without having to dive into the implementation or reading the docs.

One thing that is different about TypeScript comparing to many other languages compiled to JavaScript is that its type annotations are optional, and jQuery.ajax(url, settings) is still valid in TypeScript. So instead of an on-off switch, TypeScript’s types are more of a dial. If you find that the code is trivial to read and understand without type annotations, do not use them. Use types only when they add value.

A few words and examples regarding the typing

Dynamically-typed languages have inferior tooling, but they are more malleable and expressive. We think using TypeScript makes your code more rigid, but to a much lesser degree than people think. For example:

const PersonRecord = Record({name:null, age:null});

function createPerson(name, age) {

 return new PersonRecord({name, age});

}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");

How do we type the record? Let’s start with defining an interface called Person:

interface Person { name: string, age: number };

If we try to do the following:

function createPerson(name: string, age: number): Person {

 return new PersonRecord({name, age});

}

the TypeScript compiler will complain. It does not know that PersonRecord is actually compatible with Person because PersonRecord is created reflectively. TypeScript’s type system is not the most advanced one. But its goal is different. It is not here to prove that the program is 100% correct. It is about giving you more information and enable greater tooling. So it is alright to take shortcuts when the type system is not flexible enough. So we can just cast the created record, as follows:

function createPerson(name: string, age: number): Person {

 return <any>new PersonRecord({name, age});

}

The typed example:

interface Person { name: string, age: number };

const PersonRecord = Record({name:null, age:null});

function createPerson(name: string, age: number): Person {

 return <any>new PersonRecord({name, age});

}

const p = createPerson("Jim", 44);

expect(p.name).toEqual("Jim");

The reason why it works is because the type system is structural. As long as the created object has the right fields, name and age, we are good.

You need to embrace the mindset that it is alright to take shortcuts when working with TypeScript. Only then you will find using the language enjoyable. For instance, don’t try to add types to some funky metaprogramming code  most likely you won’t be able to express it statically. Type everything around that code, and tell the typechecker to ignore the funky bit. In this case you will not lose a lot of expressiveness, and the bulk of your code will remain toolable and analyzable.

The optional type system preserves the JavaScript development workflow. Large parts of your application’s code base can be “broken”, but you can still run it. TypeScript will keep generating JavaScript, even when the type checker complains. This is extremely useful during development

When to use TypeScript

We believe that TypeScript should be used on projects which benefit from its use as that is the most rational choice. That would refer to projects that would be much better off with structure in the object-oriented sense. For smaller projects which use small scripts only to change a few css properties of HTML elements TypeScript is unneeded and even too verbose for small tasks such as that. Of course writing plain old JavaScript in a TypeScript file is completely alright and one can do that, but in essence TypeScript’s utilities are never really used. But they are used when we have some hierarchy of classes, dependencies between information et cetera and TypeScript is the perfect candidate to sort out and structure such things.

In conclusion, we believe that small projects which use shorts scripts which do not require structure, do not require types as well and therefore would not have the specific benefit of being written in TypeScript which is the opposite for bigger projects with plenty lines of code which would greatly benefit by having structure and thus would benefit from having their programmers write TypeScript code instead of plain old JavaScript code.