TypeScript

JavaScript Development Guide

Nicholas Brown

TypeScript

JavaScript Development Guide

By Nicholas Brown

Table of Contents

Disclaimer

While all attempts have been made to verify the information provided in this book, the author does assume any responsibility for errors, omissions, or contrary interpretations of the subject matter contained within. The information provided in this book is for educational and entertainment purposes only. The reader is responsible for his or her own actions and the author does not accept any responsibilities for any liabilities or damages, real or perceived, resulting from the use of this information.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

Introduction

There is a greater need for large software applications, and especially the web applications. The reason behind this is the increase in the functionalities which are needed to be implemented in software applications. This calls for us to learn how to develop large applications. This can only be done in programming languages which are capable of supporting that. TypeScript is one of such programming languages. It supports the creation of large applications. This book is an excellent waay for you to learn how to program in TypeScript. Enjoy reading!

Chapter 1- A Brief Overview of TypeScript

TypeScript is a programming language which was developed and is maintained by Microsoft. It is an open source programming language. The language is a superset of JavaScript since it has added some extra features to it, such as support for object oriented programming and optional static typing. The language was developed for the purpose of creating large applications, and it always compiles into JavaScript. One can use TypeScript to develop JavaScript applications for both the client side and server side. Since we have said that TypeScript is a superset of javaScript, any program written in JavaScript is also viable to be considered a TypeScript program. The TypeScript compiler was also written in TypeScript.

The aim of TypeScript is to provide developers with an easy mechanism by which to develop large applications. It comes with concepts such as classes, inheritance, interfaces, modules, and others, and these make it easy for developers to create large applications with much ease.

Chapter 2- Installing TypeScript

There are two ways that we can install TypeScript into our system. These include the following:

1. By use of the npm (Node Package Manager) tool.

2. By installation of Visual studio plugins for TypeScript.

For Windows users, you can use the second option if it will be more convenient for you. Just install the Visual Studio, and then download and install the necessary plugin for programming in TypeScript. This option has been found to be a bit superior compared to the use of other code editors.

For those using platforms other than Windows, just ensure that you have a text editor, a browser, and the npm package for TypeScript so as to be able to program in TypeScript.

The following steps can be followed for this to be done:

1. Begin by installing the Node Package Manager (npm).

Just open the terminal, and execute the commands given below:

$ curl http://npmjs.org/install.sh | sh

$ npm —version

1.1.70

2. In the next step, install the TypeScript npm package, and this should be done globally on the command line. The following commands can be used for doing this:

$ npm install -g typescript

$ npm view typescript version

npm http GET https://registry.npmjs.org/typescript

npm http 304 https://registry.npmjs.org/typescript

0.8.1-1

Make sure that you use the latest version of your browser, such as Chrome. Any text editor is appropriate, and an example of this is the Sublime text editor.

Hello World Example

Now that we have set up the environment ready for programming, we can begin to program. We will begin by creating the Hello world example. Remember that TypeScript is a superset of ES5, and it has more features for ES6. TypeScript is also compiled into JavaScript.

We want to create our index file and then reference it to an external file. Here is the code for the file "*index.html"*:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>This is TypeScript</title>
</head>
<body>
```

<script src=“hello.ts”></script>

</body>

</html>

As shown in the above code, we have referenced the code in the file to the file “*hello.ts.”* This has been done in the line “<script src=“hello.ts”></script>” as the script source. The extension “*.ts”* in the file name means that it is a TypeScript file.

Now, create a new file and give it the name “*hello.ts.”* The following line of code should then be added to the file:

alert(‘hello world, this is TypeScript!’);

After that, just launch the command line, and then navigate to the folder with the file “*hello.ts.”* Compile the file by executing the following command:

tsc hello.ts

The tsc command in TypeScript represents the TypeScript compiler, and it will generate a new file with the name "*hello.js.*" From the extension in the file, we will have transformed the TypeScript to a JavaScript file, which is what normally happens in TypeScript.

Chapter 3- Type Annotations

This feature is optional in TypeScript, and it allows us to express and implement indentation in our TypeScript programs. We need to demonstrate this by creating a program which will calculate the area of a rectangle. Just open your text editor and create a new file with the name "*area.ts.*" Add the following code to the file:

```
function area(shape: string, width: number, height: number) {
    var area = width * height;
    return "The shape is " + shape + " having an area of " + area + " cm squared.";
}

document.body.innerHTML = area("rectangle", 20, 15);
```

Next, open the file "*index.html*" and change the script source from "*type.ts*" to "*area.ts.*" Open your terminal, and then execute the following command so as to compile the whole thing:

tsc area.ts

The message together with the area of the rectangle will be displayed. As you must have noticed, the type annotations are passed to the parameters of the function, and they are used for indicating the kind of parameters which are supported by the function. In the above example, the parameter "*shape*" is a string, while the parameters length and width are numbers.

You have to know that both Type annotations and other TypeScript features are enforced during compile time. In case one passes other types of values to these parameters, a compile-time error will be displayed by the compiler. This is a very good idea when it comes to working with large applications.

We need to demonstrate how errors will be detected in our case. Consider the code given below:

```
function area(shape: string, width: number, height: number) {
   var area = width * height;
   return "The shape is " + shape + " having an area of " + area + " cm squared.";
}
```

document.body.innerHTML = area(“rectangle”, “width”, 15); // wrong type of width

In the above example, the width has been passed in as a string, which is wrong. The compiler expects that the width should be a number. After compiling the program given above, you will get the following as the error:

$ tsc type.ts

type.ts(6,26): Supplied parameters do not match any signature of call target

That is the kind of error you get.

However, you must have noticed that although there was a warning due to type mismatch, this did not stop the TypeScript file from compiling and creating a JavaScript file.

As shown in the above examples, it is very clear that annotations are used for the purpose of recording the function which a function or a variable is intended to perform. Suppose that you have a function which expects that only a single parameter should be passed to it, and the parameter should be of the string type. What should happen once we pass an array

to it? Consider the example given below:

```
function welcome(name: string) {
    return “Hello, ” + name;
}

var user = [0, 1, 2];

document.body.innerHTML = wlecome(user);
```

Once you recompile the above code, you will get the following error:

```
welcome.ts(7,26): Supplied parameters do not match any signature of call target
```

Similarly, you can try to remove all the parameters from the welcome call function. You will be warned by the TypeScript compiler that you have passed in an unexpected number of parameters.

Chapter 4- Interfaces

We need to use our previous example by expanding it so as to implement an interface which will describe the shape as an object having an optional color property.

Create a new file, and give it the name "*interface.ts.*" Open the file "*index.html*" and then modify it so as to have the script source as "*interface.js.*" Add the following code to the file "*interface.ts*":

```
interface Shape {
    name: string;
    width: number;
    height: number;
    color?: string;
}

function area(shape : Shape) {
    var area = shape.width * shape.height;
```

return “The shape is ” + shape.name + ” having an area of ” + area + ” cm squared”;

}

console.log(area({name: “rectangle”, width: 20, height: 15}));

console.log(area({name: “square”, width: 20, height: 25, color: “blue”}));

Interfaces are the names which are given to object types. It is possible for us to declare interfaces, and at the same time use it in our type annotation.

Once you compile the file “*interface.js,”* you will get no errors. For us to make it give us an error, let us add a new line of code to the file “*interface.js”* having a shape with no name property and then observe the result in the console of our browser. Append the line of code given below to the file “*interface.js”*:

console.log(area({width: 20, height: 15}));

You can then run the command “tsc interface.js” so as to compile the code in the file. An error will be the output, but do not be worried by that for now. Refresh your browser, and

then observe what the console gives you. The text and the area of the rectangle will form the output on the console. The arrow should be as follows:

interface.ts(26,13): Supplied parameters do not match any signature of call target:

Could not apply type ‘Shape’ to argument 1, which is of type ‘{ width: number; height: number; }’

The error shown above is a result of the fact that the object which has been passed to the function “*area()”* will not conform to the interface named “*shape,”* and a name property is needed so as to do so.

Consider the code given below, which shows how an interface can be used in TypeScript:

```
interface Person {
    fName: string;
    lName: string;
}

function welcome(person: Person) {
```

```
    return “Hello, ” + person.fName + ” ” + person.lName;
}

var user = { fName: “Jane”, lName: “User” };

document.body.innerHTML = welcome(user);
```

In the above example, we have implemented an interface which is to be used for describing a person by use of the first and last name. For two types to be compatible in JavaScript, you have to ensure that their internal is also compatible. With this, we will be able to implement an interface just by use of the shape which is needed by the interface, and we will not need to use an explicit “*implements”* clause.

Chapter 5- Arrow Function Expressions

For one to understand the scope of the keyword "*this,*" it is a challenging practice. TypeScript supports the use of arrow functions and expressions so as to make this a bit easier for you. This is just a new feature which has been introduced in ES6 for the first time. In arrow functions, the value for keyword "*this*" is preserved, and this becomes easy for us to write and use callbacks. Consider the sample code given below:

```
var objectShape = {
    name: "rectangle",
    popup: function() {
    console.log('The inside popup(): ' + this.name);

    setTimeout(function() {
    console.log('The inside setTimeout(): ' + this.name);
    console.log("The shape is a " + this.name + "!");
    }, 3000);

    }
```

};

objectShape.popup();

The object “*this.name”* will be empty, and this will be depicted clearly on the browser console.

The TypeScript arrow function can be used for the purpose of fixing this problem. You have to replace “*function()”* with “() =>.” This is shown in the code given below:

```
var objectShape = {
    name: “rectangle”,
    popup: function() {

    console.log(‘The inside popup(): ‘ + this.name);

    setTimeout( () => {
    console.log(‘The inside setTimeout(): ‘ + this.name);
    console.log(“The shape is a ” + this.name + “!”);
```

```
    }, 3000);


    }
};

objectShape.popup();
```

You can then look at the JavaScript file which is generated. You will observe that the compiler has injected a new variable, which is “var _this = this;,” and this will be used in the callback “setTimeout(),” and the “*name”* property will be referenced.

Consider the pure JavaScript class given below:

```
function Person(age) {
    this.age = age
    this.becomeOld = function() {
    this.age++;
    }
}
```

```
var person = new Person(1);

setTimeout(person.becomeOld,1000);


setTimeout(function() { console.log(person.age); },2000); // the 1 should have been set to 2
```

Once the above code has been executed in the browser, the object "*this*" will point to "*window*" since it will be the one used for executing the "*becomeOld*" function. This can only be solved by executing the arrow function as shown below:

```
function Person(age) {
    this.age = age
    this.becomeOld = () => {
    this.age++;
    }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);


setTimeout(function() { console.log(person.age); },2000); // 2
```

The reason behind the above code functioning is that referencing to the object "*this*" will be captured from outside our function body. The above code is similar to the JavaScript code given below:

```
function Person(age) {
    this.age = age
    var _this = this;  // capture the this
    this.becomeOld = function() {
    _this.age++;   // using the already captured this
    }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2
```

Since we are using TypeScript, it is possible for us to use a sweeter syntax by combining classes and arrows. This is shown in the example code given below:

```
class Person {
    constructor(public age:number) {}
    becomeOld = () => {
    this.age++;
    }
}
var person = new Person(1);
setTimeout(person.becomeOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2
```

However, you should know that the fat arrow should only be used if you are to give the function to another individual so as to make the calling. This one shown below:

```
var becomeOld = person.becomeOld;
// This will be called by someone else later:
becomeOld();
```

If you are to call it yourself, you should do it as shown below:

person.becomeOld();

And then the "*this*" object will become the correct calling context.

In most cases, libraries will use the object "*this*" so as to pass the object which is currently being iterated on. If you are in need of accessing the library passed "*this,*" even your surrounding variable, you can make use of a temp variable like you would do in the absence of arrow functions. This is shown in the code given below:

```
let _self = this;
thing.each(function() {
    console.log(_self); // this is your lexically scoped value
    console.log(this); // this is a library passed value
});
```

Chapter 6- Classes

In TypeScript, there are classes which have a public or private accessibility. In TypeScript, classes are used in the same way as in ES6. Create a new file, and give it the name "*class.ts.*" Add the following code to the class:

```
class Shape {

    area: number;
    color: string;

    constructor ( name: string, width: number, height: number ) {
    this.area = width * height;
    this.color = "red";
    };

    shoutit() {
    return "The object is" + this.color + " " + this.name +  " having an area of " + this.area + " cm squared.";
```

```
    }
}

var square = new Shape("square", 20, 20);

console.log( square.shoutit() );
console.log( 'Area of Object: ' + square.area );
console.log( 'Name of Object: ' + square.name );
console.log( 'Color of Object: ' + square.color );
console.log( 'Width of Object: ' + square.width );
console.log( 'Height of Object: ' + square.height );
```

Our object in the above example has two properties, namely "*area*" and "*color.*" We have also implemented a constructor and the method "*shoutit().*" Our constructor arguments have a scope which is local to the constructor. Due to this, errors will be observed both in the browser and the compiler, as shown below:

```
class.ts(12,42): The 'name' property does not exist on the value of type 'Shape'
class.ts(20,40): The 'name' property does not exist on the value of type 'Shape'
class.ts(22,41): The 'width' property does not exist on the value of type 'Shape'
class.ts(23,42): The 'height' property does not exist on the value of type 'Shape'
```

TypeScript supports both “*public”* and “*private”* access modifiers. With public members, we are able to access them from everywhere, while with the private members, we can only access them from within the In JavaScript, there is no feature which can be used for the purpose of enforcing privacy, and this is why it is enforced during compile-time, and it serves to warn the developer about their intention to make it private.

We are in need of adding the public accessibility modifier to the constructor argument “*name”* and then a private accessibility modifier to a member, that is, “*color.”* Once an accessibility modifier has been added to an argument of a constructor, this will automatically become a member of class with that accessibility modifier. Consider the example given below:

…

private color: string;

…

constructor (public name: string, width: number, height: number) {

…

Consider the ES 6 class given below:

```
class Monster {
    constructor(name, initPosition) {
    this.name = name;
    this.initPosition = initPosition;
    }
}
```

In ES 6, the above class will be okay when used in the way it has been defined. However, this will not be okay in TypeScript as you are expected to define some properties on the object. This is shown in the code given below:

```
interface Point {
    a: number,
    b: number
}

class Monster {
    name: string;
    initPosition: Point
```

```
    constructor(name, initPosition) {
    this.name = name;
    this.initPosition = initPosition;
    }
}

var scaring = new Monster(“Alien”, {a: 0, b: 0});
```

However, there is a shorthand to that. The following code will just do the same:

```
interface Point {
    a: number,
    b: number
}

class Monster {
    constructor(public name: string, public initPosition: Point) {

    }
```

}

var scaring = new Monster(“Alien”, {a: 0, b: 0});

As you must have noticed, the constructor has used the keyword “*public”* before each variable name. This is an indication that we need to make the parameter to be a public property on our object.

Inheritance

In TypeScript, the *"extends"* keyword is used for the purpose of supporting a single inheritance. The code given below demonstrates how this can be done:

```
class Point3 extends Point {
    z: number;
    constructor(x: number, y: number, z: number) {
    super(x, y);
    this.z = z;
    }
    add(point: Point3) {
    var point2 = super.add(point);
    return new Point3(point2.x, point2D.y, this.z + point.z);
    }
}
```

In case there is a constructor in your class, you will be forced to call the parent constructor

from the constructor. This will ensure that the stuff which is needed for setting this will be set. Followed by the call to "*super,*" one can add any additional stuff needed for doing the constructor.

Consider the second example given below, which shows one can use the "*extends*" keyword to create an inheritance in TypeScript:

```
class MyReport {
    name: string;
    constructor (name: string) {
    this.name = name;
    }
    print() {
    alert("My Report: " + this.name);
    }
}

class CashReport extends Report {
    constructor (name: string) {
    super(name);
```

```
    }

    print() {
    alert("Cash Report: " + this.name);
    }

    getLineItems() {
    alert("5 line items");
    }
}

var myreport = new CashReport("Month's Sales");
myreport.print();
myreport.getLineItems();
```

In the above example, we have created a base class with the name "*MyReport,*" and it has a single variable "*name,*" a constructor for accepting the variable name as a string, and a function with the name "*print().*" The class "*CashReport*" uses the "*extends*" keyword so as to inherit from the class "*MyReport.*" This means that the child class will have a direct access to the "*print()*" function which has been defined in the base class. The class "*CashReport*" will override the "*print()*" method found in the base class and then create

its own. The class “*CashReport”* will also forward the name value which it receives in the constructor to its base class by use of the “*super()”* call.

Statics

Static properties are supported in TypeScript, and these are shared by all of the instances of the class. The best place to put these is in a class itself, and that is exactly what happens in TypeScript. Consider the code given below, which shows how this happens:

```
class MyClass {
    static instances = 0;
    constructor() {
    MyClass.instances++;
    }
}

var s1 = new MyClass ();
var s2 = new MyClass ();
console.log(MyClass.instances); // 2
```

It is possible for one to have both static members and static functions.

Access Modifiers

In TypeScript, access modifiers are supported and these govern how variables can be accessed. A variable can be outside a class directly by its instances, or it may be accessible in the child classes. The following are the three types of access modifiers:

1. public: available on the instances everywhere.

2. private: not available for access outside class.

3. protected: available on the child classes, but not on the instances directly.

You should be aware that it will have no great impact on your programs, but if you try to access them during runtime, they will give you some errors. Consider the example given below, which shows how this can be done:

class ClassBase {

```
    public a: number;
    private b: number;
    protected c: number;
}

// EFFECT ON the INSTANCES
var myVar = new ClassBase();
foo.a; // okay
foo.b; // ERROR : private
foo.c; // ERROR : protected

// EFFECT ON THE CHILD CLASSES
class ClassChild extends ClassBase {
    constructor() {
    super();
    this.a; // okay
    this.b; // ERROR: private
    this.c; // okay
    }
}
```

As you are aware, the above modifiers will work for both member properties and the member functions.

Define using constructor

A member in a class can be initialized as shown below:

```
class MyClass {
    x: number;
    constructor(x:number) {
    this.x = x;
    }
}
```

This is a very common scenario in TypeScript, and one can simply prefix the member using an access modifier and this will be declared automatically in the class and then copied from your constructor. With this, our previous example can be written as shown below:

```
class MyClass {
```

```
    constructor(public x:number) {

    }

}
```

Property initializer

This is a very important feature which is supported in TypeScript. It is possible for you to initialize any member of the class outside the constructor, and it is good for providing a default value. Consider the example given below:

```
class MyClass {
    members = [];  // Initializing directly
    add(x) {
    this.members.push(x);
    }
}
```

Rest Parameters

This feature allows one to accept multiple parameters in a function, and then get these as an array. This can be denoted as shown in the example given below:

```
Function allFunction(first, second, …rest) {
    console.log(rest);
}
allFunction (‘sample’, ‘bar’); // []
allFunction (‘sample’, ‘bar’, ‘bas’, ‘qux’); // [‘bas’,‘qux’]
```

Chapter 7- Destructuring

Destructuring means breaking up the structure. The following types of destructuring are supported in TypeScript:

1. Object Destructuring

2. Array Destructuring

To make it easy, see destructuring as the inverse of structuring. The example given below shows how structuring can be done in JavaScript:

```
var myVar = {
   bar: {
   bas: 789
```

```
    }
};
```

With the support for structuring in JavaScript, the process of creating new objects would have been a bit tough. With destructuring, we get the same level of convenience to obtaining data out of a particular source.

Object Destructuring

With destructuring, one can do much work which could have taken lines in just a single line. Consider the example given below:

var rectangle = { a: 0, b: 10, width: 20, height: 30 };

// Destructuring the assignment

var {a, b, width, height} = rectangle;

console.log(a, b, width, height);

If we were not to use destructing, the a, b, width, and height would have been picked separately from the rectangle. Deep data can also be obtained from a structure by use of destruction. The following code best illustrates how this can be done:

var foo = { bar: { bas: 123 } };

var {bar: {bas}} = foo; // Effectively `var bas = foo.bar.bas;`

Array Destructuring

"*const*" is a new addition which is now offered in TypeScript and ES6. With it, one can be immutable with variables. This serves a good purpose in runtime and for documentation. For you to use it, you just have to replace the keyword "*var*" with "*const.*" This is best demonstrated below:

const myConst = 123;

In TypeScript, use of const is easy compared to what happens in other programming languages. It helps us to avoid magic literals, and it is a good practice for improving readability and maintainability. Consider the code given below:

// Low readability

if (a > 10) {

}

// This one is Better!

const maximumRows = 10;

if (a > maximumRows) {

}

Initialization of "*const*" declarations has to be done. Consider the compile error given below:

const myConst // ERROR: const declarations must be initialized

Also, it is impossible for a left hand assignment to be a constant. This is because after creation of constants, they become immutable, so if you try to assign them to a new value, you will get a compile error. This is shown in the code given below:

const myConst = 123;

myConst = 540; // ERROR: Left-hand side of an assignment expression cannot be a constant

As shown above, trying to do that results in the compile errors which have been commented.

Block Scoped

A "*const*" is just a block which has been scoped like we did using "*let.*" The code given below best describes this:

```
const myConst = 123;
if (true) {
    const myConst = 500; // Allowed as its a new variable limited to this `if` block
}
```

Deep immutability

It is possible for us to use "*const*" with object literals, as far as you are concerned with protecting the "*reference*" to the variable. This is shown below:

const myConst = { bar: 123 };

myConst = { bar: 500 }; // ERROR : Left hand side of an assignment expression cannot be a constant

However, it is possible for you to mute the sub properties of objects. This is shown in the code given below:

const myConst = { bar: 123 };

myConst.bar = 500; // This one is Allowed!

console.log(myConst); // { bar: 500 }

This is why it is recommended that you use the const with immutable data structures and with literals. Constants are very good when you need to maintain the value of a variable without changing it.

Chapter 8- for…of

The "*for…of*" does not iterate over the elements of an array, and this is a common problem experienced by JavaScript beginners. This one iterates over the keys of the objects which are passed in. Consider the example given below, which best describes this:

```
var myArray = [7, 2, 5];
for (var item in myArray) {
    console.log(item); // 0,1,2
}
```

In the above example, although we expect to get 7, 2, 5, we will get the indexes which are 0, 1, 2. Tihs is why the "*for…of*" was introduced in TypeScript. Consider the example given below, which will iterate over the elements of the array and output the result which is expected:

```
var myArray = [7, 2, 5];
```

```
for (var item of myArray) {
    console.log(item); // 7,2,5
}
```

The above example will output the result which you expect rather than the indexes.

Similarly, in TypeScript, it is no trouble for one to go through a string character by character by use of “for…of.” This is demonstrated in the code given below:

```
var hi = “are you the one looking for me?”;
for (var char of hi) {
    console.log(char); // are you the one looking for me?
}
```

JS Generation

TypeScript will always generate a standard “for (var i = 0; i < list.length; i++)” for pre ES6 targets. Our previous example will generate the following:

```
var myArray = [7, 2, 5];
for (var item of myArray) {
    console.log(item);
}

// it will become //

for (var _j = 0; _j < myArray.length; _j++) {
    var item = myArray[_j];
    console.log(item);
}
```

As shown in the above example, use of “for…of” will make the “*intent*” clearer and the

amount of code you are expected to write will also be reduced, as well as the number of variables you are expected to create.

Limitations

For those who are not targeting ES6 or above, the code which is generated will assume that the property "*length*" exists on our object and it can be indexed via numbers. This means that it is only supported in arrays and string for the legacy JS engines.

In case TypeScript notices that you are not using either an array or string, then you will get an error. Consider the example given below which best describes this:

```
let artParagraphs = document.querySelectorAll("article > p");
// Error: Nodelist is not an array type or a string type
for (let par of artParagraphs) {
    par.classList.add("read");
}
```

From the above demonstrations, it is very clear that "*for…of*" should only be used for objects which are either an array or a string.

Chapter 9- Iterators

An iterator is a feature which is mostly used in object-oriented programming languages. It is an object, and is used for the purpose of implementation of the following types of interfaces:

```
interface Iterator<T> {
    next(value?: any): IteratorResult<T>;
    return?(value?: any): IteratorResult<T>;
    throw?(e?: any): IteratorResult<T>;
}
```

The interface allows us to retrieve a particular value from a collection or a sequence belonging to the object. Imagine a situation in which you have an object of a frame, and this includes a list of components of which our frame consists. With the iterator interface, it becomes possible for us to retrieve components from the frame object as shown in the code given below:

```
‘use strict’;


class Component {
 constructor (public name: string) {}
}


class Frame implements Iterator<Component> {


 private point = 0;


 constructor(public name: string, public components: Component[]) {}


 public next(): IteratorResult<Component> {
    if (this.point < this.components.length) {
    return {
    done: false,
    value: this.components[this.point++]
    }
    } else {
    return {
    done: true
```

```
    }

    }

 }


}

let fr = new Frame("Door", [new Component("top"), new Component("bottom"), new Component("left"), new Component("right")]);

let iteratorResult_1 = fr.next(); //{ done: false, value: Component { name: 'top' } }

let iteratorResult_2 = fr.next(); //{ done: false, value: Component { name: 'bottom' } }

let iteratorResult_3 = fr.next(); //{ done: false, value: Component { name: 'left' } }

let iteratorResult_4 = fr.next(); //{ done: false, value: Component { name: 'right' } }

let iteratorResult_5 = fr.next(); //{ done: true }

//It is possible for us to access a value of iterator result by use of the value property:

let component = iteratorResult_1.value; //Component { name: 'top' }
```

You have to know that the iterator itself is not a feature in TypeScript, meaning the code could have worked without having to explicitly implement the iterator and the iteratorResult. However, for the sake of code consistency, it will be good for us to use

those ES6 interface features.

Consider the code given below, which shows how this can be done:

```
//…
class Frame implements Iterable<Component> {

 constructor(public name: string, public components: Component[]) {}

 [Symbol.iterator]() {
    let p = 0;
    let components = this.components;

    return {

    next(): IteratorResult<Component> {
    if (p < components.length) {
    return {
    done: false,
    value: components[p++]
    }
```

```
    } else {
    return {
    done: true
    }
    }
    }
    }
 }
}
let fr = new Frame("Door", [new Component("top"), new Component("bottom"), new Component("left"), new Component("right")]);
for (let cmp of frame) {
 console.log(cmp);
}
```

The iterator must not iterate a finite value. A good example of this is the finonacci sequence. This is best demonstrated in the code given below:

```
class Fibonacci implements IterableIterator<number> {

 protected fbn1 = 0;

 protected fbn2 = 1;

 constructor(protected maxValue?: number) {}

 public next(): IteratorResult<number> {
    var current = this.fbn1;
    this.fbn1 = this.fbn2;
    this.fbn2 = current + this.fbn1;
    if (this.maxValue && current <= this.maxValue) {
    return {
    done: false,
    value: current
    }
    } return {
    done: true
    }
```

```
 }

 [Symbol.iterator](): IterableIterator<number> {
    return this;
 }

}

let fib = new Fibonacci();

fib.next() //{ done: false, value: 0 }
fib.next() //{ done: false, value: 1 }
fib.next() //{ done: false, value: 1 }
fib.next() //{ done: false, value: 2 }
fib.next() //{ done: false, value: 3 }
fib.next() //{ done: false, value: 5 }

let fibMax50 = new Fibonacci(50);
console.log(Array.from(fibMax50)); // [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ]
```

```
let fibMax21 = new Fibonacci(21);

for(let num of fibMax21) {

 console.log(num); //will print a fibonacci sequence of 0 to 21

}
```

Chapter 10- Template Strings

These are the strings which are to use backticks instead of the single and double quotes. They are used in three areas:

- Multiline Strings

- String Interpolation

- Tagged Templates

Multiline Strings

Sometimes, one may need to embed something into their JavaScript string. For you to do this, you are required to use an escape character of the correct choice so as to put a new line in the string manually. This is shown in the code given below:

**var lcs = “You should never give up **

\nAll shall be well with you friend”;

In TypeScript, this can be done using a template string as shown below:

var lcs = ` You should never give up

All shall be well with you friend`;

String Interpolation

You might be in need of generating some strings from a static string and some variables. Some templating logic will be needed for doing this where the name for templating strings comes from. An html string could have been generated as follows previously:

var lcs = ‘ You should never give up ‘;

var html = ‘ ’ + lcs + ‘ ’;

With template strings, this can be done as follows:

var lcs = ‘ You should never give up ‘;

var html = `<div>${lcs}</div>`;

Since the placeholder inside the interpolation is treated as an expression, and then treated as shown above, you can do the following math:

```
console.log(`1 and 1 will make ${1 + 1}`);
```

Tagged Templates

A function can be placed before a template string, and this will offer it an opportunity for pre-processing the template string literals plus the variables of the placeholder expressions and then return a result. You have to know that all your static literals should be passed in as an array for your first argument.

Consider the example given below:

```
var words = “if you are a man> you wear trousers”;
var html = htmlEscape `<div> I would like to denote that : ${words}</div>`;

// an example of tag function
function htmlEscape(literals, …placeholders) {
    let result = ””;

    // interleave your literals with the placeholders
    for (let j = 0; j < placeholders.length; j++) {
```

```
    result += literals[j];
    result += placeholders[j]
    .replace(/&/g, '&amp;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#39;')
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;');
    }

    // adding the last literal
    result += literals[literals.length - 1];
    return result;
}
```

Chapter 11- Spread Operator

This operator is intended at spreading the objects of an array.

Apply

The operator is used for spreading an array into function arguments. In the past, one was expected to use the function “Function.prototype.apply.” This is shown in the code sample given below:

```
function myFunction(a, b, c) { }

var args = [0, 1, 2];

myFunction.apply(null, args);
```

For this to be done, one can prefix the arguments with (…). This is shown below:

```
function myFunction(a, b, c) { }

var args = [0, 1, 2];

myFunction(…args);
```

Destructuring

You have already seen how this can be used in destructuring. This is shown below:

var [a, b, …remaining] = [1, 2, 3, 4];

console.log(a, b, remaining); // 1, 2, [3,4]

The motivation behind this is to make it easy for one to capture the array elements which are remaining when destructuring.

Array Assignment

The spread operator can allow you to place an array which has been expanded into another array. Consider the example given below, which demonstrates how this can be done:

```
var list = [5, 6];

list = [...list, 7, 8];

console.log(list); // [5,6,7,8]
```

Chapter 12- Enums

An enum provides us with a way to organize collections having related values. Other programming languages provide us with an enum data type except JavaScript. This is supported in TypeScript. In TypeScript, an enum can be declared as shown below:

```
enum Card {
    Clubs,
    Spades,
    Diamonds,
    Hearts
}
// A sample usage
var card = Card.Clubs;

// Safety
card = “It is not a member of the card suit”; // Error : string is not assignable to type `CardSuit`
```

Enums and Numbers

Enums in TypeScfript are number-based. This is an indication that we can assign numbers to an instance of an enum in TypeScript, and the same applies to any type which is compatible to numbers. The code given below best describes how this can be done:

```
enum Color {
    Red,
    Green,
    Blue
}
var color = Color.Red;
color = 0; // This is the same as Color.Red
```

Enums and Strings

We want to know the kind of JavaScript which is produced by enums. Consider the sample TypeScript given below:

```
enum MyEnum {
    False,
    True,
    Unknown
}
```

The following JavaScript code will be generated from the above TypeScript code:

```
var MyEnum;
(function (MyEnum) {
    MyEnum [MyEnum [“False”] = 0] = “False”;
    MyEnum [MyEnum [“True”] = 1] = “True”;
```

```
    MyEnum [MyEnum [“Unknown”] = 2] = “Unknown”;
})( MyEnum || (MyEnum = {}));
```

The variable “*MyEnum*” can be used for the purpose of converting an enum of string version into a number or maybe a number version of the enum. The code given below best describes this:

```
enum MyEnum {
    False,
    True,
    Unknown
}
console.log(MyEnum [0]); // “False”
console.log(MyEnum [“False”]); // 0
console.log(MyEnum [MyEnum.False]); // “False” because ` MyEnum.False == 0`
```

It is also possible for us to change the number which is associated with the enum. The default setting is that enums are 0 and each subsequent increment is automatically made in intervals of 1. Consider the code given below which best describes this:

```
enum Color {
    Red,     // 0
    Green,   // 1
    Blue     // 2
}
```

The number associated with a specific enum can be automatically changed by specifically assigning to it. The example given below shows how this can be done, we begin at 3, and then increment from there as shown below:

```
enum Color {
    LightRed = 3,  // 3
    DarkGreen,    // 4
    LightBlue      // 5
}
```

You also have to know that enums are open ended. Consider the JavaScript code given below, which has been generated from an enum:

```
var MyEnum;
(function (MyEnum) {
    MyEnum [MyEnum [“False”] = 0] = “False”;
    MyEnum [MyEnum [“True”] = 1] = “True”;
    MyEnum [MyEnum [“Unknown”] = 2] = “Unknown”;
})( MyEnum || (MyEnum = {}));
```

An enum definition can be spread across multiple files. Consider the example given below, in which we are spreading the definition of Color to get two blocks:

```
enum Color {
    Red,
    Green,
    Blue
}

enum Color {
    LightRed = 3,
    DarkGreen,
```

LightBlue

}

Enums as flags

Flgs provide us with an excellent ability to use enums. Consider the example given below:

```
enum Animal {
    None          = 0,
    PossesClaws       = 1 << 0,
    Flying        = 1 << 1,
    ConsumesFish      = 1 << 2,
    Endangered    = 1 << 3
}
```

In the above case, we need to get bitwise disjoint operators by moving the 1 around.

Consider the next example given below:

```
enum Animal {
```

```
    None          = 0,
    PossesClaws       = 1 << 0,
    Flies         = 1 << 1,
}

function printAnimalAbilities(animal) {
    var animFlags = animal.flags;
    if (animFlags & AnimFlags.PossesClaws) {
    console.log('The animal has claws');
    }
    if (animFlags & AnimFlags.Flies) {
    console.log('The animal can fly');
    }
    if (animFlags == AnimFlags.None) {
    console.log('nothing');
    }
}

var animal = { flags: AnimFlags.None };
printAnimalAbilities(animal); // nothing
animal.flags |= AnimFlags.PossesClaws;
```

```
printAnimalAbilities(animal); //The animal has claws

animal.flags &= ~AnimFlags.PossesClaws;

printAnimalAbilities(animal); // nothing

animal.flags |= AnimFlags.PossesClaws | AnimFlags.Flies;
printAnimalAbilities(animal); // The animal has claws and the animal it flies
```

Flags can be combined for the purpose of creating convenient shortcuts in the definition of the enum. This is shown in the code given below:

```
enum Animal {
    None           = 0,
    PossesClaws       = 1 << 0,
    Flies          = 1 << 1,
    EatsFish       = 1 << 2,
    Endangered     = 1 << 3,

    EndangeredFliesClawedEatFish = PossesClaws | Flies | EatsFish | Endangered,
}
```

Const Enums

Suppose that you have an enum defined as follows:

```
enum MyEnum {
    False,
    True,
    Unknown
}

var lie = MyEnum.False;
```

For you to boost the performance, you have to mark the enum as a “const enum.” This is shown in the code given below:

```
const enum MyEnum {
    False,
```

```
    True,
    Unknown
}

var lie = MyEnum.False;
```

The above will give the following JavaScript code after compilation:

```
var lie = 0;
```

What the compiler does is that it inilines any enum which has been used. Also, no JavaScript will be generated for the definition of the enum. This is because the usages for this will be inlined.

Chapter 13- let

In JavaScript, variables are function scoped. This is not the case with several other programming languages, such as Java and C#, in which variables are block scoped. Consider the JavaScript code given below:

```
var myVar = 123;
if (true) {
    var myVar = 500;
}
console.log(myVar); // 500
```

Although you expected the above code to print 123, it will print 500. This is because our variable "*myVar*" remains to be the same both inside and outside the "*if*" block. This is one of the most common sources of errors in JavaScript. In TypeScript, the keyword "*let*" was introduced so as to enable the programmers to define variables having a true block scope. This means that if you use the keyword "*let*" rather than the "*var*" keyword to define your variable, it will be seen and used differently as outside the scope. The previous example can be written using the "*let*" keyword as shown below:

```
let myVar = 123;
if (true) {
    let myVar = 500;
}
console.log(myVar); // 123
```

Consider the example given below, which shows how the "*let*" keyword can be used in loops:

```
var index = 0;
var myArray = [1, 2, 3];
for (let index = 0; index < myArray.length; index++) {
    console.log(myArray[index]);
}
console.log(index); // 0
```

Replacing it with the "*let*" keyword will help save you from errors of the loop. This is shown below:

```
var index = 0;
var myArray = [1, 2, 3];
for (let index = 0; index < myArray.length; index++) {
    console.log(myArray[index]);
}
console.log(index); // 0
```

As shown above, it is always good for you to use the "*let*" keyword whenever possible, and you will be saved from many surprises and errors.

Functions are also used for the purpose of creating a new scope. We are in need of demonstrating how a function can be used for creating a new variable scope in JavaScript. Consider the code given below:

```
var myVar = 123;
function testFunction() {
    var myVar = 500;
}
```

testFunction();

console.log(myVar); // 123

The above code will behave just as you would like. Without it, it will be hard for one to write their JavaScript code.

Generated JS

TypeScript generates a JS code in which it is simple to rename the let variable if there exists a similar name in the surrounding. Consider the code given below, which shows a simple replacement of the *var* with *let.* Here is the code:

```
if (true) {
    let myVar = 123;
}

// will become //

if (true) {
    var myVar = 123;
}
```

However, if the name of the variable has readily been taken by the surrounding scope, then a new variable name will be generated as shown in the code given below:

```
var myVar = ‘123’;

if (true) {

    let myVar = 123;

}


// will become //


var myVar = ‘123’;

if (true) {

    var _myVar = 123; // It has been Renamed

}
```

let in closures

Consider the simple JavaScript code given below:

```
var functions = [];
// creating a bunch of functions
for (var j = 0; j < 3; j++) {
    functions.push(function() {
    console.log(j);
    })
}
// call them
for (var k = 0; k < 3; k++) {
    functions[k]();
}
```

In most interview questions, you are asked about the log in the above JavaScript file. Although most people expect the answer to be 0, 2,3, but the real answer is 3. The reason

is that the three functions are making use of the variable "*j*" from our outer scope, and we will in turn execute them the value of the variable "*j*" will be 3. It will form the termination condition for our first loop.

The solution to this problem is creation of a variable for each variable, and this should be specifically used for iteration in that loop. As you are aware, a new scope for a variable can be created by creation of a new function and then executing it immediately. This is shown in the code given below:

```
var functions = [];
// creating a bunch of functions
for (var j = 0; j < 3; j++) {
    (function() {
    var local = j;
    functions.push(function() {
    console.log(local);
    })
    })();
}
// calling them
```

```
for (var k = 0; k < 3; k++) {
    funcs[k]();
}
```

Once you use the *let* keyword rather than the *var* keyword, a variable which is unique to each loop iteration is created.

Conclusion

We have come to the conclusion of this guide. TypeScript is a programming language which always compiles into JavaScript. The language is open source, and it was developed and maintained by Microsoft. It is good for the development of large applications in which features such as static typing and object oriented programming are supported. These two features are the main ones which suit it for use to develop large applications.

The language supports type annotations which are very good for maintaining the indentations in a program. Indentations usually ensure that a program is well readable. You have to remember that TypeScript is a superset of JavaScript, meaning that any JavaScript code is also a TypeScript code. This is the reason why TypeScript always compiles into JavaScript. For you to begin programming in TypeScript, you have to first get the environment ready for doing that. For Windows users, you just have to simply install the Visual Studio and then download and install the necessary plugin, and you will be set to get programming.

If you are using any other platform, then you have to rely on the Node package named "*npm.*" Ensure that you have a text editor and a browser. Once you have installed the npm package for TypeScript, you can go ahead and begin to write your TypeScript programs.